# The KHE Timetabling Platform and Solvers

Jeffrey H. Kingston
*jeff@it.usyd.edu.au*

Version 2.13 (December 2024)

# Contents

# Part A

# The Platform

# Chapter 1. Introduction

Some instances of high school timetabling problems, taken from institutions in several countries and specified formally in an XML format called XHSTT, have recently become available [13]. For the first time, the high school timetabling problem can be studied in its full generality.

KHE is an open-source ANSI C library, released under the GNU public licence, which aims to provide a fast and robust foundation for solving instances of high school timetabling problems expressed in the XHSTT format. Users of KHE may read and write XML files, create solutions, and add and change time and resource assignments using any algorithms they wish. The cost of the current solution is always available, kept up to date by a hand-coded constraint propagation network. KHE also offers features inherited from the author's KTS system [6, 8], notably layer trees and matchings, and solvers for several major sub-tasks.

KHE is intended for production use, but it is also a research vehicle, so new versions will not be constrained by backward compatibility. Please report bugs to me at *jeff@it.usyd.edu.au.* I will release a corrected version within a few days of receiving a bug report, wherever possible.

This introductory chapter explains how to install and use KHE, surveys its data types, and describes some operations common to many types.

## 1.1. Installation and use

KHE has a home page, at

```
http://jeffreykingston.id.au/khe/
```

The current version of KHE is a gzipped tar file in that directory. The current version of this documentation (a PDF file) is also stored there. The names of these files change with each release; they are most easily downloaded using links on the home page.

Originally, 'KHE' stood for 'Kingston's High School Timetabling Engine', but it now covers all timetabling software released by me: the platform, the solvers, HSeval (which drives the HSEval web site), my nurse rostering software, and anything else I release in the future. So 'KHE' no longer stands for anything, except possibly 'Kingston's Humungous Enterprise'.

I have used different kinds of version numbers over the years, but starting with Version 2.1 I am reverting to the traditional form, of a major release number and minor release number separated by a dot. Each KHE release is a release of all my software under a single version number.

A program that incorporates the KHE platform can gain access to the current version number by calling

```
char *KheVersionNumber(void);
char *KheVersionBanner(void);
```

For example, if Version 2.6 is compiled into the program that calls these functions, their results will be `"2.6"` and `"Version 2.6 (March 2021)"`.

To install KHE, download a release and unpack it using `gunzip` and `tar xf` as usual. The resulting directory, `khe`, contains a makefile, some `src_*` directories holding the source files of KHE, and some `doc_*` directories holding the source files of this documentation. Consult the makefile for information about what's what in the distribution, and how to install and use KHE.

Starting with Version 2.1, the KHE source files are divided into three parts: the platform (whose interface is file `khe_platform.h`), the solvers (`khe_solvers.h`), and a main program. This allows users to use only the platform, or it and the solvers, or those plus a main program. The distribution also contains two source directories holding my nurse rostering software.

## 1.2. The data types of KHE

This section is an overview of KHE's data types. The following chapters have the details.

Type `KHE_ARCHIVE` represents one archive, that is, a collection of instances plus a collection of solution groups. Type `KHE_SOLN_GROUP` represents one solution group, that is, a set of solutions of the instances of the archive it lies in. The word 'solution' is abbreviated to 'soln' wherever it appears in the KHE interface. Use of these types is optional: instances do not have to lie in archives, and solutions do not have to lie in solution groups.

Type `KHE_INSTANCE` represents one instance of the high school timetabling problem. `KHE_TIME_GROUP` represents a set of times; `KHE_TIME` represents one time. `KHE_RESOURCE_TYPE` represents a resource type (typically *Teacher*, *Room*, *Class*, or *Student*); `KHE_RESOURCE_GROUP` represents a set of resources of one type; and `KHE_RESOURCE` represents one resource.

Type `KHE_EVENT_GROUP` represents a set of events; `KHE_EVENT` represents one event, including all information about its time. Type `KHE_EVENT_RESOURCE` represents one resource element within an event. Type `KHE_CONSTRAINT` represents one constraint. This could have any of the constraint types of the XML format (it is their abstract supertype).

Type `KHE_SOLN` represents one solution, complete or partial, of a given instance, optionally lying within a solution group. Type `KHE_MEET` represents one meet (KHE's commendably brief name for what the XML format calls a solution event, split event, or sub-event: one event as it appears in a solution), including all information about its time. Type `KHE_TASK` represents one piece of work for a resource to do: one resource element within a meet.

KHE supports multi-threading by ensuring that each instance and its components (of type `KHE_INSTANCE`, `KHE_TIME_GROUP`, and so on) is immutable after loading of the instance is completed, and that operations applied to one solution object do not interfere with operations applied simultaneously to another. Thus, after instance loading is completed, it is safe to create multiple threads with different `KHE_SOLN` objects in each thread, all referring to the same instance, and operate on those solutions in parallel. No such guarantees are given for operating on the same solution from different threads.

## 1.3. Memory allocation using arenas and arena sets

Large solves of timetabling instances can make heavy demands on memory allocators. A search of the internet will show that there is no ideal memory allocator, that is, one that implements the C `malloc`, `realloc`, and `free` functions with negligible time and memory overhead, taking the demands of memory caching and multi-threading into account. What follows is an attempt to do memory allocation for KHE well at the cost of requiring the user to pay more attention to it than is customary. For serious solving this seems to the author to be the right tradeoff.

An *arena memory allocator* allocates memory from an area called an *arena*. A KHE arena enlarges, effectively without limit, as memory from it is allocated. There is no operation to free one piece of arena memory. Instead, the entire arena is freed in one operation. This allows an arena memory allocator to be faster than a general memory allocator, and to use less memory.

An arena memory allocator is used throughout KHE. It is flown in from another project of the author's called Howard, so the type name of a memory arena is not `KHE_ARENA` but `HA_ARENA`. One can create an arena, obtain memory from it, and delete (free) it. The memory can have a fixed size, or it can be resizable (supporting varying-length arrays and hash tables).

Every arena belongs to exactly one *arena set*, an object of type `HA_ARENA_SET` representing a set of arenas. When an arena has no memory available to satisfy a request for memory from a caller, it obtains a new chunk of memory from its arena set; and when an arena is deleted, its chunks return to its arena set and become available to other arenas belonging to that arena set.

There is no locking within types `HA_ARENA` and `HA_ARENA_SET`, other than within the underlying calls to `malloc` which are the ultimate source of the memory that arena sets give out to their arenas. In a multi-threaded program, this constrains how arenas and arena sets are used.

The rule for arena sets is very simple. There should be exactly one arena set per thread, and all memory allocated by code running within that thread should come from arenas that belong to that arena set. This ensures that no arena set or arena is ever accessed by two different threads, and so no locking of arena sets or arenas is required. (There is no prohibition on having two arena sets within one thread, but that is not recommended because it can waste memory.)

When a new thread is begun, a new arena set should be created and passed to all code in that thread that needs memory. When a thread terminates, the operation for merging one arena set into another should be called from the parent thread to merge the terminating child thread's arena set into the parent thread's arena set. All variables pointing to the child thread's arena set must be changed to point to the parent thread's arena set. In practice users do not have to worry about all this, because it is done for them by `KheArchiveParallelSolve` (Section 8.5).

When some single-threaded code needs memory, in simple cases it can be passed a single arena, from which it can obtain its memory. That memory remains in use until the caller who passed it the arena deletes that arena. In complex cases the code should be passed the arena set of its thread, so that it can create and delete its own arenas as required.

For example, a solution object, of type `KHE_SOLN`, is a large, complex object. It is created by function `KheSolnMake` (Section 4.2) in its own arena, so that it can be deleted individually, if required, by deleting the arena. But there are also *placeholder solutions* which are mostly deleted but which retain a small amount of information, including solution cost. So in fact a solution is created in two arenas, one holding the placeholder part and the other holding the rest. Reducing a full solution to a placeholder involves deleting the second arena but not the first. Given this complexity, the right course is to pass the enclosing thread's arena set to `KheSolnMake`, and leave it to `KheSolnMake` to create whatever arenas it needs.

Similar considerations apply to solvers created by users. Typically, a solver will create an arena, use it to hold whatever objects it uses, and then, when its solve operation is complete, it will delete its arena, freeing up the memory it used, and leaving no trace of itself other than any changes it may have made to the solution object it worked on. For this it needs to be passed its thread's arena set. But in simple cases it is also reasonable for a solver to accept an arena parameter and simply use that arena.

KHE supports multi-threading, but it assumes that each solution object will be accessed by only one thread. Each solution contains the arena set of that thread, and a solver can retrieve that arena set and use it to create arenas. Alternatively, a solver can call `KheSolnArenaBegin` (Section 4.2.2) to obtain one arena from that arena set, and `KheSolnArenaEnd` to delete that arena when it is no longer required. In practice this is the usual way to gain access to an arena.

When solutions are read single-threaded for evaluation, they all reference the same arena set. When they are created for solving in parallel, they have different arena sets. There is no simple way to transition from one of these situations to the other.

Arena sets offer a means of failing gracefully if memory runs out (quite likely during large solves). One can pass a C `longjmp` environment to an arena set, and then if any request for memory from any of that arena set's arenas cannot be satisfied, the arena code will long jump to the target of the `longjmp` environment instead of returning. There is also a simple way to prevent any one thread from monopolizing the available memory. For more on these issues and on arenas and arena sets generally, consult Appendix A.1.

## 1.4. Common operations

This section describes some miscellaneous operations that are common to many data types.

Whenever KHE creates an object, any string-valued attributes of that object passed by the user are not stored directly; instead, malloced copies are stored. If the object is later deleted, the malloced copy is deleted along with it. Thus, whatever its origin, a string-valued attribute has the same lifetime as the object itself.

Use of KHE often involves creating objects that contain references to KHE entities (objects of types defined by KHE) alongside other information. Sometimes it is necessary to go backwards, from a KHE entity to a user-defined object. Accordingly, each KHE entity contains a *back pointer* which the user is free to set and retrieve, using calls which look generically like this:

```
void KheEntitySetBack(KHE_ENTITY entity, void *back);
void *KheEntityBack(KHE_ENTITY entity);
```

All back pointers are initialized to `NULL`. In general, KHE itself does not set back pointers. The exception is that some solvers packaged with KHE set the back pointers of the solution entities they deal with. This is documented where it occurs.

Timetables often contain symmetries of various kinds. In high school timetabling, the student group resources of one form are often symmmetrical: they attend the same kinds of events over the course of the cycle.

Knowledge of similarity can be useful when solving. For example, it might be useful to timetable similar events attended by student group resources of the same form at the same time. Accordingly, several KHE entities offer an operation of the form

```
bool KheEntitySimilar(KHE_ENTITY e1, KHE_ENTITY e2);
```

which returns `true` if KHE considers that the two entities are similar. If they are the exact same entity, they are always considered similar. In other cases, the definition of similarity varies with the kind of entity, although it follows a common pattern: evidence both in favour of similarity and against it is accumulated, and there needs to be a significant amount of evidence in favour, and more evidence in favour than against. For example, an event containing no event resources will never be considered similar to any event except itself, since positive evidence, such as requests for the same kinds of teachers, is lacking.

Similarity is not a transitive relation in general. In other words, if `e1` and `e2` are similar, and `e2` and `e3` are similar, that does not imply that `e1` and `e3` are similar. There is a heuristic aspect to it that seems inevitable, although the intention is to stay on the safe side: to declare two entities to be similar only when they clearly are similar.

Another operation that applies to many entities, albeit a humble one, is printing the current state of the entity as an aid to debugging. The KHE operations for this mostly take the form

```
void KheEntityDebug(KHE_ENTITY entity, int verbosity,
  int indent, FILE *fp);
```

They produce a debug print of `entity` onto `fp`.

The `verbosity` parameter controls how much detail is printed. Any value is acceptable. A zero or negative value always prints nothing. Every positive value prints something, and as the value increases, more detail is printed, depending, naturally, on the kind of entity. Value 1 tries to print the minimum amount of information needed to identify the entity, often just its name.

If `indent` is non-negative, a multi-line format is used in which each line begins with at least `indent` spaces. If `indent` is negative, the print appears on one line with no indent and no concluding newline. Since space is limited, verbosity may be reduced when `indent` is negative.

Many entities are organized hierarchically. Depending on the verbosity, printing an entity may include printing its descendants. Their debug functions are passed a value for `indent` which is 2 larger than the value received (when non-negative), so that the hierarchy is represented in the debug output by indenting. The debug print of one entity usually begins with `[` and ends with a matching `]`, making it easy to move around the printed hierarchy using a text editor.

## 1.5. KHE for employee scheduling

Recent versions of KHE support the employee scheduling data format XESTT as well as the high school timetabling format XHSTT. XESTT is the same as XHSTT except for a few extensions, which are documented on the HSEval web site.

KHE knows whether it is dealing with XESTT or XHSTT, but it does not care—it supports XESTT, which includes supporting XHSTT. When using KHE for high school timetabling, several parameters of KHE functions have to be given values that indicate that the extensions available in XESTT are not used. This mainly affects the operations for creating cluster busy times and limit busy times constraints.

# Chapter 2. Archives and Solution Groups

This chapter describes the `KHE_ARCHIVE` and `KHE_SOLN_GROUP` data types, representing archives and solution groups as in the XML format. Their use is optional, since instances are not required to lie in archives, and solutions are not required to lie in solution groups.

## 2.1. Archives

An archive is defined in the XML format to be a collection of instances together with groups of solutions to those instances. There may be any number of instances and solution groups. To create a new, empty archive, call

```
KHE_ARCHIVE KheArchiveMake(char *id, KHE_MODEL model, HA_ARENA_SET as);
```

Parameter `id` is an identifier for the archive. It may be `NULL`, but only if the archive is not going to be written. Parameter `model` says what problem the archive models, for which see below. Parameter `as` is the thread arena set used for obtaining memory. Appendix A.1.2 introduces arena sets, and Appendix B.7 explains why one arena set per thread is good. You can also pass `NULL` for `as`, but there will be some loss of efficiency in memory allocation which could be critical when handling large archives.

To delete an archive, call

```
void KheArchiveDelete(KHE_ARCHIVE archive);
```

This includes deleting `archive`'s instances (unless they lie in other archives as well), and deleting its solution groups and solutions. The memory is returned to the arena set that was used to create `archive`, except that instances decide for themselves where to return their memory to. Also,

```
void KheArchiveClear(KHE_ARCHIVE archive);
```

deletes the instances and solution sets of `archive`, leaving it empty except for its metadata. There is also

```
void KheArchiveMeld(KHE_ARCHIVE dst_archive, KHE_ARCHIVE src_archive);
```

which melds the instances, solution sets, and solutions of `src_archive` into `dst_archive`, aborting if this is not possible (because the two archives have different models, or an instance from `src_archive` has the same Id as an instance from `dst_archive`).

Although created to support the XHSTT high school timetabling model, KHE also supports an extended version of XHSTT, used for nurse rostering. Accordingly, type `KHE_MODEL` is

```
typedef enum {
  KHE_MODEL_HIGH_SCHOOL_TIMETABLE,
  KHE_MODEL_EMPLOYEE_SCHEDULE
} KHE_MODEL;
```

The model affects the initial tag read by `KheArchiveRead` and written by `KheArchiveWrite`, which is `<HighSchoolTimetableArchive>` when it is `KHE_MODEL_HIGH_SCHOOL_TIMETABLE` and `<EmployeeScheduleArchive>` when it is `KHE_MODEL_EMPLOYEE_SCHEDULE`. Instances also have a model, which must agree with the model of any archive they lie in. Thus, it is not possible to mix instances with different models in one archive. Functions

```
char *KheArchiveId(KHE_ARCHIVE archive);
KHE_MODEL KheArchiveModel(KHE_ARCHIVE archive);
```

return these attributes of an archive. To set and retrieve the back pointer (Section 1.4), call

```
void KheArchiveSetBack(KHE_ARCHIVE archive, void *back);
void *KheArchiveBack(KHE_ARCHIVE archive);
```

Archive metadata may be set and retrieved by calling

```
void KheArchiveSetMetaData(KHE_ARCHIVE archive, char *name,
  char *contributor, char *date, char *description, char *remarks);
void KheArchiveMetaData(KHE_ARCHIVE archive, char **name,
  char **contributor, char **date, char **description, char **remarks);
```

The values retrieved are copies of those passed in, as usual. The initial values are all `NULL`. When a metadata value is required when writing an archive, any `NULL` or empty values are written as `"No name"`, `"No contributor"`, etc. There is also

```
char *KheArchiveMetaDataText(KHE_ARCHIVE archive)
```

which returns a string containing the metadata as a paragraph of English text, for example

```
This archive is XHSTT-2014, assembled by Gerhard Post on 2 March 2014.
```

The string lies in the archive's arena and is deleted when the archive is deleted.

Initially an archive contains no instances and no solution groups. Solution groups are added automatically as they are created, because every solution group lies in exactly one archive. An instance may be added to an archive by calling

```
bool KheArchiveAddInstance(KHE_ARCHIVE archive, KHE_INSTANCE ins);
```

`KheArchiveAddInstance` returns `true` if it succeeds in adding `ins` to `archive`, and `false` otherwise, which can either be because `archive` already contains an instance with `ins`'s Id, or because the instance and archive models differ. The instance will appear after any instances already present. An instance may be deleted from an archive (but not destroyed) by calling

```
void KheArchiveDeleteInstance(KHE_ARCHIVE archive, KHE_INSTANCE ins);
```

`KheArchiveDeleteInstance` aborts if `ins` is not in `archive`. If there are any solutions for `ins` in `archive`, they are deleted too. The gap left by deleting the instance is filled by shuffling subsequent instances up one place.

To visit the instances of an archive, call

```
int KheArchiveInstanceCount(KHE_ARCHIVE archive);
KHE_INSTANCE KheArchiveInstance(KHE_ARCHIVE archive, int i);
```

The first returns the number of instances in `archive`, and the second returns the `i`'th of those instances, counting from 0 as usual in C. There is also

```
bool KheArchiveRetrieveInstance(KHE_ARCHIVE archive, char *id,
  KHE_INSTANCE *ins, int *index);
```

If `archive` contains an instance with the given `id`, this function sets `ins` to that instance and `*index` to its index in `archive` and returns `true`; otherwise it sets `*ins` to `NULL` and `*index` to `-1` and returns `false`. And

```
bool KheArchiveContainsInstance(KHE_ARCHIVE archive,
  KHE_INSTANCE ins, int *index);
```

is the function to call when the instance is given and just its index is needed.

For visiting the solution groups of an archive, call

```
int KheArchiveSolnGroupCount(KHE_ARCHIVE archive);
KHE_SOLN_GROUP KheArchiveSolnGroup(KHE_ARCHIVE archive, int i);
```

similarly to visiting instances. There is also

```
bool KheArchiveRetrieveSolnGroup(KHE_ARCHIVE archive, char *id,
  KHE_SOLN_GROUP *soln_group);
```

which retrieves a solution group by `id`.

## 2.2. Solution groups

A solution group is a set of solutions to instances of its archive. To create a solution group, call

```
bool KheSolnGroupMake(KHE_ARCHIVE archive, char *id,
  KHE_SOLN_GROUP *soln_group);
```

Here `archive` is compulsory, and the solution group is added to it. Parameter `id` is the Id attribute from the XML file; it is optional, with `NULL` meaning absent, although it is compulsory if `archive` is to be written later. If the operation is successful, then `true` is returned with `*soln_group` set to the new solution group; if not (which can only be because `id` is already the Id of a solution group of `archive`), then `false` is returned with `*soln_group` set to `NULL`.

To delete a solution group, including deleting it from its archive, call

```
void KheSolnGroupDelete(KHE_SOLN_GROUP soln_group);
```

The solutions within `soln_group` are not deleted.

To set and retrieve the back pointer (Section 1.4) of a solution group, call

```
void KheSolnGroupSetBack(KHE_SOLN_GROUP soln_group, void *back);
void *KheSolnGroupBack(KHE_SOLN_GROUP soln_group);
```

as usual. To retrieve the archive and Id, call

```
KHE_ARCHIVE KheSolnGroupArchive(KHE_SOLN_GROUP soln_group);
char *KheSolnGroupId(KHE_SOLN_GROUP soln_group);
```

Solution group metadata may be set and retrieved by calling

```
void KheSolnGroupSetMetaData(KHE_SOLN_GROUP soln_group,
  char *contributor, char *date, char *description,
  char *publication, char *remarks);
void KheSolnGroupMetaData(KHE_SOLN_GROUP soln_group,
  char **contributor, char **date, char **description,
  char **publication, char **remarks);
```

As usual, copies of the strings are stored, not the originals. As for archive metadata, any of these strings may be `NULL` or empty. KHE substitutes values `"No contributor"`, `"No date"`, etc. for such values when writing an archive, or omits them altogether when XHSTT allows. Also,

```
char *KheSolnGroupMetaDataText(KHE_SOLN_GROUP soln_group);
```

returns a string containing the metadata as a paragraph of terse English text. The string lies in the solution group's arena and will be deleted when the solution group is deleted.

Initially a solution group has no solutions. These are added and deleted by calling

```
void KheSolnGroupAddSoln(KHE_SOLN_GROUP soln_group, KHE_SOLN soln);
void KheSolnGroupDeleteSoln(KHE_SOLN_GROUP soln_group, KHE_SOLN soln);
```

A solution can only be added when its instance lies in the solution group's archive.

To visit the solutions of a solution group, call

```
int KheSolnGroupSolnCount(KHE_SOLN_GROUP soln_group);
KHE_SOLN KheSolnGroupSoln(KHE_SOLN_GROUP soln_group, int i);
```

Solutions have no Ids, so there is no `KheSolnGroupRetrieveSoln` function. When solution `i` is deleted, `KheSolnGroupSolnCount` decreases by 1, solution `i+1` becomes solution `i`, and so on. To visit the solutions of a solution group that solve a particular instance, call

```
KHE_SOLN_SET KheSolnGroupInstanceSolnSet(KHE_SOLN_GROUP soln_group,
  KHE_INSTANCE ins);
```

Or if the index of the instance in the `soln_group`'s archive is known, one can call

```
KHE_SOLN_SET KheSolnGroupInstanceSolnSetByIndex(
  KHE_SOLN_GROUP soln_group, int index);
```

As described just below, `KHE_SOLN_SET` is a set of solutions. The set returned by these functions holds the solutions in `soln_group` for the indicated instance. It is stored in `soln_group` and must not be modified by the user, except that it may be sorted. KHE updates it as solutions are added and deleted from its enclosing solution group, and deletes it when its instance is deleted.

## 2.3. Solution sets

Like a solution group, a solution set contains a set of solutions. But, unlike a solution group, that is all it contains: it is not considered to lie in any archive, and it has no Id and no metadata.

To create a new, empty solution set, and to delete it (but not its solutions), call

```
KHE_SOLN_SET KheSolnSetMake(HA_ARENA a);
```

As usual it (but not its solutions) will be deleted when `a` is deleted. There is also

```
void KheSolnSetClear(KHE_SOLN_SET ss);
```

which empties out `ss` without deleting it. To add a solution, and to delete one, call

```
void KheSolnSetAddSoln(KHE_SOLN_SET ss, KHE_SOLN soln);
void KheSolnSetDeleteSoln(KHE_SOLN_SET ss, KHE_SOLN soln);
```

To find out if a solution set contains a given solution, call

```
bool KheSolnSetContainsSoln(KHE_SOLN_SET ss, KHE_SOLN soln, int *pos);
```

It returns `true` if `ss` contains `soln`, setting `*pos` to its index in `ss` if so.

To visit the elements of a solution set, call

```
int KheSolnSetSolnCount(KHE_SOLN_SET ss);
KHE_SOLN KheSolnSetSoln(KHE_SOLN_SET ss, int i);
```

They have the order they were inserted in, unless this has been changed by calling either of

```
void KheSolnSetSort(KHE_SOLN_SET ss,
  int(*compar)(const void *, const void *));
void KheSolnSetSortUnique(KHE_SOLN_SET ss,
  int(*compar)(const void *, const void *));
```

`KheSolnSetSort` sorts the solutions according to comparison function `compar`, which must be suitable for passing to `qsort`. `KheSolnSetSortUnique` is the same, but afterwards it removes all but one of each run of solutions for which `compar` returns 0.

One comparison function is already written, in one form that makes sense to people and another that makes sense to `qsort`:

```
int KheIncreasingCostTypedCmp(KHE_SOLN soln1, KHE_SOLN soln2);
int KheIncreasingCostCmp(const void *t1, const void *t2);
```

It sorts the solution set so that the solutions have increasing cost. Solutions with equal cost have increasing running time. Invalid solutions are treated as though they have infinite cost, and solutions with no running time recorded are treated as though they have infinite running time.

Finally,

```
void KheSolnSetDebug(KHE_SOLN_SET ss, int verbosity,
  int indent, FILE *fp);
```

sends a debug print of ss to fp with the given verbosity and indent.

## 2.4. Reading archives

KHE reads and writes archives in XHSTT, a standard XML format [13], and in XESTT, an extension of XHSTT for employee scheduling problems [10, 11]. To read an archive, call

```
bool KheArchiveRead(FILE *fp, HA_ARENA_SET as, KHE_ARCHIVE *archive,
  KML_ERROR *ke, bool audit_and_fix, bool resource_type_partitions,
  bool infer_resource_partitions, bool allow_invalid_solns,
  KHE_SOLN_TYPE soln_type, FILE *echo_fp);
```

File fp must be open for reading UTF-8, and it remains open after the call returns. If, starting from its current position, fp contains a legal XML archive, then KheArchiveRead sets *archive to that archive, passing it as as its arena set parameter, and *ke to NULL and returns true with fp moved to the first character after the archive. If there was a problem reading the file, then it sets *archive to NULL and *ke to an error object and returns false. Any reports in the archive are discarded without checking.

Type KML_ERROR is from the KML module packaged with KHE. A full description of the KML module appears in Section A.5. Given an object of type KML_ERROR, operations

```
int KmlErrorLineNum(KML_ERROR ke);
int KmlErrorColNum(KML_ERROR ke);
char *KmlErrorString(KML_ERROR ke);
```

return the line number, the column number, and a string description of the error.

KheArchiveRead builds the archive using the functions of this guide; there is nothing special about the archive it builds. The model, for the archive and instances, depends on the initial tag: KHE_MODEL_HIGH_SCHOOL_TIMETABLE when it is <HighSchoolTimetableArchive>, and KHE_MODEL_EMPLOYEE_SCHEDULE when it is <EmployeeScheduleArchive>.

The audit_and_fix, resource_type_partitions, and infer_resource_partitions parameters are passed on to KheInstanceMakeEnd (Section 3.1). KheArchiveRead builds complete representations of the solutions it reads. To be precise, it calls functions KheSolnMakeCompleteRepresentation, KheSolnAssignPreassignedTimes, and KheSolnAssignPreassignedResources (Section 4.3), but not KheSolnMatchingBegin or KheSolnEvennessBegin (Chapter 7).

Usually, if there are errors in the file, KheArchiveRead returns false and sets *ke to the first error. But if allow_invalid_solns is true, then some errors lying in solutions are handled differently: the erroneous solutions are converted to invalid placeholders (Section 4.2.6). Each invalid placeholder solution contains its first error, and none of its errors cause false to be returned or *ke to be set. Not all errors, not even all errors lying in solutions, can be handled in this way; those that cannot cause KheArchiveRead to return false and set *ke as usual.

Each valid solution is passed to function KheSolnTypeReduce along with parameter soln_type. If soln_type is KHE_SOLN_ORDINARY this does nothing, but other values reduce the solution to a placeholder, freeing up a lot of memory which is re-used for reading other solutions. The value of soln_type may not be KHE_SOLN_INVALID_PLACEHOLDER. See Section 4.2.6 for

KheSolnTypeReduce and the other choices for soln_type.

KheArchiveRead calls KmlReadFile (Section A.5.3), passing echo_fp to it. The characters read are echoed to echo_fp if it is non-NULL; it would normally be NULL.

A similar function to KheArchiveRead is

```
bool KheArchiveLoad(KHE_ARCHIVE archive, FILE *fp,
  KML_ERROR *ke, bool audit_and_fix, bool resource_type_partitions,
  bool infer_resource_partitions, bool allow_invalid_solns,
  KHE_SOLN_TYPE soln_type, FILE *echo_fp);
```

The only difference is that the file's contents are added to existing archive archive rather than made into a new archive. The file must have the same model (high school or employee scheduling) as archive. Any archive Id or metadata in the file is ignored. Clashes between instance and solution group Id's in archive and in the file are not resolved; they are errors.

## 2.5. Reading archives incrementally

A large archive may have to be read one solution at a time. For this, call

```
bool KheArchiveReadIncremental(FILE *fp, HA_ARENA_SET as,
  KHE_ARCHIVE *archive, KML_ERROR *ke, bool audit_and_fix,
  bool resource_type_partitions, bool infer_resource_partitions,
  bool allow_invalid_solns, KHE_SOLN_TYPE soln_type, FILE *echo_fp,
  KHE_ARCHIVE_FN archive_begin_fn, KHE_ARCHIVE_FN archive_end_fn,
  KHE_SOLN_GROUP_FN soln_group_begin_fn,
  KHE_SOLN_GROUP_FN soln_group_end_fn, KHE_SOLN_FN soln_fn, void *impl);
```

The return value and the parameters up to echo_fp inclusive are as for KheArchiveRead. The remaining parameters are callback functions, except the last, impl, which is not used by KHE but is instead passed through to the calls on the callback functions. Any or all of the callback functions may be NULL, in which case the corresponding callbacks are not made.

Callback function archive_begin_fn is called by KheArchiveReadIncremental at the start of the archive. It must be written by the user like this:

```
void archive_begin_fn(KHE_ARCHIVE archive, void *impl)
{
  ...
}
```

Its archive parameter is set to the archive that KheArchiveReadIncremental will eventually build, the one it returns in its *archive parameter; its impl parameter contains the value of the impl parameter of KheArchiveReadIncremental. At the time of this call, archive contains its Id, metadata, and model attributes, but no instances and no solution groups.

Callback function archive_end_fn is called at the end of the archive, just before KheArchiveReadIncremental itself returns:

```
void archive_end_fn(KHE_ARCHIVE archive, void *impl)
{
  ...
}
```

When this function is called, `archive` contains all of its instances and solution groups. If `KheArchiveReadIncremental` returns `true`, there has been one callback to `archive_begin_fn` and one to `archive_end_fn`, if non-`NULL`.

Callback function `soln_group_begin_fn` is called at the start of each solution group:

```
void soln_group_begin_fn(KHE_SOLN_GROUP soln_group, void *impl)
{
  ...
}
```

Its `soln_group` parameter is set to one of the solution groups that the final archive will eventually contain, and its `impl` parameter is as before. At the time of this call, `soln_group` contains its Id and MetaData, and `KheSolnGroupArchive(soln_group)` returns the enclosing archive, but there are no solutions in `soln_group`.

Callback function `soln_group_end_fn` is called at the end of each solution group:

```
void soln_group_end_fn(KHE_SOLN_GROUP soln_group, void *impl)
{
  ...
}
```

At the time of this call, `soln_group` contains all its solutions.

Finally, callback function `soln_fn` is called after each solution is read:

```
void soln_fn(KHE_SOLN soln, void *impl)
{
  ...
}
```

The solution is complete, and `KheSolnSolnGroup(soln)` returns the enclosing solution group.

The purpose of incremental reading is to process the solutions as they are read, so that they can be deleted and their memory reclaimed. For example, to replace each solution by a placeholder, pass `NULL` for all callbacks except `soln_fn`, which would be defined like this:

```
void soln_fn(KHE_SOLN soln, void *impl)
{
  if( KheSolnType(soln) == KHE_SOLN_ORDINARY )
    KheSolnReduceToPlaceholder(soln, false);
}
```

The test is needed only if `allow_invalid_solns` is `true`. `KheSolnReduceToPlaceholder` (Section 4.2.6) reclaims most of the memory of `soln`, leaving just the `soln` object itself and a few attributes, including its cost. In this way, the total memory cost is reduced to not much more than the memory needed to hold the instances, but enough information is retained to support operations which (for example) print tables of solutions and their costs. Of course, `KheArchiveRead` has the `soln_type` parameter which can be used to instruct it to do these reductions anyway.

Other applications might process `soln` in some way (print timetables, for example) before finishing with a call to `KheSolnReduceToPlaceholder`, or even `KheSolnDelete`.

## 2.6. Reading archives from the command line

Reading an archive from the command line basically means opening the file named by a command-line argument and calling `KheArchiveRead`. Beyond that, there may be a need to process the archive before using it, for example to remove its solution groups. Function

```
KHE_ARCHIVE KheArchiveReadFromCommandLine(int argc, char *argv[],
  int *pos, HA_ARENA_SET as, bool audit_and_fix,
  bool resource_type_partitions, bool infer_resource_partitions,
  bool allow_invalid_solns, KHE_SOLN_TYPE soln_type, FILE *echo_fp);
```

offers a standard way to do that. Here `argc` and `argv` are exactly as they were passed to the main program, and `*pos` is an index into `argv`, to a point where the name of an archive is expected.

`KheArchiveReadFromCommandLine` first opens the file whose name is `argv[*pos]`, calls `KheArchiveRead`, and increments `*pos` to inform the caller that the argument at `*pos` has been processed. The name may be `-`, meaning standard input. Then, while command-line arguments beginning with `-x`, `-i`, `-n`, `-X`, and `-I` follow the name, it modifies the in-memory version of the archive as instructed by those arguments. Finally, it returns the archive, with `*pos` moved to the index of the first unprocessed argument, or to `argc` if the argument list becomes exhausted.

The `-x`, `-i`, `-n`, `-X`, and `-I` arguments have this syntax and meaning:

`-x<id>{,<id>}`
: Delete instances (and their solutions) with the given Ids.

`-i<id>{,<id>}`
: Include only instances (and their solutions) with the given Ids; delete all other instances.

`-n<int>`
: Include only the first `<int>` instances (and their solutions), or fewer if the archive contains fewer instances; delete all other instances. Useful for testing the solving of an archive without doing all of it.

`-X<id>{,<id>}`
: Delete solution groups with the given Ids.

`-I<id>{,<id>}`
: Include only solution groups with the given Ids; delete all other solution groups.

As a special case, `-X` with no ids means to delete all solution groups.

Arguments `-x`, `-i`, and `-n` may not be used together, and `-X` and `-I` may not be used together. If there is a problem, `KheArchiveReadFromCommandLine` prints a message and calls exit(1).

At present there is no KheArchiveReadFromCommandLineIncremental function combining KheArchiveReadFromCommandLine with KheArchiveReadIncremental.

## 2.7. Writing archives and solution groups

To write an archive to a file, call

```
void KheArchiveWrite(KHE_ARCHIVE archive, bool with_reports, FILE *fp);
```

File fp must be open for writing UTF-8 characters, and it remains open after the call returns. If with_reports is true, each written solution contains a Report section evaluating the solution.

If the archive's model is KHE_MODEL_HIGH_SCHOOL_TIMETABLE, the initial tag written to fp will be <HighSchoolTimetableArchive>. If the model is KHE_MODEL_EMPLOYEE_SCHEDULE, the initial tag will be <EmployeeScheduleArchive>.

Ids and names are optional in KHE but compulsory when writing XML: if any are missing, KheArchiveWrite writes an incomplete file and aborts with an error message. They will all be present when archive was produced by KheArchiveRead.

If any of archive's solutions are invalid or unwritable placeholders, KheArchiveWrite aborts. If with_reports is true, any placeholder solution at all causes an abort.

When an event has a preassigned time, there is a problem if one of its meets is not assigned that time. If the meet is assigned some other time (which is possible in KHE, although not easy), then writing that time will cause the solution to be declared invalid when it is re-read. If the meet is not assigned any time, then, whether or not the preassigned time is written, the meaning is that the preassigned time is assigned, which is not the true state of the solution. The same problem arises with preassigned event resources whose tasks are not assigned the preassigned resource.

Accordingly, KheArchiveWrite also writes an incomplete file and aborts with an error message when it encounters a meet (or task) derived from a preassigned event (or event resource) whose assigned time (or resource) is unequal to the preassigned time (or resource).

When writing solutions, KheArchiveWrite writes as little as possible. It does not write an unassigned or preassigned task. It does not write a meet if its duration equals the duration of the corresponding event, its time is unassigned or preassigned, and its tasks are not written according to the rule just given (see also Section 4.3).

Two similar functions are

```
void KheArchiveWriteSolnGroup(KHE_ARCHIVE archive,
  KHE_SOLN_GROUP soln_group, bool with_reports, FILE *fp);
void KheArchiveWriteWithoutSolnGroups(KHE_ARCHIVE archive, FILE *fp);
```

They also write archive, omitting all its solution groups, or all of them except soln_group. They have been superseded, in practice, by KheArchiveReadFromCommandLine (Section 2.6).

# Chapter 3. Instances

An *instance* is a particular case of the high school timetabling problem, for a particular term or semester of a particular school. This chapter describes the `KHE_INSTANCE` data type, which represents instances as defined in the XML format.

## 3.1. Creating instances

To make a new, empty instance, call

```
KHE_INSTANCE KheInstanceMakeBegin(char *id, KHE_MODEL model,
  HA_ARENA_SET as);
```

Parameter `id` is the Id attribute from the XML file; it is optional, with `NULL` meaning absent. Parameter `model` is the model, as for `KheArchiveMake`, and `as` is the thread arena set, also as for `KheArchiveMake`.

To delete an instance, call

```
void KheInstanceDelete(KHE_INSTANCE ins);
```

Its memory will be returned to the arena set that was used to create it.

Functions

```
char *KheInstanceId(KHE_INSTANCE ins);
KHE_MODEL KheInstanceModel(KHE_INSTANCE ins);
```

retrieve these attributes.

For the convenience of functions that reorganize archives, an instance may lie in any number of archives. To add an instance to an archive and delete it from an archive, call functions `KheArchiveAddInstance` and `KheArchiveDeleteInstance` from Section 2.1. To visit the archives containing a given instance, call

```
int KheInstanceArchiveCount(KHE_INSTANCE ins);
KHE_ARCHIVE KheInstanceArchive(KHE_INSTANCE ins, int i);
```

in the usual way.

To set and retrieve the back pointer of `ins`, call

```
void KheInstanceSetBack(KHE_INSTANCE ins, void *back);
void *KheInstanceBack(KHE_INSTANCE ins);
```

as usual.

After the instance has been completed, using functions still to be defined, call

```
bool KheInstanceMakeEnd(KHE_INSTANCE ins, bool audit_and_fix,
  bool resource_type_partitions, bool infer_resource_partitions,
  char **error_message);
```

This must be done, single-threaded, before any solution is created. It checks the instance and initializes various constant data structures used to speed the solution process. Parameter `audit_and_fix` is described just below, `resource_type_partitions` is the subject of Section 3.5.5, and `infer_resource_partitions` is the subject of Section 3.5.6. `KheInstanceMakeEnd` sets `*error_message` to `NULL` and returns `true` when it finds no problems; when there is something wrong it sets `*error_message` to an error message describing the first problem and returns `false`. In principle the problem could be nearly anything, although at present the only problems detected by `KheInstanceMakeEnd` are cases where the time groups used by limit idle times constraints (Section 3.7.13) are not compact.

Even when an instance is formally valid, it may have features that suggest that something is wrong, such as resources without avoid clashes constraints. When `audit_and_fix` is `true`, KHE audits the instance and fixes any problems it finds. At present, it checks for pairs of events joined by a link events constraint whose event constraints differ, and adds events as points of application of those constraints to remove the differences. Other checks may be added in future.

Instance metadata may be set and retrieved by calling

```
void KheInstanceSetMetaData(KHE_INSTANCE ins, char *name, char *contributor,
  char *date, char *country, char *description, char *remarks);
void KheInstanceMetaData(KHE_INSTANCE ins, char **name, char **contributor,
  char **date, char **country, char **description, char **remarks);
```

Copies of the strings passed in are stored, not the originals. As for archive and solution group metadata, KHE allows any instance metadata objects or strings to be `NULL` or empty, and when writing an instance it substitutes values `"No name"`, `"No contributor"`, etc., for such values, or omits them altogether when XHSTT allows. Also,

```
char *KheInstanceMetaDataText(KHE_INSTANCE ins);
```

returns a string containing the metadata as a paragraph of English text. The string lies in the instance's arena and will be deleted when the instance is deleted.

## 3.2. Visiting and retrieving the components of instances

An instance may contain any number of time groups, times, resource types, event groups, events, and constraints. These are added by the functions that create them, to be given later.

To visit all the time groups of an instance, or retrieve a time group by `id`, call

```
int KheInstanceTimeGroupCount(KHE_INSTANCE ins);
KHE_TIME_GROUP KheInstanceTimeGroup(KHE_INSTANCE ins, int i);
bool KheInstanceRetrieveTimeGroup(KHE_INSTANCE ins, char *id,
  KHE_TIME_GROUP *tg);
```

The first returns the number of time groups in `ins`. The second returns the `i`'th time group, counting from 0 as usual in C. The third searches for a time group of `ins` with the given `id`; if found, it sets `*tg` to it and returns `true`, otherwise it leaves `*tg` unchanged and returns `false`.

Only time groups created by user calls to `KheTimeGroupMake` (Section 3.4.1) are found by `KheInstanceTimeGroupCount`, `KheInstanceTimeGroup`, and `KheInstanceRetrieveTimeGroup`. Some other time groups are created automatically by KHE, but they are accessed in other ways. They include one time group for each time, holding just that time; a time group holding the full set of times of the instance; and an empty time group. These last two are returned by

```
KHE_TIME_GROUP KheInstanceFullTimeGroup(KHE_INSTANCE ins);
KHE_TIME_GROUP KheInstanceEmptyTimeGroup(KHE_INSTANCE ins);
```

Time groups may also be created during solving (Section 4.4). Those too are not accessible via `KheInstanceTimeGroupCount`, `KheInstanceTimeGroup`, or `KheInstanceRetrieveTimeGroup`.

To visit all the times of an instance, or retrieve a time by Id, call

```
int KheInstanceTimeCount(KHE_INSTANCE ins);
KHE_TIME KheInstanceTime(KHE_INSTANCE ins, int i);
bool KheInstanceRetrieveTime(KHE_INSTANCE ins, char *id, KHE_TIME *t);
```

These work in the same way as the functions above for visiting and retrieving time groups.

To visit all the resource types of an instance, or retrieve a resource type by `id`, call

```
int KheInstanceResourceTypeCount(KHE_INSTANCE ins);
KHE_RESOURCE_TYPE KheInstanceResourceType(KHE_INSTANCE ins, int i);
bool KheInstanceRetrieveResourceType(KHE_INSTANCE ins, char *id,
  KHE_RESOURCE_TYPE *rt);
```

These work in the same way as the corresponding functions for visiting and retrieving time groups and times. Resource types have operations which give access to their resource groups and resources. For convenience there are also operations

```
bool KheInstanceRetrieveResourceGroup(KHE_INSTANCE ins, char *id,
  KHE_RESOURCE_GROUP *rg);
bool KheInstanceRetrieveResource(KHE_INSTANCE ins, char *id,
  KHE_RESOURCE *r);
```

which search all the resource types of `ins` for a resource group or resource with the given `id`. It is also possible to bypass resource types and visit all resources directly, by calling

```
int KheInstanceResourceCount(KHE_INSTANCE ins);
KHE_RESOURCE KheInstanceResource(KHE_INSTANCE ins, int i);
```

in the usual way. The resources will be visited in the order they were created.

To visit all the event groups of an instance, or to retrieve an event group by `id`, call

```
int KheInstanceEventGroupCount(KHE_INSTANCE ins);
KHE_EVENT_GROUP KheInstanceEventGroup(KHE_INSTANCE ins, int i);
bool KheInstanceRetrieveEventGroup(KHE_INSTANCE ins, char *id,
  KHE_EVENT_GROUP *eg);
```

These work in the usual way.

Some event groups are created automatically by KHE, including one event group for each event, holding just that event; an event group holding the full set of events of the instance; and an empty event group. These last two are returned by

```
KHE_EVENT_GROUP KheInstanceFullEventGroup(KHE_INSTANCE ins);
KHE_EVENT_GROUP KheInstanceEmptyEventGroup(KHE_INSTANCE ins);
```

Automatically defined event groups are not visited by `KheInstanceEventGroupCount` and `KheInstanceEventGroup`. Even more event groups may be created during solving. Those also do not appear in the list of event groups of the original instance.

To visit the events of an instance, or to retrieve an event by `id`, call

```
int KheInstanceEventCount(KHE_INSTANCE ins);
KHE_EVENT KheInstanceEvent(KHE_INSTANCE ins, int i);
bool KheInstanceRetrieveEvent(KHE_INSTANCE ins, char *id, KHE_EVENT *e);
```

Two reasons for visiting all events have already been taken care of, by functions

```
bool KheInstanceAllEventsHavePreassignedTimes(KHE_INSTANCE ins);
int KheInstanceMaximumEventDuration(KHE_INSTANCE ins);
```

`KheInstanceAllEventsHavePreassignedTimes` returns `true` if all events have preassigned times. `KheInstanceMaximumEventDuration` returns the maximum event duration, or `0` when there are no events. In the usual representation of nurse rostering, their values are `true` and `1`.

To visit the event resources of an instance, call

```
int KheInstanceEventResourceCount(KHE_INSTANCE ins);
KHE_EVENT_RESOURCE KheInstanceEventResource(KHE_INSTANCE ins, int i);
```

The event resources may also be visited via their events.

To visit all the constraints of an instance, or to retrieve a constraint by `id`, call

```
int KheInstanceConstraintCount(KHE_INSTANCE ins);
KHE_CONSTRAINT KheInstanceConstraint(KHE_INSTANCE ins, int i);
bool KheInstanceRetrieveConstraint(KHE_INSTANCE ins, char *id,
  KHE_CONSTRAINT *c);
```

These work in the usual way. There is also

```
int KheInstanceConstraintOfTypeCount(KHE_INSTANCE ins,
  KHE_CONSTRAINT_TAG constraint_tag);
```

which returns the number of constraints with the given `constraint_tag`. At present there is no way to visit these constraints, other than to visit all constraints and select the ones with that tag.

## 3.3. Constraint density

Within a given instance, the *density* of a given kind of constraint is the number of applications of constraints of that kind, divided by the number of places where constraints of that kind could apply. The density is a floating-point number, usually between 0 and 1, although it can exceed 1, since constraints of the same kind may apply at one place. KHE offers functions

```
int KheInstanceConstraintDensityCount(KHE_INSTANCE ins,
  KHE_CONSTRAINT_TAG constraint_tag);
int KheInstanceConstraintDensityTotal(KHE_INSTANCE ins,
  KHE_CONSTRAINT_TAG constraint_tag);
```

returning the number of applications of constraints of kind `constraint_tag` in `ins` (the *density count*), and the number of places where constraints of that kind could apply in `ins` (the *density total*). The density is the quotient of these two quantities, unless the density total is 0, in which case the density is undefined, although it may be reported as 0.0 in that case. Precise definitions of the density count and density total are given for each kind of constraint in Section 3.7.

The first time either of these functions is called for any value of `constraint_tag`, the results of both functions are calculated for all values of `constraint_tag` and stored in `ins`. So multi-threaded calls on these functions are only safe if one single-threaded call is made first.

## 3.4. Times

### 3.4.1. Time groups

A time group, representing a set of times, is created and added to an instance by calling

```
bool KheTimeGroupMake(KHE_INSTANCE ins, KHE_TIME_GROUP_KIND kind,
  char *id, char *name, KHE_TIME_GROUP *tg);
```

This works like all creations of named objects do in KHE: if `id` is non-`NULL` and `ins` already contains a time group with this `id`, it returns `false` and creates nothing; otherwise it creates a new time group, sets `*tg` to point to it, and returns `true`.

Parameter `kind` has type

```
typedef enum {
  KHE_TIME_GROUP_KIND_ORDINARY,
  KHE_TIME_GROUP_KIND_WEEK,
  KHE_TIME_GROUP_KIND_DAY,
  KHE_TIME_GROUP_KIND_SOLN,
  KHE_TIME_GROUP_KIND_AUTO
} KHE_TIME_GROUP_KIND;
```

`KHE_TIME_GROUP_KIND_ORDINARY` is the usual kind. The XML format allows some time groups to be referred to as Weeks and Days, although they do not differ from other time groups in any other way. Values `KHE_TIME_GROUP_KIND_WEEK` and `KHE_TIME_GROUP_KIND_DAY` record this usage; they matter only when reading and writing XML files, not when solving. The last two values cannot be passed to KheTimeGroupMake, although they may be returned by function KheTimeGroupKind below. KHE_TIME_GROUP_KIND_SOLN is the kind of time groups returned by KheSolnTimeGroupEnd (Section 4.4), and KHE_TIME_GROUP_KIND_AUTO is the kind of time groups created automatically by KHE.

The id and name parameters may be NULL; they are used only when writing XML, when they represent the compulsory Id and Name attributes of the time group. Irrespective of the order time groups are created in, to conform with the XML rules, when writing time groups KHE writes days first, then weeks, then ordinary time groups; it does not write any other time groups.

To set and retrieve the back pointer of tg, call

```
void KheTimeGroupSetBack(KHE_TIME_GROUP tg, void *back);
void *KheTimeGroupBack(KHE_TIME_GROUP tg);
```

in the usual way. The other attributes may be retrieved by calling

```
KHE_INSTANCE KheTimeGroupInstance(KHE_TIME_GROUP tg);
KHE_TIME_GROUP_KIND KheTimeGroupKind(KHE_TIME_GROUP tg);
char *KheTimeGroupId(KHE_TIME_GROUP tg);
char *KheTimeGroupName(KHE_TIME_GROUP tg);
```

Initially the time group is empty. There are several operations for changing its set of times:

```
void KheTimeGroupAddTime(KHE_TIME_GROUP tg, KHE_TIME t);
void KheTimeGroupSubTime(KHE_TIME_GROUP tg, KHE_TIME t);
void KheTimeGroupUnion(KHE_TIME_GROUP tg, KHE_TIME_GROUP tg2);
void KheTimeGroupIntersect(KHE_TIME_GROUP tg, KHE_TIME_GROUP tg2);
void KheTimeGroupDifference(KHE_TIME_GROUP tg, KHE_TIME_GROUP tg2);
```

These add a time to tg, remove a time, replace tg's set of times with its union or intersecton with the set of times of tg2, and with the difference of tg's times and tg2's times. The first two operations are treated as set operations, so KheTimeGroupAddTime does nothing if t is already present, and KheTimeGroupSubTime does nothing if t is not already present.

Changes to the time groups of an instance are not allowed after KheInstanceMakeEnd is called, since instances are immutable after that point. However, solutions may construct time groups for their own use (Section 4.4).

In addition to time groups created by the user, many time groups are created automatically by KHE, with such useful values as the full set of times of the cycle and the empty set of times (Section 3.2), all singleton sets of times (Section 3.4), and various other values, associated with constraints. All these time groups are created during KheInstanceMakeEnd, and in every case, KHE first checks whether there is a user-defined time group with the desired value, and if so, it uses that time group instead of creating a new one. When it does create a new time group, that time group has KHE_TIME_GROUP_KIND_AUTO for kind and NULL for Id and name, except that time groups returned by KheTimeGroupNeighbour may have an Id and name, as explained below.

The times of any time group are visited by

```
int KheTimeGroupTimeCount(KHE_TIME_GROUP tg);
KHE_TIME KheTimeGroupTime(KHE_TIME_GROUP tg, int i);
```

These work in the same way as the visit functions for instances above. And

```
bool KheTimeGroupContains(KHE_TIME_GROUP tg, KHE_TIME t, int *pos);
bool KheTimeGroupEqual(KHE_TIME_GROUP tg1, KHE_TIME_GROUP tg2);
bool KheTimeGroupSubset(KHE_TIME_GROUP tg1, KHE_TIME_GROUP tg2);
bool KheTimeGroupDisjoint(KHE_TIME_GROUP tg1, KHE_TIME_GROUP tg2);
```

return `true` if `tg` contains `t` (setting `*pos` to its position in the time group), if `tg1` and `tg2` contain the same times, if the times of `tg1` are a subset of the times of `tg2`, and if the times of `tg1` and `tg2` are disjoint. There is nothing to prevent two distinct time groups from containing the same times.

There are also

```
int KheTimeGroupTypedCmp(KHE_TIME_GROUP tg1, KHE_TIME_GROUP tg2);
int KheTimeGroupCmp(const void *t1, const void *t2);
```

which are typed and untyped versions of a comparison function that may be used to sort an array of time groups into a canonical order. The precise order is not specified other than that a return value of 0 indicates that the two time groups are equal.

Here are some miscellaneous time group functions. Function

```
bool KheTimeGroupIsCompact(KHE_TIME_GROUP tg);
```

returns `true` when `tg` is *compact*: when it is empty or there are no gaps in its times, taken in chronological order. Function

```
int KheTimeGroupOverlap(KHE_TIME_GROUP tg, KHE_TIME time, int durn);
```

returns the number of times that a meet starting at `time` with duration `durn` overlaps with `tg`.

A key function for KHE's handling of time is

```
KHE_TIME_GROUP KheTimeGroupNeighbour(KHE_TIME_GROUP tg, int delta);
```

It returns a time group containing `tg`'s times shifted `delta` places, where `delta` may be any integer. `KheTimeGroupNeighbour(tg, 0)`, for example, is a time group with the same times as `tg`, possibly but not necessarily `tg` itself; and `KheTimeGroupNeighbour(tg, -1)` holds the times that immediately precede `tg`'s. The time group will be empty if `delta` is such a large (positive or negative) number that all the times are shifted off the cycle.

Time group neighbours are created automatically by KHE. As explained above, KHE will use existing user-defined time groups wherever possible, to avoid creating new ones. When it does create a new one, it assigns it an Id and name. This is useful because, although time group neighbours are never printed in XML files, names for them are needed when reporting the calculation made by a monitor for a constraint with a non-`NULL` `AppliesToTimeGroup`. For example, given time group `tg` with Id `"Mon"` and name `"Monday"`, if

```
KheTimeGroupNeighbour(tg, 5)
```

has to be created it is assigned Id `"Mon+5"` and name `"Monday+5"`. It is best to avoid giving user-defined time groups names like these ones, although there can be no name clashes, strictly speaking, because time group neighbours are not stored in any table indexed by Id or name. `KheInstanceRetrieveTimeGroup`, for example, only retrieves user-defined time groups.

`KheTimeGroupNeighbour` accepts time groups returned by `KheTimeGroupNeighbour`, but the result can be odd. Suppose `tg2 = KheTimeGroupNeighbour(tg, 5)` is called, and `tg` has 7 times but `tg2` has only 4, because 3 of `tg`'s times shifted off the end. A subsequent call to `KheTimeGroupNeighbour(tg2, -5)` may return another time group with 4 times, but it is more likely to return a time group equal to `tg`. This is for efficiency: if, every time a time went off either end, a whole new neighbourhood was constructed, then neighbourhood construction would go on forever. There are no such peculiarities when times do not shift off either end.

To speed up loading nurse rostering instances with long cycles, the time group returned by `KheAvoidUnavailableTimesConstraintUnavailableTimes` usually has no neighbourhood. The same goes for `KheAvoidUnavailableTimesConstraintAvailableTimes`, and also for `KheLimitBusyTimesConstraintDomain` and `KheLimitWorkloadConstraintDomain`. A call to `KheTimeGroupNeighbour` will abort with an error message if it is given one of these time groups. The user should not worry about this until it happens; it probably never will.

As an aid to debugging, function

```
void KheTimeGroupDebug(KHE_TIME_GROUP tg, int verbosity,
  int indent, FILE *fp);
```

prints `tg` onto `fp` with the given verbosity and indent, as usual (Section 1.4). Verbosity 1 prints either the Id of the time group, or the first and last time (at most) enclosed in braces.

### 3.4.2. Times

A time is created and added to an instance by calling

```
bool KheTimeMake(KHE_INSTANCE ins, char *id, char *name,
  bool break_after, KHE_TIME *t);
```

As usual, a `false` return value is only possible when `id` is non-`NULL` and already in use by another time object. Parameters `id` and `name` may be `NULL`, and are used only when writing XML.

Parameter `break_after` says that a break occurs after this time, so that, for example, an event of duration 2 could not begin here. This is not an XML feature; when representing XML this parameter should always be `false`. Within KHE itself it is used only by function `KheSolnSplitCycleMeet` and its associated operations (Section 4.5.3).

To set and retrieve the back pointer of a time, call functions

```
void KheTimeSetBack(KHE_TIME t, void *back);
void *KheTimeBack(KHE_TIME t);
```

as usual. The other attributes are retrieved by

```
KHE_INSTANCE KheTimeInstance(KHE_TIME t);
char *KheTimeId(KHE_TIME t);
char *KheTimeName(KHE_TIME t);
bool KheTimeBreakAfter(KHE_TIME t);
int KheTimeIndex(KHE_TIME t);
```

`KheTimeIndex` returns an automatically generated index number for `time`: 0 for the first time created, 1 for the second, and so on. The times of an instance form a sequence, not a set, and must be created in chronological order. This is unlike resources, events, etc., whose order of creation does not matter. The XML format requires times to appear in this same order. Function

```
bool KheTimeHasNeighbour(KHE_TIME t, int delta);
```

returns `true` when there is a time whose index is the index of `t` plus `delta`, where `delta` may be any integer, negative, zero, or positive. Function

```
KHE_TIME KheTimeNeighbour(KHE_TIME t, int delta);
```

returns this time when it exists, and aborts when it does not.

For sorting an array of times into chronological order there is

```
int KheTimeTypedCmp(KHE_TIME t1, KHE_TIME t2);
int KheTimeCmp(const void *t1, const void *t2);
```

`KheTimeCmp` is suitable for passing to `qsort`, or to `HaArraySort`.

When calculating with the chronological ordering of time—deciding whether two meets are adjacent, and so on—it is often best to call `KheTimeIndex` to obtain the indexes of the times involved and work with them. However, these functions may help to avoid time indexes:

```
bool KheTimeLE(KHE_TIME time1, int delta1, KHE_TIME time2, int delta2);
bool KheTimeLT(KHE_TIME time1, int delta1, KHE_TIME time2, int delta2);
bool KheTimeGT(KHE_TIME time1, int delta1, KHE_TIME time2, int delta2);
bool KheTimeGE(KHE_TIME time1, int delta1, KHE_TIME time2, int delta2);
bool KheTimeEQ(KHE_TIME time1, int delta1, KHE_TIME time2, int delta2);
bool KheTimeNE(KHE_TIME time1, int delta1, KHE_TIME time2, int delta2);
```

They return `true` when `KheTimeNeighbour(time1, delta1)`'s time index is less than or equal to `KheTimeNeighbour(time2, delta2)`'s, and so on. The neighbours need not exist; the functions simply convert times into indexes and perform the indicated integer operations. Also,

```
int KheTimeIntervalsOverlap(KHE_TIME time1, int durn1,
  KHE_TIME time2, int durn2);
```

takes two time intervals, one beginning at `time1` with duration `durn1`, the other beginning at `time2` with duration `durn2`, and returns the number of times lying in both intervals. For example, the result will be 0 when either interval ends before the other begins. Similarly,

```
bool KheTimeIntervalsOverlapInterval(KHE_TIME time1, int durn1,
  KHE_TIME time2, int durn2, KHE_TIME *overlap_time, int *overlap_durn);
```

returns `true` when `KheTimeIntervalsOverlap` is non-zero, and sets `*overlap_time` and `*overlap_durn` to the starting time and duration of the overlap; otherwise it returns `false`.

For convenience, a time group is available for each time, holding just that time. Function

```
KHE_TIME_GROUP KheTimeSingletonTimeGroup(KHE_TIME t);
```

returns this time group. It cannot be changed.

The events preassigned a particular time can be visited by

```
int KheTimePreassignedEventCount(KHE_TIME t);
KHE_EVENT KheTimePreassignedEvent(KHE_TIME t, int i);
```

`KheTimePreassignedEventCount(t)` returns the number of events preassigned time `t`, and `KheTimePreassignedEvent(t, i)` returns the `i`th of these events, counting from 0 as usual.

## 3.5. Resources

### 3.5.1. Resource types

A resource type, representing one broad category of resources, such as the teachers or rooms, is created and added to an instance in the usual way by the call

```
bool KheResourceTypeMake(KHE_INSTANCE ins, char *id, char *name,
  bool has_partitions, KHE_RESOURCE_TYPE *rt);
```

Attributes `id` and `name` represent the optional XML Id and Name attributes as usual. Its back pointer may be set and retrieved by

```
void KheResourceTypeSetBack(KHE_RESOURCE_TYPE rt, void *back);
void *KheResourceTypeBack(KHE_RESOURCE_TYPE rt);
```

as usual, and its other attributes may be retrieved by

```
KHE_INSTANCE KheResourceTypeInstance(KHE_RESOURCE_TYPE rt);
int KheResourceTypeIndex(KHE_RESOURCE_TYPE rt);
char *KheResourceTypeId(KHE_RESOURCE_TYPE rt);
char *KheResourceTypeName(KHE_RESOURCE_TYPE rt);
bool KheResourceTypeHasPartitions(KHE_RESOURCE_TYPE rt);
```

`KheResourceTypeIndex(rt)` returns the index of `rt` in the enclosing instance, that is, the value of `i` for which `KheInstanceResourceType` returns `rt`.

Attribute `has_partitions` is not an XML feature, and should be given value `false` when reading an XML instance. It indicates that there is a unique partitioning of the resources of this resource type, defined by a collection of specially marked resource groups called *partitions*. For example, the resources of a student groups resource type might be partitioned into forms, or the resources of a teachers resource type might be partitioned into faculties. When a resource type has partitions, each of its resources must lie in exactly one partition.

Each resource type contains an arbitrary number of resource groups, representing sets of resources of its type. Resource groups are added to a resource type automatically by the functions that create them. To visit all the resource groups of a given resource type, or to retrieve a resource group with a given `id` from a given resource type, call

```
int KheResourceTypeResourceGroupCount(KHE_RESOURCE_TYPE rt);
KHE_RESOURCE_GROUP KheResourceTypeResourceGroup(KHE_RESOURCE_TYPE rt,
  int i);
bool KheResourceTypeRetrieveResourceGroup(KHE_RESOURCE_TYPE rt,
  char *id, KHE_RESOURCE_GROUP *rg);
```

These work in the usual way. The partitions of a resource type may be visited by

```
int KheResourceTypePartitionCount(KHE_RESOURCE_TYPE rt);
KHE_RESOURCE_GROUP KheResourceTypePartition(KHE_RESOURCE_TYPE rt, int i);
```

`KheResourceTypePartitionCount` returns 0 when `rt` does not have partitions.

Some resource groups are made automatically by KHE, including one resource group for each resource, holding just that resource; a resource group holding the full set of resources of the resource type; and an empty resource group. These last two are returned by

```
KHE_RESOURCE_GROUP KheResourceTypeFullResourceGroup(KHE_RESOURCE_TYPE rt);
KHE_RESOURCE_GROUP KheResourceTypeEmptyResourceGroup(KHE_RESOURCE_TYPE rt);
```

Automatically made resource groups are not visited by `KheResourceTypeResourceGroupCount` and `KheResourceTypeResourceGroup`. Even more resource groups may be created during solving, but those do not appear in the list of resource groups of the original instance.

To visit all the resources of a given resource type, or to retrieve a resource of a given resource type by `id`, call

```
int KheResourceTypeResourceCount(KHE_RESOURCE_TYPE rt);
KHE_RESOURCE KheResourceTypeResource(KHE_RESOURCE_TYPE rt, int i);
bool KheResourceTypeRetrieveResource(KHE_RESOURCE_TYPE rt,
  char *id, KHE_RESOURCE *r);
```

in the usual way.

Four functions, which should be called only after the instance is complete, summarize information relevant to assigning resources of a given resource type. The values of these functions are calculated as the instance is made, so one call on any of them costs practically nothing. The first is

```
bool KheResourceTypeDemandIsAllPreassigned(KHE_RESOURCE_TYPE rt);
```

It returns `true` if every event resource of type `rt` is preassigned. In practice this is always true for student group resource types, and often for teachers, but rarely for rooms. The second is

```
int KheResourceTypeAvoidSplitAssignmentsCount(KHE_RESOURCE_TYPE rt);
```

It returns the number of points of application of avoid split assignments constraints that constrain event resources of this type. The larger this number is, the more difficult the resource assignment problem for resources of this type is likely to be. The third is

```
int KheResourceTypeLimitResourcesCount(KHE_RESOURCE_TYPE rt);
```

returns the number of points of application of limit resources constraints that have this resource type. See Section 12.7.3 for an application of this function. Finally,

```
float KheResourceTypeMaxWorkloadPerTime(KHE_RESOURCE_TYPE rt);
```

returns the maximum value of `KheEventResourceWorkloadPerTime(er)` over all event resources `er` of type `rt`. Limit workload monitors use this value (Section 6.7.6).

### 3.5.2. Resource groups

A resource group is created and added to a resource type by the call

```
bool KheResourceGroupMake(KHE_RESOURCE_TYPE rt, char *id, char *name,
  bool is_partition, KHE_RESOURCE_GROUP *rg)
```

This function returns `false` only when `id` is non-`NULL` and some other resource group of type `rt` has this `id`. The resource group lies in resource type `rt` with the usual `id` and `name` attributes. Attribute `is_partition` is not an XML feature, and should be given value `false` when reading an XML instance. It may be `true` only if attribute `has_partitions` of the resource group's resource type is `true`, in which case it indicates that this resource group is a partition, that is, one of those resource groups which define the unique partitioning of the resources of that type.

To set and retrieve the back pointer of a resource group, call

```
void KheResourceGroupSetBack(KHE_RESOURCE_GROUP rg, void *back);
void *KheResourceGroupBack(KHE_RESOURCE_GROUP rg);
```

as usual. The other attributes may be retrieved by calling

```
KHE_RESOURCE_TYPE KheResourceGroupResourceType(KHE_RESOURCE_GROUP rg);
KHE_INSTANCE KheResourceGroupInstance(KHE_RESOURCE_GROUP rg);
char *KheResourceGroupId(KHE_RESOURCE_GROUP rg);
char *KheResourceGroupName(KHE_RESOURCE_GROUP rg);
bool KheResourceGroupIsPartition(KHE_RESOURCE_GROUP rg);
```

`KheResourceGroupInstance` returns the resource group's resource type's instance.

Initially the resource group is empty. Several operations change its resources:

```
void KheResourceGroupAddResource(KHE_RESOURCE_GROUP rg, KHE_RESOURCE r);
void KheResourceGroupSubResource(KHE_RESOURCE_GROUP rg, KHE_RESOURCE r);
void KheResourceGroupUnion(KHE_RESOURCE_GROUP rg, KHE_RESOURCE_GROUP rg2);
void KheResourceGroupIntersect(KHE_RESOURCE_GROUP rg, KHE_RESOURCE_GROUP rg2);
void KheResourceGroupDifference(KHE_RESOURCE_GROUP rg, KHE_RESOURCE_GROUP rg2);
```

These add `r` to `rg`, remove `r`, replace `rg`'s set of resources with its union or intersecton with the set of resources of `rg2`, and with the difference of `rg`'s resources and `rg2`'s resources. All the resources and resource groups involved must be of the same type. The first two operations are treated as set operations, so `KheResourceGroupAddResource` does nothing if `r` is already present, and `KheResourceGroupSubResource` does nothing if `r` is not already present.

These functions may not be used to alter resource groups which define partitions. When a resource type has partitions, each of its resources is added to its partition when it is created.

Changes to the resource groups of an instance are not allowed after `KheInstanceMakeEnd` is called, since instances are immutable after that point. However, solutions may construct resource groups for their own use (Section 4.4).

There are also several operations for finding the cardinality of unions, intersections, etc., without changing anything:

```
int KheResourceGroupUnionCount(KHE_RESOURCE_GROUP rg,
  KHE_RESOURCE_GROUP rg2);
int KheResourceGroupIntersectCount(KHE_RESOURCE_GROUP rg,
  KHE_RESOURCE_GROUP rg2);
int KheResourceGroupDifferenceCount(KHE_RESOURCE_GROUP rg,
  KHE_RESOURCE_GROUP rg2);
int KheResourceGroupSymmetricDifferenceCount(KHE_RESOURCE_GROUP rg,
  KHE_RESOURCE_GROUP rg2);
```

Building symmetric differences is awkward, so at present there is no operation for it, only this operation for finding its cardinality.

There are also predefined resource groups, for the complete set of resources of each resource type and the empty set of resources of each type (see Section 3.5.1 for those), and one for each resource of the instance, containing just that resource (Section 3.5). The resources in predefined resource groups may not be changed.

The resources of any resource group are visited by

```
int KheResourceGroupResourceCount(KHE_RESOURCE_GROUP rg);
KHE_RESOURCE KheResourceGroupResource(KHE_RESOURCE_GROUP rg, int i);
```

These work in the usual way. And

```
bool KheResourceGroupContains(KHE_RESOURCE_GROUP rg, KHE_RESOURCE r);
bool KheResourceGroupEqual(KHE_RESOURCE_GROUP rg1,
  KHE_RESOURCE_GROUP rg2);
bool KheResourceGroupSubset(KHE_RESOURCE_GROUP rg1,
  KHE_RESOURCE_GROUP rg2);
bool KheResourceGroupDisjoint(KHE_RESOURCE_GROUP rg1,
  KHE_RESOURCE_GROUP rg2);
```

return `true` if `rg` contains `r`, if `rg1` and `rg2` contain the same resources, if the resources of `rg1` form a subset of the resources of `rg2`, and if the resources of `rg1` and `rg2` are disjoint. Two distinct resource groups may contain the same resources, so it is best not to apply the C equality operator to resource groups.

There are also

```
int KheResourceGroupTypedCmp(KHE_RESOURCE_GROUP rg1,
  KHE_RESOURCE_GROUP rg2);
int KheResourceGroupCmp(const void *t1, const void *t2);
```

which are typed and untyped versions of a comparison function that may be used to sort an array of resource groups into a canonical order. The precise order is not specified other than that a return value of 0 indicates that the two resource groups are equal.

After a resource group is finalized, function

```
KHE_RESOURCE_GROUP KheResourceGroupPartition(KHE_RESOURCE_GROUP rg);
```

may be called. If `rg` is non-empty and its resources share a partition, the result is that partition, otherwise the result is `NULL`. Since `KheResourceGroupPartition` is called when monitoring evenness, for efficiency the result is precomputed and stored in `rg` when it is finalized.

As an aid to debugging, function

```
void KheResourceGroupDebug(KHE_RESOURCE_GROUP rg, int verbosity,
  int indent, FILE *fp);
```

prints `rg` onto `fp` with the given verbosity and indent, as described for debug functions in general in Section 1.4. Verbosity 1 prints the Id of the resource group in some cases, and the first and last resource (at most) enclosed in braces in others.

### 3.5.3. Resources

A resource is created and added to its resource type by the call

```
bool KheResourceMake(KHE_RESOURCE_TYPE rt, char *id, char *name,
  KHE_RESOURCE_GROUP partition, KHE_RESOURCE *r);
```

A resource type is compulsory; `id` and `name` are the usual optional XML Id and Name.

Unlike `KheResourceGroupMake`, which returns `false` when its `id` parameter is non-`NULL` and some other resource group of the same resource type already has that Id, `KheResourceMake` returns `false` and sets `*r` to `NULL` when its `id` parameter is non-`NULL` and some other resource *of any resource type* already has its Id. This is because predefined event resources are permitted to identify a resource by its Id alone, and so resource Ids must be unique among all the resources of the instance, not merely among resources of a given type.

The `partition` attribute is not an XML feature, and should be given value `NULL` when reading an XML instance. It must be non-`NULL` if and only if `rt`'s `has_partitions` attribute is `true`, in which case its value must be a resource group of type `rt` whose `is_partition` attribute is `true`, and it indicates that the new resource lies in the specified partition. The new resource will be added to the partition by this function, and no separate call to `ResourceGroupAddResource` to do this is necessary or even permitted.

To set and retrieve the back pointer of a resource, call

```
void KheResourceSetBack(KHE_RESOURCE r, void *back);
void *KheResourceBack(KHE_RESOURCE r);
```

as usual. The other attributes may be retrieved by the calls

```
KHE_INSTANCE KheResourceInstance(KHE_RESOURCE r);
int KheResourceInstanceIndex(KHE_RESOURCE r);
KHE_RESOURCE_TYPE KheResourceResourceType(KHE_RESOURCE r);
int KheResourceResourceTypeIndex(KHE_RESOURCE r);
char *KheResourceId(KHE_RESOURCE r);
char *KheResourceName(KHE_RESOURCE r);
KHE_RESOURCE_GROUP KheResourcePartition(KHE_RESOURCE r);
```

`KheResourceInstance` returns `r`'s instance, and `KheResourceInstanceIndex` returns `r`'s index in that instance: the value of `i` for which `KheInstanceResource(ins, i)` returns `r`. `KheResourceResourceType` returns `r`'s resource type, and `KheResourceResourceTypeIndex` returns `r`'s index in that type: the value of `i` for which `KheResourceTypeResource(rt, i)` returns `r`. Unlike the index numbers of times, which indicate chronological order, resource index numbers have no semantic significance. They are made available only for convenience.

A resource group is created automatically for each resource `r`, holding just `r`. Function

```
KHE_RESOURCE_GROUP KheResourceSingletonResourceGroup(KHE_RESOURCE r);
```

returns this resource group. This resource group may not be changed. To visit the resource groups containing `r` (not including automatically generated ones), call

```
int KheResourceResourceGroupCount(KHE_RESOURCE r);
KHE_RESOURCE_GROUP KheResourceResourceGroup(KHE_RESOURCE r, int i);
```

in the usual way.

The event resources that `r` is preassigned to are made available by calling

```
int KheResourcePreassignedEventResourceCount(KHE_RESOURCE r);
KHE_EVENT_RESOURCE KheResourcePreassignedEventResource(KHE_RESOURCE r,
  int i);
```

Naturally, the entire instance has to be loaded for these to work correctly. At present there is no way to visit events containing event resource groups containing a given resource.

Some constraints apply to resources. When these constraints are created, they are added to the resources they apply to. To visit all the constraints applicable to a given resource, call

```
int KheResourceConstraintCount(KHE_RESOURCE r);
KHE_CONSTRAINT KheResourceConstraint(KHE_RESOURCE r, int i);
```

There may be any number of avoid clashes constraints, avoid unavailable times constraints, limit idle times constraints, cluster busy times constraints, limit busy times constraints, limit workload constraints, and limit active intervals constraints, in any order. There are also

```
KHE_TIME_GROUP KheResourceHardUnavailableTimeGroup(KHE_RESOURCE r);
KHE_TIME_GROUP KheResourceHardAndSoftUnavailableTimeGroup(
  KHE_RESOURCE r);
```

`KheResourceHardUnavailableTimeGroup` returns the union of the domains of the required unavailable times constraints of `r`. `KheResourceHardAndSoftUnavailableTimeGroup` does the same, except that the domains of all unavailable times constraints are included. Both functions return the empty time group when there are no applicable constraints.

These two public functions are used by KHE when calculating lower bounds:

```
bool KheResourceHasAvoidClashesConstraint(KHE_RESOURCE r, KHE_COST cost);
int KheResourcePreassignedEventsDuration(KHE_RESOURCE r, KHE_COST cost);
```

`KheResourceHasAvoidClashesConstraint` returns `true` if some avoid clashes constraint of combined weight greater than `cost` applies to `r`; `KheResourcePreassignedEventsDuration` returns the total duration of events which are both preassigned `r` and either preassigned a time or subject to an assign time constraint of combined cost greater than `cost`.

As an aid to sorting arrays of resources, functions

```
int KheResourceTypedCmp(KHE_RESOURCE r1, KHE_RESOURCE r2);
int KheResourceCmp(const void *t1, const void *t2);
```

are offered. `KheResourceTypedCmp` returns the instance index of `r1` minus the instance index of `r2`. `KheResourceCmp` is basically the same, but in the form suited for passing to `qsort`, and hence to `HaArraySort` and `HaArraySortUnique`.

As an aid to debugging, function

```
void KheResourceDebug(KHE_RESOURCE r, int verbosity,
  int indent, FILE *fp)
```

produces a debug print of resource `r` onto file `fp` with the given verbosity and indent, as described for debug functions in general in Section 1.4.

### 3.5.4. Resource layers

A *resource layer* is the set of events containing a preassignment of a given resource `r` which is the subject of a hard avoid clashes constraint. A resource layer's events may not overlap in time: they must spread horizontally across the timetable, hence the term 'layer'. Within a solution, the meets derived from the events of one resource layer form a *solution layer*, or just *layer*.

Layers are important in high school timetabling, at least for student group resources, since the total duration of their events is often close to the total duration of the cycle, and hence these events strongly constrain each other. The following operations are available on the layer of `r`:

```
int KheResourceLayerEventCount(KHE_RESOURCE r);
KHE_EVENT KheResourceLayerEvent(KHE_RESOURCE r, int i);
int KheResourceLayerDuration(KHE_RESOURCE r);
```

The first two work together in the usual way to return the events of the resource layer. They are sorted by increasing event index. If the resource is not preassigned to any events, or has no required avoid clashes constraint, then `KheResourceLayerEventCount` returns 0. `KheResourceLayerDuration` returns the total duration of the events of the layer. In the unlikely case that `r` is assigned to the same event twice, the event still appears only once in the list of events of the layer, and contributes its duration only once to the layer duration.

### 3.5.5. Resource type partitioning

Suppose that Science laboratories are never used as ordinary classrooms, and ordinary classrooms are never used as Science laboratories. Then it doesn't matter whether Science laboratories are considered to have resource type `Room` or some other type specific to them. The advantage of giving them their own type is that it makes it clear to solvers that assigning Science laboratories is a completely separate problem from assigning other rooms.

*Resource type partitioning* is KHE's name for a radical kind of resource partitioning, in which each partition becomes a resource type. Under suitable circumstances it will recognize, for example, that Science laboratories can be given their own resource type, and it will transform the instance accordingly. It is attempted only when the user explicitly asks for it, by setting the `resource_type_partitions` parameter of `KheInstanceMakeEnd` to `true`.

Consider any resource type `rt` (before partitioning). Suppose that there is an event resource of type `rt` which is not subject to a prefer resources constraint with non-zero hard cost. Then this event resource could be assigned any resource of type `rt`, and so partitioning will not succeed and will not be attempted, even when requested.

So suppose now that there are none of these event resources. Initialize by placing each resource in its own partition. For each pair of resources referenced (either directly or via a resource group) by a prefer resources constraint with non-zero hard cost, merge their partitions. If, at the end, there are two or more partitions, create new resource types to hold these partitions and replace each reference to `rt` in the instance by a reference to one of these new resource types. (Actually, `rt` is retained and used to hold one of its own partitions.)

After this process ends, resource groups may exist that contain resources of two or more types. These resource groups are arbitrarily assigned the resource type of their first resource; they are exceptions to the usual rule that all resources of a resource group have the same type.

Resource types for which `has_partition` is `true` are ignored by resource type partitioning. But `KheInstanceMakeEnd` does resource type partitioning before inferring resource partitions (Section 3.5.6), so a resource type created by resource type partitioning can have partitions.

There is no way to undo resource type partitioning. However, if the instance is written to a file it will display no trace of it: the resources, resource groups, and event resources all revert to their original types, and the resource types created by partitioning are not written. It is done this way because resource type partitioning is offered to help solvers, not to transform instances.

The implementation of resource type partitioning is incomplete in one respect: although `KheResourceGroupResourceType` returns a new resource type created by partitioning whenever its first resource is moved to such a type, the resource types themselves do not know that the resource groups have changed their types, so functions `KheResourceTypeResourceGroupCount`, `KheResourceTypeResourceGroup`, and `KheResourceTypeRetrieveResourceGroup` behave as though no partitioning has occurred. Functions `KheResourceTypeDemandIsAllPreassigned` and `KheResourceTypeAvoidSplitAssignmentsCount` may also return incorrect values, as may `KheResourceTypeLimitResourcesCount`. These problems will be corrected if needed.

The names assigned to resource types created by partitioning don't matter very much, but some attempt has been made to choose reasonable names, to help make debug and testing output readable. One of the resource types is the original one and it retains its original name. If there is a partition that contains more than half of the affected resources, that partition will be represented by this original resource type, otherwise there is no simple rule to say which partition it will represent. The other, newly created resource types will have names of the form `part1:part2`. Here `part1` is the name of the original resource type; `part2` is the name of a resource group if that resource group contains exactly the resources of the new resource type (as often happens), or the name of one of the resources of the newly created type otherwise.

### 3.5.6. Resource similarity and inferring resource partitions

Following the general approach introduced in Section 1.4, KHE offers function

```
bool KheResourceSimilar(KHE_RESOURCE r1, KHE_RESOURCE r2);
```

which returns `true` when resources `r1` and `r2` are similar: when they lie in similar resource groups and are preassigned to similar events. The exact definition is given below.

`KheResourceSimilar` often succeeds in recognising that student group resources from the same form are similar, and that teacher resources from the same faculty are similar. However, it needs positive evidence to work with. For example, when there are no student or teacher resource groups, and each event contains one preassigned student group resource, one preassigned teacher resource, and a request for one ordinary classroom, there is no basis for grouping the resources and each will be considered similar only to itself.

Resource partitions (Section 3.5.1) are not part of the XML format. But they are useful when solving, so `KheInstanceMakeEnd` has an `infer_resource_partitions` parameter which, when `true`, causes partitions to be added to each resource type `rt` that lacks them. Afterwards, `KheResourceTypeHasPartitions(rt)` will be `true`, `KheResourceGroupIsPartition(rg)` will be `true` for some of the resource groups of `rt`, and `KheResourcePartition(r)` will return a non-`NULL` partition for each resource `r`. All this is exactly as though the partitions had been entered explicitly, except that any specially created resource groups will not be visited by `KheResourceTypeResourceGroupCount` and `KheResourceTypeResourceGroup`.

The algorithm for inferring resource partitions is a simple application of resource similarity. Build a graph in which each node corresponds to one resource, and an edge joins two nodes when their resources are similar. The partitions are the connected components of this graph.

To decide whether two resources are similar or not, two non-negative integers, the *positive evidence* and the *negative evidence*, are calculated as explained below. The two resources are similar if the positive evidence exceeds the negative evidence by at least two.

Evidence comes from two sources: the resource groups that the resources lie in, and the events that the resources are preassigned to. A resource group is *admissible* (i.e. admissible as evidence) if its number of resources is at least two and at most one third of the number of resources of its resource type. Inadmissible resource groups are considered to contain no useful information and are ignored. Each case of an admissible resource group containing both resources counts as two units of positive evidence, and each case of an admissible resource group containing one resource but not the other counts as one unit of negative evidence.

A definition of what it means for two events to be similar appears in Section 3.6.2. Each case of an event preassigned one resource being similar to an event preassigned the other counts as two units of positive evidence. Each case of an event preassigned one resource for which there is no similar event preassigned the other counts as one unit of negative evidence. The cases are distinct, in the sense that each event participates in at most one case.

## 3.6. Events

### 3.6.1. Event groups

An event group, representing a set of events, is created and added to an instance by calling

```
bool KheEventGroupMake(KHE_INSTANCE ins, KHE_EVENT_GROUP_KIND kind,
  char *id, char *name, KHE_EVENT_GROUP *eg);
```

As usual, it returns `false` only when `id` is non-`NULL` and `ins` already contains an event group with this `id`. To set and retrieve the back pointer, call

```
void KheEventGroupSetBack(KHE_EVENT_GROUP eg, void *back);
void *KheEventGroupBack(KHE_EVENT_GROUP eg);
```

as usual. The other attributes may be retrieved by the calls

```
KHE_INSTANCE KheEventGroupInstance(KHE_EVENT_GROUP eg);
KHE_EVENT_GROUP_KIND KheEventGroupKind(KHE_EVENT_GROUP eg);
char *KheEventGroupId(KHE_EVENT_GROUP eg);
char *KheEventGroupName(KHE_EVENT_GROUP eg);
```

The event group kind is a value of type

```
typedef enum {
  KHE_EVENT_GROUP_KIND_COURSE,
  KHE_EVENT_GROUP_KIND_ORDINARY
} KHE_EVENT_GROUP_KIND;
```

The XML format allows some event groups to be referred to as Courses, although they do not differ from other event groups in any other way. The `kind` attribute records this distinction; it is only used by KHE when reading and writing XML files, not when solving.

Irrespective of the order event groups are created in, to conform with the XML rules, when writing event groups KHE writes courses first, then ordinary event groups.

Initially the event group is empty. There are several operations for changing its events:

```
void KheEventGroupAddEvent(KHE_EVENT_GROUP eg, KHE_EVENT e);
void KheEventGroupSubEvent(KHE_EVENT_GROUP eg, KHE_EVENT e);
void KheEventGroupUnion(KHE_EVENT_GROUP eg, KHE_EVENT_GROUP eg2);
void KheEventGroupIntersect(KHE_EVENT_GROUP eg, KHE_EVENT_GROUP eg2);
void KheEventGroupDifference(KHE_EVENT_GROUP eg, KHE_EVENT_GROUP eg2);
```

These add an event to `eg`, remove an event, replace `eg`'s set of events with its union or intersecton with the set of events of `eg2`, and with the difference of `eg`'s events and `eg2`'s events. The first two operations are treated as set operations, so `KheEventGroupAddEvent` does nothing if `e` is already present, and `KheEventGroupSubEvent` does nothing if `e` is not already present.

Changes to the event groups of an instance are not allowed after `KheInstanceMakeEnd` is called, since instances are immutable after that point. However, solutions may construct event groups for their own use (Section 4.4).

There are also predefined event groups, for the complete set of events of the instance and for the empty set of events (Section 3), and one for each event of the instance, containing just that event (Section 3.6). The events in predefined event groups may not be changed.

To visit the events of an event group, functions

```
int KheEventGroupEventCount(KHE_EVENT_GROUP eg);
KHE_EVENT KheEventGroupEvent(KHE_EVENT_GROUP eg, int i);
```

are used in the usual way. And

```
bool KheEventGroupContains(KHE_EVENT_GROUP eg, KHE_EVENT e);
bool KheEventGroupEqual(KHE_EVENT_GROUP eg1, KHE_EVENT_GROUP eg2);
bool KheEventGroupSubset(KHE_EVENT_GROUP eg1, KHE_EVENT_GROUP eg2);
bool KheEventGroupDisjoint(KHE_EVENT_GROUP eg1, KHE_EVENT_GROUP eg2);
```

return `true` if `eg` contains `e`, if `eg1` and `eg2` contain the same events, if the events of `eg1` are a subset of the events of `eg2`, and if the events of `eg1` and `eg2` are disjoint. There is nothing to prevent two distinct event groups from containing the same events.

Some constraints apply to event groups. When these are created, they are added to the event groups they apply to. To visit all the constraints that apply to a given event group, call

```
int KheEventGroupConstraintCount(KHE_EVENT_GROUP eg);
KHE_CONSTRAINT KheEventGroupConstraint(KHE_EVENT_GROUP eg, int i);
```

There may be any number of avoid split assignments constraints, spread events constraints, link events constraints, and limit resources constraints, in any order. Function

```
void KheEventGroupDebug(KHE_EVENT_GROUP eg, int verbosity,
  int indent, FILE *fp);
```

produces a debug print of `eg` onto `fp` with the given verbosity and indent, in the usual way.

### 3.6.2. Events

An event is created and added to an instance by calling

```
bool KheEventMake(KHE_INSTANCE ins, char *id, char *name, char *color,
  int duration, int workload, KHE_TIME preassigned_time, KHE_EVENT *e);
```

This returns `false` only if `id` is non-`NULL` and is already the `id` of an event of `ins`. Parameter `color` is an optional color for use when printing timetables. If non-`NULL`, its value must be a legal Web colour (`"#7CFC00"` for example, or a colour name). A duration and workload are compulsory (the XML specification states that a missing workload is taken to be equal to the duration), but the preassigned time may be NULL. The back pointer is set and retrieved by

```
void KheEventSetBack(KHE_EVENT e, void *back);
void *KheEventBack(KHE_EVENT e);
```

as usual, and the other attributes may be retrieved by

```
KHE_INSTANCE KheEventInstance(KHE_EVENT e);
char *KheEventId(KHE_EVENT e);
char *KheEventName(KHE_EVENT e);
char *KheEventColor(KHE_EVENT e);
int KheEventDuration(KHE_EVENT e);
int KheEventWorkload(KHE_EVENT e);
KHE_TIME KheEventPreassignedTime(KHE_EVENT e);
```

There are two other useful query functions. First,

```
int KheEventIndex(KHE_EVENT e);
```

returns the index number of e (0 for the first event inserted, 1 for the next, etc.). This number has no timetabling significance; it is included merely for convenience. Second,

```
int KheEventDemand(KHE_EVENT e);
```

returns the *demand* of e, defined to be its duration multiplied by the number of its event resources (in matching terms, the number of demand tixels). This is included as a measure of the overall bulk of an event, useful for sorting events by estimated difficulty of timetabling.

Each event also contains any number of event resources. These are added to their events as they are created. To visit them, call

```
int KheEventResourceCount(KHE_EVENT e);
KHE_EVENT_RESOURCE KheEventResource(KHE_EVENT e, int i);
```

in the usual way. There is also

```
bool KheEventRetrieveEventResource(KHE_EVENT e, char *role,
  KHE_EVENT_RESOURCE *er);
```

which retrieves an event resource from e with the given role. If there is such an event resource, it sets *er to it and returns true. If not, *er is not changed and false is returned.

Each event also contains any number of event resource groups. These are added to their events as they are created. To visit them, call

```
int KheEventResourceGroupCount(KHE_EVENT e);
KHE_EVENT_RESOURCE_GROUP KheEventResourceGroup(KHE_EVENT e, int i);
```

as usual.

For convenience, an event group is created for each event, holding just that event. Call

```
KHE_EVENT_GROUP KheEventSingletonEventGroup(KHE_EVENT event);
```

to retrieve this event group. Other events may not be added to it.

Some constraints apply to events. When these constraints are created, they are added to the events they apply to. To visit all the constraints applicable to a given event, call

```
int KheEventConstraintCount(KHE_EVENT e);
KHE_CONSTRAINT KheEventConstraint(KHE_EVENT e, int i);
```

There may be any number of assign time constraints, prefer times constraints, split events constraints, and distribute split events constraints, in any order, except that an event with a preassigned time cannot have assign time constraints and prefer times constraints.

Following the general pattern given in Section 1.4, function

```
bool KheEventSimilar(KHE_EVENT e1, KHE_EVENT e2);
```

returns `true` if `e1` and `e2` are similar: if they have the same duration and similar event resources. The exact definition is as follows. An event is *admissible* if it has one or more admissible event resources. An event resource is admissible if its hard domain (reflecting its prefer resources constraints and any preassignment) is an admissible resource group, as defined in Section 3.5.6. An event is always similar to itself. Two distinct events are similar if they are admissible, have equal durations, and their admissible event resources (taken in any order) have equal hard domains.

There is also

```
bool KheEventMergeable(KHE_EVENT e1, KHE_EVENT e2, int slack);
```

which returns `true` if `e1` and `e2` could reasonably be considered to be split fragments of a single larger event: if their event resources correspond, ignoring differences in the order in which they appear in the two events. If `slack` is non-zero, `KheEventMergeable` returns `true` even if up to `slack` event resources in `e1` do not correspond with any event resource in `e2` and vice versa. Two event resources correspond when they have the same resource type, the same preassigned resource, equal hard domains as returned by `KheEventResourceHardDomain`, and equal hard-and-soft domains as returned by `KheEventResourceHardAndSoftDomain`. Like those two functions, `KheEventMergeable` can only be called after the instance is complete.

A reasonable way to decide whether two events must be disjoint in time is to call

```
bool KheEventSharePreassignedResource(KHE_EVENT e1, KHE_EVENT e2,
  KHE_RESOURCE *r);
```

If `e1` and `e2` share a preassigned resource which has a required avoid clashes constraint, this function returns `true` and sets `r` to one such resource; otherwise it returns `false` and sets `r` to `NULL`. It should only be called after the instance is complete.

Function

```
void KheEventDebug(KHE_EVENT e, int verbosity, int indent, FILE *fp);
```

produces a debug print of `e` onto `fp` with the given verbosity and indent, in the usual way.

### 3.6.3. Event resources

An event resource is created and added to an event by the call

```
bool KheEventResourceMake(KHE_EVENT event, KHE_RESOURCE_TYPE rt,
  KHE_RESOURCE preassigned_resource, char *role, int workload,
  KHE_EVENT_RESOURCE *er);
```

This returns `false` only when the optional `role` parameter (used only when writing XML) is non-`NULL` and there is already an event resource within `event` with this value for `role`. Parameter `preassigned_resource` is an optional resource preassignment and may be `NULL`.

To set and retrieve the back pointer of an event resource, call

```
void KheEventResourceSetBack(KHE_EVENT_RESOURCE er, void *back);
void *KheEventResourceBack(KHE_EVENT_RESOURCE er);
```

as usual. The other attributes may be retrieved by

```
KHE_INSTANCE KheEventResourceInstance(KHE_EVENT_RESOURCE er);
int KheEventResourceInstanceIndex(KHE_EVENT_RESOURCE er);
KHE_EVENT KheEventResourceEvent(KHE_EVENT_RESOURCE er);
int KheEventResourceEventIndex(KHE_EVENT_RESOURCE er);
KHE_RESOURCE_TYPE KheEventResourceResourceType(KHE_EVENT_RESOURCE er);
KHE_RESOURCE KheEventResourcePreassignedResource(KHE_EVENT_RESOURCE er);
char *KheEventResourceRole(KHE_EVENT_RESOURCE er);
int KheEventResourceWorkload(KHE_EVENT_RESOURCE er);
float KheEventResourceWorkloadPerTime(KHE_EVENT_RESOURCE er);
```

`KheEventResourceInstance` is the enclosing instance; `KheEventResourceInstanceIndex` is the index of `er` in that instance (the number `i` such that `KheInstanceEventResource(ins, i)` returns `er`). `KheEventResourceEvent` is the enclosing event; `KheEventResourceEventIndex` is the index of `er` in that event (the number `i` such that `KheEventResource(e, i)` returns `er`). The next three functions just echo parameters. Finally, `KheEventResourceWorkloadPerTime` returns the floating-point workload per time of the event resource: its workload divided by the duration of the enclosing event.

Some constraints apply to event resources. When these are created, they are added to the event resources they apply to. To visit the constraints that apply to a given event resource, call

```
int KheEventResourceConstraintCount(KHE_EVENT_RESOURCE er);
KHE_CONSTRAINT KheEventResourceConstraint(KHE_EVENT_RESOURCE er, int i);
```

There may be any number of assign resource constraints, prefer resources constraints, and avoid split assignments constraints, in any order, except that an event resource with a preassigned resource cannot have assign resource constraints and prefer resources constraints. If the `i`'th constraint is an avoid split assignments constraint, function

```
int KheEventResourceConstraintEventGroupIndex(KHE_EVENT_RESOURCE er, int i);
```

may be called to find the event group index within that constraint that contains `er`. (It returns `-1` if the `i`'th constraint is not an avoid split assignments constraint.)

After the instance is complete but not before, function

```
KHE_MAYBE_TYPE KheEventResourceNeedsAssignment(KHE_EVENT_RESOURCE er);
```

may be called to determine whether the constraints on `er` mean that it needs assignment (i.e. that not assigning it would produce a positive hard or soft cost). Its return type is

```
typedef enum {
  KHE_NO,
  KHE_MAYBE,
  KHE_YES
} KHE_MAYBE_TYPE;
```

`KHE_YES` means that it does need assignment, because at least one assign resource constraint with positive cost applies to it; `KHE_MAYBE` means that there is no case for `KHE_YES`, but at least one limit resources constraint with positive cost and positive minimum limit applies to it; and `KHE_NO` means that there is no case for `KHE_YES` or `KHE_MAYBE`.

Also after the instance is complete, functions

```
KHE_RESOURCE_GROUP KheEventResourceHardDomain(KHE_EVENT_RESOURCE er);
KHE_RESOURCE_GROUP KheEventResourceHardAndSoftDomain(KHE_EVENT_RESOURCE er);
```

return domains suited to `er`. The resource group returned by `KheEventResourceHardDomain` is the intersection of the domains of the required prefer resources constraints, with weight greater than 0, of `er` and other event resources that share a required avoid split assignments constraint of weight greater than 0 with `er`, either directly or indirectly via any number of intermediate event resources. If any of these event resources is preassigned, then the singleton resource groups containing the preassigned resources are intersected along with the other groups. The same is true of `KheEventResourceHardAndSoftDomain`, except that both hard and soft prefer resources and avoid split assignments constraints are used, producing smaller domains in general.

These functions are not recommended for use when solving, since `KheTaskTreeMake` offers a more sophisticated way of initializing the domains of tasks. `KheEventResourceHardDomain` is used when deciding whether events are similar.

Formerly at this point a function called `KheEventResourceEquivalent` was introduced, which returned `true` when two given event resources are equivalent, in the sense that they lie in the same events and are monitored by similar constraints. This function has been withdrawn, because multi-tasks (Section 11.9) now do the same job rather better.

Function

```
void KheEventResourceDebug(KHE_EVENT_RESOURCE er, int verbosity,
  int indent, FILE *fp);
```

produces a debug print of `er` onto `fp` with the given verbosity and indent, in the usual way.

### 3.6.4. Event resource groups

An event resource group is created and added to an event by the call

```
KHE_EVENT_RESOURCE_GROUP KheEventResourceGroupMake(KHE_EVENT event,
  KHE_RESOURCE_GROUP rg);
```

Its attributes may be retrieved by calling

```
KHE_EVENT KheEventResourceGroupEvent(KHE_EVENT_RESOURCE_GROUP erg);
KHE_RESOURCE_GROUP KheEventResourceGroupResourceGroup(
  KHE_EVENT_RESOURCE_GROUP erg);
```

In addition to making a new event resource group object, `KheEventResourceGroupMake` calls `KheEventResourceMake` once for each resource of `rg`, with the resource for its `preassigned_resource` parameter and the obvious values for its other parameters. This satisfies the semantic requirement that adding a resource group should be just like adding its resources individually. These added event resources appear on the list of event resources of the event just like other event resources; they can be distinguished from them only by calling

```
KHE_EVENT_RESOURCE_GROUP KheEventResourceEventResourceGroup(
  KHE_EVENT_RESOURCE er);
```

which returns the event resource group that caused `er` to be created when there is one, and `NULL` when `er` was created directly. For example, when printing XML files, KHE calls this function once for each event resource, to decide whether it should be printed explicitly or omitted because it is part of an event resource group. Function

```
void KheEventResourceGroupDebug(KHE_EVENT_RESOURCE_GROUP erg,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of `erg` onto `fp` with the given verbosity and indent, in the usual way.

## 3.7. Constraints

Some attributes of constraints are common to all kinds of constraints; others vary from one kind of constraint to another. Accordingly, KHE offers type `KHE_CONSTRAINT`, which is the abstract supertype of all kinds of constraints, and one subtype of this type for each kind of constraint.

To set and retrieve the back pointer of a constraint object, call

```
void KheConstraintSetBack(KHE_CONSTRAINT c, void *back);
void *KheConstraintBack(KHE_CONSTRAINT c);
```

as usual. To retrieve the other attributes common to all kinds of constraints, use functions

```
KHE_INSTANCE KheConstraintInstance(KHE_CONSTRAINT c);
char *KheConstraintId(KHE_CONSTRAINT c);
char *KheConstraintName(KHE_CONSTRAINT c);
bool KheConstraintRequired(KHE_CONSTRAINT c);
int KheConstraintWeight(KHE_CONSTRAINT c);
KHE_COST KheConstraintCombinedWeight(KHE_CONSTRAINT c);
KHE_COST_FUNCTION KheConstraintCostFunction(KHE_CONSTRAINT c);
int KheConstraintIndex(KHE_CONSTRAINT c);
KHE_CONSTRAINT_TAG KheConstraintTag(KHE_CONSTRAINT c);
```

`KheConstraintInstance` returns the instance; `KheConstraintId` and `KheConstraintName` return the constraint's Id and Name (as usual, these are optional in KHE, needed only when writing XML). `KheConstraintRequired` is `true` when the Required attribute is true.

`KheConstraintWeight` is the weight given to violations of the constraint. As explained in Section 6.1, `KheConstraintCombinedWeight` is similar, except that hard constraints are weighted more heavily; `KHE_COST` is also defined there. However it is usually a mistake to call `KheConstraintCombinedWeight`; `KheMonitorCombinedWeight` (Section 6.2) is better.

`KheConstraintCostFunction` is the cost function used when calculating costs, of type

```
typedef enum {
  KHE_STEP_COST_FUNCTION,
  KHE_LINEAR_COST_FUNCTION,
  KHE_QUADRATIC_COST_FUNCTION
} KHE_COST_FUNCTION;
```

`KheConstraintIndex` returns an automatically generated index number for `c`: 0 for the first constraint created, 1 for the second, and so on. `KheConstraintTag` is the type tag which determines which concrete kind of constraint this is, with type

```
typedef enum {
  KHE_ASSIGN_RESOURCE_CONSTRAINT_TAG,
  KHE_ASSIGN_TIME_CONSTRAINT_TAG,
  KHE_SPLIT_EVENTS_CONSTRAINT_TAG,
  KHE_DISTRIBUTE_SPLIT_EVENTS_CONSTRAINT_TAG,
  KHE_PREFER_RESOURCES_CONSTRAINT_TAG,
  KHE_PREFER_TIMES_CONSTRAINT_TAG,
  KHE_AVOID_SPLIT_ASSIGNMENTS_CONSTRAINT_TAG,
  KHE_SPREAD_EVENTS_CONSTRAINT_TAG,
  KHE_LINK_EVENTS_CONSTRAINT_TAG,
  KHE_ORDER_EVENTS_CONSTRAINT_TAG,
  KHE_AVOID_CLASHES_CONSTRAINT_TAG,
  KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT_TAG,
  KHE_LIMIT_IDLE_TIMES_CONSTRAINT_TAG,
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT_TAG,
  KHE_LIMIT_BUSY_TIMES_CONSTRAINT_TAG,
  KHE_LIMIT_WORKLOAD_CONSTRAINT_TAG,
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT_TAG,
  KHE_LIMIT_RESOURCES_CONSTRAINT_TAG,
  KHE_CONSTRAINT_TAG_COUNT
} KHE_CONSTRAINT_TAG;
```

The last value is not a valid tag; it counts the number of constraints, allowing code of the form

```
for( tag = 0;  tag < KHE_CONSTRAINT_TAG_COUNT;  tag++ )
  ...
```

to be written which visits every tag, now and in the future.

The number of points of application of a constraint is returned by

```
int KheConstraintAppliesToCount(KHE_CONSTRAINT c);
```

For an assign resource constraint this is the total number of event resources; for a split events constraint it is the total number of events plus the sizes of the event groups; and so on.

Given a tag, one can obtain a string representation of the constraint name by calling

```
char *KheConstraintTagShow(KHE_CONSTRAINT_TAG tag);
char *KheConstraintTagShowSpaced(KHE_CONSTRAINT_TAG tag);
```

The first returns an unspaced form (`"AssignResourceConstraint"` and so on), the second returns a spaced form (`"Assign Resource Constraint"` and so on). There is also

```
KHE_CONSTRAINT_TAG KheStringToConstraintTag(char *str);
```

which implements the inverse function, from unspaced constraint names to constraint tags, and

```
char *KheCostFunctionShow(KHE_COST_FUNCTION cf);
```

which returns a cost function's string representation, and

```
void KheConstraintDebug(KHE_CONSTRAINT c, int verbosity,
  int indent, FILE *fp);
```

which produces a debug print of `c` onto `fp` with the given verbosity and indent. This just calls the appropriate debug function for the downcast value: `KheAssignResourceConstraintDebug`, `KheAssignTimeConstraintDebug`, and so on.

The names of the concrete subtypes themselves are

```
KHE_ASSIGN_RESOURCE_CONSTRAINT
KHE_ASSIGN_TIME_CONSTRAINT
KHE_SPLIT_EVENTS_CONSTRAINT
KHE_DISTRIBUTE_SPLIT_EVENTS_CONSTRAINT
KHE_PREFER_RESOURCES_CONSTRAINT
KHE_PREFER_TIMES_CONSTRAINT
KHE_AVOID_SPLIT_ASSIGNMENTS_CONSTRAINT
KHE_SPREAD_EVENTS_CONSTRAINT
KHE_LINK_EVENTS_CONSTRAINT
KHE_ORDER_EVENTS_CONSTRAINT
KHE_AVOID_CLASHES_CONSTRAINT
KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT
KHE_LIMIT_IDLE_TIMES_CONSTRAINT
KHE_CLUSTER_BUSY_TIMES_CONSTRAINT
KHE_LIMIT_BUSY_TIMES_CONSTRAINT
KHE_LIMIT_WORKLOAD_CONSTRAINT
KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT
KHE_LIMIT_RESOURCES_CONSTRAINT
```

Downcasting and upcasting between `KHE_CONSTRAINT` and each of these subtypes, using C casts, is a normal part of the use of KHE. Alternatively, since C casts can also be used for unsafe things, explicit functions are offered for upcasting:

```
    KHE_CONSTRAINT KheFromAssignResourceConstraint(
      KHE_ASSIGN_RESOURCE_CONSTRAINT c);
    KHE_CONSTRAINT KheFromAssignTimeConstraint(
      KHE_ASSIGN_TIME_CONSTRAINT c);
    KHE_CONSTRAINT KheFromSplitEventsConstraint(
      KHE_SPLIT_EVENTS_CONSTRAINT c);
    KHE_CONSTRAINT KheFromDistributeSplitEventsConstraint(
      KHE_DISTRIBUTE_SPLIT_EVENTS_CONSTRAINT c);
    KHE_CONSTRAINT KheFromPreferResourcesConstraint(
      KHE_PREFER_RESOURCES_CONSTRAINT c);
    KHE_CONSTRAINT KheFromPreferTimesConstraint(
      KHE_PREFER_TIMES_CONSTRAINT c);
    KHE_CONSTRAINT KheFromAvoidSplitAssignmentsConstraint(
      KHE_AVOID_SPLIT_ASSIGNMENTS_CONSTRAINT c);
    KHE_CONSTRAINT KheFromSpreadEventsConstraint(
      KHE_SPREAD_EVENTS_CONSTRAINT c);
    KHE_CONSTRAINT KheFromLinkEventsConstraint(
      KHE_LINK_EVENTS_CONSTRAINT c);
    KHE_CONSTRAINT KheFromOrderEventsConstraint(
      KHE_ORDER_EVENTS_CONSTRAINT c);
    KHE_CONSTRAINT KheFromAvoidClashesConstraint(
      KHE_AVOID_CLASHES_CONSTRAINT c);
    KHE_CONSTRAINT KheFromAvoidUnavailableTimesConstraint(
      KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT c);
    KHE_CONSTRAINT KheFromLimitIdleTimesConstraint(
      KHE_LIMIT_IDLE_TIMES_CONSTRAINT c);
    KHE_CONSTRAINT KheFromClusterBusyTimesConstraint(
      KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c);
    KHE_CONSTRAINT KheFromLimitBusyTimesConstraint(
      KHE_LIMIT_BUSY_TIMES_CONSTRAINT c);
    KHE_CONSTRAINT KheFromLimitWorkloadConstraint(
      KHE_LIMIT_WORKLOAD_CONSTRAINT c);
    KHE_CONSTRAINT KheFromLimitActiveIntervalsConstraint(
      KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c);
    KHE_CONSTRAINT KheFromLimitResourcesConstraint(
      KHE_LIMIT_RESOURCES_CONSTRAINT c);
```

and for downcasting:

```
KHE_ASSIGN_RESOURCE_CONSTRAINT
  KheToAssignResourceConstraint(KHE_CONSTRAINT c);
KHE_ASSIGN_TIME_CONSTRAINT
  KheToAssignTimeConstraint(KHE_CONSTRAINT c);
KHE_SPLIT_EVENTS_CONSTRAINT
  KheToSplitEventsConstraint(KHE_CONSTRAINT c);
KHE_DISTRIBUTE_SPLIT_EVENTS_CONSTRAINT
  KheToDistributeSplitEventsConstraint(KHE_CONSTRAINT c);
KHE_PREFER_RESOURCES_CONSTRAINT
  KheToPreferResourcesConstraint(KHE_CONSTRAINT c);
KHE_PREFER_TIMES_CONSTRAINT
  KheToPreferTimesConstraint(KHE_CONSTRAINT c);
KHE_AVOID_SPLIT_ASSIGNMENTS_CONSTRAINT
  KheToAvoidSplitAssignmentsConstraint(KHE_CONSTRAINT c);
KHE_SPREAD_EVENTS_CONSTRAINT
  KheToSpreadEventsConstraint(KHE_CONSTRAINT c);
KHE_LINK_EVENTS_CONSTRAINT
  KheToLinkEventsConstraint(KHE_CONSTRAINT c);
KHE_ORDER_EVENTS_CONSTRAINT
  KheToOrderEventsConstraint(KHE_CONSTRAINT c);
KHE_AVOID_CLASHES_CONSTRAINT
  KheToAvoidClashesConstraint(KHE_CONSTRAINT c);
KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT
  KheToAvoidUnavailableTimesConstraint(KHE_CONSTRAINT c);
KHE_LIMIT_IDLE_TIMES_CONSTRAINT
  KheToLimitIdleTimesConstraint(KHE_CONSTRAINT c);
KHE_CLUSTER_BUSY_TIMES_CONSTRAINT
  KheToClusterBusyTimesConstraint(KHE_CONSTRAINT c);
KHE_LIMIT_BUSY_TIMES_CONSTRAINT
  KheToLimitBusyTimesConstraint(KHE_CONSTRAINT c);
KHE_LIMIT_WORKLOAD_CONSTRAINT
  KheToLimitWorkloadConstraint(KHE_CONSTRAINT c);
KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT
  KheToLimitActiveIntervalsConstraint(KHE_CONSTRAINT c);
KHE_LIMIT_RESOURCES_CONSTRAINT
  KheToLimitResourcesConstraint(KHE_CONSTRAINT c);
```

The downcasting functions check that their parameter is of the correct type, and abort if not.

### 3.7.1. Assign resource constraints

An assign resource constraint is created and added to an instance by

```
bool KheAssignResourceConstraintMake(KHE_INSTANCE ins, char *id,
  char *name, bool required, int weight, KHE_COST_FUNCTION cf,
  char *role, KHE_ASSIGN_RESOURCE_CONSTRAINT *c);
```

This accepts the attributes common to all constraints, followed by an optional `role`, which is specific to this kind of constraint. As usual, if successful it returns `true`, setting `*c` to the new constraint; if not (which can only be because `id` is non-`NULL` and equal to the Id of an existing constraint of `ins`), then it returns `false`, setting `*c` to `NULL`.

The attributes common to all kinds of constraints may be retrieved by upcasting to `KHE_CONSTRAINT` and calling the relevant operations on that type. The attribute specific to assign resources constraints may be retrieved by calling

```
char *KheAssignResourceConstraintRole(KHE_ASSIGN_RESOURCE_CONSTRAINT c);
```

Initially the constraint has no points of application. There are two ways to add them. The first is to give `NULL` for `role`, then add the event resources that this constraint applies to by calling

```
void KheAssignResourceConstraintAddEventResource(
  KHE_ASSIGN_RESOURCE_CONSTRAINT c, KHE_EVENT_RESOURCE er);
```

as often as necessary. It is an error to call this function when `er` contains a preassigned resource, since assign resource constraints do not apply to event resources with preassigned resources. To visit the event resources of `c`, call

```
int KheAssignResourceConstraintEventResourceCount(
  KHE_ASSIGN_RESOURCE_CONSTRAINT c);
KHE_EVENT_RESOURCE KheAssignResourceConstraintEventResource(
  KHE_ASSIGN_RESOURCE_CONSTRAINT c, int i);
```

as usual.

The second way to add event resources, used when reading XML files, is to give a non-`NULL` value for `role`, then add events and event groups. To add events and visit them, the calls are

```
void KheAssignResourceConstraintAddEvent(
  KHE_ASSIGN_RESOURCE_CONSTRAINT c, KHE_EVENT e);
int KheAssignResourceConstraintEventCount(
  KHE_ASSIGN_RESOURCE_CONSTRAINT c);
KHE_EVENT KheAssignResourceConstraintEvent(
  KHE_ASSIGN_RESOURCE_CONSTRAINT c, int i);
```

To add event groups and visit them, the calls are

```
void KheAssignResourceConstraintAddEventGroup(
  KHE_ASSIGN_RESOURCE_CONSTRAINT c, KHE_EVENT_GROUP eg);
int KheAssignResourceConstraintEventGroupCount(
  KHE_ASSIGN_RESOURCE_CONSTRAINT c);
KHE_EVENT_GROUP KheAssignResourceConstraintEventGroup(
  KHE_ASSIGN_RESOURCE_CONSTRAINT c, int i);
```

When this is done, KHE stores the events and event groups in the constraint so that they can be written out again correctly later, but it also works out which event resources the constraint applies to and calls `KheAssignResourceConstraintAddEventResource` for each of them, taking due note of the XML rule that it does not apply when an event does not contain an event resource with the specified role, or when such an event resource has a preassigned resource.

Function

```
void KheAssignResourceConstraintDebug(KHE_ASSIGN_RESOURCE_CONSTRAINT c,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of `c` onto `fp` with the given verbosity and indent, in the usual way.

The constraint density of the assign resources constraints of an instance (Section 3.3) is their number of their points of application divided by the number of event resources without preassigned resources.

### 3.7.2. Assign time constraints

An assign time constraint is created and added to an instance by

```
bool KheAssignTimeConstraintMake(KHE_INSTANCE ins, char *id,
  char *name, bool required, int weight, KHE_COST_FUNCTION cf,
  KHE_ASSIGN_TIME_CONSTRAINT *c);
```

As usual, if successful it returns `true`, setting `*c` to the new constraint; if not (which can only be because `id` is non-`NULL` and equal to the Id of an existing constraint of `ins`), then it returns `false`, setting `*c` to `NULL`. The attributes may be retrieved by upcasting to `KHE_CONSTRAINT` and calling the relevant operations on that type.

The points of application of an assign time constraint are events, and the XML file allows them to be given individually and in groups. To add individual events and visit them, call

```
void KheAssignTimeConstraintAddEvent(KHE_ASSIGN_TIME_CONSTRAINT c,
  KHE_EVENT e);
int KheAssignTimeConstraintEventCount(KHE_ASSIGN_TIME_CONSTRAINT c);
KHE_EVENT KheAssignTimeConstraintEvent(KHE_ASSIGN_TIME_CONSTRAINT c,
  int i);
```

To add groups of events and visit them, call

```
void KheAssignTimeConstraintAddEventGroup(KHE_ASSIGN_TIME_CONSTRAINT c,
  KHE_EVENT_GROUP eg);
int KheAssignTimeConstraintEventGroupCount(
  KHE_ASSIGN_TIME_CONSTRAINT c);
KHE_EVENT_GROUP KheAssignTimeConstraintEventGroup(
  KHE_ASSIGN_TIME_CONSTRAINT c, int i);
```

The XML specification states that assign time constraints skip events with preassigned times, whether those events are mentioned or not. Accordingly, although such events are added to constraints by the calls just given, the reverse links, from the events to the constraint, are added only to events that do not have preassigned times.

Function

```
void KheAssignTimeConstraintDebug(KHE_ASSIGN_TIME_CONSTRAINT c,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of c onto fp with the given verbosity and indent, in the usual way.

The constraint density of the assign times constraints of an instance (Section 3.3) is their number of points of application divided by the number of events without preassigned times.

### 3.7.3. Split events constraints

A split events constraint is created and added to an instance by

```
bool KheSplitEventsConstraintMake(KHE_INSTANCE ins, char *id,
  char *name, bool required, int weight, KHE_COST_FUNCTION cf,
  int min_duration, int max_duration, int min_amount,
  int max_amount, KHE_SPLIT_EVENTS_CONSTRAINT *c);
```

in the usual way. Most of the attributes may be retrieved by upcasting to KHE_CONSTRAINT and calling the relevant operation on that type. The exceptions are

```
int KheSplitEventsConstraintMinDuration(KHE_SPLIT_EVENTS_CONSTRAINT c);
int KheSplitEventsConstraintMaxDuration(KHE_SPLIT_EVENTS_CONSTRAINT c);
int KheSplitEventsConstraintMinAmount(KHE_SPLIT_EVENTS_CONSTRAINT c);
int KheSplitEventsConstraintMaxAmount(KHE_SPLIT_EVENTS_CONSTRAINT c);
```

which return the various attributes specific to split events constraints.

The points of application are events, and, as for assign time constraints, these may be added and visited individually:

```
void KheSplitEventsConstraintAddEvent(KHE_SPLIT_EVENTS_CONSTRAINT c,
  KHE_EVENT e);
int KheSplitEventsConstraintEventCount(KHE_SPLIT_EVENTS_CONSTRAINT c);
KHE_EVENT KheSplitEventsConstraintEvent(KHE_SPLIT_EVENTS_CONSTRAINT c,
  int i);
```

and also in groups:

```
void KheSplitEventsConstraintAddEventGroup(
  KHE_SPLIT_EVENTS_CONSTRAINT c, KHE_EVENT_GROUP eg);
int KheSplitEventsConstraintEventGroupCount(
  KHE_SPLIT_EVENTS_CONSTRAINT c);
KHE_EVENT_GROUP KheSplitEventsConstraintEventGroup(
  KHE_SPLIT_EVENTS_CONSTRAINT c, int i);
```

All the events are linked to the constraint, unlike for assign time constraints.

Function

```
void KheSplitEventsConstraintDebug(KHE_SPLIT_EVENTS_CONSTRAINT c,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of c onto fp with the given verbosity and indent, in the usual way.

The constraint density of the split events constraints of an instance (Section 3.3) is their number of points of application divided by the total number of events.

### 3.7.4. Distribute split events constraints

A distribute split events constraint is created and added to an instance by

```
bool KheDistributeSplitEventsConstraintMake(KHE_INSTANCE ins, char *id,
  char *name, bool required, int weight, KHE_COST_FUNCTION cf,
  int duration, int minimum, int maximum,
  KHE_DISTRIBUTE_SPLIT_EVENTS_CONSTRAINT *c);
```

in the usual way. Most of the attributes may be retrieved by upcasting to `KHE_CONSTRAINT` and calling the relevant operation on that type. The exceptions are

```
int KheDistributeSplitEventsConstraintDuration(
  KHE_DISTRIBUTE_SPLIT_EVENTS_CONSTRAINT c);
int KheDistributeSplitEventsConstraintMinimum(
  KHE_DISTRIBUTE_SPLIT_EVENTS_CONSTRAINT c);
int KheDistributeSplitEventsConstraintMaximum(
  KHE_DISTRIBUTE_SPLIT_EVENTS_CONSTRAINT c);
```

which return the various attributes specific to distribute split events constraints.

The points of application are events, and, as for split events constraints, these may be added and visited individually:

```
void KheDistributeSplitEventsConstraintAddEvent(
  KHE_DISTRIBUTE_SPLIT_EVENTS_CONSTRAINT c, KHE_EVENT e);
int KheDistributeSplitEventsConstraintEventCount(
  KHE_DISTRIBUTE_SPLIT_EVENTS_CONSTRAINT c);
KHE_EVENT KheDistributeSplitEventsConstraintEvent(
  KHE_DISTRIBUTE_SPLIT_EVENTS_CONSTRAINT c, int i);
```

and also in groups:

```
void KheDistributeSplitEventsConstraintAddEventGroup(
  KHE_DISTRIBUTE_SPLIT_EVENTS_CONSTRAINT c, KHE_EVENT_GROUP eg);
int KheDistributeSplitEventsConstraintEventGroupCount(
  KHE_DISTRIBUTE_SPLIT_EVENTS_CONSTRAINT c);
KHE_EVENT_GROUP KheDistributeSplitEventsConstraintEventGroup(
  KHE_DISTRIBUTE_SPLIT_EVENTS_CONSTRAINT c, int i);
```

All the events are linked to the constraint.

Function

```
void KheDistributeSplitEventsConstraintDebug(
  KHE_DISTRIBUTE_SPLIT_EVENTS_CONSTRAINT c,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of `c` onto `fp` with the given verbosity and indent, in the usual way.

The constraint density of the distribute split events constraints of an instance (Section 3.3) is their number of points of application divided by the total number of events.

### 3.7.5. Prefer resources constraints

A prefer resources constraint is created and added to an instance by

```
bool KhePreferResourcesConstraintMake(KHE_INSTANCE ins, char *id,
  char *name, bool required, int weight, KHE_COST_FUNCTION cf,
  char *role, KHE_PREFER_RESOURCES_CONSTRAINT *c);
```

As usual, the only reason for returning `false` is that `id` is non-`NULL` and there is already a constraint in `ins` with this `id`. Most of the attributes may be retrieved by upcasting to `KHE_CONSTRAINT` and calling the relevant operations on that type; the exception is `role`, which is retrieved by calling

```
char *KhePreferResourcesConstraintRole(KHE_PREFER_RESOURCES_CONSTRAINT c);
```

since it is specific to this constraint type.

In the XML specification, the resources that make up the domain of the constraint may be added in groups or individually. To add them in groups, and to visit the groups, call

```
bool KhePreferResourcesConstraintAddResourceGroup(
  KHE_PREFER_RESOURCES_CONSTRAINT c, KHE_RESOURCE_GROUP rg);
int KhePreferResourcesConstraintResourceGroupCount(
  KHE_PREFER_RESOURCES_CONSTRAINT c);
KHE_RESOURCE_GROUP KhePreferResourcesConstraintResourceGroup(
  KHE_PREFER_RESOURCES_CONSTRAINT c, int i);
```

The `bool` result type of `KhePreferResourcesConstraintAddResourceGroup` (and other functions below) is explained at the end of this section. To add and visit resources individually, call

```
bool KhePreferResourcesConstraintAddResource(
  KHE_PREFER_RESOURCES_CONSTRAINT c, KHE_RESOURCE r);
int KhePreferResourcesConstraintResourceCount(
  KHE_PREFER_RESOURCES_CONSTRAINT c);
KHE_RESOURCE KhePreferResourcesConstraintResource(
  KHE_PREFER_RESOURCES_CONSTRAINT c, int i);
```

After the instance is complete, but not before, function

```
KHE_RESOURCE_GROUP KhePreferResourcesConstraintDomain(
  KHE_PREFER_RESOURCES_CONSTRAINT c);
```

returns the domain of `c` as a single resource group. If exactly one resource group or one resource was added, this resource group will be that resource group or the automatically created singleton resource group for that resource; otherwise it will be created by taking the union of everything added. This resource group may be used like any other, except for a problem in one special case: when no resource groups or resources are added, the domain is not only an empty resource group but also has a `NULL` resource type.

There is also

```
KHE_RESOURCE_GROUP KheLimitResourcesConstraintDomainComplement(
  KHE_LIMIT_RESOURCES_CONSTRAINT c);
```

which returns the complement of the domain, that is, the set of resources of the same type as the domain that are not in it. This will not work when `c`'s domain is empty.

The points of application of prefer resources constraints are event resources, and they are handled in the same way as for assign resource constraints. That is, one can load the event resources directly by having a `NULL` value for `role` and calling

```
bool KhePreferResourcesConstraintAddEventResource(
  KHE_PREFER_RESOURCES_CONSTRAINT c, KHE_EVENT_RESOURCE er);
int KhePreferResourcesConstraintEventResourceCount(
  KHE_PREFER_RESOURCES_CONSTRAINT c);
KHE_EVENT_RESOURCE KhePreferResourcesConstraintEventResource(
  KHE_PREFER_RESOURCES_CONSTRAINT c, int i);
```

or load them indirectly by loading events:

```
bool KhePreferResourcesConstraintAddEvent(
  KHE_PREFER_RESOURCES_CONSTRAINT c, KHE_EVENT e);
int KhePreferResourcesConstraintEventCount(
  KHE_PREFER_RESOURCES_CONSTRAINT c);
KHE_EVENT KhePreferResourcesConstraintEvent(
  KHE_PREFER_RESOURCES_CONSTRAINT c, int i);
```

and event groups:

```
bool KhePreferResourcesConstraintAddEventGroup(
  KHE_PREFER_RESOURCES_CONSTRAINT c, KHE_EVENT_GROUP eg,
  KHE_EVENT *problem_event);
int KhePreferResourcesConstraintEventGroupCount(
  KHE_PREFER_RESOURCES_CONSTRAINT c);
KHE_EVENT_GROUP KhePreferResourcesConstraintEventGroup(
  KHE_PREFER_RESOURCES_CONSTRAINT c, int i);
```

When `KhePreferResourcesConstraintAddEventGroup` returns `false`, `problem_event` is set to the first event that caused the problem. The rules for skipping inappropriate events are as for assign resource constraints.

The resources, resource groups, and event resources of a prefer resources constraint all have a resource type attribute. All these resources types must be equal. This is why the operations above for adding a resource, resource group, event resource, event, or event group all have a `bool` result type: they all return `false` and add nothing if the operation would add an entity with a different resource type from something added previously.

Function

```
void KhePreferResourcesConstraintDebug(KHE_PREFER_RESOURCES_CONSTRAINT c,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of `c` onto `fp` with the given verbosity and indent, in the usual way.

The constraint density of prefer resources constraints (Section 3.3) is the number of points of application divided by the number of event resources without preassigned resources.

### 3.7.6. Prefer times constraints

A prefer times constraint is created and added to an instance by

```
bool KhePreferTimesConstraintMake(KHE_INSTANCE ins, char *id,
  char *name, bool required, int weight, KHE_COST_FUNCTION cf,
  int duration, KHE_PREFER_TIMES_CONSTRAINT *c);
```

As usual, the only possible reason for returning `false` is that `id` is non-`NULL` and there is already a constraint in `ins` with this `id`. A duration is optional; to not give one (meaning that the constraint applies for all durations), use the special value `KHE_ANY_DURATION`, a synonym for 0.

Most of the attributes may be retrieved by upcasting to `KHE_CONSTRAINT` and calling the relevant operations on that type; the exception is `duration`, which is retrieved by calling

```
int KhePreferTimesConstraintDuration(KHE_PREFER_TIMES_CONSTRAINT c);
```

since it is specific to this constraint type.

In the XML specification, the times that make up the domain of the constraint may be added in groups or individually. To add them in groups, and to visit the groups, call

```
void KhePreferTimesConstraintAddTimeGroup(
  KHE_PREFER_TIMES_CONSTRAINT c, KHE_TIME_GROUP tg);
int KhePreferTimesConstraintTimeGroupCount(
  KHE_PREFER_TIMES_CONSTRAINT c);
KHE_TIME_GROUP KhePreferTimesConstraintTimeGroup(
  KHE_PREFER_TIMES_CONSTRAINT c, int i);
```

To add and visit times individually, call

```
void KhePreferTimesConstraintAddTime(
  KHE_PREFER_TIMES_CONSTRAINT c, KHE_TIME t);
int KhePreferTimesConstraintTimeCount(
  KHE_PREFER_TIMES_CONSTRAINT c);
KHE_TIME KhePreferTimesConstraintTime(
  KHE_PREFER_TIMES_CONSTRAINT c, int i);
```

After the instance is complete, but not before, function

```
KHE_TIME_GROUP KhePreferTimesConstraintDomain(
  KHE_PREFER_TIMES_CONSTRAINT c);
```

returns the domain of `c` as a single time group. If exactly one time group or one time was added, this time group will be that time group or the automatically created singleton time group for that time; otherwise it will be created by taking the union of everything added. This time group may be used like any other.

The points of application of prefer times constraints are events, and they can be added and visited individually:

```
void KhePreferTimesConstraintAddEvent(
  KHE_PREFER_TIMES_CONSTRAINT c, KHE_EVENT e);
int KhePreferTimesConstraintEventCount(
  KHE_PREFER_TIMES_CONSTRAINT c);
KHE_EVENT KhePreferTimesConstraintEvent(
  KHE_PREFER_TIMES_CONSTRAINT c, int i);
```

or in groups:

```
void KhePreferTimesConstraintAddEventGroup(
  KHE_PREFER_TIMES_CONSTRAINT c, KHE_EVENT_GROUP eg);
int KhePreferTimesConstraintEventGroupCount(
  KHE_PREFER_TIMES_CONSTRAINT c);
KHE_EVENT_GROUP KhePreferTimesConstraintEventGroup(
  KHE_PREFER_TIMES_CONSTRAINT c, int i);
```

The XML specification states that prefer times constraints skip events with preassigned times, whether those events are mentioned or not. Accordingly, although such events are added to constraints by the calls just given, the reverse links, from the events to the constraint, are added only to events that do not have preassigned times.

Function

```
void KhePreferTimesConstraintDebug(KHE_PREFER_TIMES_CONSTRAINT c,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of `c` onto `fp` with the given verbosity and indent, in the usual way.

The constraint density of the prefer times constraints of an instance (Section 3.3) is their number of points of application divided by the number of events without preassigned times.

### 3.7.7. Avoid split assignments constraints

An avoid split assignments constraint is created and added to an instance by

```
bool KheAvoidSplitAssignmentsConstraintMake(KHE_INSTANCE ins, char *id,
  char *name, bool required, int weight, KHE_COST_FUNCTION cf,
  char *role, KHE_AVOID_SPLIT_ASSIGNMENTS_CONSTRAINT *c);
```

As usual, the attributes may be retrieved by upcasting to `KHE_CONSTRAINT` and calling the relevant operation on that type, except that to retrieve the `role` attribute the call is

```
char *KheAvoidSplitAssignmentsConstraintRole(
  KHE_AVOID_SPLIT_ASSIGNMENTS_CONSTRAINT c);
```

The `role` attribute may be `NULL`.

The handling of the points of application of an avoid split assignments constraint is somewhat complex, because one point of application is fundamentally a set of event resources (the XML file identifies each set by an event group and a role), so that the points of application overall form a set of sets of event resources. We will first explain how to add these points of application when reading an XML file, and then how to do it directly.

When reading an XML file, a non-`NULL` `role` is passed, and then each event group is added in the usual way. To add an event group and to visit the event groups, the calls are

```
bool KheAvoidSplitAssignmentsConstraintAddEventGroup(
  KHE_AVOID_SPLIT_ASSIGNMENTS_CONSTRAINT c, KHE_EVENT_GROUP eg,
  KHE_EVENT *problem_event);
int KheAvoidSplitAssignmentsConstraintEventGroupCount(
  KHE_AVOID_SPLIT_ASSIGNMENTS_CONSTRAINT c);
KHE_EVENT_GROUP KheAvoidSplitAssignmentsConstraintEventGroup(
  KHE_AVOID_SPLIT_ASSIGNMENTS_CONSTRAINT c, int i);
```

Behind the scenes, the appropriate event resources are retrieved from the events of each event group and added automatically, so that nothing further needs to be done. A `false` result returned by `KheAvoidSplitAssignmentsConstraintAddEventGroup` indicates that one of the events of `eg` does not contain an event resource with the required non-`NULL` `role`. In this case, `*problem_event` will contain the first event of `eg` with this problem on return.

When the instance is not derived from an XML file it may be more convenient to add event resources directly. For the sake of this case, `role` may be `NULL`, and the `eg` parameter of `KheAvoidSplitAssignmentsConstraintAddEventGroup` may also be `NULL`. If either is `NULL`, event resources are not added automatically.

To add event resources manually, and to visit event resources (whether added automatically or manually), the calls are

```
void KheAvoidSplitAssignmentsConstraintAddEventResource(
  KHE_AVOID_SPLIT_ASSIGNMENTS_CONSTRAINT c, int eg_index,
  KHE_EVENT_RESOURCE er);
int KheAvoidSplitAssignmentsConstraintEventResourceCount(
  KHE_AVOID_SPLIT_ASSIGNMENTS_CONSTRAINT c, int eg_index);
KHE_EVENT_RESOURCE KheAvoidSplitAssignmentsConstraintEventResource(
  KHE_AVOID_SPLIT_ASSIGNMENTS_CONSTRAINT c, int eg_index, int er_index);
```

These functions add an event resource to the `eg_index`'th point of application of `c`, return the number of event resources at that point, and return the `er_index`'th event resource at that point. They define the required set of sets of event resources.

Usually, constraints are added to the instance and to the entities they apply to. For avoid split assignments constraints this would mean adding the constraint to the instance and the event groups. This is done, but, for convenience, each avoid split assignments constaint is also added to each of its event resources.

Function

```
void KheAvoidSplitAssignmentsConstraintDebug(
  KHE_AVOID_SPLIT_ASSIGNMENTS_CONSTRAINT c,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of `c` onto `fp` with the given verbosity and indent, in the usual way.

The constraint density (Section 3.3) is the number of event resources in all points of application divided by the number of event resources without preassigned resources.

### 3.7.8. Spread events constraints

A spread events constraint is created and added to an instance by

```
bool KheSpreadEventsConstraintMake(KHE_INSTANCE ins, char *id,
  char *name, bool required, int weight, KHE_COST_FUNCTION cf,
  KHE_TIME_SPREAD ts, KHE_SPREAD_EVENTS_CONSTRAINT *c);
```

where type `KHE_TIME_SPREAD` is explained below. Most of the attributes may be retrieved by upcasting to `KHE_CONSTRAINT` and calling the relevant operation on that type. The exception is

```
KHE_TIME_SPREAD KheSpreadEventsConstraintTimeSpread(
  KHE_SPREAD_EVENTS_CONSTRAINT c);
```

which returns the time spread. Type `KHE_TIME_SPREAD` is an object which describes the time groups that the constraint requires the event group to spread through, and the limits on the number of events that may touch each time group. Time spread objects are immutable, and may be shared among any number of constraints. To create a time spread object, call

```
KHE_TIME_SPREAD KheTimeSpreadMake(KHE_INSTANCE ins);
```

Initially this has no time groups. To add them, call

```
void KheTimeSpreadAddLimitedTimeGroup(KHE_TIME_SPREAD ts,
  KHE_LIMITED_TIME_GROUP ltg);
```

repeatedly. To retrieve the limited time groups of a time spread, call

```
int KheTimeSpreadLimitedTimeGroupCount(KHE_TIME_SPREAD lts);
KHE_LIMITED_TIME_GROUP KheTimeSpreadLimitedTimeGroup(
  KHE_TIME_SPREAD lts, int i);
```

An object of type `KHE_LIMITED_TIME_GROUP` contains what one element of a time spread needs: a time group plus a minimum and maximum number of events. It may be created by calling

```
KHE_LIMITED_TIME_GROUP KheLimitedTimeGroupMake(KHE_TIME_GROUP tg,
  int minimum, int maximum);
```

and functions

```
KHE_TIME_GROUP KheLimitedTimeGroupTimeGroup(KHE_LIMITED_TIME_GROUP ltg);
int KheLimitedTimeGroupMinimum(KHE_LIMITED_TIME_GROUP ltg);
int KheLimitedTimeGroupMaximum(KHE_LIMITED_TIME_GROUP ltg);
```

retrieve its attributes.

Two other operations on time spreads, available only after the instance is complete, provide information that may be useful to solvers:

```
bool KheTimeSpreadTimeGroupsDisjoint(KHE_TIME_SPREAD ts);
bool KheTimeSpreadCoversWholeCycle(KHE_TIME_SPREAD ts);
```

`KheTimeSpreadTimeGroupsDisjoint` returns `true` when the time groups of `ts`'s limited time groups are pairwise disjoint. `KheTimeSpreadCoversWholeCycle` returns `true` when every time of the cycle appears in at least one of the time groups of `ts`'s limited time groups.

Spread events apply to event groups; the operations for adding and visiting them are

```
void KheSpreadEventsConstraintAddEventGroup(
  KHE_SPREAD_EVENTS_CONSTRAINT c, KHE_EVENT_GROUP eg);
int KheSpreadEventsConstraintEventGroupCount(
  KHE_SPREAD_EVENTS_CONSTRAINT c);
KHE_EVENT_GROUP KheSpreadEventsConstraintEventGroup(
  KHE_SPREAD_EVENTS_CONSTRAINT c, int i);
```

as usual.

Function

```
void KheSpreadEventsConstraintDebug(KHE_SPREAD_EVENTS_CONSTRAINT c,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of `c` onto `fp` with the given verbosity and indent, in the usual way.

The constraint density of the spread events constraints of an instance (Section 3.3) is the number of events in their points of application, divided by the number of events.

### 3.7.9. Link events constraints

A link events constraint is created and added to an instance by

```
bool KheLinkEventsConstraintMake(KHE_INSTANCE ins, char *id,
  char *name, bool required, int weight, KHE_COST_FUNCTION cf,
  KHE_LINK_EVENTS_CONSTRAINT *c);
```

Most of the attributes may be retrieved by upcasting to `KHE_CONSTRAINT` and calling the relevant operation on that type. One point of application of a link events constraint is an event group; one constraint may contain any number of these. The operations for adding them are

```
void KheLinkEventsConstraintAddEventGroup(KHE_LINK_EVENTS_CONSTRAINT c,
  KHE_EVENT_GROUP eg);
int KheLinkEventsConstraintEventGroupCount(KHE_LINK_EVENTS_CONSTRAINT c);
KHE_EVENT_GROUP KheLinkEventsConstraintEventGroup(
  KHE_LINK_EVENTS_CONSTRAINT c, int i);
```

as usual.

Function

```
void KheLinkEventsConstraintDebug(KHE_LINK_EVENTS_CONSTRAINT c,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of c onto fp with the given verbosity and indent, in the usual way.

The constraint density of the link events constraints of an instance (Section 3.3) is the number of events in their points of application, divided by the number of events.

### 3.7.10. Order events constraints

An order events constraint is created and added to an instance by

```
bool KheOrderEventsConstraintMake(KHE_INSTANCE ins, char *id,
  char *name, bool required, int weight, KHE_COST_FUNCTION cf,
  KHE_ORDER_EVENTS_CONSTRAINT *c);
```

Most of the attributes may be retrieved by upcasting to `KHE_CONSTRAINT` and calling the relevant operation on that type.

One point of application of an order events constraint is a pair of instance events, together with integer minimum and maximum separations. To add one point of application, call

```
void KheOrderEventsConstraintAddEventPair(KHE_ORDER_EVENTS_CONSTRAINT c,
  KHE_EVENT first_event, KHE_EVENT second_event, int min_separation,
  int max_separation);
```

Both `min_separation` and `max_separation` must be non-negative. Infinity, the default value of `max_separation` in the XML format, is implemented by passing `INT_MAX`.

To retrieve the number of points of application and the attributes of each, call

```
int KheOrderEventsConstraintEventPairCount(
  KHE_ORDER_EVENTS_CONSTRAINT c);
KHE_EVENT KheOrderEventsConstraintFirstEvent(
  KHE_ORDER_EVENTS_CONSTRAINT c, int i);
KHE_EVENT KheOrderEventsConstraintSecondEvent(
  KHE_ORDER_EVENTS_CONSTRAINT c, int i);
int KheOrderEventsConstraintMinSeparation(
  KHE_ORDER_EVENTS_CONSTRAINT c, int i);
int KheOrderEventsConstraintMaxSeparation(
  KHE_ORDER_EVENTS_CONSTRAINT c, int i);
```

in the usual way. The value of `KheOrderEventsConstraintEventPairCount(c)` is the same as the value of `KheConstraintAppliesToCount((KHE_CONSTRAINT) c)`.

Function

```
void KheOrderEventsConstraintDebug(KHE_ORDER_EVENTS_CONSTRAINT c,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of c onto fp with the given verbosity and indent, in the usual way.

The constraint density of the order events constraints of an instance (Section 3.3) is their number of points of application divided by the number of events.

### 3.7.11. Avoid clashes constraints

An avoid clashes constraint is created and added to an instance by

```
bool KheAvoidClashesConstraintMake(KHE_INSTANCE ins, char *id,
  char *name, bool required, int weight, KHE_COST_FUNCTION cf,
  KHE_AVOID_CLASHES_CONSTRAINT *c);
```

as usual. The attributes may be retrieved by upcasting to `KHE_CONSTRAINT` and calling the relevant operation on that type.

Avoid clashes constraints apply to resource groups and resources. To add and visit resource groups, the operations are

```
void KheAvoidClashesConstraintAddResourceGroup(
  KHE_AVOID_CLASHES_CONSTRAINT c, KHE_RESOURCE_GROUP rg);
int KheAvoidClashesConstraintResourceGroupCount(
  KHE_AVOID_CLASHES_CONSTRAINT c);
KHE_RESOURCE_GROUP KheAvoidClashesConstraintResourceGroup(
  KHE_AVOID_CLASHES_CONSTRAINT c, int i);
```

while to add and visit resources the operations are

```
void KheAvoidClashesConstraintAddResource(
  KHE_AVOID_CLASHES_CONSTRAINT c, KHE_RESOURCE r);
int KheAvoidClashesConstraintResourceCount(
  KHE_AVOID_CLASHES_CONSTRAINT c);
KHE_RESOURCE KheAvoidClashesConstraintResource(
  KHE_AVOID_CLASHES_CONSTRAINT c, int i);
```

These all work in the usual way. There is also

```
int KheAvoidClashesConstraintResourceOfTypeCount(
  KHE_AVOID_CLASHES_CONSTRAINT c, KHE_RESOURCE_TYPE rt);
```

which returns the number of resources of type `rt` which are points of application of `c`. In practice the resources of one constraint always have the same type, but the rules do not guarantee this.

Function

```
void KheAvoidClashesConstraintDebug(KHE_AVOID_CLASHES_CONSTRAINT c,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of `c` onto `fp` with the given verbosity and indent, in the usual way.

The constraint density of the avoid clashes constraints of an instance (Section 3.3) is the number of points of application divided by the number of resources.

### 3.7.12. Avoid unavailable times constraints

An avoid unavailable times constraint is created and added to an instance by

```
bool KheAvoidUnavailableTimesConstraintMake(KHE_INSTANCE ins, char *id,
  char *name, bool required, int weight, KHE_COST_FUNCTION cf,
  KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT *c);
```

in the usual way. To add the resource groups and resources defining the points of application, and to visit them, call

```
void KheAvoidUnavailableTimesConstraintAddResourceGroup(
  KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT c, KHE_RESOURCE_GROUP rg);
int KheAvoidUnavailableTimesConstraintResourceGroupCount(
  KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT c);
KHE_RESOURCE_GROUP KheAvoidUnavailableTimesConstraintResourceGroup(
  KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT c, int i);
```

for resource groups and

```
void KheAvoidUnavailableTimesConstraintAddResource(
  KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT c, KHE_RESOURCE r);
int KheAvoidUnavailableTimesConstraintResourceCount(
  KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT c);
KHE_RESOURCE KheAvoidUnavailableTimesConstraintResource(
  KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT c, int i);
```

for individual resources. The XML format allows the unavailable times themselves to be defined by both time groups and times. To add time groups and visit them, call

```
void KheAvoidUnavailableTimesConstraintAddTimeGroup(
  KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT c, KHE_TIME_GROUP tg);
int KheAvoidUnavailableTimesConstraintTimeGroupCount(
  KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT c);
KHE_TIME_GROUP KheAvoidUnavailableTimesConstraintTimeGroup(
  KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT c, int i);
```

To add individual times and visit them, call

```
void KheAvoidUnavailableTimesConstraintAddTime(
  KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT c, KHE_TIME t);
int KheAvoidUnavailableTimesConstraintTimeCount(
  KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT c);
KHE_TIME KheAvoidUnavailableTimesConstraintTime(
  KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT c, int i);
```

These functions all work in the usual way. Function

```
KHE_TIME_GROUP KheAvoidUnavailableTimesConstraintUnavailableTimes(
  KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT c);
```

returns a time group containing the union of the time groups and times of `c`, and

```
KHE_TIME_GROUP KheAvoidUnavailableTimesConstraintAvailableTimes(
  KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT c);
```

returns a time group containing the complement of those times. Both functions may be called only after construction of the instance is complete. The time groups they return will usually not have neighbourhoods (Section 3.4.1). This is not likely to cause problems.

Function

```
void KheAvoidUnavailableTimesConstraintDebug(
  KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT c,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of `c` onto `fp` with the given verbosity and indent, in the usual way.

The constraint density of the avoid unavailable times constraints of an instance (Section 3.3) is the number of points of application divided by the number of resources.

### 3.7.13. Limit idle times constraints

A limit idle times constraint is created and added to an instance by

```
bool KheLimitIdleTimesConstraintMake(KHE_INSTANCE ins, char *id,
  char *name, bool required, int weight, KHE_COST_FUNCTION cf,
  int minimum, int maximum, KHE_LIMIT_IDLE_TIMES_CONSTRAINT *c);
```

Most of the attributes may be retrieved by upcasting to `KHE_CONSTRAINT` and calling the relevant operation on that type; the exceptions are

```
int KheLimitIdleTimesConstraintMinimum(KHE_LIMIT_IDLE_TIMES_CONSTRAINT c);
int KheLimitIdleTimesConstraintMaximum(KHE_LIMIT_IDLE_TIMES_CONSTRAINT c);
```

which are specific to this kind of constraint.

A limit idle times constraint requires time groups, which are added and visited by calling

```
void KheLimitIdleTimesConstraintAddTimeGroup(
  KHE_LIMIT_IDLE_TIMES_CONSTRAINT c, KHE_TIME_GROUP tg);
int KheLimitIdleTimesConstraintTimeGroupCount(
  KHE_LIMIT_IDLE_TIMES_CONSTRAINT c);
KHE_TIME_GROUP KheLimitIdleTimesConstraintTimeGroup(
  KHE_LIMIT_IDLE_TIMES_CONSTRAINT c, int i);
```

After the instance ends, the following queries are available:

```
bool KheLimitIdleTimesConstraintTimeGroupsDisjoint(
  KHE_LIMIT_IDLE_TIMES_CONSTRAINT c);
bool KheLimitIdleTimesConstraintTimeGroupsCoverWholeCycle(
  KHE_LIMIT_IDLE_TIMES_CONSTRAINT c);
```

They return `true` when the time groups of `c` are pairwise disjoint, and when their union covers the whole cycle.

A limit idle times constraint also requires the resource groups and resources which define its points of application. Resource groups are added and visited by calling

```
void KheLimitIdleTimesConstraintAddResourceGroup(
  KHE_LIMIT_IDLE_TIMES_CONSTRAINT c, KHE_RESOURCE_GROUP rg);
int KheLimitIdleTimesConstraintResourceGroupCount(
  KHE_LIMIT_IDLE_TIMES_CONSTRAINT c);
KHE_RESOURCE_GROUP KheLimitIdleTimesConstraintResourceGroup(
  KHE_LIMIT_IDLE_TIMES_CONSTRAINT c, int i);
```

and individual resources are added and visited by calling

```
void KheLimitIdleTimesConstraintAddResource(
  KHE_LIMIT_IDLE_TIMES_CONSTRAINT c, KHE_RESOURCE r);
int KheLimitIdleTimesConstraintResourceCount(
  KHE_LIMIT_IDLE_TIMES_CONSTRAINT c);
KHE_RESOURCE KheLimitIdleTimesConstraintResource(
  KHE_LIMIT_IDLE_TIMES_CONSTRAINT c, int i);
```

in the usual way.

Function

```
void KheLimitIdleTimesConstraintDebug(KHE_LIMIT_IDLE_TIMES_CONSTRAINT c,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of `c` onto `fp` with the given verbosity and indent, in the usual way.

The constraint density of the limit idle times constraints of an instance (Section 3.3) is the number of points of application divided by the number of resources.

### 3.7.14. Cluster busy times constraints

A cluster busy times constraint is created and added to an instance by

```
bool KheClusterBusyTimesConstraintMake(KHE_INSTANCE ins, char *id,
  char *name, bool required, int weight, KHE_COST_FUNCTION cf,
  KHE_TIME_GROUP applies_to_tg, int minimum, int maximum,
  bool allow_zero, KHE_CLUSTER_BUSY_TIMES_CONSTRAINT *c);
```

Most of the attributes may be retrieved by upcasting to `KHE_CONSTRAINT` and calling the relevant operation on that type; the exceptions are

```
KHE_TIME_GROUP KheClusterBusyTimesConstraintAppliesToTimeGroup(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c);
int KheClusterBusyTimesConstraintMinimum(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c);
int KheClusterBusyTimesConstraintMaximum(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c);
bool KheClusterBusyTimesConstraintAllowZero(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c);
```

which are specific to this kind of constraint. In the high school timetabling model, `applies_to_tg` must be `NULL` and `allow_zero` must be `false`. There is also

```
bool KheClusterBusyTimesConstraintLimitBusyRecode(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c);
```

It returns `true` when `c` is a recoded limit busy times constraint, for which see Section 3.7.15.

After the instance is complete, functions

```
int KheClusterBusyTimesConstraintAppliesToOffsetCount(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c);
int KheClusterBusyTimesConstraintAppliesToOffset(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c, int i);
```

may be used to visit the *applies-to offsets*, or just *offsets*, of `c`. If `applies_to_tg` is `NULL`, there is one offset, with value 0. If `applies_to_tg` is empty, there are no offsets. Otherwise, let `t0` be the first time in `applies_to_tg`. There is one offset for each time `ti` in `applies_to_tg`, including `t0`, such that when `KheTimeIndex(ti) - KheTimeIndex(t0)` is added to the index of each time in `c`, the result is a legal time index.

A cluster busy times constraint requires time groups, which are added and visited by

```
void KheClusterBusyTimesConstraintAddTimeGroup(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c, KHE_TIME_GROUP tg, KHE_POLARITY po);
int KheClusterBusyTimesConstraintTimeGroupCount(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c);
KHE_TIME_GROUP KheClusterBusyTimesConstraintTimeGroup(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c, int i, int offset, KHE_POLARITY *po);
```

where type `KHE_POLARITY` is

```
typedef enum {
  KHE_NEGATIVE,
  KHE_POSITIVE
} KHE_POLARITY;
```

In the high school model, the polarity must be `KHE_POSITIVE`. When visiting, to get the original time groups, set `offset` to `0`; to get the time groups being monitored by monitor `m`, set it to `KheClusterBusyTimesMonitorOffset(m)`.

Convenience functions

```
bool KheClusterBusyTimesConstraintAllPositive(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c);
bool KheClusterBusyTimesConstraintAllNegative(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c);
```

return `true` when all of the time groups added so far have polarity `KHE_POSITIVE`, or all have polarity `KHE_NEGATIVE`. In real instances one of these two functions will usually return `true`. In nurse rostering the main exceptions are constraints that implement unwanted patterns. Also,

```
bool KheClusterBusyTimesConstraintTimeGroupsDisjoint(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c);
bool KheClusterBusyTimesConstraintTimeGroupsCoverWholeCycle(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c);
```

return `true` when the time groups of `c` are pairwise disjoint, and when their union covers the whole cycle. These functions should only be called after the instance is complete. Also,

```
bool KheClusterBusyTimesConstraintRange(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c, int offset,
  KHE_TIME *first_time, KHE_TIME *last_time);
```

sets `*first_time` and `*last_time` to the chronologically first and last times monitored by `c` at `offset`, and returns `true`. Here `offset` must be a legal offset (a value returned by `KheClusterBusyTimesConstraintAppliesToOffset` above). In the unlikely event of `c` having no time groups, the function returns `false` with `*first_time` and `*last_time` set to `NULL`.

To add the resource groups and resources defining the points of application, use

```
void KheClusterBusyTimesConstraintAddResourceGroup(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c, KHE_RESOURCE_GROUP rg);
int KheClusterBusyTimesConstraintResourceGroupCount(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c);
KHE_RESOURCE_GROUP KheClusterBusyTimesConstraintResourceGroup(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c, int i);
```

for resource groups and

```
void KheClusterBusyTimesConstraintAddResource(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c, KHE_RESOURCE r);
int KheClusterBusyTimesConstraintResourceCount(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c);
KHE_RESOURCE KheClusterBusyTimesConstraintResource(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c, int i);
```

for individual resources. There is also

```
int KheClusterBusyTimesConstraintResourceOfTypeCount(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c, KHE_RESOURCE_TYPE rt);
```

which returns the number of resources of type `rt` which are points of application of `c`. In practice the resources of one constraint always have the same type, but the rules do not guarantee this.

For employee scheduling only, to add and retrieve a value representing the number of time groups preceding this constraint, called $a_i$ in Jeff Kingston's paper on history [10], call

```
void KheClusterBusyTimesConstraintAddHistoryBefore(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c, int val);
int KheClusterBusyTimesConstraintHistoryBefore(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c);
```

When `KheClusterBusyTimesConstraintAddHistoryBefore` is not called, the value is 0.

For employee scheduling only, to add and retrieve a value representing the number of time groups following this constraint, called $c_i$ in the history paper, call

```
void KheClusterBusyTimesConstraintAddHistoryAfter(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c, int val);
int KheClusterBusyTimesConstraintHistoryAfter(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c);
```

When `KheClusterBusyTimesConstraintAddHistoryAfter` is not called, the value is 0.

For employee scheduling only, to add and retrieve a value for one resource representing the number of active time groups preceding this constraint, called $x_i$ in the history paper, call

```
void KheClusterBusyTimesConstraintAddHistory(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c, KHE_RESOURCE r, int val);
int KheClusterBusyTimesConstraintHistory(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c, KHE_RESOURCE r);
```

When `KheClusterBusyTimesConstraintAddHistory` is not called for some `r`, the value is 0.

KHE does not check that resources in history calls are points of application of `c`. It aborts if any conflicting history values are received.

Function

```
void KheClusterBusyTimesConstraintDebug(
  KHE_CLUSTER_BUSY_TIMES_CONSTRAINT c,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of `c` onto `fp` with the given verbosity and indent, in the usual way.

The number of points of application of a cluster busy times constraint `c` is its total number of resources multiplied by `KheClusterBusyTimesConstraintAppliesToOffsetCount(c)`. The constraint density of the cluster busy times constraints of an instance (Section 3.3) is their total number of points of application divided by the number of resources in the instance.

Suppose that a cluster busy times constraint requires some resource to be busy for at most 20 out of 28 days. This is the same as requiring the resource to be free for at least 8 out of the 28 days. Here is a general statement of what is going on here, along with a proof.

**Theorem**. Suppose cluster busy times constraint $c$ has minimum limit $a$, maximum limit $b$, and $n$ time groups. Suppose cluster busy times constraint $c'$ has minimum limit $n - b$, maximum limit $n - a$, the same hardness, cost function, and weight as $c$, and the same time groups as $c$, only with their polarities reversed. If $c$ has history values $a_i$, $x_i$ (one for each resource), and $c_i$, suppose that $c'$ has the same $a_i$ and $c_i$ values as $c$, but that each of its $x_i$ values is changed to $a_i - x_i$. Then in every solution, $c$ and $c'$ have equal cost.

**Proof**. The proof depends on the fact that when a time group's polarity is reversed, so is its activity. If positive time group $g$ is active, it is busy, so one of its times is busy. If $g$ is made negative, one of its times is still busy, so it is inactive. If positive time group $g$ is inactive, none of its times is busy. If $g$ is made negative, still none of its times is busy, so it is active. And so on.

Let $S$ be an arbitrary solution, and suppose that $c$ has $k$ active time groups in $S$. Then the deviation of $c$ is

$$d(c) = \max(0, a - k, k - b)$$

But $c'$ has $n - k$ active time groups in $S$, because the time groups are the same as in $c$ but their polarity, and hence their activity as we have seen, is reversed. So the deviation of $c'$ is

$$d(c') = \max(0, (n - b) - (n - k), (n - k) - (n - a))$$

which simplifies to $\max(0, k - b, a - k)$ which equals $d(c)$.

When history is present, there are $a_i$ time groups preceding the first time group but not explicitly represented, $x_i$ of which are active; and there are $c_i$ time groups following the last time group, again not explicitly represented, whose activity is unknown. When these times groups' polarities are reversed, there will still be $a_i$ time groups preceding the first time group, but now $a_i - x_i$ of them will be active; and there will still be $c_i$ time groups following the last time group, whose activity remains unknown.

### 3.7.15. Limit busy times constraints

A limit busy times constraint is created and added to an instance by

```
bool KheLimitBusyTimesConstraintMake(KHE_INSTANCE ins, char *id,
  char *name, bool required, int weight, KHE_COST_FUNCTION cf,
  KHE_TIME_GROUP applies_to_tg, int minimum, int maximum,
  bool allow_zero, KHE_LIMIT_BUSY_TIMES_CONSTRAINT *c);
```

Most of these attributes may be retrieved by upcasting to `KHE_CONSTRAINT` and calling the relevant operation on that type. The exceptions are

```
KHE_TIME_GROUP KheLimitBusyTimesConstraintAppliesToTimeGroup(
  KHE_LIMIT_BUSY_TIMES_CONSTRAINT c);
int KheLimitBusyTimesConstraintMinimum(
  KHE_LIMIT_BUSY_TIMES_CONSTRAINT c);
int KheLimitBusyTimesConstraintMaximum(
  KHE_LIMIT_BUSY_TIMES_CONSTRAINT c);
bool KheLimitBusyTimesConstraintAllowZero(
  KHE_LIMIT_BUSY_TIMES_CONSTRAINT c);
```

which are specific to this kind of constraint. In the high school timetabling model, `applies_to_tg` must be `NULL` and `allow_zero` must be `false`.

After the instance is complete, functions

```
int KheLimitBusyTimesConstraintAppliesToOffsetCount(
  KHE_LIMIT_BUSY_TIMES_CONSTRAINT c);
int KheLimitBusyTimesConstraintAppliesToOffset(
  KHE_LIMIT_BUSY_TIMES_CONSTRAINT c, int i);
```

may be used to visit the *applies-to offsets*, or just *offsets*, of `c`. If `applies_to_tg` is `NULL`, there is one offset, with value 0. If `applies_to_tg` is empty, there are no offsets. Otherwise, let `t0` be the first time in `applies_to_tg`. There is one offset for each time `ti` in `applies_to_tg`, including `t0`, such that when `KheTimeIndex(ti) - KheTimeIndex(t0)` is added to the index of each time in `c`, the result is a legal time index.

A limit busy times constraint requires time groups, which are added and visited by

```
void KheLimitBusyTimesConstraintAddTimeGroup(
  KHE_LIMIT_BUSY_TIMES_CONSTRAINT c, KHE_TIME_GROUP tg);
int KheLimitBusyTimesConstraintTimeGroupCount(
  KHE_LIMIT_BUSY_TIMES_CONSTRAINT c);
KHE_TIME_GROUP KheLimitBusyTimesConstraintTimeGroup(
  KHE_LIMIT_BUSY_TIMES_CONSTRAINT c, int offset, int i);
```

To get the original time groups, set `offset` to `0`; to get the time groups monitored by monitor `m`, set it to `KheLimitBusyTimesMonitorOffset(m)`.

After the instance is complete, these two functions may be called:

```
KHE_TIME_GROUP KheLimitBusyTimesConstraintDomain(
  KHE_LIMIT_BUSY_TIMES_CONSTRAINT c);
bool KheLimitBusyTimesConstraintLimitsWholeCycle(
  KHE_LIMIT_BUSY_TIMES_CONSTRAINT c);
```

`KheLimitBusyTimesConstraintDomain` returns the *domain* of `c`: the union of its time groups. It may be used like any time group, except that it may have no neighbourhood (Section 3.4.1). This function should probably not exist; it is irrelevant to solving, because the limits are applied to each time group separately. `KheLimitBusyTimesConstraintLimitsWholeCycle` returns `true` when `c` contains a time group equal to the whole cycle.

A limit busy times constraint also requires the resource groups and resources which define the points of application of the constraint. Resource groups are added and visited by calling

```
void KheLimitBusyTimesConstraintAddResourceGroup(
  KHE_LIMIT_BUSY_TIMES_CONSTRAINT c, KHE_RESOURCE_GROUP rg);
int KheLimitBusyTimesConstraintResourceGroupCount(
  KHE_LIMIT_BUSY_TIMES_CONSTRAINT c);
KHE_RESOURCE_GROUP KheLimitBusyTimesConstraintResourceGroup(
  KHE_LIMIT_BUSY_TIMES_CONSTRAINT c, int i);
```

and individual resources are added and visited by calling

```
void KheLimitBusyTimesConstraintAddResource(
  KHE_LIMIT_BUSY_TIMES_CONSTRAINT c, KHE_RESOURCE r);
int KheLimitBusyTimesConstraintResourceCount(
  KHE_LIMIT_BUSY_TIMES_CONSTRAINT c);
KHE_RESOURCE KheLimitBusyTimesConstraintResource(
  KHE_LIMIT_BUSY_TIMES_CONSTRAINT c, int i);
```

in the usual way. There is also

```
int KheLimitBusyTimesConstraintResourceOfTypeCount(
  KHE_LIMIT_BUSY_TIMES_CONSTRAINT c, KHE_RESOURCE_TYPE rt);
```

which returns the number of resources of type `rt` which are points of application of `c`. In practice the resources of one constraint always have the same type, but the rules do not guarantee this.

Function

```
void KheLimitBusyTimesConstraintDebug(KHE_LIMIT_BUSY_TIMES_CONSTRAINT c,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of `c` onto `fp` with the given verbosity and indent, in the usual way.

The number of points of application of a limit busy times constraint `c` is its total number of resources multiplied by `KheLimitBusyTimesConstraintAppliesToOffsetCount(c)`. The constraint density of the limit busy times constraints of an instance (Section 3.3) is their total number of points of application divided by the number of resources in the instance.

What happens, precisely, is this. For each time group of each limit busy times constraint that has a minimum limit, a cluster busy times constraint is added to the instance which has the exact same meaning as the limit busy times constraint does on that time group. (It has a singleton time group for each time of the time group, and the same limits and cost function.) This constraint appears on lists of constraints in the usual way, but if the instance is printed out later it is omitted from the print. Furthermore, when a solution object is created, monitors are created for the cluster busy times constraints but not for the original limit busy times constraints.

### 3.7.16. Limit workload constraints

A limit workload constraint is created and added to an instance by

```
bool KheLimitWorkloadConstraintMake(KHE_INSTANCE ins, char *id,
  char *name, bool required, int weight, KHE_COST_FUNCTION cf,
  KHE_TIME_GROUP applies_to_tg, int minimum, int maximum,
  bool allow_zero, KHE_LIMIT_WORKLOAD_CONSTRAINT *c);
```

Most of these attributes may be retrieved by upcasting to `KHE_CONSTRAINT` and calling the relevant operation on that type. The exceptions are

```
KHE_TIME_GROUP KheLimitWorkloadConstraintAppliesToTimeGroup(
  KHE_LIMIT_WORKLOAD_CONSTRAINT c);
int KheLimitWorkloadConstraintMinimum(KHE_LIMIT_WORKLOAD_CONSTRAINT c);
int KheLimitWorkloadConstraintMaximum(KHE_LIMIT_WORKLOAD_CONSTRAINT c);
bool KheLimitWorkloadConstraintAllowZero(
  KHE_LIMIT_WORKLOAD_CONSTRAINT c);
```

which are specific to this kind of constraint. In the high school timetabling model, `applies_to_tg` must be `NULL` and `allow_zero` must be `false`.

After the instance is complete, functions

```
int KheLimitWorkloadConstraintAppliesToOffsetCount(
  KHE_LIMIT_WORKLOAD_CONSTRAINT c);
int KheLimitWorkloadConstraintAppliesToOffset(
  KHE_LIMIT_WORKLOAD_CONSTRAINT c, int i);
```

may be used to visit the *applies-to offsets*, or just *offsets*, of `c`. If `applies_to_tg` is `NULL`, there is one offset, with value 0. If `applies_to_tg` is empty, there are no offsets. Otherwise, let `t0` be the first time in `applies_to_tg`. There is one offset for each time `ti` in `applies_to_tg`, including `t0`, such that when `KheTimeIndex(ti) - KheTimeIndex(t0)` is added to the index of each time in `c`, the result is a legal time index.

A limit workload constraint has optional time groups (not permitted in the high school model), which are added and visited by

```
void KheLimitWorkloadConstraintAddTimeGroup(
  KHE_LIMIT_WORKLOAD_CONSTRAINT c, KHE_TIME_GROUP tg);
int KheLimitWorkloadConstraintTimeGroupCount(
  KHE_LIMIT_WORKLOAD_CONSTRAINT c);
KHE_TIME_GROUP KheLimitWorkloadConstraintTimeGroup(
  KHE_LIMIT_WORKLOAD_CONSTRAINT c, int offset, int i);
```

To get the original time groups, set `offset` to `0`; to get the time groups monitored by monitor `m`, set it to `KheLimitWorkloadMonitorOffset(m)`. Adding no time groups is semantically equivalent to adding one time group holding all the times of the instance. So when no time groups are added, after the instance is finalized, `KheLimitWorkloadConstraintTimeGroupCount(c)` is `1`, and `KheLimitWorkloadConstraintTimeGroup(c, 0, 0)` is `KheInstanceFullTimeGroup(ins)`. Nevertheless, in this special case `KheArchiveWrite` does not write any time groups.

Also after the instance is complete, these two functions may be called:

```
KHE_TIME_GROUP KheLimitWorkloadConstraintDomain(
  KHE_LIMIT_WORKLOAD_CONSTRAINT c);
bool KheLimitWorkloadConstraintLimitsWholeCycle(
  KHE_LIMIT_WORKLOAD_CONSTRAINT c);
```

`KheLimitWorkloadConstraintDomain` returns the *domain* of `c`: the union of its time groups. If no time groups were added, it returns the set of all the times in the instance. This time group may be used like any other, except that it might have no neighbourhood (Section 3.4.1). This function should probably not exist; it is irrelevant to solving, because the limits are applied to each time group separately. `KheLimitWorkloadConstraintLimitsWholeCycle` returns `true` when `c` contains a time group equal to the whole cycle.

A limit workload constraint also requires the resource groups and resources which define the points of application of the constraint. Resource groups are added and visited by calling

```
void KheLimitWorkloadConstraintAddResourceGroup(
  KHE_LIMIT_WORKLOAD_CONSTRAINT c, KHE_RESOURCE_GROUP rg);
int KheLimitWorkloadConstraintResourceGroupCount(
  KHE_LIMIT_WORKLOAD_CONSTRAINT c);
KHE_RESOURCE_GROUP KheLimitWorkloadConstraintResourceGroup(
  KHE_LIMIT_WORKLOAD_CONSTRAINT c, int i);
```

and individual resources are added and visited by calling

```
void KheLimitWorkloadConstraintAddResource(
  KHE_LIMIT_WORKLOAD_CONSTRAINT c, KHE_RESOURCE r);
int KheLimitWorkloadConstraintResourceCount(
  KHE_LIMIT_WORKLOAD_CONSTRAINT c);
KHE_RESOURCE KheLimitWorkloadConstraintResource(
  KHE_LIMIT_WORKLOAD_CONSTRAINT c, int i);
```

in the usual way.

Function

```
void KheLimitWorkloadConstraintDebug(KHE_LIMIT_WORKLOAD_CONSTRAINT c,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of `c` onto `fp` with the given verbosity and indent, in the usual way.

The number of points of application of a limit workload constraint `c` is its total number of resources multiplied by `KheLimitWorkloadConstraintAppliesToOffsetCount(c)`. The constraint density of the limit workload constraints of an instance (Section 3.3) is their total number of points of application divided by the number of resources in the instance.

### 3.7.17. Limit active intervals constraints

Limit active intervals constraints are allowed only with `KHE_MODEL_EMPLOYEE_SCHEDULE`. Although they have their own semantics, syntactically they are almost the same as cluster busy times constraints: the only differences are the change of name and the absence of `AllowZero`.

A limit active intervals constraint is created and added to an instance by

```
bool KheLimitActiveIntervalsConstraintMake(KHE_INSTANCE ins, char *id,
  char *name, bool required, int weight, KHE_COST_FUNCTION cf,
  KHE_TIME_GROUP applies_to_tg, int minimum, int maximum,
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT *c);
```

Most of the attributes may be retrieved by upcasting to `KHE_CONSTRAINT` and calling the relevant operation on that type; the exceptions are

```
KHE_TIME_GROUP KheLimitActiveIntervalsConstraintAppliesToTimeGroup(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c);
int KheLimitActiveIntervalsConstraintMinimum(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c);
int KheLimitActiveIntervalsConstraintMaximum(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c);
```

which are specific to this kind of constraint.

After the instance is complete, functions

```
int KheLimitActiveIntervalsConstraintAppliesToOffsetCount(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c);
int KheLimitActiveIntervalsConstraintAppliesToOffset(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c, int i);
```

may be used to visit the *applies-to offsets*, or just *offsets*, of c. If applies_to_tg is NULL, there is one offset, with value 0. If applies_to_tg is empty, there are no offsets. Otherwise, let t0 be the first time in applies_to_tg. There is one offset for each time ti in applies_to_tg, including t0, such that when KheTimeIndex(ti) - KheTimeIndex(t0) is added to the index of each time in c, the result is a legal time index.

A limit active intervals constraint requires time groups, which are added and visited by

```
void KheLimitActiveIntervalsConstraintAddTimeGroup(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c, KHE_TIME_GROUP tg,
  KHE_POLARITY po);
int KheLimitActiveIntervalsConstraintTimeGroupCount(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c);
KHE_TIME_GROUP KheLimitActiveIntervalsConstraintTimeGroup(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c, int i, int offset,
  KHE_POLARITY *po);
```

where type KHE_POLARITY is as for cluster busy times constraints. When visiting, to get the original time groups, set offset to 0; to get the time groups being monitored by monitor m, set it to KheLimitActiveIntervalsMonitorOffset(m).

Convenience functions

```
bool KheLimitActiveIntervalsConstraintAllPositive(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c);
bool KheLimitActiveIntervalsConstraintAllNegative(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c);
```

return true when all of the time groups added so far have polarity KHE_POSITIVE, or all have polarity KHE_NEGATIVE. In real instances it is almost certain that one of these will return true.

To add the resource groups and resources defining the points of application, use

```
void KheLimitActiveIntervalsConstraintAddResourceGroup(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c, KHE_RESOURCE_GROUP rg);
int KheLimitActiveIntervalsConstraintResourceGroupCount(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c);
KHE_RESOURCE_GROUP KheLimitActiveIntervalsConstraintResourceGroup(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c, int i);
```

for resource groups and

```
void KheLimitActiveIntervalsConstraintAddResource(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c, KHE_RESOURCE r);
int KheLimitActiveIntervalsConstraintResourceCount(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c);
KHE_RESOURCE KheLimitActiveIntervalsConstraintResource(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c, int i);
```

for individual resources. There is also

```
int KheLimitActiveIntervalsConstraintResourceOfTypeCount(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c, KHE_RESOURCE_TYPE rt);
```

which returns the number of resources of type `rt` which are points of application of `c`. In practice the resources of one constraint always have the same type, but the rules do not guarantee this.

To add and retrieve a value representing the number of time groups preceding this constraint, called $a_i$ in Jeff Kingston's paper on history [10], call

```
void KheLimitActiveIntervalsConstraintAddHistoryBefore(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c, int val);
int KheLimitActiveIntervalsConstraintHistoryBefore(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c);
```

When `KheLimitActiveIntervalsConstraintAddHistoryBefore` is not called, the value is 0.

To add and retrieve a value representing the number of time groups following this constraint, called $c_i$ in the history paper, call

```
void KheLimitActiveIntervalsConstraintAddHistoryAfter(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c, int val);
int KheLimitActiveIntervalsConstraintHistoryAfter(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c);
```

When `KheLimitActiveIntervalsConstraintAddHistoryAfter` is not called, the value is 0.

To add and retrieve a value for one resource representing the number of active time groups preceding this constraint, called $x_i$ in the history paper, call

```
void KheLimitActiveIntervalsConstraintAddHistory(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c, KHE_RESOURCE r, int val);
int KheLimitActiveIntervalsConstraintHistory(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c, KHE_RESOURCE r);
```

When `KheLimitActiveIntervalsConstraintAddHistory` is not called for `r`, the value is 0.

KHE does not check that resources in history calls are points of application of `c`. It aborts if a history value is given twice in the same constraint.

Function

```
void KheLimitActiveIntervalsConstraintDebug(
  KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT c,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of `c` onto `fp` with the given verbosity and indent, in the usual way.

The number of points of application of a limit active intervals constraint `c` is its number of resources times `KheLimitActiveIntervalsConstraintAppliesToOffsetCount(c)`. The constraint density of the limit active intervals constraints of an instance (Section 3.3) is their total number of points of application divided by the number of resources in the instance.

### 3.7.18. Limit resources constraints

Limit resources constraints are allowed only with `KHE_MODEL_EMPLOYEE_SCHEDULE`.

A limit resources constraint is created and added to an instance by

```
bool KheLimitResourcesConstraintMake(KHE_INSTANCE ins, char *id,
  char *name, bool required, int weight, KHE_COST_FUNCTION cf,
  int minimum, int maximum, KHE_LIMIT_RESOURCES_CONSTRAINT *c);
```

Most of these attributes may be retrieved by upcasting to `KHE_CONSTRAINT` and calling the relevant operation on that type; the exceptions are

```
int KheLimitResourcesConstraintMinimum(KHE_LIMIT_RESOURCES_CONSTRAINT c);
int KheLimitResourcesConstraintMaximum(KHE_LIMIT_RESOURCES_CONSTRAINT c);
```

which are specific to this kind of constraint. These values are optional in XESTT files; a missing minimum is represented by 0, and a missing maximum is represented by `INT_MAX`.

To add and visit the resource groups and resources required by this constraint, call

```
bool KheLimitResourcesConstraintAddResourceGroup(
  KHE_LIMIT_RESOURCES_CONSTRAINT c, KHE_RESOURCE_GROUP rg);
int KheLimitResourcesConstraintResourceGroupCount(
  KHE_LIMIT_RESOURCES_CONSTRAINT c);
KHE_RESOURCE_GROUP KheLimitResourcesConstraintResourceGroup(
  KHE_LIMIT_RESOURCES_CONSTRAINT c, int i);
```

and

```
bool KheLimitResourcesConstraintAddResource(
  KHE_LIMIT_RESOURCES_CONSTRAINT c, KHE_RESOURCE r);
int KheLimitResourcesConstraintResourceCount(
  KHE_LIMIT_RESOURCES_CONSTRAINT c);
KHE_RESOURCE KheLimitResourcesConstraintResource(
  KHE_LIMIT_RESOURCES_CONSTRAINT c, int i);
```

After the instance has ended, function

```
KHE_RESOURCE_GROUP KheLimitResourcesConstraintDomain(
  KHE_LIMIT_RESOURCES_CONSTRAINT c);
```

returns a resource group containing the union of all these resource groups and resources (which must all have the same type). There is also

```
KHE_RESOURCE_GROUP KheLimitResourcesConstraintDomainComplement(
  KHE_LIMIT_RESOURCES_CONSTRAINT c);
```

which returns the complement of the domain, that is, the set of resources of the same type as the domain that are not in it.

To add and visit the event groups which are this constraint's points of application, call

```
void KheLimitResourcesConstraintAddEventGroup(
  KHE_LIMIT_RESOURCES_CONSTRAINT c, KHE_EVENT_GROUP eg);
int KheLimitResourcesConstraintEventGroupCount(
  KHE_LIMIT_RESOURCES_CONSTRAINT c);
KHE_EVENT_GROUP KheLimitResourcesConstraintEventGroup(
  KHE_LIMIT_RESOURCES_CONSTRAINT c, int i);
```

XESTT also allows individual events to be given, interpreted as singleton event groups. When KHE reads an XESTT file, an individual event `e` is added by a call to

```
KheLimitResourcesConstraintAddEventGroup(c, KheEventSingletonEventGroup(e));
```

When KHE writes an XESTT file, it makes two passes over the list of event groups, first writing all event groups whose number of events is not 1, then writing all event groups whose number of events is 1, the latter written as individual events rather than as event groups.

To add and visit the roles of the constraint, call

```
void KheLimitResourcesConstraintAddRole(
  KHE_LIMIT_RESOURCES_CONSTRAINT c, char *role);
int KheLimitResourcesConstraintRoleCount(
  KHE_LIMIT_RESOURCES_CONSTRAINT c);
char *KheLimitResourcesConstraintRole(
  KHE_LIMIT_RESOURCES_CONSTRAINT c, int i);
```

In practice, these should all be distinct, but no-one is checking.

Although the points of application are described as event groups, at the implementation level they are really sets of event resources. There is a way to bypass event groups and roles and create these sets of event resources directly. First, to create one point of application, call `KheLimitResourcesConstraintAddEventGroup` with `NULL` for the event group. Then call

```
void KheLimitResourcesConstraintAddEventResource(
  KHE_LIMIT_RESOURCES_CONSTRAINT c, int eg_index, KHE_EVENT_RESOURCE er);
```

to add an event resources to the `eg_index`'th point of application. Instances containing points of application created in this way cannot be written.

To visit the event resources of the `eg_index`'th point of application, call

```
int KheLimitResourcesConstraintEventResourceCount(
  KHE_LIMIT_RESOURCES_CONSTRAINT c, int eg_index);
KHE_EVENT_RESOURCE KheLimitResourcesConstraintEventResource(
  KHE_LIMIT_RESOURCES_CONSTRAINT c, int eg_index, int er_index);
```

Before the instance ends, these functions only visit the event resources added by `KheLimitResourcesConstraintAddEventResource`. After the instance ends, they also visit the event resources defined by the event group (if present) and roles.

Function

```
void KheLimitResourcesConstraintDebug(KHE_LIMIT_RESOURCES_CONSTRAINT c,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of `c` onto `fp` with the given verbosity and indent, in the usual way.

The number of points of application of a limit resources constraint is its number of event groups. The constraint density of the limit resources constraints of an instance is the number of event resources in all points of application divided by the number of event resources without preassigned resources.

# Chapter 4. Solutions

## 4.1. Overview

A solution is represented by an object of type `KHE_SOLN` ('solution' is always abbreviated to 'soln' in the KHE interface). Any number of solutions may exist and be operated on simultaneously. Instances are immutable after creation, and operations that change instances only assemble them, they do not disassemble them. In contrast, each operation that changes a solution is paired with one that changes it back. This supports not just the assembly of a fixed solution, such as one read from a file, but also the changes and testing of alternatives needed when solving an instance.

Within each solution are `KHE_MEET` objects representing meets (also called split events or sub-events), each of which may be assigned a time, and `KHE_TASK` objects representing the resource elements of meets, each of which may be assigned a resource. Although most meets are derived from events and most tasks are derived from event resources, these derivations are optional. Only meets and tasks that are so derived are considered part of the solution to the original instance, but other meets and tasks may be present to help with solving. Several meets may be derived from one event; these are the split events or sub-events of that event in the solution.

At all times, the solution (however incomplete it may be) has a definite numerical *cost*, a 64-bit integer measuring the badness of the solution which is always available via function `KheSolnCost` (Chapter 6). It may be used to guide the search for good solutions.

A solution must obey a condition called the *solution invariant* throughout its lifetime; this is an unbreakable constraint. A precise statement of the solution invariant appears in Section 4.9. Every operation that changes a solution in a way that could violate the invariant is implemented with two functions, which look generically like this:

```
bool KheOperationCheck(...);
bool KheOperation(...);
```

The two functions accept the same inputs and return the same value in a given solution state. The first returns `true` if the change would not violate the invariant, but itself changes nothing. The second also returns `true` if the change would not violate the invariant, but in that case it also makes the change. It changes nothing if the change would violate the invariant.

The relationship between the solution invariant and the constraints of the original instance is rather subtle. Should a constraint be incorporated into the invariant, so that no solution (not even a partial solution) will ever violate it? KHE leaves this question to the user. Some operations do incorporate constraints into the solution invariant, but those operations are all optional.

Some aspects of solution entities that may be changed have operations of the form

```
void KheEntityAspectFix(ENTITY e);
void KheEntityAspectUnFix(ENTITY e);
bool KheEntityAspectIsFixed(ENTITY e);
```

The first fixes that aspect of the entity—prevents later operations from changing it; the second removes the fix; the third returns `true` when the fix is in place. Initially everything is unfixed. Fixing a fixed aspect, and unfixing an unfixed aspect, do nothing. When the current value of some aspect will remain unchanged for a long time, fixing that aspect may have a significant efficiency payoff. This is because fixing detaches attached monitors (Chapter 6) whose cost is 0 and cannot change while the current fixes are in place, which can save a lot of time. Unfixing attaches those unattached monitors which could have non-zero cost given the unfix.

There are three levels of operations. At the lowest level are *basic operations*, which carry out basic queries and changes to a solution, such as assigning or unassigning the time of a meet. Above them are *helper functions*, which implement commonly needed sequences of basic operations, such as swaps. Some helper functions utilize optimizations that make them significantly more efficient than the equivalent sequences of basic operations.

At the highest level are *solvers*, which make large-scale changes to solutions. A complete algorithm for solving an instance is a solver, but so are operations with more modest scope, such as assigning times to the meetings of one form, assigning rooms, and so on.

KHE supplies many solvers, documented in later chapters, and the user is free to write others. As a matter of good design, solvers should not have behind-the-scenes access to KHE's data structures; they should use only the operations described in this guide and made available by header file `khe_platform.h`. They may of course call other solvers. The solvers supplied by KHE follow this rule.

## 4.2. Top-level operations

This section presents functions that operate on objects of type `KHE_SOLN`. Later sections present functions that operate on the components of solutions (meets, tasks, and so on).

### 4.2.1. Creation, deletion, and copy

To create a solution for a given instance, initially with no meets or tasks, call

```
KHE_SOLN KheSolnMake(KHE_INSTANCE ins, HA_ARENA_SET as);
```

`KheInstanceMakeEnd(ins)` must have been called and returned before `KheSolnMake` is called.

To delete `soln` and everything in it, and remove it from its solution groups, if any, call

```
void KheSolnDelete(KHE_SOLN soln);
```

The memory consumed by `soln` and everything in it will be freed. Each solution lies in its own memory arena, allowing its deletion to be carried out very efficiently: just delete its arena. Actually, there are two arenas, one holding the `soln` object, the other holding everything else. This is needed in case the user chooses to reduce a solution to a placeholder (Section 4.2.6).

Another way to create a solution is

```
KHE_SOLN KheSolnCopy(KHE_SOLN soln, HA_ARENA_SET as);
```

It returns a copy of `soln`. Parameter `as` is as for `KheSolnMake`. The copy is exact except that it does not lie in any solution groups. Immutable elements, such as anything from the instance, and time, resource, and event groups created within the solution, are shared, as are back pointers.

Copying is useful when forking a solution process part-way through: the original solution may continue down one thread, and the copy, which is quite independent, may be given to the other thread. Care is needed in one respect, however: it is not safe to make two copies of one solution simultaneously, even though the original solution is unaffected by copying it. This is because the copy algorithm uses temporary forwarding pointers in the objects of the solution.

Even semantically unimportant things, such as the order of items in sets, are preserved by `KheSolnCopy`. If the same solution algorithm is run on the original and the copy, and it does not depend on anything peculiar such as elapsed time or the memory addresses of its objects, it should produce the same solution. The author has verified this for `KheGeneralSolve2024` (Section 8.4). Diversity can be obtained by changing the copy's diversifier (Section 4.2.4).

The specification of `qsort` states that when two elements compare equal, their order in the final result is undefined. So the author has tried to eliminate all such cases in the comparison functions packaged with KHE. Index numbers, returned by functions such as `KheMeetSolnIndex` and `KheTaskSolnIndex`, are useful for breaking ties consistently as a last resort.

As an aid to debugging, function

```
void KheSolnDebug(KHE_SOLN soln, int verbosity, int indent, FILE *fp);
```

prints information about the current solution onto file `fp` with the given verbosity and indent, as described for debug functions in general in Section 1.4. Verbosity 1 prints just the instance name and current cost, verbosity 2 adds a breakdown of the current cost by constraint type (only constraint types with non-zero cost are printed), verbosity 3 adds debug prints of the solution's defects (Section 6.2), and verbosity 4 prints further details.

### 4.2.2. Solutions and arenas

Solutions and solvers can take up a lot of memory, and memory allocation and deallocation can become a serious bottleneck. KHE has a strategy for mitigating this problem. The idea is to create one arena set `as` (Appendix A.1.2) per thread, and pass `as` to each call to `KheSolnMake` made by the thread. Then the arenas needed to construct the solution are taken from `as`.

KHE does not make the mistake of sharing one arena set across threads. That would require arena sets to be lockable, which they are not. Appendix B.7 has more on these kinds of issues.

Solvers can and should participate in this memory allocation strategy. A solver can call

```
HA_ARENA KheSolnArenaBegin(KHE_SOLN soln);
```

This will return a currently unused arena from `soln`'s arena set. When the arena is no longer required and its memory can be made available for other uses, the solver can call

```
void KheSolnArenaEnd(KHE_SOLN soln, HA_ARENA a);
```

This makes the memory used by arena `a` available to the other arenas of `soln`'s arena set. (These two functions simply call `HaArenaMake` and `HaArenaDelete` from Appendix A.1.1). There is also a function to retrieve a solution's arena set:

```
HA_ARENA_SET KheSolnArenaSet(KHE_SOLN soln);
```

Formerly there was a function (`KheSolnSetArenaSet`) to set a solution's arena set. However, a redesign of arenas and arena sets made this wrong, so it has been withdrawn. The recommended alternative is to use `KheSolnCopy` (Section 4.2.1) to copy the solution into the desired arena set.

For responding gracefully when memory runs out, there are functions

```
void KheSolnJmpEnvBegin(KHE_SOLN soln, jmp_buf *env);
void KheSolnJmpEnvEnd(KHE_SOLN soln);
```

These just call `HaArenaSetJmpEnvBegin` and `HaArenaSetJmpEnvEnd` on `soln`'s arena set. See Appendix A.1.2 for a description of these functions and an example of how to use them.

### 4.2.3. Simple attributes

A solution may lie in any number of solution groups. To add it to a solution group and delete it from a solution group, use functions `KheSolnGroupAddSoln` and `KheSolnGroupDeleteSoln` from Section 2.2. To visit the solution groups containing `soln`, call

```
int KheSolnSolnGroupCount(KHE_SOLN soln);
KHE_SOLN_GROUP KheSolnSolnGroup(KHE_SOLN soln, int i);
```

in the usual way.

A solution is always for a particular instance, fixed when the solution is created. Function

```
KHE_INSTANCE KheSolnInstance(KHE_SOLN soln);
```

returns the instance that the solution is for.

A solution has an optional Description attribute which may contain arbitrary text saying what is distinctive about the solution. This attribute may be set and retrieved by calling

```
void KheSolnSetDescription(KHE_SOLN soln, char *description);
char *KheSolnDescription(KHE_SOLN soln);
```

The default value is `NULL`, meaning no description.

A solution also has an optional RunningTime attribute giving the wall clock time to produce the solution, in seconds. This attribute may be set and retrieved by calling

```
void KheSolnSetRunningTime(KHE_SOLN soln, float running_time);
bool KheSolnHasRunningTime(KHE_SOLN soln, float *running_time);
```

If `KheSolnSetRunningTime` has been called, then `KheSolnHasRunningTime` returns `true` with `*running_time` set to the most recent value passed by `KheSolnSetRunningTime`. Otherwise it returns `false` with `*running_time` set to `-1.0`. It would be impossible for KHE to ensure that the value stored in this field is honest, and it does not try to.

There is also a function for comparing two solutions by their running times. It comes in two versions, one which makes sense to people, and another which makes sense to `qsort`:

```
int KheSolnIncreasingRunningTimeTypedCmp(KHE_SOLN soln1, KHE_SOLN soln2);
int KheSolnIncreasingRunningTimeCmp(const void *t1, const void *t2);
```

Solutions without a running time are treated as though they have a very large running time.

Solution objects and their components have back pointers in the usual way. These may be changed at any time. To set and retrieve the back pointer of a solution object, call

```
void KheSolnSetBack(KHE_SOLN soln, void *back);
void *KheSolnBack(KHE_SOLN soln);
```

as usual.

### 4.2.4. Diversification

One strategy for finding good solutions is to find many solutions and choose the best. This only works when the solutions are diverse, creating a need to find ways to produce diversity.

Each solution contains a non-negative integer *diversifier*. Its initial value is 0, but it may be set and retrieved at any time by

```
void KheSolnSetDiversifier(KHE_SOLN soln, int val);
int KheSolnDiversifier(KHE_SOLN soln);
```

When solutions are created that need to be diverse, each is given a different diversifier. When an algorithm reaches a point where it could equally well follow any one of several paths, it consults the diversifier when making its choice.

Suppose the diversifier has value $d$ and a point is reached where there are $c$ alternatives, for some $c \geq 1$. A simple approach is to choose the `i`th alternative (counting from 0), where

```
i = d % c;
```

We call a function $D(d,c)$ which returns an integer $i$ s.t. $0 \leq i < c$ a *diversification function.*

How should we choose diversifiers and diversification functions to ensure that we really do get diversity? One possibility is to start with a random integer and change it using a random number generator, passing the current value as seed, each time the diversifier is consulted. But there is no way to analyse the effect of this, so instead we are going to examine what happens when the diversifiers are fixed successive integers starting from 0.

What we want is a little hard to grasp. Suppose that, at some points in the algorithm, it is offered a choice between 1 alternative; at others, there are 2 alternatives, and so on, with a maximum of $n$ alternatives. For a given diversifier, there are $n!$ different functions of the number of choices. Ideally we would want all of these functions to turn up as $d$ varies over its range.

It is not obvious, but it is a fact that the modulus function above does turn up every function when $n$ is 1, 2 or 3, but when $n$ is 4 it produces 12 distinct functions, only half the possible 24 functions, as the following tables, obtained by running `khe -d4`, show:

```
   d | 1 2        d | 1 2 3         d | 1 2 3 4
 ----+------    ----+---------    ----+-----------
   0 | 0 0        0 | 0 0 0         0 | 0 0 0 0
   1 | 0 1        1 | 0 1 1         1 | 0 1 1 1
 ----+------      2 | 0 0 2         2 | 0 0 2 2
                  3 | 0 1 0         3 | 0 1 0 3
                  4 | 0 0 1         4 | 0 0 1 0
                  5 | 0 1 2         5 | 0 1 2 1
                ----+---------      6 | 0 0 0 2
                                    7 | 0 1 1 3
                                    8 | 0 0 2 0
                                    9 | 0 1 0 1
                                   10 | 0 0 1 2
                                   11 | 0 1 2 3
                                   12 | 0 0 0 0   (same as 0)
                                   13 | 0 1 1 1   (same as 1)
                                   14 | 0 0 2 2   (same as 2)
                                   15 | 0 1 0 3   (same as 3)
                                   16 | 0 0 1 0   (same as 4)
                                   17 | 0 1 2 1   (same as 5)
                                   18 | 0 0 0 2   (same as 6)
                                   19 | 0 1 1 3   (same as 7)
                                   20 | 0 0 2 0   (same as 8)
                                   21 | 0 1 0 1   (same as 9)
                                   22 | 0 0 1 2   (same as 10)
                                   23 | 0 1 2 3   (same as 11)
                                  ----+-----------
```

Each row is one value of *d*, and each column is one value of *c*. What this means is that if, during the course of one run, no more than 4 choices are offered at any one point, then only 12 distinct solutions can emerge, no matter how many are begun.

The most natural diversification function which produces distinct outcomes is probably

```
(d / fact(c - 1)) % c
```

where `fact` is the factorial function. (To avoid overflow, in practice one stops multiplying as soon as the value exceeds `d`.) Each line is something like the binary representation of `d`, only in a factorial number system rather than binary:

```
   d | 1  2        d | 1  2  3        d | 1  2  3  4
 ----+------    ----+---------     ----+------------
   0 | 0  0        0 | 0  0  0        0 | 0  0  0  0
   1 | 0  1        1 | 0  1  0        1 | 0  1  0  0
 ----+------       2 | 0  0  1        2 | 0  0  1  0
                   3 | 0  1  1        3 | 0  1  1  0
                   4 | 0  0  2        4 | 0  0  2  0
                   5 | 0  1  2        5 | 0  1  2  0
                ----+---------        6 | 0  0  0  1
                                      7 | 0  1  0  1
                                      8 | 0  0  1  1
                                      9 | 0  1  1  1
                                     10 | 0  0  2  1
                                     11 | 0  1  2  1
                                     12 | 0  0  0  2
                                     13 | 0  1  0  2
                                     14 | 0  0  1  2
                                     15 | 0  1  1  2
                                     16 | 0  0  2  2
                                     17 | 0  1  2  2
                                     18 | 0  0  0  3
                                     19 | 0  1  0  3
                                     20 | 0  0  1  3
                                     21 | 0  1  1  3
                                     22 | 0  0  2  3
                                     23 | 0  1  2  3
                                   ----+------------
```

But there is still a problem: if all alternatives have 4 choices, say, then the first 6 threads will produce the same result despite differing in `d`. The solution to this seems to be function

```
(d / fact(c - 1) + d % fact(c - 1)) % c
```

Delightfully, it produces

```
   d | 1  2        d | 1  2  3        d | 1  2  3  4
 ----+------    ----+---------     ----+------------
   0 | 0  0        0 | 0  0  0        0 | 0  0  0  0
   1 | 0  1        1 | 0  1  1        1 | 0  1  1  1
 ----+------       2 | 0  0  1        2 | 0  0  1  2
                   3 | 0  1  2        3 | 0  1  2  3
                   4 | 0  0  2        4 | 0  0  2  0
                   5 | 0  1  0        5 | 0  1  0  1
                ----+---------        6 | 0  0  0  1
                                      7 | 0  1  1  2
                                      8 | 0  0  1  3
                                      9 | 0  1  2  0
                                     10 | 0  0  2  1
                                     11 | 0  1  0  2
                                     12 | 0  0  0  2
                                     13 | 0  1  1  3
                                     14 | 0  0  1  0
                                     15 | 0  1  2  1
                                     16 | 0  0  2  2
                                     17 | 0  1  0  3
                                     18 | 0  0  0  3
                                     19 | 0  1  1  0
                                     20 | 0  0  1  1
                                     21 | 0  1  2  2
                                     22 | 0  0  2  3
                                     23 | 0  1  0  0
                                   ----+------------
```

and is diverse up to $c = 8$ at least. Function

```
int KheSolnDiversifierChoose(KHE_SOLN soln, int c);
```

implements this function, returning a non-negative integer less than `c`.

It is quite reasonable for *every* algorithm faced with an arbitrary choice to diversify. It is easy to do, and it provides a continual prodding towards diversity that should drive solutions with different diversifiers further and further apart as solving continues, always provided that there are sufficiently many choices.

### 4.2.5. Visit numbers

Some algorithms, such as tabu search and ejection chains, need to know whether some part of the solution has changed recently. KHE supports this with a system of *visit numbers*.

A visit number is just an integer stored at some point in the solution. The KHE platform initializes visit numbers (to 0) and copies them, but does not otherwise use them. The user is free to set their values in any way at any time, using operations that look generically like this:

```
void KheSolnEntitySetVisitNum(KHE_SOLN_ENTITY e, int num);
int KheSolnEntityVisitNum(KHE_SOLN_ENTITY e);
```

But there is also a conventional way to use visit numbers, as follows.

The solution object contains a *global visit number* which is used differently from the others. The following operations are applicable to it:

```
void KheSolnSetGlobalVisitNum(KHE_SOLN soln, int num);
int KheSolnGlobalVisitNum(KHE_SOLN soln);
void KheSolnNewGlobalVisit(KHE_SOLN soln);
```

The first two operations are not usually used directly. The third increases the global visit number by one. This new value has not previously been assigned to any visit number.

The visit numbers of other solution entities should never exceed the global visit number. The operations for other solution entities look generically like this:

```
void KheSolnEntitySetVisitNum(KHE_SOLN_ENTITY e, int num);
int KheSolnEntityVisitNum(KHE_SOLN_ENTITY e);
bool KheSolnEntityVisited(KHE_SOLN_ENTITY e, int slack);
void KheSolnEntityVisit(KHE_SOLN_ENTITY e);
void KheSolnEntityUnVisit(KHE_SOLN_ENTITY e);
```

Type `SOLN_ENTITY` is fictitious and so are these functions; they just display the standard pattern. The first two are the standard ones. The third returns the value of the condition

```
KheSolnVisitNum(soln) - KheSolnEntityVisitNum(e) <= slack
```

where `soln` is the solution containing `e`. The fourth sets `e`'s visit number to its solution object's visit number, and the last sets it to one less than its solution's visit number.

These operations may be used to implement tabu search efficiently as follows. Suppose for example that a change to the assignment of `meet` is to remain tabu until at least `tabu_len` other changes have been made. The code for this is

```
if( !KheMeetVisited(meet, tabu_len) )
{
  KheSolnNewVisit(KheMeetSoln(meet));
  KheMeetVisit(meet);
  ... change the assignment of meet ...
}
```

To ensure that everything is visitable initially, call

```
KheSolnSetVisitNum(soln, tabu_len);
```

It is easy to generalize this code to other operations.

One form of the ejection chains algorithm requires that once a meet (or other entity) has been changed during the current visit, it must remain tabu until a new visit is started in the outer loop of the algorithm. The code for this is

```
if( !KheMeetVisited(meet, 0) )
{
  KheMeetVisit(meet);
  ... change the assignment of meet ...
}
```

A variant of this idea makes `meet` tabu to recursive calls, but not tabu for the entire remainder of the current visit. The code for this is

```
if( !KheMeetVisited(meet, 0) )
{
  KheMeetVisit(meet);
  ... change the assignment of meet and recurse ...
  KheMeetUnVisit(meet);
}
```

Only meets in the direct line of the recursion are tabu. At present, the author's ejection code follows this pattern for visiting monitors only, and does it behind the scenes is the ejection chains framework code (in type `KHE_EJECTOR`). The user does not have to worry about visiting.

### 4.2.6. Placeholder and invalid solutions

A placeholder solution is a solution which is missing most of what an ordinary solution has, either because it is invalid, or to save memory. Function

```
KHE_SOLN_TYPE KheSolnType(KHE_SOLN soln);
```

may be used to find out whether a solution is a placeholder. Its return value has type

```
typedef enum {
  KHE_SOLN_INVALID_PLACEHOLDER,
  KHE_SOLN_BASIC_PLACEHOLDER,
  KHE_SOLN_WRITABLE_PLACEHOLDER,
  KHE_SOLN_ORDINARY
} KHE_SOLN_TYPE;
```

The first three values indicate that `soln` is a placeholder of some kind, as follows.

`KHE_SOLN_INVALID_PLACEHOLDER` means that `soln` is an *invalid placeholder*: it became a placeholder because it has some problem. In practice this can only happen when reading a solution from an archive (Section 2.4). We usually just say that `soln` is *invalid*. Function

```
KML_ERROR KheSolnInvalidError(KHE_SOLN soln);
```

returns the first error that made `soln` invalid, or `NULL` if `soln` is not invalid. For type `KML_ERROR`, see Section A.5.2. An invalid solution offers few functions: for example, it has no cost.

`KHE_SOLN_BASIC_PLACEHOLDER` means that `soln` is a *basic placeholder*: all of the objects below `soln` (all its meets, tasks, and so on) have been deleted. This frees a great deal of memory, which is the point of it, but it makes `soln` unusable except that the following functions remain available and return their previous values:

```
char *KheSolnDescription(KHE_SOLN soln);
void *KheSolnBack(KHE_SOLN soln);
KHE_INSTANCE KheSolnInstance(KHE_SOLN soln);
bool KheSolnHasRunningTime(KHE_SOLN soln, float *running_time);
int KheSolnSolnGroupCount(KHE_SOLN soln);
KHE_SOLN_GROUP KheSolnSolnGroup(KHE_SOLN soln, int i);
void *KheSolnImpl(KHE_SOLN soln);
int KheSolnDiversifier(KHE_SOLN soln);
int KheSolnVisitNum(KHE_SOLN soln);
KHE_COST KheSolnCost(KHE_SOLN soln);
```

Function `KheSolnTypeReduce` below is also still available.

`KHE_SOLN_WRITABLE_PLACEHOLDER` is like `KHE_SOLN_BASIC_PLACEHOLDER` except that the solution can also be written by `KheArchiveWrite` (Section 2.7), because a brief private record of who is assigned to what is retained. `KheArchiveWrite` will abort if it is asked to write an invalid or basic placeholder. Even a writable placeholder cannot be written if `KheArchiveWrite` has been asked to write a report along with each solution.

Finally, `KHE_SOLN_ORDINARY` indicates that `soln` is an ordinary solution (not a placeholder), supporting the full range of operations including access to its meets, tasks, and so on. When a solution is created, it is an ordinary solution. A placeholder solution cannot be created directly; an ordinary solution must be created and then reduced to a placeholder, using function `KheSolnTypeReduce` below. This ensures that the solution cost is correct.

Placeholder solutions may be used to build tables of solutions showing costs and running times; but they cannot be used to find cost breakdowns by constraint type, or to print timetables, and so on. Writable placeholder solutions are good when solving, both for solutions produced by the solver and for solutions which are already in the archive and just need to be read in and written out again. Function

```
void KheSolnTypeReduce(KHE_SOLN soln, KHE_SOLN_TYPE soln_type,
  KML_ERROR ke);
```

changes the type of `soln` to `soln_type`. If `soln_type` is `KHE_SOLN_INVALID_PLACEHOLDER`, `ke` must be non-`NULL`, and a copy of it becomes the value returned by `KheSolnInvalidError`. Otherwise `ke` is not used and should be `NULL`.

`KheSolnTypeReduce` can only change the type to something equal or lower. For example, it can reduce an ordinary solution to any kind of placeholder, but it cannot reduce a placeholder to an ordinary solution, because the data is lost. Changing the type to what it already is does nothing except replace `KheSolnInvalidError` if the type is `KHE_SOLN_INVALID_PLACEHOLDER`.

### 4.2.7. Traversing the components of solutions

A solution has many components: principally tasks and meets, but also other objects. They can all be visited, using the functions defined in this section.

To visit the meets of a solution, in an unspecified order, call

```
int KheSolnMeetCount(KHE_SOLN soln);
KHE_MEET KheSolnMeet(KHE_SOLN soln, int i);
```

The meets visited include the *cycle meets* described in Section 4.5.3. To visit the meets of a solution derived from a given event, call

```
int KheEventMeetCount(KHE_SOLN soln, KHE_EVENT e);
KHE_MEET KheEventMeet(KHE_SOLN soln, KHE_EVENT e, int i);
```

The first returns the number of meets derived from `e` (possibly 0), and the second returns the `i`'th of these meets, in an unspecified order.

To visit the tasks of a solution, in an unspecified order, call

```
int KheSolnTaskCount(KHE_SOLN soln);
KHE_TASK KheSolnTask(KHE_SOLN soln, int i);
```

To visit the tasks derived from a given event resource, call

```
int KheEventResourceTaskCount(KHE_SOLN soln, KHE_EVENT_RESOURCE er);
KHE_TASK KheEventResourceTask(KHE_SOLN soln, KHE_EVENT_RESOURCE er,
  int i);
```

There is one for each meet derived from the event containing `er`.

A solution may also contain *nodes* and *taskings*, as explained in Chapter 5. To visit the nodes in an unspecified order, call

```
int KheSolnNodeCount(KHE_SOLN soln);
KHE_NODE KheSolnNode(KHE_SOLN soln, int i);
```

To visit the taskings, call

```
int KheSolnTaskingCount(KHE_SOLN soln);
KHE_TASKING KheSolnTasking(KHE_SOLN soln, int i);
```

in the usual way.

## 4.3. Complete representation and preassignment conversion

A solution is a *complete representation* when it satisfies the following two conditions:

- For each event `e` of the solution's instance, the total duration of the meets derived from `e` is equal to the duration of `e`;
- For each event resource `er` of the solution's instance, each meet derived from the event containing `er` contains a task derived from `er`.

Complete representation does not rule out extra meets or tasks. It has nothing to do with being a complete solution, in the sense of assigning a time to every meet and a resource to every task.

KHE does not require a solution to be a complete representation, since that would be too restrictive when building and modifying solutions. However, the cost it reports for a solution is correct only when that solution is a complete representation. This is because, behind the scenes, KHE needs to be able to see a meet with no assigned time in order for it to realize that an assign time constraint is being violated, and similarly for the other constraints.

There is a standard procedure, part of the XML specification, for converting a solution into a complete representation:

1. For each event `e` of the solution's instance, if there are no meets derived from `e`, then insert one meet whose duration is the duration of `e`, and whose assigned time is the preassigned time of `e`, or is absent if `e` has no preassigned time. Initially, this meet contains no tasks, but that may be changed by the third rule.
2. If now there is an event `e` such that the total duration of the meets derived from `e` is not equal to the duration of `e`, then that is an error and the XML file is rejected.
3. For each event resource `er` of each event `e` of the instance, for each meet derived from `e`, if that meet does not contain a task derived from `er`, then add one. Its assigned resource is the preassigned resource of `er` if there is one, or is absent if `er` has no preassigned resource.

This procedure, minus the conversions from preassignments to assignments, is implemented by

```
bool KheSolnMakeCompleteRepresentation(KHE_SOLN soln,
  KHE_EVENT *problem_event);
```

For each event `e`, it finds the total duration of the meets derived from `e`. If that is greater than the duration of `e` it returns `false` with `*problem_event` set to `e`. If it is less, then one meet derived from `e` is added whose duration makes up the difference. The domain of this meet has the usual default value: the preassigned time of `e` if any, or else the largest legal domain, `KheSolnPackingTimeGroup(soln)` (Section 4.5.3). Then, within each meet derived from an event, just created or not, it adds a task for each event resource `er` not already represented. The domain of this task has the usual default value: the preassigned resource of `er` if any, or else the largest legal domain, `KheResourceTypeFullResourceGroup(rt)`, where `rt` is `er`'s resource type.

`KheSolnMakeCompleteRepresentation` has two uses. The first is in `KheArchiveRead` (Section 2.4), which applies it to each solution it reads, as the XML specification requires, and then calls these two public functions to convert preassignments into assignments:

```
void KheSolnAssignPreassignedTimes(KHE_SOLN soln);
void KheSolnAssignPreassignedResources(KHE_SOLN soln,
  KHE_RESOURCE_TYPE rt);
```

`KheSolnAssignPreassignedTimes` assigns the obvious time to each preassigned unassigned meet. `KheSolnAssignPreassignedResources` assigns the obvious resource to each preassigned unassigned task of type `rt` (any type if `rt` is `NULL`).

The second use for `KheSolnMakeCompleteRepresentation` is to build a solution from scratch, ready for solving. The solution returned by `KheSolnMake` has no meets except for the initial cycle meet, and it has no tasks. `KheSolnMakeCompleteRepresentation` is a very convenient way to add both. When solving, it is usually called immediately after `KheSolnMake` and `KheSolnSplitCycleMeet` (Section 4.5.3). The solution changes as solving proceeds, but it remains a complete representation throughout, except perhaps during brief reconstructions. A call to `KheSolnAssignPreassignedResources` is also a good idea, since it does no harm and ensures that resource constraints involving preassigned resources will contribute to the cost of the solution as soon as the meets they are preassigned to are assigned times. On the other hand, it may be better not to assign preassigned times at this point; Section 10.4 has the alternatives.

## 4.4. Solution time, resource, and event groups

Groups are important in solving. A solver needs to be able to construct its own, since the ones declared in the instance might not be enough. (Conceivably, a solver could need its own times and resources as well, but that possibility is not currently supported.) Accordingly, the following functions are provided for constructing a time group while solving:

```
void KheSolnTimeGroupBegin(KHE_SOLN soln);
void KheSolnTimeGroupAddTime(KHE_SOLN soln, KHE_TIME t);
void KheSolnTimeGroupSubTime(KHE_SOLN soln, KHE_TIME t);
void KheSolnTimeGroupUnion(KHE_SOLN soln, KHE_TIME_GROUP tg2);
void KheSolnTimeGroupIntersect(KHE_SOLN soln, KHE_TIME_GROUP tg2);
void KheSolnTimeGroupDifference(KHE_SOLN soln, KHE_TIME_GROUP tg2);
KHE_TIME_GROUP KheSolnTimeGroupEnd(KHE_SOLN soln);
```

The first operation begins the process; the next five do what the corresponding operations for instance time groups do, and the last operation returns the finished time group. Its kind will be `KHE_TIME_GROUP_KIND_ORDINARY`, and its `id` and `name` attributes will be `NULL`.

A similar set of operations constructs a resource group:

```
void KheSolnResourceGroupBegin(KHE_SOLN soln, KHE_RESOURCE_TYPE rt);
void KheSolnResourceGroupAddResource(KHE_SOLN soln, KHE_RESOURCE r);
void KheSolnResourceGroupSubResource(KHE_SOLN soln, KHE_RESOURCE r);
void KheSolnResourceGroupUnion(KHE_SOLN soln, KHE_RESOURCE_GROUP rg2);
void KheSolnResourceGroupIntersect(KHE_SOLN soln, KHE_RESOURCE_GROUP rg2);
void KheSolnResourceGroupDifference(KHE_SOLN soln, KHE_RESOURCE_GROUP rg2);
KHE_RESOURCE_GROUP KheSolnResourceGroupEnd(KHE_SOLN soln);
```

and an event group:

```
void KheSolnEventGroupBegin(KHE_SOLN soln);
void KheSolnEventGroupAddEvent(KHE_SOLN soln, KHE_EVENT e);
void KheSolnEventGroupSubEvent(KHE_SOLN soln, KHE_EVENT e);
void KheSolnEventGroupUnion(KHE_SOLN soln, KHE_EVENT_GROUP eg2);
void KheSolnEventGroupIntersect(KHE_SOLN soln, KHE_EVENT_GROUP eg2);
void KheSolnEventGroupDifference(KHE_SOLN soln, KHE_EVENT_GROUP eg2);
KHE_EVENT_GROUP KheSolnEventGroupEnd(KHE_SOLN soln);
```

All the usual operations may be applied to these groups. The functions use `soln` as a factory object instead of the group itself, to ensure that groups are complete and immutable (apart from their back pointers) by the time they are given to the user. Groups are deleted when their solution is deleted. They know which instance they are for, but the instance, being immutable after creation, is not aware of their existence.

Within one solution, when calls to `KheSolnTimeGroupEnd` return groups containing the same elements, the objects returned are the same too. This is done using a hash table of time groups. It allows the user to experiment with many time groups, without worrying about their memory cost. This is not being done for resource and event groups yet; it should be.

## 4.5. Meets

A meet is created by calling

```
KHE_MEET KheMeetMake(KHE_SOLN soln, int duration, KHE_EVENT e);
```

This creates and adds to `soln` a new meet of the given duration, which must be at least 1. If `e` is non-`NULL`, it indicates that this meet is derived from event `e`. Initially the meet contains no tasks; they must be added separately. A meet may be deleted from its solution by calling

```
void KheMeetDelete(KHE_MEET meet);
```

Any tasks within `meet` are also deleted. If `meet` is assigned to another meet, or any other meets are assigned to it, all those assignments are removed. The meet is also deleted from any node (Section 5.2) it may lie in.

The back pointer of a meet may be set and retrieved by

```
void KheMeetSetBack(KHE_MEET meet, void *back);
void *KheMeetBack(KHE_MEET meet);
```

and the visit number by

```
void KheMeetSetVisitNum(KHE_MEET meet, int num);
int KheMeetVisitNum(KHE_MEET meet);
bool KheMeetVisited(KHE_MEET meet, int slack);
void KheMeetVisit(KHE_MEET meet);
void KheMeetUnVisit(KHE_MEET meet);
```

Function

```
char *KheMeetId(KHE_MEET meet);
```

returns a string which is supposed to uniquely identify the meet. Most of the time, this is the Id of the meet's event, followed, if the event is split into more than one meet, by a colon and an index number identifying the meet within the event. Some special meets (e.g. cycle meets) have an Id beginning and ending with `"/"`.

The result of `KheMeetId(meet)` is created when `KheMeetId(meet)` is first called, and stored in `meet` so that it does not have to be created over and over. If it is used only for debugging, as is the intention, there is virtually no cost in running time or memory when debugging is off. There is some uncertainty over the choice of index number when meets are split and joined.

The other attributes of a meet are accessed by

```
KHE_SOLN KheMeetSoln(KHE_MEET meet);
int KheMeetSolnIndex(KHE_MEET meet);
int KheMeetDuration(KHE_MEET meet);
KHE_EVENT KheMeetEvent(KHE_MEET meet);
```

These return the enclosing solution, `meet`'s index in that solution (that is, the value of `i` for which `KheSolnMeet(soln, i)` returns `meet`), its duration, and the event that `meet` is derived from (possibly `NULL`). Index numbers change when meets are deleted (the hole left by the deletion of a meet, if not last, is plugged by the last meet), so care is needed. There is also

```
bool KheMeetIsPreassigned(KHE_MEET meet, TIME *time);
```

which returns `true` when `KheMeetEvent(meet) != NULL` and that event has a preassigned time; `meet` is called a *preassigned meet* in that case. If `time != NULL`, then `*time` is set to the event's preassigned time if `meet` is preassigned, and to `NULL` otherwise.

When deciding what order to assign meets in, it is handy to have some measure of how difficult they are to timetable. Functions

```
int KheMeetAssignedDuration(KHE_MEET meet);
int KheMeetDemand(KHE_MEET meet);
```

attempt to provide this. `KheMeetAssignedDuration` is the duration of `meet` if it is assigned, or 0 otherwise. `KheMeetDemand(meet)` is the sum, over `meet` and all meets assigned to `meet`, directly or indirectly, of the product of the duration of the meet and the number of tasks it contains. This value is stored in the meet and kept up to date as solutions change, so a call on `KheMeetDemand` costs almost nothing.

A task is added to its meet when it is created, and removed from its meet when it is deleted.

To visit the tasks of a meet, call

```
int KheMeetTaskCount(KHE_MEET meet);
KHE_TASK KheMeetTask(KHE_MEET meet, int i);
bool KheMeetRetrieveTask(KHE_MEET meet, char *role, KHE_TASK *task);
bool KheMeetFindTask(KHE_MEET meet, KHE_EVENT_RESOURCE er,
  KHE_TASK *task);
```

The first two traverse the tasks. The order of tasks within meets is not significant, and it may change as tasks are created and deleted. `KheMeetRetrieveTask` retrieves a task which is derived from an event resource with the given `role`, if present. `KheMeetFindTask` is similar, but it looks for a task derived from event resource `er`, rather than for a role. There are also

```
bool KheMeetContainsResourcePreassignment(KHE_MEET meet,
  KHE_RESOURCE r, KHE_TASK *task);
bool KheMeetContainsResourceAssignment(KHE_MEET meet,
  KHE_RESOURCE r, KHE_TASK *task);
```

which return `true` if `meet` contains a task preassigned or assigned `r`, setting `*task` to one if so. Here a task is considered to be preassigned if it is derived from a preassigned event resource.

A meet contains an optional *assignment*, which assigns the meet to a particular offset in another meet, thereby fixing its time relative to the starting time of the other meet, and a *time domain* which restricts the times it may start at to an arbitrary subset of the times of the cycle. These attributes are described in detail in later sections.

A meet may optionally be contained in one node (Chapter 5). Functions

```
KHE_NODE KheMeetNode(KHE_MEET meet);
int KheMeetNodeIndex(KHE_MEET meet);
```

return the node containing `meet`, and the index of `meet` in that node, or `NULL` and `-1` if none.

As an aid to debugging, function

```
void KheMeetDebug(KHE_MEET meet, int verbosity, int indent, FILE *fp);
```

prints `meet` onto `fp` with the given verbosity and indent (for which see Section 1.4). Verbosity 1 prints just an identifying name; verbosity 2 adds the chain of assignments leading out of `meet`.

The name is usually the name of `meet`'s event, between quotes. If there is more than one meet corresponding to that event, this will be followed by a colon and the number `i` for which `KheEventMeet(soln, e, i)` equals `meet`. Alternatively, if `meet` is a cycle meet (Section 4.5.3), the name is its starting time (a time name or else an index) between slashes.

### 4.5.1. Splitting and merging

A meet may be split into two meets whose durations sum to the duration of the original meet:

```
bool KheMeetSplitCheck(KHE_MEET meet, int duration1, bool recursive);
bool KheMeetSplit(KHE_MEET meet, int duration1, bool recursive,
  KHE_MEET *meet1, KHE_MEET *meet2);
```

These functions follow the pattern described earlier for operations that might violate the solution invariant, in that both return `true` if the split is permitted. The second actually carries out the split, setting `*meet1` and `*meet2` to the new meets if the split is permitted, and leaving them unchanged if not. The original meet, `meet`, is undefined after a successful split, unless `meet1` or `meet2` is set to `&meet` (this may seem dangerous, but it does what is wanted whether the split succeeds or not). The split meet may be a cycle meet, in which case so are the two fragments.

The first new meet, `*meet1`, has duration `duration1`, and the second, `*meet2`, has the remaining duration. Parameter `duration1` must be such that both meets have duration at least 1, otherwise both functions abort. Their back pointers are set to the back pointer of `meet`. If `meet` is assigned, `*meet1` has the same target meet and offset as `meet`, while `*meet2` has the same target meet, but its offset is `duration1` larger, making the two meets adjacent in time.

If `recursive` is `true`, any meets assigned to `meet` that span the split point will also be split, into one meet for the part overlapping `*meet1` and one for the part overlapping `*meet2`. This process proceeds recursively as deeply as required.

The two split functions return `true` if these two conditions hold:

- Either `recursive` is `true`, or else no meets assigned to `meet` span the split point.

- The meets resulting from each split have copies of the meet bounds (Section 4.5.4) of the meets they are fragments of. Nevertheless their domains usually change, owing to meet bounds with specific `duration` attributes. This must cause no incompatibilities with the domains of other meets connected to them by assignments, allowing for offsets. When a cycle meet (Section 4.5.3) splits, the two fragments have the appropriate singleton domains. Domain incompatibilities cannot occur in that case.

If these conditions hold, `meet` is said to be *splittable* at `duration1`.

When a meet splits, its tasks split too. This produces what is typically required when assigning rooms: the fragments are free to be assigned different resources. The other possibility, where the fragments are required to be assigned the same resource, can be obtained by assigning the fragmentary tasks to each other. This must be done separately.

The next two functions are concerned with merging two meets into one:

```
bool KheMeetMergeCheck(KHE_MEET meet1, KHE_MEET meet2);
bool KheMeetMerge(KHE_MEET meet1, KHE_MEET meet2, bool recursive,
  KHE_MEET *meet);
```

Parameters `meet1` and `meet2` become undefined after a successful merge, unless `meet` is set to `&meet1` or `&meet2`.

If `recursive` is `true`, after merging `meet1` and `meet2`, `KheMeetMerge` searches for pairs of meets, one formerly assigned to the end of `meet1`, the other formerly assigned to the beginning of `meet2`, which are mergeable according to `KheMeetMergeCheck`, and merges each such pair. This process proceeds recursively as deeply as required. `KheMeetMergeCheck` has no `recursive` parameter because its result does not depend on whether the merge is recursive.

The functions return `true` if all these conditions hold:

- The two meets are distinct.

- The two meets have the same value of `KheMeetIsCycleMeet` (Section 4.5.3).
- The two meets have the same value of `KheMeetEvent`, possibly `NULL`.
- The two meets have the same value of `KheMeetNode`, possibly `NULL`.
- The two meets are both either assigned to the same meet, or not assigned. If assigned, the offset of one (it may be either) must equal the offset plus duration of the other, ensuring they are adjacent in time. Cycle meets, although never assigned, must also be adjacent in time.
- The two meets have the same number of tasks, and the order of their tasks may be permuted so that corresponding tasks are compatible. Two tasks are compatible when they have the same taskings, domains, event resources, and assignments.
- The result meet takes over the meet bounds (Section 4.5.4) of one of the meets being merged. Nevertheless its domain usually changes, owing to meet bounds with non-zero `duration` attributes. This must cause no incompatibilities with the domains of other meets connected to it by assignments, allowing for offsets. When cycle meets (Section 4.5.3) merge, the result meet has the singleton domain of the chronologically first meet. Domain incompatibilities cannot occur in that case.

If all these conditions hold, `meet1` and `meet2` are said to be *mergeable*. These conditions usually hold trivially when merging the results of a previous split. The merged meet's attributes (including its meet bounds and the order of its tasks) may come from either `meet1` or `meet2`; the choice is deliberately left unspecified, and the user must not depend on it.

It is now clear why `KheMeetMergeCheck` does not need a `recursive` parameter: because none of the conditions just given depend on whether the merge is recursive. Recursive merges are only attempted when `KheMergeCheck` says they will succeed. So instead of preventing the top-level merge, an unacceptable recursive merge simply does not happen.

### 4.5.2. Assignment

KHE's basic operations do not include assigning a time to a meet. A meet is either unassigned or else assigned to another meet at a given offset, fixing the starting times of the two meets relative to each other, but not assigning a specific time to either. For example, if `m1` is assigned to `m2` at offset 2, then whatever time `m2` eventually starts at, `m1` will start two times later. Of course, ultimately meets need to be assigned times. This is done by assigning them to special meets called *cycle meets* (Section 4.5.3).

Assigning one meet to another supports *hierarchical timetabling*, in which several meets are timetabled relative to each other, then the whole group is timetabled into a larger context, and so on. One simple application is in handling link events constraints. Assigning all the linked events except one to that exception guarantees that the linked events will be simultaneous; the time eventually assigned to the exception becomes the time assigned to all.

The fundamental meet assignment operations are

```
bool KheMeetMoveCheck(KHE_MEET meet, KHE_MEET target_meet, int offset);
bool KheMeetMove(KHE_MEET meet, KHE_MEET target_meet, int offset);
```

`KheMeetMove` changes the assignment of `meet` from whatever it is now to `target_meet` at `offset`. If `target_meet` is `NULL`, the move is an unassignment and `offset` is ignored.

These functions follow the usual pattern, returning `true` if the move can be carried out, with `KheMeetMove` actually doing it if so. They return `true` if all of the following conditions hold:

- `KheMeetAssignIsFixed` (see below) returns `false`.
- The `meet` parameter is not a cycle meet.
- The move actually changes the assignment: either `target_meet` is `NULL` and `meet`'s current assignment is non-`NULL`, or `target_meet` is non-`NULL` and `meet`'s current assignment is not to `target_meet` at `offset`.
- The `offset` parameter is in range: if `target_meet` is non-`NULL`, then `offset >= 0` and `offset <= KheMeetDuration(target_meet) - KheMeetDuration(meet);`
- If `target_meet` is non-`NULL`, then the time domain (Section 4.5.4) of `target_meet` is a subset of the time domain of `meet`, allowing for offsets.
- The node rule (Section 4.9) would not be violated if the move was carried out.

If all these conditions hold, then `meet` is said to be *moveable* to `target_meet` at `offset`. Returning `false` when the move changes nothing reflects the practical reality that no solver wants to waste time on such moves.

KHE offers several convenience functions based on `KheMeetMoveCheck` and `KheMeetMove`. For assigning a meet there is

```
bool KheMeetAssignCheck(KHE_MEET meet, KHE_MEET target_meet, int offset);
bool KheMeetAssign(KHE_MEET meet, KHE_MEET target_meet, int offset);
```

Assigning is the same as moving except that `meet` is expected to be unassigned to begin with, and `KheMeetAssignCheck` and `KheMeetAssign` return `false` if not. For unassigning there is

```
bool KheMeetUnAssignCheck(KHE_MEET meet);
bool KheMeetUnAssign(KHE_MEET meet);
```

Unassigning is the same as moving to `NULL`. For swapping there is

```
bool KheMeetSwapCheck(KHE_MEET meet1, KHE_MEET meet2);
bool KheMeetSwap(KHE_MEET meet1, KHE_MEET meet2);
```

A swap is two moves, one of `meet1` to whatever `meet2` is assigned to, and the other of `meet2` to whatever `meet1` is assigned to. It succeeds whenever those two moves succeed.

`KheMeetSwap` has two useful properties. First, exchanging the order of its parameters never affects what it does. Second, the code fragment

```
if( KheMeetSwap(meet1, meet2) )
  KheMeetSwap(meet1, meet2);
```

leaves the solution in its original state whether the swap occurs or not.

A variant of the swapping idea called *block swapping* is offered:

```
bool KheMeetBlockSwapCheck(KHE_MEET meet1, KHE_MEET meet2);
bool KheMeetBlockSwap(KHE_MEET meet1, KHE_MEET meet2);
```

Block swapping is the same as ordinary swapping except that it treats one very special case in a different way: the case when both meets are initially assigned to the same meet, at different offsets which cause them to be adjacent, but not overlapping, in time. In this case, both meets remain assigned to the same meet afterwards, and the later meet is assigned the offset of the earlier one, but the earlier one is not necessarily assigned the offset of the later one. Instead, it is assigned that offset which places it adjacent to the other meet.

For example, when swapping a meet of duration 1 assigned to the first time on Monday with a meet of duration 2 assigned to the second time on Monday, `KheMeetBlockSwap` would move the first meet to the third time on Monday, not the second time. This is much more likely to work well when the two meets have preassigned resources in common. It is the same as an ordinary swap when the meets have the same duration, but it is different when their durations differ. The two useful properties of ordinary swaps also hold for block swaps.

A meet's assignment may be retrieved by calling

```
KHE_MEET KheMeetAsst(KHE_MEET meet);
int KheMeetAsstOffset(KHE_MEET meet);
```

These return the meet that `meet` is assigned to, and the offset into that meet. If there is no assignment, the values returned are `NULL` and `-1`.

Although a meet may only be assigned to one meet, any number of meets may be assigned to a meet, each with its own offset. Functions

```
int KheMeetAssignedToCount(KHE_MEET target_meet);
KHE_MEET KheMeetAssignedTo(KHE_MEET target_meet, int i);
```

visit all the meets that are assigned to a given meet, in an unspecified order which could change when a meet is assigned to or unassigned from `target_meet`. (What actually happens is that an assignment is added to the end, and the hole created by the unassignment of any element other than the last is plugged with the last element.)

Given that a meet can be assigned to another meet at some offset, it follows that a chain of assignments can be built up, from one meet to another and another and so on. Function

```
KHE_MEET KheMeetRoot(KHE_MEET meet, int *offset_in_root);
```

returns the *root* of `meet`: the last meet on the chain of assignments leading out of `meet`. It also sets `*offset_in_root` to the offset of `meet` in its root meet, which is just the sum of the offsets along the assignment path. One function which uses `KheMeetRoot` is

```
bool KheMeetOverlap(KHE_MEET meet1, KHE_MEET meet2);
```

This returns `true` if `meet1` and `meet2` can be proved to overlap in time, because they have the same root meet, and their offsets in that root meet and durations make them overlap. Also,

```
bool KheMeetAdjacent(KHE_MEET meet1, KHE_MEET meet2, bool *swap);
```

returns `true` if `meet1` and `meet2` can be proved to be immediately adjacent in time (but not overlapping), because they have the same root meet, and their offsets in that root meet and durations make them adjacent. If so, it also sets `*swap` to `true` if `meet2` precedes `meet1`, and to `false` otherwise. Again, the meets are required to have the same root meet. This implies that a meet assigned to the end of one cycle meet (Section 4.5.3) is not reported to be adjacent to a meet assigned to the start of the next cycle meet. This is usually what is wanted in practice.

Meet assignments may be fixed and unfixed, by calling

```
void KheMeetAssignFix(KHE_MEET meet);
void KheMeetAssignUnFix(KHE_MEET meet);
bool KheMeetAssignIsFixed(KHE_MEET meet);
```

Any attempt to change the assignment of `meet` will fail while the fix is in place. We often say that the meet is fixed, although strictly speaking it is its assignment that is fixed. When several events are linked by a link events constraint, assigning the meets of all but one of them to the meets of that one and fixing those assignments, or assigning the meets of all of them to some other set of meets and fixing those assignments, has a significant efficiency payoff.

A call to `KheMeetMoveCheck(meet, target_meet, offset)` returns `false` irrespective of `target_meet` and `offset` when `meet` is a cycle meet or its assignment is fixed. Function

```
bool KheMeetIsMovable(KHE_MEET meet);
```

returns `true` when neither of these conditions holds, so that `KheMeetMoveCheck` can be expected to return `true` for at least some target meets and offsets.

Two similar functions follow chains of fixed assignments:

```
KHE_MEET KheMeetFirstMovable(KHE_MEET meet, int *offset_in_result);
KHE_MEET KheMeetLastFixed(KHE_MEET meet, int *offset_in_result);
```

`KheMeetFirstMovable` returns the first meet `m` on the chain of assignments out of `meet` such that `KheMeetIsMovable(m)` holds. If there is no such meet it returns `NULL`. It is used when changing the time assigned to `meet`: this can be done only by changing the assignment of `KheMeetFirstMovable(meet)`, or of a movable meet further along the chain, and this is only possible when the result is non-`NULL`. `KheMeetLastFixed` returns the last meet on the chain of fixed assignments out of `meet`; that is, it follows the chain of assignments out of `meet` until it reaches a meet whose target meet is `NULL` or whose assignment is not fixed, and returns that meet. Its result is always non-`NULL`, and could be a cycle meet. It is used to decide whether two meets are fixed to the same meet, directly or indirectly. In both functions, the result could be `meet` itself, and `*offset_in_result` is set to the offset of `meet` in the result, if non-`NULL`.

### 4.5.3. Cycle meets and time assignment

Even if most meets are assigned to other meets, there must be a way to associate a particular starting time with a meet eventually. Rather than having two kinds of assignment, one to a meet and one to a time, which might conflict, KHE has a special kind of meet called a *cycle meet*. A cycle meet has type `KHE_MEET` as usual, and it has many of the properties of ordinary meets. But it is also associated with a particular starting time (and its domain is fixed to just that time and cannot be changed), and so by assigning a meet to a cycle meet one also assigns a time.

A cycle meet cannot be assigned to another meet; its assignment is fixed to `NULL` and cannot be changed. Cycle meets may be split (their offspring are also cycle meets) and merged. They may even be deleted, but that is not likely to ever be a good idea.

The user cannot create cycle meets directly. Instead, one cycle meet is created automatically whenever a solution is created. The starting time of this *initial cycle meet* is the first time of the cycle, and its duration is the number of times of the cycle. When solving, it is usual to split the initial cycle meet into one meet for each block of times not separated by a meal break or the end of a day, to prevent other meets from being assigned times which cause them to span these breaks. A function for this appears below. When evaluating a fixed solution, it is usual to not split the initial cycle meet, since the other meets already have unchangeable starting times and durations, and splitting the initial cycle meet might prevent them from being assigned to cycle meets.

To find out whether a given meet is a cycle meet, call

```
bool KheMeetIsCycleMeet(KHE_MEET meet);
```

Cycle meets appear on the list of all meets contained in a solution. They are not stored separately anywhere. So the way to find them all is

```
for( i = 0;  i < KheSolnMeetCount(soln);  i++ )
{
  meet = KheSolnMeet(soln, i);
  if( KheMeetIsCycleMeet(meet) )
    visit_cycle_meet(meet);
}
```

However, cycle meets are usually near the front of the list, so this can be optimized as follows:

```
time_count = KheInstanceTimeCount(KheSolnInstance(soln));
durn = 0;
for( i = 0;  i < KheSolnMeetCount(soln) && durn < time_count;  i++ )
{
  meet = KheSolnMeet(soln, i);
  if( KheMeetIsCycleMeet(meet) )
  {
    visit_cycle_meet(meet);
    durn += KheMeetDuration(meet);
  }
}
```

The loop terminates as soon as the total duration of the cycle meets visited reaches the number of times in the instance.

Solutions offer several functions whose results depend on cycle meets. They notice when cycle meets are split, and adjust their results accordingly. Functions

```
KHE_MEET KheSolnTimeCycleMeet(KHE_SOLN soln, KHE_TIME t);
int KheSolnTimeCycleMeetOffset(KHE_SOLN soln, KHE_TIME t);
```

return the unique cycle meet running at time `t`, and the offset of `t` within that meet. Function

```
KHE_TIME_GROUP KheSolnPackingTimeGroup(KHE_SOLN soln, int duration);
```

returns a time group containing the times at which a meet of the given duration may begin. For example, if the initial cycle meet has not been split, `KheSolnPackingTimeGroup(soln, 2)` will contain every time except the last in the cycle; if the initial cycle meet has been split into one meet for each day, it will contain every time except the last in each day; and so on.

As mentioned earlier, when solving it is usual to split the initial cycle meet into one fragment for each maximal block of times not spanning a meal break or end of day. The XML format does not record this information, but solver

```
void KheSolnSplitCycleMeet(KHE_SOLN soln);
```

is able to infer it, as follows. Say that two events of `soln`'s instance are related if they share a required link events constraint with non-zero weight. Find the equivalence classes of the reflexive transitive closure of this relation. For each class, examine the required split events constraints with non-zero weight of the events of the class to determine what durations the meets derived from the events of this class may have. Also determine whether the starting time of the class is preassigned, because one of its events has a preassigned time.

For each permitted duration, consult the required prefer times constraints of non-zero weight of the events of the class to see when its meets of that duration could begin. If a meet `m` with duration 2 can begin at time `t`, there cannot be a break after time `t`; if a meet `m` with duration 3 can begin at time `t`, there cannot be a break after time `t` or after the time following `t`, if any; and so on. Accumulating all this information for all classes determines the set of times which cannot be followed by a break. All other times can be followed by a break, and the initial cycle event is split at these times, and also at times where a break is explicitly allowed by function `KheTimeBreakAfter` from Section 3.4.2.

These functions move a meet to a time, following the familiar pattern:

```
bool KheMeetMoveTimeCheck(KHE_MEET meet, KHE_TIME t);
bool KheMeetMoveTime(KHE_MEET meet, KHE_TIME t);
```

They work by converting `t` into a cycle meet and offset, via functions `KheSolnTimeCycleMeet` and `KheSolnTimeCycleMeetOffset` above, and calling `KheMeetMoveCheck` and `KheMeetMove`. Meets may also be assigned to cycle meets directly, using `KheMeetMove` and the rest. The direct route is more convenient in general solving, since time assignment is then not a special case.

The following functions are also offered:

```
bool KheMeetAssignTimeCheck(KHE_MEET meet, KHE_TIME t);
bool KheMeetAssignTime(KHE_MEET meet, KHE_TIME t);
bool KheMeetUnAssignTimeCheck(KHE_MEET meet);
bool KheMeetUnAssignTime(KHE_MEET meet);
KHE_TIME KheMeetAsstTime(KHE_MEET meet);
```

The first four are wrappers for `KheMeetAssignCheck`, `KheMeetAssign`, `KheMeetUnAssignCheck`, and `KheMeetUnAssign`. `KheMeetAsstTime` follows the assignments of `meet` as far as possible, and if it arrives in a cycle meet, it returns the starting time of `meet`; otherwise it returns `NULL`.

### 4.5.4. Meet domains and bounds

Each meet contains a time group called its *domain*, retrievable by calling

```
KHE_TIME_GROUP KheMeetDomain(KHE_MEET meet);
```

When a meet is assigned a time, that time must be an element of its domain.

More precisely, the solution invariant says that `meet`'s domain must be a superset of the domain of the meet it is assigned to, if any, adjusted for offsets. So, given a chain of assignments beginning at `meet` and ending at a cycle meet, the domain of `meet` must be a superset of the domain of the cycle meet, adjusted for offsets. Since the domain of a cycle meet is a singleton set defining a time, the time assigned to `meet` by this chain of assignments lies in `meet`'s domain.

Meet domains cannot be set directly. Instead, *meet bound* objects influence them. This may seem unnecessarily complicated, but meet bounds have several major advantages over setting domains directly, including allowing restrictions on domains to be added and removed independently, and doing the right thing when meets split and merge.

When meets split and merge, their durations change, and this usually requires a change of domain. For example, a meet of duration 2 cannot be assigned the last time on any day, but if it is split, the fragments may be. Accordingly, a meet bound object stores a whole set of time groups, one for each possible duration. Only one time group influences a meet's domain at any moment: the one corresponding to the meet's current duration. But the others remain in reserve for when the meet's duration is changed by a split or merge.

To create a meet bound object, call

```
KHE_MEET_BOUND KheMeetBoundMake(KHE_SOLN soln,
  bool occupancy, KHE_TIME_GROUP dft_tg);
```

See below for the `occupancy` and `dft_tg` parameters. To delete a meet bound object, call

```
bool KheMeetBoundDeleteCheck(KHE_MEET_BOUND mb);
bool KheMeetBoundDelete(KHE_MEET_BOUND mb);
```

This includes deleting `mb` from each meet it is added to, and is permitted when all of those deletions are permitted, according to `KheMeetDeleteMeetBoundCheck`, defined below.

To retrieve the attributes defined when a meet bound is created, call

```
KHE_SOLN KheMeetBoundSoln(KHE_MEET_BOUND mb);
bool KheMeetBoundOccupancy(KHE_MEET_BOUND mb);
KHE_TIME_GROUP KheMeetBoundDefaultTimeGroup(KHE_MEET_BOUND mb);
```

These are rarely accessed in practice.

As mentioned above, a meet bound is supposed to define a time group for each possible duration. These time groups can be set manually by making any number of calls to

```
void KheMeetBoundAddTimeGroup(KHE_MEET_BOUND mb,
  int duration, KHE_TIME_GROUP tg);
```

Each declares that when `mb` is applied to a meet of the given `duration`, it restricts its domain to be a subset of tg. They may be retrieved by

```
KHE_TIME_GROUP KheMeetBoundTimeGroup(KHE_MEET_BOUND mb, int duration);
```

In both functions, duration may be any positive integer, provided it is not unreasonably large. Two calls to KheMeetBoundAddTimeGroup with the same duration are pointless, but if they occur, the second takes effect. There is no need to specify a time group for every possible duration: durations other than those covered by calls to KheMeetBoundAddTimeGroup are assigned time groups using the occupancy and dft_tg arguments of KheMeetBoundMake. To explain them we need to delve deeper.

There are really two kinds of domains. Those we have dealt with so far may be called *starting-time domains*, because they restrict the starting times of meets. They are appropriate, for example, when expressing prefer times and spread events constraints (which constrain starting times) structurally. The others may be called *occupancy domains*, because they restrict the whole set of times a meet occupies, not just its starting time. For example, a meet of duration 2 should not start immediately before a time when one of its resources is unavailable: the complement of a resource's set of unavailable times is an occupancy domain, not a starting-time domain.

KHE works directly only with starting-time domains, not occupancy domains, so what is needed is a function to convert an occupancy domain into a starting-time domain:

```
KHE_TIME_GROUP KheSolnStartingTimeGroup(KHE_SOLN soln, int duration,
  KHE_TIME_GROUP tg);
```

This returns the set of times that a meet of the given duration could start without any part of it lying outside tg. In other words, it accepts occupancy domain tg and returns the equivalent starting-time domain for a meet of the given duration. When duration is 1, the result is just tg. As duration increases the result shrinks, eventually becoming empty.

To return to meet bounds. When occupancy is false, the time group used by the meet bound for durations not set explicitly is dft_tg. It may be best to set all durations explicitly in this case. When occupancy is true, the value used for any unspecified duration is

```
KheSolnStartingTimeGroup(soln, duration, dft_tg);
```

These values could be passed explicitly, but this way they can be (and are) created only when needed, and there is no need to know the maximum duration. For example, let available_tg be the set of times that some resource is available. Then the meet bound created by

```
KheMeetBoundMake(soln, true, available_tg);
```

ensures that a meet lies entirely within this set of times, whatever duration it has.

Function

```
void KheMeetBoundDebug(KHE_MEET_BOUND mb, int verbosity,
  int indent, FILE *fp);
```

produces the usual debug print of mb onto fp with the given verbosity and indent.

A meet m may have any number of meet bounds. Its domain is the intersection, over all its meet bounds mb, of KheMeetBoundTimeGroup(mb, KheMeetDuration(m)), or the full cycle if none. A meet bound may be added to any number of meets. To add a meet bound, call

```
bool KheMeetAddMeetBoundCheck(KHE_MEET meet, KHE_MEET_BOUND mb);
bool KheMeetAddMeetBound(KHE_MEET meet, KHE_MEET_BOUND mb);
```

These follow the usual form, returning `true` when the addition is permitted (when the change in `meet`'s domain it causes does not violate the solution invariant), with `KheMeetAddMeetBound` actually carrying out the addition in that case. To delete a meet bound from a meet, call

```
bool KheMeetDeleteMeetBoundCheck(KHE_MEET meet, KHE_MEET_BOUND mb);
bool KheMeetDeleteMeetBound(KHE_MEET meet, KHE_MEET_BOUND mb);
```

This too is not always permitted, because it may increase `meet`'s domain, which may violate the solution invariant with respect to the domains of meets assigned to `meet`.

While a meet bound is added to at least one meet, it is not permitted to change its time groups (that is, calls to `KheMeetBoundAddTimeGroup` are prohibited).

To visit the meet bounds added to a given meet, call

```
int KheMeetMeetBoundCount(KHE_MEET meet);
KHE_MEET_BOUND KheMeetMeetBound(KHE_MEET meet, int i);
```

as usual. To visit the meets to which a given meet bound has been added, call

```
int KheMeetBoundMeetCount(KHE_MEET_BOUND mb);
KHE_MEET KheMeetBoundMeet(KHE_MEET_BOUND mb, int i);
```

A meet may have any number of meet bounds; a meet bound may have any number of meets.

It is acceptable for a given meet bound to be added to a meet more than once. In that case, the meet bound appears more than once in the meet's list of meet bounds, and the meet appears more than once in the meet bound's list of meets. A call to `KheMeetDeleteMeetBound` deletes one of these occurrences in each list, not all of them.

When a meet is split, its meet bounds are added to both fragments; and when two meets are merged, one (either) of the two sets of meet bounds is used for the merged meet. Although the meet bounds are the same, the durations change, so the domains may change too. Splits and merges are only permitted when the new domains do not violate the solution invariant.

Adding a meet bound to a meet has some cost in run time, but is fast enough to use within solvers. Meet bound objects are obtained from free lists held in the solution object. Time groups are immutable during solving and may be shared.

When `KheMeetMake` makes a meet derived from an event with a preassigned time, it adds to the meet a meet bound whose default time group is the singleton time group containing that time. No other special arrangements are made for meets derived from preassigned events.

### 4.5.5. Automatic domains

Cycle meets have fixed singleton domains, and meets derived from events can also be assigned fixed domains, based on their durations and the constraints that apply to them.

When solving hierarchically there may be other meets, lying at intermediate levels, for which there is no obvious fixed domain. Instead, the domain of such a meet needs to be the largest domain consistent with the domains of the meets assigned to it: the intersection of those domains, allowing for offsets, or the full set of times if no meets are assigned to it.

As meets are assigned to and unassigned from such a meet, its domain changes automatically. At any moment it does have a domain, however, defined by the rule just given, and this domain must satisfy the solution invariant as usual.

A newly created meet has a fixed domain. To convert it to the automatic form, call

```
bool KheMeetSetAutoDomainCheck(KHE_MEET meet, bool automatic);
bool KheMeetSetAutoDomain(KHE_MEET meet, bool automatic);
```

Assigning `true` to `automatic` gives the meet an automatic domain. This will return `false` if `meet` is a cycle meet, or if `meet` is derived from an event or contains tasks, as discussed below. Assigning `false` returns the meet to a fixed domain. Meet bounds are not affected by automatic domains; what is affected is whether they are used to construct the domain or not.

`KheMeetDomain` returns `NULL` when the meet has an automatic domain. It is important not to mistake this for ‘having no domain,’ a concept not defined by KHE. Function

```
KHE_TIME_GROUP KheMeetDescendantsDomain(KHE_MEET meet);
```

returns the intersection of the domains of the descendants of `meet`, including `meet` itself, adjusted for offsets, or the full time group if there are no such meets or they all have automatic domains. It may thus be used to find the true domain of a meet when `KheMeetDomain` returns `NULL`. It is relatively slow and not intended for use during solving.

When a meet with an automatic domain is split, its two fragments have automatic domains. When two meets are joined, they must both either have automatic domains or not; and if both do, then the joined meet has an automatic domain.

A meet with an automatic domain may not be derived from an event, and it may not have tasks. These two conditions are naturally satisfied by the kinds of meets that need automatic domains. They are necessary, since otherwise KHE would be forced to maintain explicit domains as meets are assigned and unassigned, which would not be efficient. As it is, automatic domains are implemented by having the domain test bypass meets whose domains are automatic, as though each such meet was replaced by the collection of meets assigned to it.

## 4.6. Tasks

A task is a demand for one resource. It is created by calling

```
KHE_TASK KheTaskMake(KHE_SOLN soln, KHE_RESOURCE_TYPE rt,
  KHE_MEET meet, KHE_EVENT_RESOURCE er);
```

The task lies in `soln` and has resource type `rt`. When parameter `meet` is non-`NULL`, the task lies within `meet`, representing a demand for one resource, of type `rt`, at the times when `meet` is running. When `meet` is `NULL`, the task still demands a resource, but at no times, making it useful only as a target for the assignment of other tasks, as explained below.

Parameter `er` may be non-`NULL` only when `meet` is non-`NULL` and derived from some event `e`. In that case, `er` must be one of `e`'s event resources. Its presence causes the task to consider itself to be derived from event resource `er`.

When first created, a meet has no tasks. They must be created separately by calls to `KheTaskMake`. Function `KheSolnMakeCompleteRepresentation` (Section 4.3) does this. When a task's enclosing meet splits, the task splits too. And when two meets merge, their tasks must be compatible and are merged pairwise, inversely to the split.

A task contains an optional *assignment* to another task, and a *resource domain* which restricts the resources it may be assigned to an arbitrary subset of the resources of its type. These attributes are described in detail in later sections.

A task may be deleted by calling

```
void KheTaskDelete(KHE_TASK task);
```

This removes the task from its meet, if any, and unassigns any assignments involving the task.

The back pointer of a task may be set and retrieved by

```
void KheTaskSetBack(KHE_TASK task, void *back);
void *KheTaskBack(KHE_TASK task);
```

as usual, and the usual visit number operations are available:

```
void KheTaskSetVisitNum(KHE_TASK task, int num);
int KheTaskVisitNum(KHE_TASK task);
bool KheTaskVisited(KHE_TASK task, int slack);
void KheTaskVisit(KHE_TASK task);
void KheTaskUnVisit(KHE_TASK task);
```

Function

```
char *KheTaskId(KHE_TASK task);
```

returns a string which is supposed to uniquely identify the task. Most of the time, this is the Id of the task's meet, followed by a dot and an index number identifying the task within the meet (the first task has index 0, the second has index 1, and so on). Some special tasks (e.g. cycle meets) have an Id beginning and ending with `"/"`.

The result of `KheTaskId(task)` is created when `KheTaskId(task)` is first called, and stored in `task` so that it does not have to be created over and over. If it is used only for debugging, as is the intention, there is virtually no cost in running time or memory when debugging is off.

The attributes of a task related to its meet may be retrieved by

```
KHE_MEET KheTaskMeet(KHE_TASK task);
int KheTaskMeetIndex(KHE_TASK task);
int KheTaskDuration(KHE_TASK task);
float KheTaskWorkload(KHE_TASK task);
```

If there is no meet, `KheTaskMeet` returns `NULL` and `KheTaskDuration` and `KheTaskWorkload` return 0. If there is a meet and event resource, `KheTaskWorkload` returns the workload of the task, defined in accord with the XML format's definition to be

$$w(task) = \frac{d(meet)w(er)}{d(e)}$$

where $d(meet)$ is the duration of `task`'s meet, $w(er)$ is the workload of `task`'s event resource, and $d(e)$ is the duration of `task`'s meet's event. See below for the similar and more generally useful `KheTaskTotalDuration` and `KheTaskTotalWorkload` operations. There is also

```
float KheTaskWorkloadPerTime(KHE_TASK task);
```

which returns the workload per time of the task: the workload per time of its event resource, or 0.0 if there is no event resource. Other attributes of a task may be accessed by

```
KHE_SOLN KheTaskSoln(KHE_TASK task);
int KheTaskSolnIndex(KHE_TASK task);
KHE_RESOURCE_TYPE KheTaskResourceType(KHE_TASK task);
KHE_EVENT_RESOURCE KheTaskEventResource(KHE_TASK task);
```

These return the solution containing `task`, the index of `task` in its solution (the value of `i` for which `KheSolnTask(soln, i)` returns `task`), the task's resource type, and its event resource (if any). Index numbers may change when tasks are deleted (what actually happens is that the hole left by the deletion of a task, if not last, is plugged by the last task), so care is needed. Also,

```
bool KheTaskIsPreassigned(KHE_TASK task, KHE_RESOURCE *r);
```

returns `true` when `KheTaskEventResource(task) != NULL` and that event resource has a preassigned resource; `task` is called a *preassigned task* in that case. If `r != NULL`, then `*r` is set to the event resource's preassigned resource if `task` is preassigned, and to `NULL` otherwise.

When a task is preassigned, its domain (Section 4.6.3) contains just the preassigned resource. This prevents any other resource from being assigned, directly or indirectly, to the task. However there is no prohibition on unassigning the task, or on moving it to a task whose domain makes it also preassigned that same resource. Allowing such changes can be awkward; it has been done to permit preassigned tasks to participate in structures such as task trees.

Formerly at this point a function called `KheTaskEquivalent` was introduced which returned `true` when two given tasks are equivalent, in the sense that it makes no difference which of the two is assigned some resource. This function has been withdrawn, because multi-tasks (Section 11.9) now do the same job rather better.

A task may lie in a *tasking*, which is an arbitrary set of tasks (Section 5.5). Functions

```
KHE_TASKING KheTaskTasking(KHE_TASK task);
int KheTaskTaskingIndex(KHE_TASK task);
```

return the tasking containing `task` and the index of `task` in that tasking, or `NULL` and `-1` if the task does not lie in a tasking. Finally,

```
void KheTaskDebug(KHE_TASK task, int verbosity, int indent, FILE *fp);
```

produces the usual debug print of `task` onto `fp` with the given verbosity and indent.

### 4.6.1. Assignment

Just as KHE assigns one meet to another meet, not to a time, so it assigns one task to another task, not to a resource. Accordingly, the assignment operations for tasks parallel those for meets, the main difference being that there is no offset.

The fundamental task assignment operations are

```
bool KheTaskMoveCheck(KHE_TASK task, KHE_TASK target_task);
bool KheTaskMove(KHE_TASK task, KHE_TASK target_task);
```

`KheTaskMove` changes the assignment of `task` to `target_task`. If `target_task` is `NULL`, the move is an unassignment. These operations follow the usual pattern, returning `false` and changing nothing if they cannot be carried out. Here is the full list of reasons why this could happen:

- `task`'s assignment is fixed;
- `task` is a cycle task (Section 4.6.2);
- the move changes nothing: `target_task` is the same as `task`'s current assignment;
- `target_task` is non-`NULL` and the resource domain (Section 4.6.3) of `target_task` is not a subset of the resource domain of `task`.

As for meet moves, returning `false` when the move changes nothing reflects the practical reality that no solver wants to waste time on such moves.

KHE offers several convenience functions based on `KheTaskMoveCheck` and `KheTaskMove`. For assigning a task there is

```
bool KheTaskAssignCheck(KHE_TASK task, KHE_TASK target_task);
bool KheTaskAssign(KHE_TASK task, KHE_TASK target_task);
```

Assigning is the same as moving except that `task` is expected to be unassigned to begin with, and `KheTaskAssignCheck` and `KheTaskAssign` return `false` if not. For unassigning there is

```
bool KheTaskUnAssignCheck(KHE_TASK task);
bool KheTaskUnAssign(KHE_TASK task);
```

Unassigning is the same as moving to `NULL`. For swapping there is

```
bool KheTaskSwapCheck(KHE_TASK task1, KHE_TASK task2);
bool KheTaskSwap(KHE_TASK task1, KHE_TASK task2);
```

A swap is two moves, one of `task1` to whatever `task2` is assigned to, and the other of `task2` to whatever `task1` is assigned to. It succeeds whenever those two moves succeed. As for meet swaps, exchanging the parameters changes nothing, and code fragment

```
if( KheTaskSwap(task1, task2) )
  KheTaskSwap(task1, task2);
```

leaves the solution in its original state whether the swap occurs or not.

A task's assignment may be retrieved by calling

```
KHE_TASK KheTaskAsst(KHE_TASK task);
```

If there is no assignment, `NULL` is returned. Although a task may only be assigned to one task, any number of tasks may be assigned to a task. Functions

```
int KheTaskAssignedToCount(KHE_TASK target_task);
KHE_TASK KheTaskAssignedTo(KHE_TASK target_task, int i);
```

visit all the tasks that are assigned to `target_task`, in an unspecified order which could change when a task is assigned or unassigned from `target_task`. (What actually happens is that an assignment is added to the end, and the hole created by the unassignment of any element other than the last is plugged with the last element.) Functions

```
int KheTaskTotalDuration(KHE_TASK task);
float KheTaskTotalWorkload(KHE_TASK task);
```

return the total duration and workload of `task` and the tasks assigned to it, directly or indirectly. These functions are usually more appropriate than `KheTaskDuration` and `KheTaskWorkload`.

Given that a task can be assigned to another task, a chain of assignments can be built up, from one task to another and so on. Function

```
KHE_TASK KheTaskRoot(KHE_TASK task);
```

returns the *root* of `task`: the last task on the chain of assignments leading out of `task`, possibly `task` itself. The result is never `NULL`, but it could be a cycle task (Section 4.6.2). Function

```
KHE_TASK KheTaskProperRoot(KHE_TASK task);
```

is like `KheTaskRoot` except that it excludes assignments to cycle tasks from the chain of assignments it follows. The result is a cycle task only when `task` itself is a cycle task. Also,

```
bool KheTaskIsProperRoot(KHE_TASK task);
```

returns `true` when `task` is a proper root task: when it is not a cycle task, and is either unassigned or assigned directly to a cycle task.

The next function is offered as an aid to solvers, to help them to decide whether they should try to assign a resource to a given task, or not:

```
bool KheTaskNeedsAssignment(KHE_TASK task);
```

Irrespective of whether `task` is currently assigned or not, this function returns `true` when `task` needs to be assigned a resource in order to avoid a positive cost (hard or soft) among the event resource constraints that apply to it, taking the rest of the current solution as fixed.

This function is mainly useful when repairing solutions. When constructing initial solutions it will often be misleading, since when none of the tasks subject to a limit resources constraint with a positive minimum limit is assigned (as is the case initially), it will say that all of them need assignment, when in fact only some of them (enough to reach the limit) need assignment.

Although the idea of `KheTaskNeedsAssignment` is simple enough, there are several wrinkles, which we explain now by describing the implementation.

First, KheTaskNeedsAssignment finds the proper root of task, as defined just above, and applies itself to that task. This is because the intention is to determine whether task needs assignment to a resource, not to another task, and assignments to other tasks are taken as fixed. It's best, on the whole, if task itself is already a proper root task.

The next step is to check the tasks assigned to task recursively. If any of them need assignment, then so does task. Otherwise, it remains to check task itself.

If task is not derived from an event resource, then it does not need assignment. Otherwise, KheTaskNeedsAssignment calls KheEventResourceNeedsAssignment (Section 3.6.3). If this returns KHE_NO or KHE_YES, KheTaskNeedsAssignment returns false or true immediately. If it returns KHE_MAYBE, then task's monitors are searched for limit resources monitors m with a positive minimum limit, and each is handled as follows.

If m is below the limit, then irrespective of whether or not task is assigned, clearly it needs to be assigned. Otherwise, m is at or above the limit. If task is either unassigned or assigned a resource of no interest to m, then it does not need to be assigned, since other tasks are satisfying m. This leaves one awkward case: m is satisfied, but task is assigned in a way that contributes to that satisfaction, and it may be that if it was not assigned, m would not be satisfied.

We need to work out what would happen if the task was unassigned. We do that by finding the total duration of all descendant tasks of the proper root task that are monitored by m, and comparing their total duration with the amount by which m exceeds its limit.

A similar function is

```
KHE_COST KheTaskAssignmentCostReduction(KHE_TASK task);
```

This returns the total amount by which the cost of the event resource constraints that monitor task (and any tasks assigned, directly or indirectly, to task) reduce when task is assigned. As for KheTaskNeedsAssignment, this function does not care whether task is assigned or not, and it applies itself to the proper root of task, so it is probably best if task is its own proper root. The result could be negative, if assigning task has bad consequences: causing the maximum limit of a limit resources constraint to be exceeded, or when task is subject to a prefer resources constraint with an empty domain. The result is inexact when the cost function is not linear, and also when two tasks are subject to the same limit resources constraint and one is assigned, directly or indirectly, to the other.

Task assignments may be fixed and unfixed as usual, by calling

```
void KheTaskAssignFix(KHE_TASK task);
void KheTaskAssignUnFix(KHE_TASK task);
bool KheTaskAssignIsFixed(KHE_TASK task);
```

The assignment of task cannot be changed while the fix is in place. We often say that the task is fixed, although strictly speaking it is its assignment that is fixed. When several tasks are linked by an avoid split assignments constraint, assigning all but one of them to that one and fixing those assignments, or assigning all of them to some other task and fixing those assignments, has a significant efficiency payoff. Function

```
KHE_TASK KheTaskFirstUnFixed(KHE_TASK task);
```

returns the first task on the chain of assignments out of `task` whose assignment is not fixed (possibly `task`), or `NULL` if none. A solver can change the resource assigned to `task` only by changing the assignment of `KheTaskFirstUnFixed(task)`, or of a task further along the chain.

### 4.6.2. Cycle tasks and resource assignment

Just as meets are assigned times by assigning them, directly or indirectly, to cycle meets, so tasks are assigned resources by assigning them, directly or indirectly, to *cycle tasks*. A cycle task has type `KHE_TASK` as usual, and it has many of the properties of ordinary tasks. But it is also associated with a particular resource (and its domain is fixed to just that resource and cannot be changed), and so by assigning a task to a cycle task one also assigns a resource.

The user cannot create cycle tasks directly. Instead, one cycle task is created automatically for each resource whenever a solution is created. The first `KheInstanceResourceCount` tasks of a solution are its cycle tasks, in the order the resources appear in the instance. Function

```
bool KheTaskIsCycleTask(KHE_TASK task);
```

returns `true` when `task` is a cycle task. Function

```
KHE_TASK KheSolnResourceCycleTask(KHE_SOLN soln, KHE_RESOURCE r);
```

returns the cycle task representing `r` in `soln`.

These functions move a task to a resource, following the familiar pattern:

```
bool KheTaskMoveResourceCheck(KHE_TASK task, KHE_RESOURCE r);
bool KheTaskMoveResource(KHE_TASK task, KHE_RESOURCE r);
```

They first produce a target task. If `r` is non-`NULL` this is the cycle task returned by function `KheSolnResourceCycleTask` above, otherwise it is `NULL`. Then they call `KheTaskMoveCheck` and `KheTaskMove`. Tasks may also be assigned to cycle tasks directly, using `KheTaskMove` etc.

The following functions are also offered:

```
bool KheTaskAssignResourceCheck(KHE_TASK task, KHE_RESOURCE r);
bool KheTaskAssignResource(KHE_TASK task, KHE_RESOURCE r);
bool KheTaskUnAssignResourceCheck(KHE_TASK task);
bool KheTaskUnAssignResource(KHE_TASK task);
KHE_RESOURCE KheTaskAsstResource(KHE_TASK task);
```

The first four are wrappers for `KheTaskAssignCheck`, `KheTaskAssign`, `KheTaskUnAssignCheck`, and `KheTaskUnAssign`. `KheTaskAsstResource` follows the assignments of `task` as far as possible. If it arrives at a cycle task, it returns the resource represented by that task, else it returns `NULL`.

To find the tasks assigned a given resource, either directly or indirectly via other tasks, call

```
int KheResourceAssignedTaskCount(KHE_SOLN soln, KHE_RESOURCE r);
KHE_TASK KheResourceAssignedTask(KHE_SOLN soln, KHE_RESOURCE r, int i);
```

When a resource `r` is assigned to a task, the task and all tasks assigned to it, directly or indirectly, go on the end of `r`'s sequence. When `r` is unassigned from a task, the task and all tasks assigned to it, directly or indirectly, are removed, and the gaps are plugged by tasks taken from the end.

The sequence does not include `r`'s cycle task.

In practice, tasks are of three kinds: *cycle tasks*, which represent resources; *unfixed tasks*, which require assignment to cycle tasks; and *fixed tasks*, whose assignments are fixed to unfixed tasks, relinquishing responsibility for assigning a resource to those tasks. Resource assignment algorithms are concerned with assigning or reassigning unfixed tasks.

### 4.6.3. Task domains and bounds

Each task contains a resource group called its *domain*, retrievable by calling

```
KHE_RESOURCE_GROUP KheTaskDomain(KHE_TASK task);
```

When a task is assigned a resource, that resource must be an element of its domain.

More precisely, the solution invariant says that `task`'s domain must be a superset of the domain of the task it is assigned to, if any. So, given a chain of assignments beginning at `task` and ending at a cycle task, the domain of `task` must be a superset of the domain of the cycle task. Since the domain of a cycle task is a singleton set defining a resource, the resource assigned to `task` by this chain of assignments lies in `task`'s domain.

Task domains cannot be set directly. Instead, *task bound* objects influence them. Task bounds work in the same way as meet bounds, except that the complications introduced by meet splitting are absent. To create a task bound object, call

```
KHE_TASK_BOUND KheTaskBoundMake(KHE_SOLN soln, KHE_RESOURCE_GROUP rg);
```

To delete a task bound object, call

```
bool KheTaskBoundDeleteCheck(KHE_TASK_BOUND tb);
bool KheTaskBoundDelete(KHE_TASK_BOUND tb);
```

This includes deleting `tb` from each task it is added to, and is permitted when all of those deletions are permitted, according to `KheTaskDeleteTaskBoundCheck`, defined below.

To retrieve the attributes defined when a task bound is created, call

```
KHE_SOLN KheTaskBoundSoln(KHE_TASK_BOUND tb);
KHE_RESOURCE_GROUP KheTaskBoundResourceGroup(KHE_TASK_BOUND tb);
```

These are rarely accessed in practice.

A task may have any number of task bounds. Its domain is the intersection, over all its task bounds `tb`, of `KheTaskBoundResourceGroup(tb)`, or the full set of resources of its type if none. A task bound may be added to any number of tasks. To add a task bound, call

```
bool KheTaskAddTaskBoundCheck(KHE_TASK task, KHE_TASK_BOUND tb);
bool KheTaskAddTaskBound(KHE_TASK task, KHE_TASK_BOUND tb);
```

These follow the usual form, returning `true` when the addition is permitted (when the change in `task`'s domain it causes does not violate the solution invariant), with `KheTaskAddTaskBound` actually carrying out the addition in that case. To delete a task bound from a task, call

```
bool KheTaskDeleteTaskBoundCheck(KHE_TASK task, KHE_TASK_BOUND tb);
bool KheTaskDeleteTaskBound(KHE_TASK task, KHE_TASK_BOUND tb);
```

This too is not always permitted, because it may increase `task`'s domain, which may violate the solution invariant with respect to the domains of tasks assigned to `task`.

To visit the task bounds added to a given task, call

```
int KheTaskTaskBoundCount(KHE_TASK task);
KHE_TASK_BOUND KheTaskTaskBound(KHE_TASK task, int i);
```

as usual. To visit the tasks to which a given task bound has been added, call

```
int KheTaskBoundTaskCount(KHE_TASK_BOUND tb);
KHE_TASK KheTaskBoundTask(KHE_TASK_BOUND tb, int i);
```

A task may have any number of task bounds; a task bound may have any number of tasks.

It is acceptable for a given task bound to be added to a task more than once. In that case, the task bound appears more than once in the task's list of task bounds, and the task appears more than once in the task bound's list of tasks. A call to `KheTaskDeleteTaskBound` deletes one of these occurrences in each list, not all of them.

Adding a task bound to a task has some cost in run time, but is fast enough to use within solvers. The implementation parallels the one described previously for meet bounds.

When `KheTaskMake` makes a task derived from an event resource which has a preassigned resource, it adds to the task a task bound whose resource group is the singleton resource group containing that resource. No other special arrangements are made for tasks derived from preassigned event resources.

## 4.7. Resource availability

Evaluators and solvers may wish to know how available a resource is: how much more work it could do without becoming overloaded. This section presents KHE's functions for this.

### 4.7.1. Resource availability functions

The *maximum load* of a resource $r$ is the maximum amount of work that $r$ could do without violating any preassignment or resource constraint of non-zero weight (hard or soft). The *current load* is the amount of work that $r$ is doing now (in a given solution); its *available load* is its maximum load minus its current load. Available load could be negative, in which case some preassignment or resource constraint of non-zero weight must be violated: $r$ is *overloaded*.

Here 'load' refers to either of two measures: the total number of times occupied by the tasks that $r$ is assigned to, or their total workload.

The maximum load is the maximum, over all timetables for $r$ which do not violate any of $r$'s preassignments or resource constraints, of the load of the timetable. These two functions return an estimate of the maximum load, which is usually the true value but may be higher:

```
bool KheResourceMaxBusyTimes(KHE_SOLN soln, KHE_RESOURCE r, int *res);
bool KheResourceMaxWorkload(KHE_SOLN soln, KHE_RESOURCE r, float *res);
```

If they can show that `r`'s maximum load is limited to a non-trivial value, they return `true` and set `*res` to that value. Otherwise they return `false` with `*res` set to `INT_MAX` or `FLT_MAX`. A max busy times value is non-trivial when it is influenced by at least one constraint or unassignable time; a max workload value is also non-trivial when it is influenced by at least one constraint or unassignable time, but it must also be influenced by at least one limit workload constraint (since otherwise it would offer nothing substantially different from the max busy times value).

`KheResourceMaxBusyTimes` and `KheResourceMaxWorkload` depend only on the instance, not on the solution. They are presented as they are because their results are cached in the solution by the first call, ensuring that subsequent calls take almost no time. This is important, because they are slow. (Another way to support caching, which is to calculate these numbers for every resource while finalizing the instance, seems too burdensome for users who do not need them.)

Next come two functions which calculate the current load:

```
int KheResourceBusyTimes(KHE_SOLN soln, KHE_RESOURCE r);
float KheResourceWorkload(KHE_SOLN soln, KHE_RESOURCE r);
```

These return the total duration of the tasks currently assigned `r` in `soln`, and their total workload. They could be implemented by traversing the tasks assigned `r` using functions `KheResourceAssignedTaskCount` and `KheResourceAssignedTask` (Section 4.6.2), but in fact KHE keeps track of their values as tasks are assigned and unassigned, so they are very fast.

Finally come two functions that calculate availability:

```
bool KheResourceAvailableBusyTimes(KHE_SOLN soln, KHE_RESOURCE r, int *res);
bool KheResourceAvailableWorkload(KHE_SOLN soln, KHE_RESOURCE r, float *res);
```

These are the same as `KheResourceMaxBusyTimes` and `KheResourceMaxWorkload`, except they subtract the current load from `*res` when they return `true`. So `*res` could be negative here.

### 4.7.2. How the maximum number of busy times is calculated

This section explains how `KheResourceMaxBusyTimes` is implemented. It aims to find a good limit, not to run quickly.

A resource's maximum number of busy times depends on preassignments and on its avoid unavailable times, limit busy times, cluster busy times, and limit workload monitors (Chapter 6) of non-zero weight, soft as well as hard. It could be very easy to find. For example, it could be the maximum limit of a limit busy times constraint whose time group is the entire cycle. But there are more complicated cases. A cluster busy times constraint might limit the number of busy days, and limit busy times constraints might limit the number of busy times on each day. There might be limits on each day or week, which need to be added to give the overall limit.

Cases like these explain why `KheResourceMaxBusyTimes` is not always exact. It proceeds as follows, for each resource separately. The following applies to one resource, *r*.

A *busy-times avail node*, or just *avail node*, is a set of times together with a non-negative integer limit. Its meaning is that *r* is constrained to be busy for at most the limit number of times from the set, in the sense that any larger number would contradict some preassignment or cause some monitor to have non-zero cost.

At various points in the following description, it says that an avail node $x$ with a given set of times and limit is created. Such statements are to be understood as subject to these rules:

(a) When the limit is 0, the avail node is created as usual, but in addition, all its times are marked as being *inaccessible to r*. When other avail nodes are created later, all inaccessible times are deleted from their time sets (without changing their limits) as the nodes are created, before rules (b) and (c) are applied. This ensures that each inaccessible time appears in exactly one avail node, so that it is sure to appear in every independent set (see below for these), yet not prevent the construction of any independent set.

(b) If $x$'s limit is equal to or larger than its number of times, then $x$ offers no useful information and it is not created. This includes all avail nodes whose set of times is empty.

(c) If several avail nodes containing the same set of times are created for $r$, only one of them, one whose limit is minimal, is kept; the others are either not created at all, or destroyed when a node with a smaller limit is created.

Here is the algorithm for `KheResourceMaxBusyTimes`. Its first phase uses preassignments and $r$'s monitors to create avail nodes, as follows. These four cases are handled first:

1. If preassignments prevent $r$ from being busy at time $t$, then create one avail node containing time set $\{t\}$ and limit 0. This will be the case when the set $S(t, r)$ (Section 4.7.3) is empty.

2. If $r$ is subject to avoid unavailable times monitor $m$ of non-zero weight, then create one avail node for each time $t$ of $m$, containing time set $\{t\}$ and limit 0.

3. If $r$ is subject to limit busy times monitor $m$ of non-zero weight with maximum limit 0, then create one avail node for each time group $g$ of $m$, containing time set $g$ and limit 0.

4. If $r$ is subject to cluster busy times monitor $m$ of non-zero weight with maximum limit 0, then create one avail node for each positive time group $g$ of $m$, containing $g$ and limit 0.

As explained, in these cases the implementation marks times inaccessible as well as creating avail nodes. These cases are the main sources of inaccessible times; by handling them first, we ensure that these times are deleted from time sets, as described in (a) above, in the cases to follow.

Next, the algorithm handles these four cases. It makes two passes over the relevant monitors, because an avail node derived from one can open the way to avail nodes derived from others.

5. If $r$ is subject to limit busy times monitor $m$ of non-zero weight with maximum limit $U > 0$, then create one avail node for each time group $g$ of $m$ with time set $g$ and limit $U$.

6. Suppose that $r$ is subject to cluster busy times monitor $m$ of non-zero weight with maximum limit $U > 0$. For each positive time group $g$ of $m$, define a set of times and a limit as follows. The set of times consists of the times of $g$, minus any inaccessible times. The limit is the number of times in that set, unless there is already an avail node whose times are the times of that set with a smaller limit, in which case the limit is that node's limit. Then define one avail node as follows. Sort the limits of the positive time groups, as just defined, into decreasing order. The avail node's times are the times of the positive time groups, and its limit is the sum of the first $U$ of the sorted limits.

   When history is present, the maximum limit $U$ is replaced by $\max(0, U - x_i)$ in accordance with the meaning of history. If $U < x_i$ the resource is overloaded even if every time group is inactive, but that possibility is not fully taken into account here.

7. Suppose that $r$ is subject to cluster busy times monitor $m$ of non-zero weight with a non-zero minimum limit (including value `false` for the `allow_zero` flag). This may be the same monitor as in the previous point. Then $m$ may be converted into an equivalent cluster busy times monitor $m'$ with the same time groups, but with their polarities reversed, and maximum limit equal to the number of time groups minus the minimum limit. For example, if $m$ says that $r$ must be free on at least 8 out of 28 days, then $m'$ says that $r$ must be busy on at most 20 out of 28 days. So make this conversion (notionally) and apply the previous point. For a proof that the conversion is correct in general, see the end of Section 3.7.14.

   When history is present, suppose that $m$ has $n$ time groups, minimum limit $L$, and history values $a_i$ and $x_i$. According to the conversion, the revised history value is $a_i - x_i$, and the revised limit (now a maximum limit) is $n - L$. So altogether the maximum limit comes to $\max(0, (n - L) - (a_i - x_i))$.

8. Suppose that $r$ is subject to limit workload monitor $m$ of non-zero weight with maximum limit $U$. For each time group $g$ of $m$, proceed as follows. For each time $t$ of $g$, let $L(t, r)$ be a lower bount on the workload per time that $r$ could incur when it is busy at $t$. (For how to find $L(t, r)$, consult Section 4.7.3.) Sort the $L(t, r)$ of $g$ into increasing order, and let $k$ be the largest integer such that the sum of the first $k$ of the $L(t, r)$ does not exceed $U$. Then $k$ is the largest number of times that $r$ can be busy within $g$ without violating $m$, so create an avail node containing the times of $g$ with limit $k$.

This ends the first phase. Its result is a set of avail nodes. Each inaccessible time appears in exactly one avail node, because it is deleted from any others it could appear in.

The second phase uses a graph whose nodes are the first phase's avail nodes. An edge joins two nodes when their sets of times have a non-empty intersection. Any independent set in this graph (any set of nodes such that no two are connected by an edge) defines a larger avail node whose set of times $S$ is the union of its nodes' sets, and whose limit $U$ is the sum of their limits.

Let the set of times of the whole cycle be $C$. Given an independent set with times $S$ and limit $U$, classify the times of $C$ into two disjoint parts: $S$ (including all inaccessible times, as we'll see), and $C - S$. The limits on these are respectively $U$ and $|C - S|$. Adding these gives a maximum limit of

$$U' = U + |C - S|$$

on the number of times that $r$ can be busy.

So the second phase finds an independent set for which $U'$ is as small as possible. This problem is closely related to the problem of finding a maximum independent set, making it NP-complete, so `KheResourceMaxBusyTimes` uses a simple heuristic. It sorts the avail nodes into decreasing time set size order. Then, for each node in that order, it finds one independent set, by starting with that node and then examining each following node in order, adding a node whenever its times do not intersect with the times of the previously added nodes. It then chooses, from these independent sets, one for which $U'$ is minimum, and returns that $U'$ as its result.

Because each inaccessible time appears in exactly one avail node, all avail nodes with limit 0 will be part of the chosen independent set.

To limit running time on large instances, the algorithm exits early when 20 candidate independent sets have been tried since the most recent new best.

At the very end, when $|C - S| > 0$ an avail node is added containing times $C - S$ and limit $|C - S|$. Normally such a node would be rejected for not contributing useful information, but this node is let through so that functions `KheAvailSolverMaxBusyTimesAvailNodeCount` and `KheAvailSolverMaxBusyTimesAvailNode` can report the presence of these 'leftover' times.

The cases covered here are not the only possibilities. Limit active intervals monitors force resources to have some free time, for example. Pairs of nodes whose time sets have a non-empty intersection can still be useful, if the intersection is small. But we have to stop somewhere, and the independent sets suggest that finding the true limit is likely to be an NP-complete problem.

### 4.7.3. How the maximum workload is calculated

This section explains how `KheResourceMaxWorkload` is implemented. But first, we need to calculate two quantities. Recall that each event resource has a non-negative integer workload and a positive integer duration, and that its workload per time is its workload divided by its duration (a floating-point number). We denote the workload per time of event resource $s$ by $w(s)$.

Let $S(t, r)$ be the set of all event resources that could possibly be running at time $t$ and be assigned resource $r$ without violating preassignments or $r$'s resource constraints. Define

$$L(t,r) = \min_{s \in S(t,r)} w(s)$$

and

$$U(t,r) = \max_{s \in S(t,r)} w(s)$$

$L(t, r)$ is a lower bound on the workload per time that $r$ could incur at time $t$, assuming that $r$ is assigned at least one event resource at $t$; and $U(t, r)$ is an upper bound on the workload per time that $r$ could incur at time $t$, assuming that $r$ is assigned at most one event resource at $t$.

When $S(t, r)$ is empty, there is no event resource by whose means $r$ could incur any workload at time $t$. This creates two issues. First, $L(t, r)$ and $U(t, r)$ are undefined then. We leave $L(t, r)$ undefined (it will not be used), but we define $U(t, r)$ to be 0.0, which is reasonable since $r$ cannot be busy at $t$. Second, as Section 4.7.2 mentioned, we create a busy-times avail node with limit 0 containing $t$, and a workload avail node with limit 0.0 containing $t$, again because $r$ cannot be busy at $t$. The presence of these avail nodes ensures that $L(t, r)$ is not used, as it turns out.

To find $S(t, r)$, proceed as follows. Let $S$ be the set of all event resources whose type is the type of $r$. (More detailed domain information, as expressed by prefer resources constraints, is not taken into account.) Make the following definitions:

- Let $S_{uu}$ be the event resources of $S$ that lie in unpreassigned events and are themselves unpreassigned.
- Let $S_{pu}(t)$ be the event resources of $S$ that lie in preassigned events that run during time $t$, and are themselves unpreassigned.
- Let $S_{up}(r)$ be the event resources of $S$ that lie in unpreassigned events and are themselves preassigned $r$.
- Let $S_{pp}(t, r)$ be the event resources of $S$ that lie in preassigned events that run during time $t$, and are themselves preassigned $r$.

Set $S(t, r)$ to $S_{pp}(t, r)$ when $S_{pp}(t, r)$ is non-empty, and to $S_{uu} \cup S_{pu}(t) \cup S_{up}(r)$ otherwise.

We turn now to how `KheResourceMaxWorkload` is implemented. It works in a similar way to `KheResourceMaxBusyTimes`, but using its own set of avail nodes called *workload avail nodes*, which place limits on the total workload per time of the times of the node. In this case the limits are floating-point values rather than integers. The rules for creating these nodes are:

1. Each busy-times avail node with limit 0 is re-used as a workload avail node with limit 0.0. Clearly, if $r$ cannot be busy at time $t$, then its workload per time at time $t$ must also be 0.0.
2. For each time group of each limit workload monitor of non-zero weight with a maximum limit, build one workload avail node containing $S$, the times of the time group minus any inaccessible times, and $U$, the maximum limit. But only include this node if $U < \sum_{t \in S} U(t, r)$, since otherwise it contributes no useful information.

Then independent sets are generated in the usual way, and one is kept which leads to the minimum maximum workload. The formula for the maximum workload of an independent set $S$ with limit $U$ is

$$U' = U + \sum_{t \in C - S} U(t, r)$$

The value is a floating-point number since the $U(t, r)$ are floating-point. This formula follows from the usual division of the set of all times $C$ into those in $S$ (including all inaccessible times) with limit $U$, and the rest, each $t$ of which can contribute up to $U(t, r)$.

Once again, if $|C - S| > 0$, then as a last step an avail node containing times $C - S$ and limit $\sum_{t \in C - S} U(t, r)$ is added, to assist in the reporting of leftover times.

### 4.7.4. Detailed querying of resource availability

KHE offers functions for querying in detail how resource availability is calculated. The first step is to obtain a *resource availability solver* by calling

```
KHE_AVAIL_SOLVER KheSolnAvailSolver(KHE_SOLN soln);
```

Each solution object has one resource availability solver, which is created the first time it is needed (e.g. when `KheSolnAvailSolver` is first called) and stored in the solution object. It uses `soln`'s memory arena, so it will be deleted when `soln` is deleted or made into a placeholder. It uses memory fairly efficiently, recycling what it uses through its own free lists. Given the complexity of the algorithm above, it is not particularly quick. It will run as quickly as possible if all calls that concern a particular resource type `rt` occur together, and all calls that concern a particular resource `r` occur together.

The availability solver offers several query functions. To begin with,

```
bool KheAvailSolverMaxBusyTimes(KHE_AVAIL_SOLVER as,
  KHE_RESOURCE r, int *res);
bool KheAvailSolverMaxWorkload(KHE_AVAIL_SOLVER as,
  KHE_RESOURCE r, float *res);
```

are the same as `KheResourceMaxBusyTimes` and `KheResourceMaxWorkload` except that they query the avail solver about `r`.

The solver recognises these types of avail node:

```
typedef enum {
  KHE_AVAIL_NODE_UNASSIGNABLE_TIME,
  KHE_AVAIL_NODE_UNAVAILABLE_TIME,
  KHE_AVAIL_NODE_LIMIT_BUSY_ZERO,
  KHE_AVAIL_NODE_CLUSTER_BUSY_ZERO,
  KHE_AVAIL_NODE_LIMIT_BUSY,
  KHE_AVAIL_NODE_CLUSTER_BUSY,
  KHE_AVAIL_NODE_CLUSTER_BUSY_MIN,
  KHE_AVAIL_NODE_WORKLOAD
} KHE_AVAIL_NODE_TYPE;
```

These follow the cases given in the previous sections, so should be self-explanatory. Function

```
char *KheAvailNodeTypeShow(KHE_AVAIL_NODE_TYPE type);
```

returns a short string in static memory describing in general terms what a node with the given type was derived from: `"Unavailable time"`, and so on.

To find out how the maximum number of busy times was calculated, call

```
int KheAvailSolverMaxBusyTimesAvailNodeCount(KHE_AVAIL_SOLVER as,
  KHE_RESOURCE r);
void KheAvailSolverMaxBusyTimesAvailNode(KHE_AVAIL_SOLVER as,
  KHE_RESOURCE r, int i, KHE_AVAIL_NODE_TYPE *type, int *limit,
  KHE_TIME_SET *ts, KHE_MONITOR *m);
```

`KheAvailSolverMaxBusyTimesAvailNodeCount` returns the number of avail nodes in the independent set chosen to define the limit, or 0 if the solver was unable to find a non-trivial limit. `KheAvailSolverMaxBusyTimesAvailNode` visits the `i`th avail node of the chosen independent set, returning its type, its limit, its set of times, and the monitor that gave rise to it, or NULL if none. For type KHE_TIME_SET, see Section 5.8.

To do the same job for workload, the calls are

```
int KheAvailSolverMaxWorkloadAvailNodeCount(KHE_AVAIL_SOLVER as,
  KHE_RESOURCE r);
void KheAvailSolverMaxWorkloadAvailNode(KHE_AVAIL_SOLVER as,
  KHE_RESOURCE r, int i, KHE_AVAIL_NODE_TYPE *type, int *limit,
  KHE_TIME_SET *ts, KHE_MONITOR *m);
```

In this case *type is always KHE_AVAIL_NODE_WORKLOAD and *m is never NULL. Again, the count is 0 if the solver could not find a non-trivial limit.

The avail solver does not report current or available load. Current load may be found by calling KheResourceBusyTimes and KheResourceWorkload, as explained earlier. For a detailed analysis, call functions KheResourceAssignedTaskCount and KheResourceAssignedTask (Section 4.6.2) to visit the tasks assigned r, and KheTaskDuration and KheTaskWorkload to find their load. Available load is just maximum load minus current load.

## 4.8. Marks and paths

Suppose you want to make the best time assignment for a meet. You try each assignment in turn, remembering the best so far and its solution cost, then finish off by re-doing the best one.

Now suppose the alternative operations are more complicated. For example, they might be Kempe meet moves (Section 10.2.2), each consisting of an unpredictable number of time assignments. The same program structure works, but undoing one alternative is much more complicated. Marks and paths solve these kinds of problems.

A *mark* is like a waymark on a journey: it marks a particular point, or state, that a solution has reached. It is created and deleted by

```
KHE_MARK KheMarkBegin(KHE_SOLN soln);
void KheMarkEnd(KHE_MARK mark, bool undo);
```

These operations must be called in matching pairs: for each call to KheMarkBegin there must be one later call to KheMarkEnd with the same mark object. Between these two calls there may be other calls to KheMarkBegin and KheMarkEnd, and those calls must occur in matching pairs.

KheMarkEnd deletes the mark created by the corresponding KheMarkBegin. If its undo parameter is true, it also undoes all operations on soln since the corresponding KheMarkBegin, returning the solution to its state when that call was made. Another way to undo is

```
void KheMarkUndo(KHE_MARK mark);
```

It undoes all operations on soln since the call to KheMarkBegin which returned mark, only without removing mark. It can only be called when it would be legal to call KheMarkEnd with the same value of mark: when mark is the mark returned most recently by a call to KheMarkBegin, apart from marks already completed by KheMarkEnd.

When undoing by either method, the resulting value of the solution may differ from the original in its naturally nondeterministic aspects, such as the set of unmatched demand monitors (but not their number), and the order of elements in arrays representing sets (of meets, etc.). But as a solution it will be the same as the original. KHE objects deleted while doing and re-created while undoing are re-created with the same memory addresses as the originals.

At any time between `KheMarkBegin` and its corresponding `KheMarkEnd`, functions

```
KHE_SOLN KheMarkSoln(KHE_MARK mark);
KHE_COST KheMarkSolnCost(KHE_MARK mark);
```

may be called to obtain `mark`'s solution and the solution cost at the time `KheMarkBegin` was called. Exploring the result of `KheMarkSoln` will reveal the solution as it is now, not as it was when `KheMarkBegin` was called.

All mark objects share access to one sequence, stored in the solution object, of records of the operations performed on the solution since the first call to `KheMarkBegin` whose corresponding `KheMarkEnd` has not occurred yet. When undoing, these operations are undone in reverse order and removed from the sequence. All changes to solutions, including changes to back pointers, are recorded, except changes to visit numbers, since undoing them would be inappropriate. A mark object holds a pointer to the solution object, its cost when `KheMarkBegin` was called, an index into the sequence saying where to stop undoing, and a sequence of paths, described below.

Changes to monitors (attaching, unattaching, grouping, ungrouping, and `SetCeiling` and similar operations) are not recorded in the operation sequence, and so are not undone by `KheMarkEnd`. The author is unsure whether this counts as a defect in KHE or not. Certainly, it removes any guarantee that undoing will ensure that a solution re-attains its previous cost, because if monitors have changed and the changes are not undone, different costs will be reported. Solvers that make changes to monitors should probably change them back again before they return, even if other changes to the solution, recording a successful outcome for the solver, remain in place.

A *path* is like the route between two waymarks. A path is created by calling

```
KHE_PATH KheMarkAddPath(KHE_MARK mark);
```

and represents the route from the state of `mark`'s solution represented by `mark` to the state of that solution at the moment `KheMarkAddPath` is called. Concretely, a path holds a copy of the shared sequence of operations, taken at the moment `KheMarkAddPath` is called, from its mark's index to the end. As well as being returned, a path is stored in its mark and deleted by that mark's `KheMarkEnd`, if it has not been deleted before then. A path is meaningless after its mark ends.

In practice, this helper function may be more useful than `KheMarkAddPath`:

```
KHE_PATH KheMarkAddBestPath(KHE_MARK mark, int k);
```

It is written using the more basic functions given below. Its behaviour is equivalent to calling `KheMarkAddPath(mark)`, then sorting `mark`'s paths into increasing cost order, then deleting paths from the end as required to ensure that not more than `k` paths are kept. But rather than following this description literally, it uses an optimized method that only calls `KheMarkAddPath(mark)` when the resulting path would be one of those kept; it returns the new path in that case, and `NULL` otherwise. For example, `KheMarkAddBestPath(mark, 1)` saves only the best path, and only creates a path when it would be a new best.

Any number of paths may be stored in a mark, and they may be visited using

```
int KheMarkPathCount(KHE_MARK mark);
KHE_PATH KheMarkPath(KHE_MARK mark, int i);
```

as usual, and sorted by calling

```
void KheMarkPathSort(KHE_MARK mark,
  int(*compar)(const void *, const void *));
```

where `compar` is a function suited to passing to `qsort` when sorting an array of `KHE_PATH` objects. One such function, `KhePathIncreasingSolnCostCmp`, is provided, such that after calling

```
KheMarkPathSort(mark, &KhePathIncreasingSolnCostCmp);
```

the paths will be sorted into increasing solution cost order, so that the path with the smallest solution cost comes first. The following operations on paths are also available:

```
KHE_SOLN KhePathSoln(KHE_PATH path);
KHE_COST KhePathSolnCost(KHE_PATH path);
KHE_MARK KhePathMark(KHE_PATH path);
void KhePathDelete(KHE_PATH path);
void KhePathRedo(KHE_PATH path);
```

`KhePathSoln` returns `path`'s solution, and `KhePathSolnCost` returns the solution cost at the moment the path was created by `KheMarkAddPath`. `KhePathMark` returns `path`'s mark. `KhePathDelete` deletes `path`, including removing it from its mark. `KheMarkEnd` calls `KhePathDelete` for each of its paths; once a mark is deleted, its paths have no meaning.

When `KhePathRedo(path)` is called, the solution must be in the state it was in when `path`'s mark was created. It redoes `path`, without deleting or otherwise disturbing its mark, so that the state after it returns is the state at the end of `path`. This is the only way to redo a path, and because it checks that it starts from the same state that the path started from originally, it guarantees that the operations executed while redoing the path cannot fail. KHE objects created along the path and deleted during the undo (which must have occurred in order to return the solution to its original state) are re-created during the redo with the same memory addresses as the originals.

One application of marks and paths is the conversion of a sequence of operations into an *atomic sequence*, one which is either carried out completely or not at all:

```
mark = KheMarkBegin(soln);
success = SomeSequenceOfOperations(...);
KheMarkEnd(mark, !success);
```

If the sequence of operations is successful, it remains in place; otherwise the unsuccessful sequence, or whatever part of it was completed before failure occurred, is undone. Similarly,

```
mark = KheMarkBegin(soln);
SomeSequenceOfOperations(...);
KheMarkEnd(mark, KheSolnCost(soln) >= KheMarkSolnCost(mark));
```

keeps the sequence of operations if it reduces the cost of the solution, but not otherwise.

Another application is the coordination of complex searches, such as tree searches, which try many alternatives and keep the best. Before the search begins, create a mark, and pass it to the search function, so that whenever it finds a worthwhile state it can record it in the mark by calling `KheMarkAddPath` or `KheMarkAddBestPath`. (If the initial state is a valid solution, one that the rest of the search is trying to improve on, call `KheMarkAddPath` immediately after `KheMarkBegin`.) Within the search function, create other marks as required so that subtrees can be undone by calling `KheMarkEnd(sub_mark, true)`. At the end, all worthwhile states are paths in the original mark, where they can be examined, sorted, or whatever—like this, perhaps:

```
if( KheMarkPathCount(mark) > 0 )
  KhePathRedo(KheMarkPath(mark, 0));
KheMarkEnd(mark, false);
```

when only the best path is kept. If it is safe to redo that path, there can be nothing to undo.

Marks and paths have been implemented carefully, and their running time is small. Indeed, it is usually faster to use marks and undoing to return a solution to a previous state, than to use operations opposite to the originals. This is because `KheMarkBegin` and `KheMarkEnd` call `KheSolnMatchingMarkBegin` and `KheSolnMatchingMarkEnd` (Section 7.2), and because there is no need to check that an undo is safe, as there is when carrying out an opposite operation.

## 4.9. The solution invariant

Here is the condition, called the solution invariant, that every solution always satisfies. The last three rules relate to data types introduced in Chapter 5.

1. The *meet rule*: if `meet` is assigned to `target_meet` at offset `offset`, then:

   (a) The value of `offset` is at least 0 and at most the duration of `target_meet` minus the duration of `meet`;

   (b) The time domain of `target_meet`, shifted right `offset` places, is a subset of the time domain of `meet`;

2. The *task rule*: if `task` is assigned to `target_task`, then the resource domain of `target_task` is a subset of the resource domain of `task`.

3. The *cycle rule*: the parent links of nodes may not form a cycle.

4. The *node rule*: if meet `meet` is assigned to meet `target_meet` and lies in node `n`, then `n` has a parent node and `target_meet` lies in that parent node.

5. The *layer rule*: every node of a layer has the same parent node as the layer.

No sequence of operations can bring a solution to a state that violates this invariant.

# Chapter 5. Extra Types for Solving

This chapter introduces several types of objects that help with solving. Four of them (*nodes*, *layers*, *zones*, and *taskings*) are integral to solutions, being copied when they are copied, for example. But they are not part of the XML model, so their use is optional. Nodes and layers together define the *layer tree*, a data structure invented by the author [7] for use in time assignment. Zones help to make time assignments regular, and taskings are used in resource assignment.

## 5.1. Layer trees

The layer tree is a data structure for organizing solutions during time assignment. It supports *hierarchical timetabling*, in which meets are timetabled together into small timetables called *tiles*, the tiles are timetabled together, and so on until a complete timetable is produced. Layer trees are recommended when solving general instances, since they gracefully handle awkward cases, such as linked events whose durations differ.

Layer trees are made of *nodes*, which form a tree (actually, a forest). Each node has an optional *parent node*. The nodes with a given parent are its *children*.

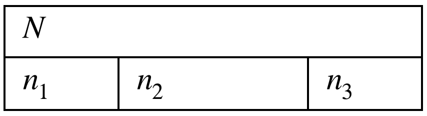
Within each node lie any number of meets. The *node rule*, part of the solution invariant (Section 4.9), imposes a structure on how the meets of nodes may be assigned: if `meet` is assigned to `target_meet` and lies in node `n`, then `n` has a parent node and `target_meet` lies in that parent node. A layer tree usually has a single root node containing the cycle meets, called the *cycle node*. If there is a cycle node, the node rule guarantees that if every non-cycle meet lying in a node is assigned to some meet, then every such meet is assigned a time.

A meet may lie in at most one node. When using layer trees, it is conventional for every meet to lie in a node except when it has received its final assignment. Omitting meets from nodes hides them from time assignment algorithms, which typically access meets via nodes.

When a meet splits, it is replaced in its node (if any) by the two fragments. When two meets merge, they must lie in the same node (or none), and they are replaced by the merged meet.

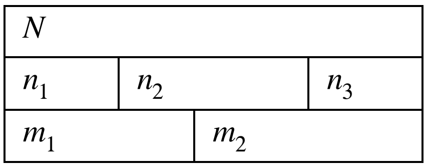
A *layer* is a subset of the children of some node with the property that none of the meets in the nodes of the layer may overlap in time. This could be for any reason, but it is usually because their meets all share a preassigned resource which possesses a required avoid clashes constraint. The property is not enforced by KHE; it is merely a convention.

Here are some examples of layer trees. The first has four nodes, $N$, $n_1$, $n_2$, and $n_3$. The $n_i$ share a layer and are children of $N$, so their meets must be assigned to meets of $N$ and should not overlap in time:



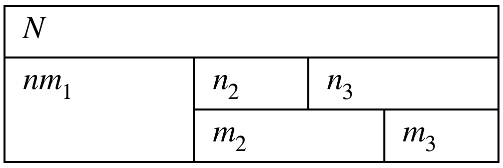
The nodes are shown as rectangles. The horizontal direction represents time. That the $n_i$ share a layer is indicated by placing them alongside each other, and that they are children of $N$ is indicated by placing them vertically below $N$.

In the next example, $N$ has five children, lying in two layers, $\{n_1, n_2, n_3\}$ and $\{m_1, m_2\}$:



This could arise when one group of students attends the $n_i$ while another group attends the $m_i$.

Finally, here is an example where a node lies in two layers (but still has only one parent):



The two layers $\{nm_1, n_2, n_3\}$ and $\{nm_1, m_2, m_3\}$ both contain node $nm_1$. This case arises naturally when an event (or a set of linked events) is attended by two groups of students, so that their timetables coincide at that event but may differ elsewhere.

The key operation in hierarchical timetabling is the assignment of the meets of the children of a node to the meets of the node, so that meets that share a layer do not overlap. One way to construct a timetable is to build a layer tree containing every meet, whose root node contains the cycle meets, and apply this operation at each node, visiting the nodes in postorder (bottom up).

## 5.2. Nodes

To create a layer tree node, initially with no meets, no parent, and no children, call

```
KHE_NODE KheNodeMake(KHE_SOLN soln);
```

Its back pointer may be accessed by

```
void KheNodeSetBack(KHE_NODE node, void *back);
void *KheNodeBack(KHE_NODE node);
```

and its visit number by

```
void KheNodeSetVisitNum(KHE_NODE n, int num);
int KheNodeVisitNum(KHE_NODE n);
bool KheNodeVisited(KHE_NODE n, int slack);
void KheNodeVisit(KHE_NODE n);
void KheNodeUnVisit(KHE_NODE n);
```

as usual, and its other attributes may be retrieved by calling

```
KHE_SOLN KheNodeSoln(KHE_NODE node);
int KheNodeSolnIndex(KHE_NODE node);
```

`KheNodeSolnIndex` returns the *index* of `node`: the value of `i` for which `KheSolnNode(soln, i)` (Section 4.2.7) returns `node`. The index may change when nodes are deleted (what actually happens is that the hole left by the deletion of a node, if not last, is plugged by the last node) so care is needed if indexes are stored. To visit the nodes of a solution in increasing index order, use functions `KheSolnNodeCount` and `KheSolnNode` from Section 4.2.7. To delete a node, call

```
bool KheNodeDeleteCheck(KHE_NODE node);
bool KheNodeDelete(KHE_NODE node);
```

This deletes all parent-child links involving `node`, and deletes all meets from `node` (but does not delete them). It is permitted only when no meets assigned to `node`'s meets lie in a node.

To make one node the parent of another, call

```
bool KheNodeAddParentCheck(KHE_NODE child_node, KHE_NODE parent_node);
bool KheNodeAddParent(KHE_NODE child_node, KHE_NODE parent_node);
```

These abort if `child_node` already has a parent; they return `false` and do nothing when adding the link would cause a cycle. To delete a parent-child link, call

```
bool KheNodeDeleteParentCheck(KHE_NODE child_node);
bool KheNodeDeleteParent(KHE_NODE child_node);
```

Deletion is permitted only when none of the meets of `child_node` is assigned. The gap created in the list of child nodes of the parent node by the deletion of `child_node` is filled by shuffling the following nodes down one place. To retrieve the parent of a node, call

```
KHE_NODE KheNodeParent(KHE_NODE node);
```

This returns `NULL` when `node` has no parent. Children are added and deleted, obviously, by adding and deleting parents. Functions

```
int KheNodeChildCount(KHE_NODE node);
KHE_NODE KheNodeChild(KHE_NODE node, int i);
```

visit a node's children in the usual way. There are also

```
bool KheNodeIsDescendant(KHE_NODE node, KHE_NODE ancestor_node);
bool KheNodeIsProperDescendant(KHE_NODE node, KHE_NODE ancestor_node);
```

`KheNodeIsDescendant` returns `true` when `node` is a descendant of `ancestor_node`, possibly `ancestor_node` itself; `KheNodeIsProperDescendant` returns `true` when `node` is a proper descendant of `ancestor_node`, that is, a descendant other than `ancestor_node` itself. They work in the obvious way, searching upwards from `node` for `ancestor_node`.

Several helper functions for rearranging nodes appear in Section 9.5. They are often more useful than `KheNodeAddParent` and `KheNodeDeleteParent`. Some of them call

```
void KheNodeSwapChildNodesAndLayers(KHE_NODE node1, KHE_NODE node2);
```

This function makes all the child nodes and child layers of `node1` into child nodes and child layers of `node2` and vice versa. The child nodes and layers are the exact same objects as before, stored in the same order as before; only their parent node is changed. Any assigned meets lying in child nodes of either node are unassigned (otherwise the node rule would be violated).

A meet may lie in at most one node, and function `KheMeetNode` (Section 4.5) returns the node containing a given meet, if any. To add a meet to a node and delete it, the operations are

```
bool KheNodeAddMeetCheck(KHE_NODE node, KHE_MEET meet);
bool KheNodeAddMeet(KHE_NODE node, KHE_MEET meet);
bool KheNodeDeleteMeetCheck(KHE_NODE node, KHE_MEET meet);
bool KheNodeDeleteMeet(KHE_NODE node, KHE_MEET meet);
```

`KheNodeAddMeetCheck` and `KheNodeAddMeet` abort if `meet` already lies in a node, and return `false` if it is already assigned to a meet not in the parent of `node`. `KheNodeDeleteMeetCheck` and `KheNodeDeleteMeet` abort if `meet` does not lie in `node`, and return `false` if a meet from a child of `node` is assigned to `meet`. Functions

```
int KheNodeMeetCount(KHE_NODE node);
KHE_MEET KheNodeMeet(KHE_NODE node, int i);
```

visit the meets of a node in the usual way. The order that meets are stored in nodes and returned by these functions is arbitrary, and the user can change it by calling

```
void KheNodeMeetSort(KHE_NODE node,
  int(*compar)(const void *, const void *))
```

where `compar` is a comparison function suitable for passing to `qsort`. Two such comparison functions are supplied. One sorts the meets into decreasing duration order:

```
int KheMeetDecreasingDurationCmp(const void *p1, const void *p2);
```

Here is the implementation:

```
int KheMeetDecreasingDurationCmp(const void *p1, const void *p2)
{
  KHE_MEET meet1 = * (KHE_MEET *) p1;
  KHE_MEET meet2 = * (KHE_MEET *) p2;
  if( KheMeetDuration(meet1) != KheMeetDuration(meet2) )
    return KheMeetDuration(meet2) - KheMeetDuration(meet1);
  else
    return KheMeetIndex(meet1) - KheMeetIndex(meet2);
}
```

Ties are broken by referring to the meet index. The other sorts meets by increasing value of the index of the target meet, breaking ties by increasing value of the target offset:

```
int KheMeetIncreasingAsstCmp(const void *p1, const void *p2)
```

This brings together meets whose assignments place them adjacent in time. Unassigned meets appear after assigned ones, but are not themselves sorted into any particular order.

Unlike cycle meets, which are different behind the scenes from other meets, cycle nodes are just ordinary nodes whose meets happen to be cycle meets. Accordingly, function

```
bool KheNodeIsCycleNode(KHE_NODE node);
```

merely returns `true` if `node` contains at least one meet, and its first meet is a cycle meet.

The total duration, assigned duration, and demand of the meets of `node` are returned by

```
int KheNodeDuration(KHE_NODE node);
int KheNodeAssignedDuration(KHE_NODE node);
int KheNodeDemand(KHE_NODE node);
```

The duration is kept up to date and stored in the node, so `KheNodeDuration` costs almost nothing. The other two have to sum values stored in the meets, which is slower but still fast.

Following the pattern laid down in Section 1.4, function

```
bool KheNodeSimilar(KHE_NODE node1, KHE_NODE node2);
```

returns `true` when `node1` and `node2` are similar: when they contain similar events. The exact rule is as follows. If `node1` and `node2` are the same node, they are similar. A node is *admissible* if all of its meets are derived from events, and for each event found among those meets, all of the meets of that event lie in the node. Thus, an admissible node can be considered as a set of events. Two distinct nodes are similar if they are admissible and each event in one can be matched up with a similar event in the other. The definition of similarity for events is as in Section 3.6.2.

A similar property is *regularity* (Section 5.4). Two nodes are regular when they are the same node or contain meets of equal durations and equal time domains. Function

```
bool KheNodeRegular(KHE_NODE node1, KHE_NODE node2, int *regular_count);
```

returns `true` when `node1` and `node2` are regular, and `false` otherwise. Either way, it reorders the meets of both nodes so that corresponding meets have equal durations and equal time domains, as far as possible; `*regular_count` is the number of such pairs. (So `true` is returned when `*regular_count` equals the number of meets in both nodes.)

Another function useful to solvers is

```
int KheNodeResourceDuration(KHE_NODE node, KHE_RESOURCE r);
```

This returns the total duration of meets in `node` and its descendants that contain a preassignment of `r`. If a meet contains two such preassignments, its duration is only counted once.

To make a debug print of `node` onto file `fp` with a given verbosity and indent, call

```
void KheNodeDebug(KHE_NODE node, int verbosity, int indent, FILE *fp);
```

Verbosity 1 prints just the node index number, verbosity 2 adds the duration and meets, verbosity 3 adds the node's children, and verbosity 4 adds its segments. There is also

```
void KheNodePrintTimetable(KHE_NODE node, int cell_width,
  int indent, FILE *fp);
```

which prints a timetable showing the meets of `node` across the top, and the assigned meets lying in child nodes of `node` on subsequent lines, one line per child layer. (So `node` needs to have child layers when it is called.) Parameter `cell_width` is the width of each cell, in characters.

## 5.3. Layers

A *layer* (not to be confused with the resource layer of Section 3.5.4) is a subset of the child nodes of some node. The intention is that the meets of a layer's nodes should not overlap in time, although this condition is not enforced.

For a given node there are two sets of layers of interest: the node's *parent layers*, which are the layers it lies in (it may lie in several), and its *child layers*, which are subsets of its child nodes. A node is a member of all of its parent layers and none of its child layers.

To create a layer of children of a given parent node, initially with no nodes, call

```
KHE_LAYER KheLayerMake(KHE_NODE parent_node);
```

It has a back pointer and a visit number, accessed by

```
void KheLayerSetBack(KHE_LAYER layer, void *back);
void *KheLayerBack(KHE_LAYER layer);

void KheLayerSetVisitNum(KHE_LAYER layer, int num);
int KheLayerVisitNum(KHE_LAYER layer);
bool KheLayerVisited(KHE_LAYER layer, int slack);
void KheLayerVisit(KHE_LAYER layer);
void KheLayerUnVisit(KHE_LAYER layer);
```

as usual. Functions

```
KHE_NODE KheLayerParentNode(KHE_LAYER layer);
int KheLayerParentNodeIndex(KHE_LAYER layer);
```

return the parent node of `layer` and the value of `i` for which `KheNodeChildLayer(KheLayerParentNode(layer), i)` returns `layer`. For convenience the solution containing it can be found by

```
KHE_SOLN KheLayerSoln(KHE_LAYER layer);
```

To delete the layer (but not its nodes), call

```
void KheLayerDelete(KHE_LAYER layer);
```

To add and delete a child node of `parent_node` from a layer, call

```
void KheLayerAddChildNode(KHE_LAYER layer, KHE_NODE node);
void KheLayerDeleteChildNode(KHE_LAYER layer, KHE_NODE node);
```

`KheLayerAddChildNode` aborts if `node`'s parent node and `layer`'s parent node are different, and `KheLayerDeleteChildNode` aborts if `node` does not lie in `layer`; otherwise, both succeed. When a child node is deleted from a layer, all later nodes are shuffled up one place to fill the gap. To visit the child nodes of a layer, call

```
int KheLayerChildNodeCount(KHE_LAYER layer);
KHE_NODE KheLayerChildNode(KHE_LAYER layer, int i);
```

To sort the child nodes of a layer, call

```
void KheLayerChildNodesSort(KHE_LAYER layer,
  int(*compar)(const void *, const void *));
```

where `compar` is a function suited to passing to `qsort` when it sorts an array of nodes.

Although much about layers is taken on trust, the *layer rule* is enforced: the parent node of each node of a layer equals the parent node of the layer. When the parent of a node is changed, the node is deleted from all the layers it lies in.

The usual reason why nodes are placed into a layer together is because their meets have one or more preassigned resources in common, and the resources have hard avoid clashes constraints, preventing the meets from overlapping in time. To document this reason when it applies, a layer contains a set of resources. To add and delete a resource from this set, the functions are

```
void KheLayerAddResource(KHE_LAYER layer, KHE_RESOURCE r);
void KheLayerDeleteResource(KHE_LAYER layer, KHE_RESOURCE r);
```

To visit this set of resources, the functions are

```
int KheLayerResourceCount(KHE_LAYER layer);
KHE_RESOURCE KheLayerResource(KHE_LAYER layer, int i);
```

There is no check that these resources are actually preassigned to the layer's meets.

When `KheLayerMake(parent_node)` is called, the resulting layer becomes a *child layer* of `parent_node`. To visit the child layers of a given node, call

```
int KheNodeChildLayerCount(KHE_NODE parent_node);
KHE_LAYER KheNodeChildLayer(KHE_NODE parent_node, int i);
```

Also,

```
void KheNodeChildLayersSort(KHE_NODE parent_node,
  int(*compar)(const void *, const void *));
```

sorts the child layers of `parent_node`, using `compar` (a function suited to passing to `qsort`) as the comparison function, and

```
void KheNodeChildLayersDelete(KHE_NODE parent_node);
```

deletes all the child layers of `parent_node`, without deleting any nodes.

When `KheLayerAddChildNode(layer, node)` is called, `layer` becomes a *parent layer* of `node`. To visit a node's parent layers, call

```
int KheNodeParentLayerCount(KHE_NODE child_node);
KHE_LAYER KheNodeParentLayer(KHE_NODE child_node, int i);
```

It is important to allow multiple parent layers in this way. For example, suppose there is one layer for the meets attended by Year 12 students and another for the meets attended by Year 11 students. If one of the Year 11 events is linked to one of the Year 12 events by a link events constraint, then there will usually be a single node whose subtree contains the meets of both events, and this node will appear in both layers. Function

```
bool KheNodeSameParentLayers(KHE_NODE node1, KHE_NODE node2);
```

returns `true` when `node1` and `node2` have the same parent layers.

Functions

```
int KheLayerDuration(KHE_LAYER layer);
int KheLayerMeetCount(KHE_LAYER layer);
```

return the total duration of `layer`'s child nodes and the number of meets in them. These values are stored in the layer and kept up to date as it changes, in the expectation that they will be used when sorting layers. Similarly,

```
int KheLayerAssignedDuration(KHE_LAYER layer);
int KheLayerDemand(KHE_LAYER layer);
```

return the total duration of the assigned meets of `layer`'s child nodes, and their total demand. These values are calculated on demand, not stored, so the functions are a bit slower. There are also set operations, implemented efficiently using bit vectors of node indexes:

```
bool KheLayerEqual(KHE_LAYER layer1, KHE_LAYER layer2);
bool KheLayerSubset(KHE_LAYER layer1, KHE_LAYER layer2);
bool KheLayerDisjoint(KHE_LAYER layer1, KHE_LAYER layer2);
bool KheLayerContains(KHE_LAYER layer, KHE_NODE node);
```

These return `true` if `layer1` and `layer2` contain the same nodes, if every node of `layer1` is a node of `layer2`, if `layer1` and `layer2` contain no nodes in common, and if `node` lies in `layer`.

Three functions offer more complex comparisons between layers:

```
bool KheLayerSame(KHE_LAYER layer1, KHE_LAYER layer2, int *same_count);
bool KheLayerSimilar(KHE_LAYER layer1, KHE_LAYER layer2,
  int *similar_count);
bool KheLayerRegular(KHE_LAYER layer1, KHE_LAYER layer2,
  int *regular_count);
```

These work in the same general way: they reorder the nodes in the two layers so that the first `*same_count` (etc.) nodes in `layer1` are equivalent in some way to the corresponding nodes in `layer2`, returning `true` if this accounts for all the nodes in both layers. `KheLayerSame` aligns nodes that are the identical same node; `KheLayerSimilar` aligns nodes that are similar, according to `KheNodeSimilar` from Section 5.2; and `KheLayerRegular` aligns nodes that are regular, according to `KheNodeRegular` from Section 5.2. If `layer1` and `layer2` are the same layer, all three functions return `true` and set their count variable to the number of nodes in the layer. If some nodes are shared between the two layers, these are always considered equivalent and they always appear first after the layers are ordered.

These functions are implemented by calls to a more general function:

```
bool KheLayerAlign(KHE_LAYER layer1, KHE_LAYER layer2,
  bool (*node_equiv)(KHE_NODE node1, KHE_NODE node2), int *count);
```

which does the same kind of alignment, first bringing identical nodes to the front of both layers, then ordering the other nodes, calling `node_equiv` to decide whether two nodes are equivalent.

Two layers that share a common parent node may be merged:

```
void KheLayerMerge(KHE_LAYER layer1, KHE_LAYER layer2, KHE_LAYER *res);
```

The layers are deleted and replaced by layer `*res`, containing the nodes and resources of `layer1` and `layer2`. It makes sense to merge, for example, when one layer is a subset of the other.

As an aid to debugging, KHE offers function

```
void KheLayerDebug(KHE_LAYER layer, int verbosity, int indent, FILE *fp);
```

It sends a debug print of `layer` to `fp` in the usual way.

## 5.4. Zones

A *regular* timetable is one which has a pattern that makes it easy to understand. For example, if a train comes every 15 minutes, then that is a regular train timetable.

In high school timetabling, two forms of regularity are important. *Meet regularity* is achieved when meets which overlap in time have the same starting times and durations. It is automatic when all meets have duration 1, but not otherwise. For example, if there are two meets of duration 2, and one starts at the first time on Mondays while the second starts at the second time, that is not regular. Most instances seem to have meets of durations 1 and 2, with just a few meets of higher durations, and under those circumstances meet regularity is easy to achieve.

*Node regularity* is achieved when the meets of two nodes which overlap in time have the same starting times and durations. Node regularity makes a timetable easy to understand, and simplifies resource assignment by reducing the number of pairs of events whose meets overlap in time, by ensuring that they generally either overlap completely or not at all.

There seems to be little value in measuring regularity formally; the important thing is to encourage it. This is what zones are for.

For any node $n$, consider the set of all pairs of the form $(m, o)$, where $m$ is a meet lying in $n$, and $o$ is a legal offset of $m$: if $m$ has duration 1, $o$ may only be 0; if $m$ has duration 2, $o$ may be 0 or 1; and so on. Such a pair is called a *meet-offset of n*. For example, if $n$ contains the cycle meets, then there is a meet-offset of $n$ for each time of the cycle.

A *zone* of node $n$ is a subset of the meet-offsets of $n$. A zone may be created by calling

```
KHE_ZONE KheZoneMake(KHE_NODE node);
```

Initially it contains no meet-offsets. Functions

```
KHE_NODE KheZoneNode(KHE_ZONE zone);
int KheZoneNodeIndex(KHE_ZONE zone);
```

return `zone`'s node, which never changes, and the value of `i` for which `KheNodeZone(node, i)` returns `zone`. When a zone is deleted, the indexes of other zones in its node may change. (As usual, the gap left by the deletion of the zone is plugged by moving the last zone into it, unless the deleted zone was the last zone.) For convenience there is also

```
KHE_SOLN KheZoneSoln(KHE_ZONE zone);
```

which returns the solution containing `zone`'s node.

A zone has has the usual back pointer and visit number:

```
void KheZoneSetBack(KHE_ZONE zone, void *back);
void *KheZoneBack(KHE_ZONE zone);

void KheZoneSetVisitNum(KHE_ZONE zone, int num);
int KheZoneVisitNum(KHE_ZONE zone);
bool KheZoneVisited(KHE_ZONE zone, int slack);
void KheZoneVisit(KHE_ZONE zone);
void KheZoneUnVisit(KHE_ZONE zone);
```

A zone may be deleted by calling

```
void KheZoneDelete(KHE_ZONE zone);
```

and all the zones of a node may be deleted by calling

```
void KheNodeDeleteZones(KHE_NODE node);
```

Each meet-offset may lie in at most one zone. To add a meet-offset to a zone, and to delete a meet-offset from a zone, the operations are

```
void KheZoneAddMeetOffset(KHE_ZONE zone, KHE_MEET meet, int offset);
void KheZoneDeleteMeetOffset(KHE_ZONE zone, KHE_MEET meet, int offset);
```

To retrieve the zone of a meet-offset, call

```
KHE_ZONE KheMeetOffsetZone(KHE_MEET meet, int offset);
```

All these functions abort if `offset` is not a legal offset of `meet`. `KheZoneAddMeetOffset` also aborts if the meet-offset already lies in a zone, or `zone` is `NULL`, or `meet` does not lie in a node, or `zone` is not a zone of the node containing `meet`. `KheMeetOffsetZone` returns `NULL` if the meet-offset does not lie in any zone, as is the case by default.

The zones of a node may be accessed from the node in the usual way:

```
int KheNodeZoneCount(KHE_NODE node);
KHE_ZONE KheNodeZone(KHE_NODE node, int i);
```

They are returned in an arbitrary order. The meet-offsets of a zone may be accessed by calling

```
int KheZoneMeetOffsetCount(KHE_ZONE zone);
void KheZoneMeetOffset(KHE_ZONE zone, int i, KHE_MEET *meet, int *offset);
```

They are returned in an arbitrary order. Function

```
void KheZoneDebug(KHE_ZONE zone, int verbosity, int indent, FILE *fp);
```

produces a debug print of `zone` onto `fp` in the usual way.

When a meet is deleted from a node or deleted altogether, all the meet-offsets involving that meet are removed from their zones. When a meet is split or merged, the meet-offsets mutate in the appropriate way, but preserve their zones. For example, when a meet $m$ of duration 3 is split into a meet $m_1$ of duration 1 and a meet $m_2$ of duration 2, the meet-offsets mutate as follows:

$$(m, 0), (m, 1), (m, 2) \rightarrow (m_1, 0), (m_2, 0), (m_2, 1)$$

Nothing constrains a zone to hold any particular meet-offsets, and indeed nothing requires zones to be created at all. The basic operations of KHE are not restricted in any way by zones. By convention only, some solvers use zones to encourage meet and node regularity. See Section 9.6 for solvers that install zones.

A useful helper function when using zones is

```
bool KheMeetMovePreservesZones(KHE_MEET meet1, int offset1,
  KHE_MEET meet2, int offset2, int durn);
```

Assuming that a meet of duration `durn` may be assigned to `meet1` at `offset1` and to `meet2` at `offset2`, this function returns `true` if that meet would be assigned to the same zones either way. It treats the `NULL` value returned at times by `KheMeetOffsetZone` as though it was a zone.

Another useful function is

```
int KheNodeIrregularity(KHE_NODE node);
```

It returns the *irregularity* of `node`: 0 if none of its meets is assigned, else the number of distinct zones of `n`'s parent node that the assigned meets of `n` are assigned to (counting `NULL` as a zone), minus one. For example, when `n`'s parent node has no zones, or all of the meets of `n` are assigned to the same zone, `n`'s irregularity is 0. One reasonable way to preserve existing regularity is to measure the irregularity of the nodes affected by an operation beforehand, measure it again afterwards, and undo the operation if irregularity has increased.

## 5.5. Taskings

A *tasking* is an object of type `KHE_TASKING` representing a set of tasks. A task may lie in at most one tasking at any one time. Taskings make useful parameters to resource solvers: the solver's job can be to assign resources to the tasks of the tasking—any subset of the tasks of a solution. For a deeper analysis of the role of taskings, see Section 11.2.

To create a tasking, initially with no tasks, call

```
KHE_TASKING KheTaskingMake(KHE_SOLN soln, KHE_RESOURCE_TYPE rt);
```

When `rt` is non-`NULL`, it signifies that all the tasks of the tasking have that type; but it may also be `NULL`, in which case there is no restriction. To retrieve the two attributes, call

```
KHE_SOLN KheTaskingSoln(KHE_TASKING tasking);
KHE_RESOURCE_TYPE KheTaskingResourceType(KHE_TASKING tasking);
```

To visit the taskings of a solution, call functions `KheSolnTaskingCount` and `KheSolnTasking` from Section 4.2.7. To delete a tasking, without deleting its tasks, call

```
void KheTaskingDelete(KHE_TASKING tasking);
```

To add a task to a tasking, and to delete it from a tasking, call

```
void KheTaskingAddTask(KHE_TASKING tasking, KHE_TASK task);
void KheTaskingDeleteTask(KHE_TASKING tasking, KHE_TASK task);
```

`KheTaskingAddTask` aborts if `task` already lies in a tasking, or if the resource type of `tasking` is non-`NULL` and `task` does not have that resource type. `KheTaskingDeleteTask` aborts if `task` does not lie in `tasking`. Functions

```
int KheTaskingTaskCount(KHE_TASKING tasking);
KHE_TASK KheTaskingTask(KHE_TASKING tasking, int i);
```

visit the tasks of a tasking in the usual way, and

```
void KheTaskingDebug(KHE_TASKING tasking, int verbosity,
  int indent, FILE *fp);
```

produces a debug print of `tasking`.

## 5.6. Task sets

A *task set* is like a tasking in that it represents a set of tasks. It is different in that a task may lie in any number of task sets, but it does not know which task sets it lies in.

To create a new, empty task set for holding tasks from `soln`, call

```
KHE_TASK_SET KheTaskSetMake(KHE_SOLN soln);
```

The `soln` attribute is stored in the task set and may be accessed by calling

```
KHE_SOLN KheTaskSetSoln(KHE_TASK_SET ts);
```

To delete a task set (but not its tasks), call

```
void KheTaskSetDelete(KHE_TASK_SET ts);
```

This places the task set object on a free list in its solution object, where it is available for use by subsequent calls to `KheTaskSetMake` on the same solution object.

To clear a task set back to the empty set of tasks, call

```
void KheTaskSetClear(KHE_TASK_SET ts);
```

To clear a task set from the end back to a point where it contains `count` elements, call

```
void KheTaskSetClearFromEnd(KHE_TASK_SET ts, int count);
```

To remove the last `n` tasks from a task set, call

```
void KheTaskSetDropFromEnd(KHE_TASK_SET ts, int n);
```

To add a task to a task set, call

```
void KheTaskSetAddTask(KHE_TASK_SET ts, KHE_TASK task);
```

To add the tasks of `ts2` to `ts`, call

```
void KheTaskSetAddTaskSet(KHE_TASK_SET ts, KHE_TASK_SET ts2);
```

To delete a task, call

```
void KheTaskSetDeleteTask(KHE_TASK_SET ts, KHE_TASK task);
```

`KheTaskSetDeleteTask` aborts if `task` is not in `ts`. If the tasks of `ts` are equivalent, the best way to extract one task is

```
KHE_TASK KheTaskSetLastAndDelete(KHE_TASK_SET ts);
```

This deletes and returns the last task of `ts`; it aborts if `ts` is empty.

To search a task set for a given task, call

```
bool KheTaskSetContainsTask(KHE_TASK_SET ts, KHE_TASK task, int *pos);
```

If this returns `true`, it sets `*pos` to the index of `task` in `ts`. To visit the tasks of a task set, call

```
int KheTaskSetTaskCount(KHE_TASK_SET ts);
KHE_TASK KheTaskSetTask(KHE_TASK_SET ts, int i);
```

as usual. There are also

```
KHE_TASK KheTaskSetFirst(KHE_TASK_SET ts);
KHE_TASK KheTaskSetLast(KHE_TASK_SET ts);
```

which return the first and last tasks. To sort the tasks, call

```
void KheTaskSetSort(KHE_TASK_SET ts,
  int(*compar)(const void *, const void *));
void KheTaskSetSortUnique(KHE_TASK_SET ts,
  int(*compar)(const void *, const void *));
```

`KheTaskSetSortUnique` calls `HaArraySortUnique` (Section A.1.3).

Functions

```
int KheTaskSetTotalDuration(KHE_TASK_SET ts);
float KheTaskSetTotalWorkload(KHE_TASK_SET ts);
```

return the total duration or total workload of the task set: the sum, over all tasks `t`, of the total duration or total workload of `t`.

Function

```
void KheTaskSetUnGroup(KHE_TASK_SET ts);
```

is useful when `ts` is being used to record a set of tasks which were assigned to other tasks in order to ensure that they would be assigned the same resource. It removes the assignments of the tasks of `ts`, but then assigns the tasks directly to the resources (cycle tasks) that they were previously indirectly assigned to, if any, or unassigns them otherwise.

There is another possible specification for `KheTaskSetUnGroup`, saying that the assignment in each task of `ts` is changed to the grandparent task (whatever the current assignment is assigned to). This was rejected because it misbehaves in some cases when groupings are made in stages, with one task assigned to another and then that task assigned to a third. The preferred specification cuts the knot by ungrouping the tasks from all groupings.

Function

```
bool KheTaskSetNeedsAssignment(KHE_TASK_SET ts);
```

returns `true` if `KheTaskNeedsAssignment` returns `true` for at least one task in `ts`.

There are functions for visiting the tasks of a task set, following the usual pattern:

```
void KheTaskSetSetVisitNum(KHE_TASK_SET ts, int num);
int KheTaskSetGetVisitNum(KHE_TASK_SET ts);
bool KheTaskSetAllVisited(KHE_TASK_SET ts, int slack);
bool KheTaskSetAnyVisited(KHE_TASK_SET ts, int slack);
void KheTaskSetAllVisit(KHE_TASK_SET ts);
void KheTaskSetAllUnVisit(KHE_TASK_SET ts);
```

These just call the corresponding task visit operation on each task of `ts`, except that `KheTaskSetGetVisitNum` returns the visit number of `ts`'s first task, aborting if `ts` is empty. `KheTaskSetAllVisited` returns `true` when all the calls on individual tasks return `true`, and `KheTaskSetAnyVisited` returns `true` when any of the calls on individual tasks return true. Which of these two truly represents the condition '`ts` has been visited' is a matter of opinion.

There are also functions for moving, assigning, and unassigning all the tasks of a task set:

```
bool KheTaskSetMoveResourceCheck(KHE_TASK_SET ts, KHE_RESOURCE r);
bool KheTaskSetMoveResource(KHE_TASK_SET ts, KHE_RESOURCE r);
bool KheTaskSetAssignResourceCheck(KHE_TASK_SET ts, KHE_RESOURCE r);
bool KheTaskSetAssignResource(KHE_TASK_SET ts, KHE_RESOURCE r);
bool KheTaskSetUnAssignResourceCheck(KHE_TASK_SET ts);
bool KheTaskSetUnAssignResource(KHE_TASK_SET ts);
```

These are like `KheTaskMoveResourceCheck` and so on, except that they apply to all the tasks of `ts`: `KheTaskSetMoveResourceCheck` checks that all the tasks of `ts` can be moved to `r`, `KheTaskSetMoveResource` checks and moves, and so on. If `false` is returned, some tasks may have been changed and others not. If that does not suit, check first before trying to change anything. There are also

```
bool KheTaskSetPartMoveResourceCheck(KHE_TASK_SET ts,
  int first_index, int last_index, KHE_RESOURCE r);
bool KheTaskSetPartMoveResource(KHE_TASK_SET ts,
  int first_index, int last_index, KHE_RESOURCE r);
```

which are like `KheTaskSetMoveResourceCheck` and `KheTaskSetMoveResource`, but applied only to the tasks with indexes between `first_index` and `last_index` (inclusive).

KHE's policy is for operations to return `false` when they change nothing, on the grounds that no solver wants to waste time on operations that do nothing. However this policy does not seem to work very well here, because very often the task set move or assignment is just one part of a larger operation. Certainly, we do not want to waste time on the larger operation if it does nothing, but that does not prevent one part of it from doing nothing. Accordingly, these operations all succeed (return `true`) when `ts` is empty.

Functions

```
void KheTaskSetAssignFix(KHE_TASK_SET ts);
void KheTaskSetAssignUnFix(KHE_TASK_SET ts);
```

call `KheTaskAssignFix` or `KheTaskAssignUnFix` on each task of `ts`.

Finally,

```
void KheTaskSetDebug(KHE_TASK_SET ts, int verbosity, int indent, FILE *fp);
```

produces a debug print of `ts` onto `fp` with the given verbosity and indent.

## 5.7. Meet sets

A *meet set* is like a node in that it represents a set of meets. It is different in that a meet may lie in any number of meet sets, but it does not know which. Meet sets correspond closely with task sets, so we will be brief. To create a new, empty meet set for holding meets from `soln`, call

```
KHE_MEET_SET KheMeetSetMake(KHE_SOLN soln);
```

To delete a meet set (but not its meets), call

```
void KheMeetSetDelete(KHE_MEET_SET ms);
```

A deleted meet set goes on a free list in the solution object and becomes available for re-use.

```
void KheMeetSetClear(KHE_MEET_SET ms);
```

clears `ms` back to the empty set of meets, and

```
void KheMeetSetDropFromEnd(KHE_MEET_SET ms, int n);
```

removes the last `n` meets from `ms`. To add and delete a meet, call

```
void KheMeetSetAddMeet(KHE_MEET_SET ms, KHE_MEET meet);
void KheMeetSetDeleteMeet(KHE_MEET_SET ms, KHE_MEET meet);
```

`KheMeetSetDeleteMeet` aborts if `meet` is not present. To search a meet set, call

```
bool KheMeetSetContainsMeet(KHE_MEET_SET ms, KHE_MEET meet, int *pos);
```

If this returns `true`, it sets `*pos` to the index of `meet` in `ms`. To visit the meets, call

```
int KheMeetSetMeetCount(KHE_MEET_SET ms);
KHE_MEET KheMeetSetMeet(KHE_MEET_SET ms, int i);
```

as usual. To sort the meets, call

```
void KheMeetSetSort(KHE_MEET_SET ms,
  int(*compar)(const void *, const void *));
void KheMeetSetSortUnique(KHE_MEET_SET ms,
  int(*compar)(const void *, const void *));
```

`KheMeetSetSortUnique` calls `HaArraySortUnique` (Section A.1.3). Function

```
int KheMeetSetTotalDuration(KHE_MEET_SET ms);
```

the sum, over all meets `m` in `ms`, of the duration of `m`.

There are functions for visiting the meets of a meet set, following the usual pattern:

```
void KheMeetSetSetVisitNum(KHE_MEET_SET ms, int num);
int KheMeetSetGetVisitNum(KHE_MEET_SET ms);
bool KheMeetSetVisited(KHE_MEET_SET ms, int slack);
void KheMeetSetVisit(KHE_MEET_SET ms);
void KheMeetSetUnVisit(KHE_MEET_SET ms);
```

These just call the corresponding meet visit operation on each meet of `ms`, except that `KheMeetSetGetVisitNum` returns the visit number of `ms`'s first meet, aborting if `ms` is empty. `KheMeetSetVisited` returns `true` when all the calls on individual meets return `true`. Finally,

```
void KheMeetSetDebug(KHE_MEET_SET ms, int verbosity, int indent, FILE *fp);
```

produces a debug print of `ms` onto `fp` with the given verbosity and indent.

## 5.8. Time sets

A *time set* is like a time group in that it represents a set of times. However, it carries less baggage: it has no name, and there is nothing equivalent to `KheTimeGroupNeighbour`. It is a convenient type to use when a set of times is needed during solving. Internally, a time set holds the instance that the times come from and a sorted array of time indexes, nothing more.

To create a new, empty time set, call

```
KHE_TIME_SET KheTimeSetMake(KHE_INSTANCE ins, HA_ARENA a);
```

Another way to make a time set is

```
KHE_TIME_SET KheTimeSetCopy(KHE_TIME_SET ts, HA_ARENA a);
```

This makes a fresh copy of `ts` in arena `a`. There is also

```
void KheTimeSetCopyElements(KHE_TIME_SET dst_ts, KHE_TIME_SET src_ts);
```

which replaces the times of time set `dst_ts`, whatever they are, with the times of `src_ts`.

To retrieve a time set's instance, call

```
KHE_INSTANCE KheTimeSetInstance(KHE_TIME_SET ts);
```

There is no function to delete a time set; it is deleted when its arena is deleted. But a time set can be cleared back to the empty set of times, by calling

```
void KheTimeSetClear(KHE_TIME_SET ts);
```

To add times to a time set, the following operations are available:

```
void KheTimeSetAddTime(KHE_TIME_SET ts, KHE_TIME t);
void KheTimeSetAddTimeGroup(KHE_TIME_SET ts, KHE_TIME_GROUP tg);
void KheTimeSetAddTaskTimes(KHE_TIME_SET ts, KHE_TASK task);
```

These add a time, or the times of a time group, or the times a task is running (including tasks assigned, directly or indirectly, to that task). To add the times of a time set, call `KheTimeSetUnion` below. Here and elsewhere, adding a time that is already present does nothing.

For deleting times there are

```
void KheTimeSetDeleteTime(KHE_TIME_SET ts, KHE_TIME t);
void KheTimeSetDeleteLastTime(KHE_TIME_SET ts);
```

`KheTimeSetDeleteTime` deletes `t` from `ts`, or does nothing if it is not present. `KheTimeSetDeleteLastTime` deletes the last time from `ts`; it must be present.

To visit the times of a time set, call

```
int KheTimeSetTimeCount(KHE_TIME_SET ts);
KHE_TIME KheTimeSetTime(KHE_TIME_SET ts, int i);
```

in the usual way. There is also

```
int KheTimeSetTimeIndex(KHE_TIME_SET ts, int i);
```

which returns the index of the `i`th time, rather than the time itself. Irrespective of the order in which the times were added, they are stored and visited in order of increasing index.

There are also set operations on time sets:

```
void KheTimeSetUnion(KHE_TIME_SET ts1, KHE_TIME_SET ts2);
void KheTimeSetIntersect(KHE_TIME_SET ts1, KHE_TIME_SET ts2);
void KheTimeSetDifference(KHE_TIME_SET ts1, KHE_TIME_SET ts2);
```

These update `ts1` to hold its union, intersection, or difference with `ts2`. Also,

```
int KheTimeSetUnionCount(KHE_TIME_SET ts1, KHE_TIME_SET ts2);
int KheTimeSetIntersectCount(KHE_TIME_SET ts1, KHE_TIME_SET ts2);
int KheTimeSetDifferenceCount(KHE_TIME_SET ts1, KHE_TIME_SET ts2);
```

return the cardinality of the union, intersection, and difference without building the actual set. `KheTimeSetIntersectCount` is optimized for the case of intersecting a small (and presumably localized) set with a large one: it uses binary search to retrieve the indexes of the first and last elements of the smaller set in the larger one, then only traverses the larger one in that range. This idea could be applied to other operations, but so far it has not been.

Several set queries are available:

```
bool KheTimeSetEmpty(KHE_TIME_SET ts);
bool KheTimeSetEqual(KHE_TIME_SET ts1, KHE_TIME_SET ts2);
bool KheTimeSetSubset(KHE_TIME_SET ts1, KHE_TIME_SET ts2);
bool KheTimeSetDisjoint(KHE_TIME_SET ts1, KHE_TIME_SET ts2);
bool KheTimeSetContainsTime(KHE_TIME_SET ts, KHE_TIME t);
```

These return `true` when `ts` is empty, when `ts1` is equal to `ts2`, when `ts1` is a subset of `ts2`, when `ts1` is disjoint from `ts2`, and when `ts` contains `t`.

For applications in which a time set is used as the key into a hash table, there is

```
int KheTimeSetHash(KHE_TIME_SET ts);
```

At present the value returned is the sum of the indexes of the first, middle, and last times, after shifting left 16, 8, and 0 places respectively, or 0 if `ts` is empty.

Four other comparison functions are available:

```
int KheTimeSetCmp(const void *t1, const void *t2);
int KheTimeSetTypedCmp(KHE_TIME_SET ts1, KHE_TIME_SET ts2);
```

`KheTimeSetCmp` is suitable for passing to `HaArraySort`, to bring equal time sets together. `KheTimeSetTypedCmp` is the typed equivalent of `KheTimeSetCmp`. And

```
int KheTimeSetCmpReverse(const void *t1, const void *t2);
int KheTimeSetTypedCmpReverse(KHE_TIME_SET ts1, KHE_TIME_SET ts2);
```

are like `KheTimeSetCmp` and `KheTimeSetTypedCmp`, except that they sort in the reverse order.

Unlike time groups, time sets allow `NULL` to be a member. It is handled like any other time: it can be added and deleted, and it participates in set operations. It has index `-1`, which means that, if present, it is the result of `KheTimeSetTime(ts, 0)`.

Finally,

```
void KheTimeSetDebug(KHE_TIME_SET ts, int verbosity, int indent, FILE *fp);
```

produces a debug print of `ts` onto `fp` with the given verbosity and indent, as usual. Since the time set has no name, this can only be done by printing its elements. When `verbosity` is 1 or `indent` is negative, only the first and last elements (at most) are printed.

## 5.9. Resource sets

A *resource set* is like a resource group in that it represents a set of resources of a particular type. However, it carries less baggage: it has no name, for example. It is a convenient type to use when a set of resources is needed during solving. Internally, a resource set holds the resource type that the resources must share, and a sorted array of resource indexes in that resource type.

Resource sets are virtually clones of time sets, with some extra operations that might find their way into time sets eventually. To create a new, empty resource set of a given type, call

```
KHE_RESOURCE_SET KheResourceSetMake(KHE_RESOURCE_TYPE rt, HA_ARENA a);
```

Another way to make a resource set is

```
KHE_RESOURCE_SET KheResourceSetCopy(KHE_RESOURCE_SET rs, HA_ARENA a);
```

It makes a fresh copy of `rs` in arena `a`. Either way, it will be deleted when `a` is deleted. Also,

```
void KheResourceSetCopyElements(KHE_RESOURCE_SET dst_rs,
  KHE_RESOURCE_SET src_rs);
```

replaces the resources of resource set `dst_rs`, whatever they are, with the resources of `src_rs`.

To retrieve a resource set's resource type, call

```
KHE_RESOURCE_TYPE KheResourceSetResourceType(KHE_RESOURCE_SET rs);
```

To clear a resource set back to the empty set of resources, call

```
void KheResourceSetClear(KHE_RESOURCE_SET rs);
```

To add resources to a resource set, the following operations are available:

```
void KheResourceSetAddResource(KHE_RESOURCE_SET rs, KHE_RESOURCE r);
void KheResourceSetAddResourceGroup(KHE_RESOURCE_SET rs,
  KHE_RESOURCE_GROUP rg);
```

These add a resource, or the resources of a resource group. To add the resources of a resource set, call `KheResourceSetUnion` below. Here and elsewhere, adding a resource that is already present does nothing.

For deleting resources there are

```
void KheResourceSetDeleteResource(KHE_RESOURCE_SET rs, KHE_RESOURCE r);
void KheResourceSetDeleteLastResource(KHE_RESOURCE_SET rs);
```

`KheResourceSetDeleteResource` deletes `r` from `rs`, or does nothing if it is not present. `KheResourceSetDeleteLastResource` deletes the last resource from `rs`; it must be present.

To visit the resources of a resource set, call

```
int KheResourceSetResourceCount(KHE_RESOURCE_SET rs);
KHE_RESOURCE KheResourceSetResource(KHE_RESOURCE_SET rs, int i);
```

in the usual way. There is also

```
int KheResourceSetResourceIndex(KHE_RESOURCE_SET rs, int i);
```

which returns the index in the resource set's resource type of the `i`th resource, rather than the resource itself. Irrespective of the order in which the resources were added, they are stored and visited in order of increasing index.

There are also set operations on resource sets. These come in various forms, all of which are illustrated by the operations for set union:

```
void KheResourceSetUnion(KHE_RESOURCE_SET rs1, KHE_RESOURCE_SET rs2);
void KheResourceSetUnionGroup(KHE_RESOURCE_SET rs1,
  KHE_RESOURCE_GROUP rg2);
int KheResourceSetUnionCount(KHE_RESOURCE_SET rs1, KHE_RESOURCE_SET rs2);
int KheResourceSetUnionCountGroup(KHE_RESOURCE_SET rs1,
  KHE_RESOURCE_GROUP rg2);
```

`KheResourceSetUnion` updates `rs1` to hold the set union of itself with `rs2`, eliminating duplicates. `KheResourceSetUnionGroup` is the same, except that the second parameter is a resource group. `KheResourceSetUnionCount` and `KheResourceSetUnionCountGroup` return the number of elements in the union, without actually building it. This is faster, when it suits.

A resource group does actually hold a resource set, but it would not be safe to make that set available directly, because resource groups are supposed to be immutable after their creation ends. A copy of it is easily made, by starting with an empty resource set and calling `KheResourceSetUnionGroup`.

Corresponding operations are available for set intersection:

```
void KheResourceSetIntersect(KHE_RESOURCE_SET rs1, KHE_RESOURCE_SET rs2);
void KheResourceSetIntersectGroup(KHE_RESOURCE_SET rs1,
  KHE_RESOURCE_GROUP rg2);
int KheResourceSetIntersectCount(KHE_RESOURCE_SET rs1,
  KHE_RESOURCE_SET rs2);
int KheResourceSetIntersectCountGroup(KHE_RESOURCE_SET rs1,
  KHE_RESOURCE_GROUP rg2);
```

and set difference:

```
void KheResourceSetDifference(KHE_RESOURCE_SET rs1,KHE_RESOURCE_SET rs2);
void KheResourceSetDifferenceGroup(KHE_RESOURCE_SET rs1,
  KHE_RESOURCE_GROUP rg2);
int KheResourceSetDifferenceCount(KHE_RESOURCE_SET rs1,
  KHE_RESOURCE_SET rs2);
int KheResourceSetDifferenceCountGroup(KHE_RESOURCE_SET rs1,
  KHE_RESOURCE_GROUP rg2);
```

For symmetric difference, at present only the count operations are offered:

```
int KheResourceSetSymmetricDifferenceCount(KHE_RESOURCE_SET rs1,
  KHE_RESOURCE_SET rs2);
int KheResourceSetSymmetricDifferenceCountGroup(KHE_RESOURCE_SET rs1,
  KHE_RESOURCE_GROUP rg2);
```

This is because building the actual set is awkward.

For the Boolean-valued set operations, again the second parameter may be a resource group, but there is no need for count operations. So for equality we have

```
bool KheResourceSetEqual(KHE_RESOURCE_SET rs1, KHE_RESOURCE_SET rs2);
bool KheResourceSetEqualGroup(KHE_RESOURCE_SET rs1,
  KHE_RESOURCE_GROUP rg2);
```

and similarly for subset:

```
bool KheResourceSetSubset(KHE_RESOURCE_SET rs1, KHE_RESOURCE_SET rs2);
bool KheResourceSetSubsetGroup(KHE_RESOURCE_SET rs1,
  KHE_RESOURCE_GROUP rg2);
```

and testing for disjointness:

```
bool KheResourceSetDisjoint(KHE_RESOURCE_SET rs1, KHE_RESOURCE_SET rs2);
bool KheResourceSetDisjointGroup(KHE_RESOURCE_SET rs1,
  KHE_RESOURCE_GROUP rg2);
```

There is also

```
bool KheResourceSetContainsResource(KHE_RESOURCE_SET rs, KHE_RESOURCE r);
```

which returns `true` when `rs` contains `r`.

Two other comparison functions are available:

```
int KheResourceSetCmp(const void *t1, const void *t2);
int KheResourceSetTypedCmp(KHE_RESOURCE_SET rs1, KHE_RESOURCE_SET rs2);
```

`KheResourceSetCmp` is suitable for passing to `HaArraySort`, to bring equal resource sets together. (It is not for sorting the resources of one resource set; their order is fixed.) `KheResourceSetTypedCmp` is the typed equivalent of `KheResourceSetCmp`.

Unlike resource groups, resource sets allow `NULL` to be a member. It is handled like any other resource: it can be added and deleted, and it participates in set operations. It has index `-1`, which means that, if present, it is the result of `KheResourceSetResource(rs, 0)`.

Finally,

```
void KheResourceSetDebug(KHE_RESOURCE_SET rs, int verbosity,
  int indent, FILE *fp);
```

produces a debug print of `rs` onto `fp` with the given verbosity and indent, as usual. Since the resource set has no name, this can only be done by printing its elements. When `verbosity` is 1 or `indent` is negative, only the first and last elements (at most) are printed.

## 5.10. Time frames

A *time frame*, or just *frame*, is a sequence of time groups. Frames satisfy a practical need during solving; they help to bridge the gap between the high school and nurse rostering time models.

A frame has type `KHE_FRAME`, the usual pointer to a private struct, lying in heap memory and holding the enclosing solution, the time groups, and some other things.

Frames are immutable after creation. To help enforce this, they are created indirectly via another type, `KHE_FRAME_MAKE`. The operations for creating a frame are

```
KHE_FRAME_MAKE KheFrameMakeBegin(KHE_SOLN soln);
void KheFrameMakeAddTimeGroup(KHE_FRAME_MAKE fm, KHE_TIME_GROUP tg);
KHE_FRAME KheFrameMakeEnd(KHE_FRAME_MAKE fm, bool sort_time_groups);
```

`KheFrameMakeBegin` starts the creation of the frame by creating a `KHE_FRAME_MAKE` object. This is followed by any number of calls to `KheFrameMakeAddTimeGroup`, which add the time groups. The creation ends with a call to `KheFrameMakeEnd`, which returns the actual frame.

If the `sort_time_groups` parameter of `KheFrameMakeEnd` is `true`, `KheFrameMakeEnd` sorts the time groups into increasing first time order.

To delete a frame returned by `KheFrameMakeEnd`, call

```
void KheFrameDelete(KHE_FRAME frame);
```

This frees the memory consumed by `frame`; it goes on a free list in `frame`'s solution object, where it can be re-used by a later call to `KheFrameMakeBegin`.

The usual operations are available for retrieving the attributes of a frame. To retrieve the enclosing solution, call

```
KHE_SOLN KheFrameSoln(KHE_FRAME frame);
```

To visit the time groups, call

```
int KheFrameTimeGroupCount(KHE_FRAME frame);
KHE_TIME_GROUP KheFrameTimeGroup(KHE_FRAME frame, int i);
```

`KheFrameTimeGroup` returns the `i`th time group of `frame`.

A frame is *disjoint* when its time groups are pairwise disjoint, and *complete* when every time in the cycle lies in at least one of its time groups. Frames do not have to satisfy these conditions, but some applications of frames require them. They are returned by functions

```
bool KheFrameIsDisjoint(KHE_FRAME frame, int *problem_index1,
  int *problem_index2);
bool KheFrameIsComplete(KHE_FRAME frame, KHE_TIME *problem_time);
```

If the frame is disjoint, `KheFrameIsDisjoint` returns `true` with `*problem_index1` and `*problem_index2` set to `-1`; otherwise it returns `false` with `*problem_index1` and `*problem_index2` set to the indexes of two overlapping time groups. If the frame is complete, `KheFrameIsComplete` returns `true` with `*problem_time` set to `NULL`; otherwise it returns `false` with `*problem_time` set to a time of the instance which is not in any of `frame`'s time groups.

`KheFrameIsDisjoint` and `KheFrameIsComplete` are typically called at most once per frame, after `KheFrameMakeEnd`. An efficient implementation has not been thought necessary. But this function is implemented efficiently:

```
int KheFrameTimeIndex(KHE_FRAME frame, KHE_TIME t);
```

It returns the index in `frame` of the time group containing time `t`. If there is no such time group (implying that the frame is not complete), `-1` is returned. If there is more than one such time group (implying that the frame is not disjoint), the index of one of the time groups is returned. The time group itself can then be retrieved using `KheFrameTimeGroup`. There is also

```
KHE_TIME_GROUP KheFrameTimeTimeGroup(KHE_FRAME frame, KHE_TIME t);
```

which combines the two steps, returning the time group of `frame` that contains `t`, or aborting if there is no such time group. And

```
int KheFrameTimeOffset(KHE_FRAME frame, KHE_TIME t);
```

returns the offset of time `t` in its frame time group; that is, the index of `t` minus the index of the first time in time group `KheFrameTimeTimeGroup(frame, t)`. In nurse rostering, this offset identifies the shift type (morning, afternoon, or whatever).

Frames arise naturally in employee scheduling when each employee can work at most one shift per day (evidenced by a hard limit busy times constraint with non-zero cost, maximum limit 1, and one time group for each day). When this is true of all resources, function

```
KHE_FRAME KheFrameMakeCommon(KHE_SOLN soln);
```

returns a frame with one time group per day, each with positive polarity. The time groups do not have to actually represent days, they merely need to be the same for all resources and to be disjoint and complete. When there is no common frame, `NULL` is returned.

When `KheFrameMakeCommon` returns `NULL`, as a fallback there is

```
KHE_FRAME KheFrameMakeSingletons(KHE_SOLN soln);
```

This returns a frame with one time group for each time, containing just that single time.

Once created, frames of this kind do not change. So it makes sense to share a single one between solvers, by storing it in the solvers' shared options object. A convenient way do this is

```
KHE_FRAME KheOptionsFrame(KHE_OPTIONS options, char *key, KHE_SOLN soln);
```

from Section 8.2. This returns the frame stored in `options` under the given `key`. If there is no object in `options` under that key, it first creates one, by calling `KheFrameMakeCommon`, followed by `KheFrameMakeSingletons` if necessary, and storing the result in `options` under `key`. Thus, if all solvers that need a frame call this function to obtain it, they will all share the same frame, the one created the first time this function is called. By convention, the key to use is `"gs_common_frame"`, and so

```
frame = KheOptionsFrame(options, "gs_common_frame", soln);
```

is the recommended way to obtain this kind of frame.

Function

```
bool KheFrameIntersectsTimeGroup(KHE_FRAME frame, KHE_TIME_GROUP tg);
```

returns `true` when `tg` shares at least one time with at least one of the time groups of `frame`.

There is the usual debug function:

```
void KheFrameDebug(KHE_FRAME frame, int verbosity, int indent, FILE *fp);
```

This prints `frame` onto `fp` with the given verbosity and indent. Here `frame` may be `NULL`.

Finally, here are three related miscellaneous functions:

```
bool KheFrameResourceHasClashes(KHE_FRAME frame, KHE_RESOURCE r);
void KheFrameResourceAssertNoClashes(KHE_FRAME frame, KHE_RESOURCE r);
void KheFrameAssertNoClashes(KHE_FRAME frame);
```

These help to debug solvers that preserve an invariant stating that each resource attends at most one task during each time group of `frame`. `KheFrameResourceHasClashes` returns `true` if `r` violates this condition; `KheFrameResourceAssertNoClashes` aborts the run if it is violated for resource `r`, after printing out information about which resource and time group is involved; and `KheFrameAssertNoClashes` calls `KheFrameResourceAssertNoClashes` for all resources.

# Chapter 6. Solution Monitoring

As a solution changes, it is continuously *monitored* by a hand-tuned constraint network.

## 6.1. Measuring cost

KHE measures the badness of a solution as a single integral value called the *cost*, or sometimes the *combined cost* because it includes the cost of both hard and soft constraint deviations. Storing costs in this way is convenient, because it allows costs to be assigned using =, added using +, and compared using < and so on in the usual way. The hard cost is shifted left by 32 bits, to ensure that it is more significant than any reasonable total soft cost, then added to the soft cost.

The type of a combined cost is `KHE_COST`, a synonym for the standard C 64-bit integer type `int64_t` (a fact best forgotten). To find the current combined cost of a solution, call

```
KHE_COST KheSolnCost(KHE_SOLN soln);
```

This value is stored explicitly in `soln`, so this function takes virtually no time to execute. Call

```
KHE_COST KheCost(int hard_cost, int soft_cost);
```

to create a combined cost. The two components of a combined cost may be accessed by

```
int KheHardCost(KHE_COST combined_cost);
int KheSoftCost(KHE_COST combined_cost);
```

There is also the constant `KheCostMax`, which returns the maximum value storable in a variable of type `KHE_COST` (a synonym for `INT64_MAX`) and the function

```
int KheCostCmp(KHE_COST cost1, KHE_COST cost2);
```

which returns an `int` which is less than, equal to, or greater than zero if the first argument is respectively less than, equal to, or greater than the second, as needed when sorting items by cost. The implementation does not make the mistake of merely subtracting `cost2` from `cost1`; the result then would be a `KHE_COST` which will usually overflow the `int` result.

The suggested way to display a combined cost is as a decimal number with the hard cost before the decimal point and the soft cost after. Five decimal places are displayed, allowing for soft costs up to 99999. Larger soft costs are displayed as 99999. To assist with this, function

```
double KheCostShow(KHE_COST combined_cost);
```

returns a value which, when printed with `printf` format `"%.5f"`, prints the cost in this format.

These functions assume that both components of the cost are non-negative. There is no problem with negative combined costs in themselves, but when a hard and soft cost are combined together, if either is negative they may be different if they are separated again.

## 6.2. Monitors

A *monitor* is an object, of type `KHE_MONITOR`, that monitors one part of a solution: typically, one point of application of one constraint. It contains the usual back pointer and visit number:

```
void KheMonitorSetBack(KHE_MONITOR m, void *back);
void *KheMonitorBack(KHE_MONITOR m);
void KheMonitorSetVisitNum(KHE_MONITOR m, int num);
int KheMonitorVisitNum(KHE_MONITOR m);
bool KheMonitorVisited(KHE_MONITOR m, int slack);
void KheMonitorVisit(KHE_MONITOR m);
void KheMonitorUnVisit(KHE_MONITOR m);
```

Operations

```
KHE_SOLN KheMonitorSoln(KHE_MONITOR m);
int KheMonitorSolnIndex(KHE_MONITOR m);
KHE_COST KheMonitorCost(KHE_MONITOR m);
KHE_COST KheMonitorLowerBound(KHE_MONITOR m);
```

return the enclosing solution, the index of `m` in that solution, the cost of what `m` is monitoring (kept up to date by KHE as the solution changes), and a constant lower bound on `KheMonitorCost`, which is usually 0 but will be non-zero when KHE can prove the lower bound easily.

Each monitor contains its own combined weight, used when calculating its cost. This is initially (and usually) the combined weight of the constraint that the monitor is derived from, but it is possible to set it independently:

```
KHE_COST KheMonitorCombinedWeight(KHE_MONITOR m);
void KheMonitorSetCombinedWeight(KHE_MONITOR m,
  KHE_COST combined_weight);
void KheMonitorResetCombinedWeight(KHE_MONITOR m);
```

`KheMonitorCombinedWeight` returns the combined weight, `KheMonitorSetCombinedWeight` sets it, and `KheMonitorResetCombinedWeight` resets it to its initial value, the value from `m`'s constraint. (If `m` is not derived from any constraint, for example if it is one of the redundancy prefer resources monitors described in Section 7.6, then `KheMonitorResetCombinedWeight` does nothing.

Any change to `m`'s weight is reflected immediately in the costs of `m` and its ancestors, for example in solution cost. However `KheMonitorLowerBound` does not change; it depends only on the initial combined weight.

Here are two other functions related to cost:

```
KHE_COST_FUNCTION KheMonitorCostFunction(KHE_MONITOR m);
KHE_COST KheMonitorDevToCost(KHE_MONITOR m, int dev);
```

`KheMonitorCostFunction` returns `m`'s cost function, which is the same as `m`'s constraint's cost function. `KheMonitorDevToCost` returns the cost that `m` would report if it had deviation `dev`. Apart from `dev`, this depends only on `m`'s cost function and combined weight.

Type `KHE_MONITOR` is the abstract supertype of many concrete subtypes, with these tags:

```
typedef enum {
  KHE_ASSIGN_RESOURCE_MONITOR_TAG,
  KHE_ASSIGN_TIME_MONITOR_TAG,
  KHE_SPLIT_EVENTS_MONITOR_TAG,
  KHE_DISTRIBUTE_SPLIT_EVENTS_MONITOR_TAG,
  KHE_PREFER_RESOURCES_MONITOR_TAG,
  KHE_PREFER_TIMES_MONITOR_TAG,
  KHE_AVOID_SPLIT_ASSIGNMENTS_MONITOR_TAG,
  KHE_SPREAD_EVENTS_MONITOR_TAG,
  KHE_LINK_EVENTS_MONITOR_TAG,
  KHE_ORDER_EVENTS_MONITOR_TAG,
  KHE_AVOID_CLASHES_MONITOR_TAG,
  KHE_AVOID_UNAVAILABLE_TIMES_MONITOR_TAG,
  KHE_LIMIT_IDLE_TIMES_MONITOR_TAG,
  KHE_CLUSTER_BUSY_TIMES_MONITOR_TAG,
  KHE_LIMIT_BUSY_TIMES_MONITOR_TAG,
  KHE_LIMIT_WORKLOAD_MONITOR_TAG,
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR_TAG,
  KHE_LIMIT_RESOURCES_MONITOR_TAG,
  KHE_EVENT_TIMETABLE_MONITOR_TAG,
  KHE_RESOURCE_TIMETABLE_MONITOR_TAG,
  KHE_ORDINARY_DEMAND_MONITOR_TAG,
  KHE_WORKLOAD_DEMAND_MONITOR_TAG,
  KHE_EVENNESS_MONITOR_TAG,
  KHE_GROUP_MONITOR_TAG,
  KHE_MONITOR_TAG_COUNT
} KHE_MONITOR_TAG;
```

Each monitor object contains a tag identifying its subtype, returned by

```
KHE_MONITOR_TAG KheMonitorTag(KHE_MONITOR m);
```

Monitors of the first eighteen types monitor one point of application of one constraint; their cost is the total cost of deviations at that point. They are described in detail in later sections of this chapter. Monitors of the last six types (from `KHE_EVENT_TIMETABLE_MONITOR_TAG` onwards) do not monitor constraints. Timetable monitors hold the timetables of events and resources (Section 6.8) Ordinary and workload demand monitors monitor matchings, and evenness monitors monitor evenness (Chapter 7). Group monitors group together other monitors (Section 6.9). The last value is not a tag; it is a count of the number of monitor types, allowing code of the form

```
for( tag = 0;  tag < KHE_MONITOR_TAG_COUNT;  tag++ )
  ... do something for monitors of type tag ...
```

For those monitors that monitor a point of application of a constraint, functions

```
KHE_CONSTRAINT KheMonitorConstraint(KHE_MONITOR m);
char *KheMonitorAppliesToName(KHE_MONITOR m);
```

return the constraint and the name of the point of application (if this point is an event resource, the name of the enclosing event is returned). For other monitors they return `NULL`. It is quite dangerous to assume that the result of `KheMonitorConstraint` is non-`NULL`; in particular, there are prefer resources monitors for which `KheMonitorConstraint` returns `NULL` (Section 7.6).

`KheMonitorAppliesToName` is more or less obsolete; the author prefers now to call

```
char *KheMonitorPointOfApplication(KHE_MONITOR m);
```

which returns a more precise indication of the point of application. Each constraint monitor also has functions which return the specific constraint and point of application.

The result of `KheMonitorPointOfApplication(m)` is created afresh on each call. This is not very efficient, but if the function is called only when generating evaluation tables, as is the intention, that will not matter.

A similar function to `KheMonitorPointOfApplication` is

```
char *KheMonitorId(KHE_MONITOR m);
```

It returns a string composed of two or three fields separated by `/` characters. Each field is an Id from the instance or an integer. The fields are supposed to uniquely identify the monitor, although in a few cases this is doubtful. The first field is always a constraint Id, identifying the constraint that the monitor is derived from, and the second is usually an event, event group, or resource Id, identifying the point of application. There may be a third field, holding a second event Id (for order events monitors) or an offset (for resource constraints with an `AppliesToTimeGroup` attribute). When the offset is 0 the offset field and preceding `/` are omitted. A non-zero offset may be replaced by the Id of the first time of the first monitored time group.

The result of `KheMonitorId(m)` is created when `KheMonitorId(m)` is first called, and stored in `m` so that it does not have to be created over and over. If it is used only for debugging, as is the intention, there is virtually no cost in running time or memory when debugging is off.

The cost of a monitor is a function of its *deviation*, a non-negative integer:

```
int KheMonitorDeviation(KHE_MONITOR m);
char *KheMonitorDeviationDescription(KHE_MONITOR m);
```

These functions are intended for reporting, not solving. `KheMonitorDeviation` returns the deviation, and `KheMonitorDeviationDescription` returns a description of it: an expression, augmented with brief text, showing how it is calculated. The result string does not necessarily lie in heap memory, and should not be freed.

For limit active intervals monitors, `KheMonitorDeviation` returns the sum of the deviations of the active intervals. Exceptionally, the cost of the monitor is not a function of this deviation; instead, it is the sum of the costs of the deviations of the active intervals taken separately.

To visit the full set of monitors monitoring `soln`, call

```
int KheSolnMonitorCount(KHE_SOLN soln);
KHE_MONITOR KheSolnMonitor(KHE_SOLN soln, int i);
```

Although KHE does not fully specify the order in which these monitors appear, it does guarantee that the monitors which monitor constraints will appear together in the list in the order that their constraints appear in the input. It is best to select these monitors by testing whether the result of `KheMonitorConstraint` above is non-`NULL`.

There is also function

```
bool KheSolnRetrieveMonitor(KHE_SOLN soln, char *id, KHE_MONITOR *m);
```

It searches for a monitor whose Id, as returned by `KheMonitorId` above, is equal to `id`. If it finds one, it returns `true` with `*m` set to that monitor; otherwise it returns `false` with `*m` set to `NULL`.

Although every monitor has an Id, at present `KheSolnRetrieveMonitor` does not retrieve all monitors. It retrieves resource monitors, and event monitors that monitor a single event.

`KheSolnRetrieveMonitor` is intended for debugging and is not very efficient. It works by finding the entity (event, event group, or resource) identified by the second field of `id` and searching that entity's list of monitors for one for which `KheMonitorId` returns `id`.

To debug a monitor `m` with a given verbosity and indent, call

```
void KheMonitorDebug(KHE_MONITOR m, int verbosity, int indent, FILE *fp);
```

There are also versions of this function for each of the specific monitor types. These all work in the same way. The output starts with a `G`, `A` or `D` indicating whether the monitor is a group monitor, an attached non-group monitor, or a detached non-group monitor. This is followed by the number of paths up from the monitor to the solution (Section 6.9), usually 0 or 1. Then comes the monitor's tag and cost, then other information depending on the monitor type and verbosity. There is also

```
char *KheMonitorTagShow(KHE_MONITOR_TAG tag);
```

which returns a string representation of `tag`. In practice a more useful function is

```
char *KheMonitorLabel(KHE_MONITOR m);
```

This returns `KheMonitorTagShow(KheMonitorTag(m))` if `m` is not a group monitor, and `m`'s subtag label if `m` is a group monitor.

## 6.3. Attaching, detaching, and provably zero fixed cost

For a monitor to be updated when the solution changes, there must be links from the appropriate points within the solution to the monitor. When these links are present, the monitor is said to be *attached to the solution*, or just *attached*. Most monitors are attached to begin with, but they can be detached at any time, and even reattached later, by calling

```
void KheMonitorDetachFromSoln(KHE_MONITOR m);
void KheMonitorAttachToSoln(KHE_MONITOR m);
```

Even when detached, a monitor remembers which parts of the solution it is supposed to monitor, so the attach operation does not have to tell the monitor where to attach itself. To find out whether a monitor is currently attached or detached, call

```
bool KheMonitorAttachedToSoln(KHE_MONITOR m);
```

Another function, highly recommended for calling at the end of a solve, is

```
void KheSolnEnsureOfficialCost(KHE_SOLN soln);
```

This ensures that all constraint monitors are both attached to the solution and reporting their cost to the solution, directly or indirectly via group monitors, that the multipliers of all cluster busy times monitors are 1 and their minimums have their original value, and that all demand and evenness monitors are detached from the solution. This guarantees that the solution cost is the official cost.

While a monitor is detached, it receives no information about changes to the solution, and, by definition, its deviation and cost are 0. Detaching a monitor may therefore change its cost. If there is a change in cost, it is reported to the monitor's parents (if it has any) as usual. Conversely, attaching a monitor brings it up to date with the current state of the solution, which again may change its cost; and again, if there is a change in cost it is reported to its parents (if it has any).

There are two main reasons for detaching a monitor. First, the user might make a deliberate choice to ignore some constraints. For example, a solver that works in two phases, first finding a solution that satisfies the hard constraints, and then attacking the soft ones, might detach the monitors for the soft constraints during its first phase. An example of this kind of deliberate choice is KHE's matching feature (Chapter 7), which is implemented with monitors. Unlike other monitors, matching monitors are detached initially. KHE makes this choice deliberately, on the grounds that the cost of the matching is not officially part of the cost function.

The second reason for detaching a monitor is that it may be clear that its cost will be zero for a long time. In that case, detaching it means that no time is spent keeping it up to date, yet it still reports the correct cost. For example, if the meets of one point of application of a link events constraint are assigned to each other and those assignments will not be removed, then it is safe to save time by detaching the corresponding monitor.

This reasoning was formerly embodied in a function called `KheMonitorAttachCheck`, which assumed that certain elements of the solution were unlikely to change, and detached monitors accordingly. `KheMonitorAttachCheck` has been withdrawn; the equivalent functionality is now obtained, more reliably, by calling the `Fix` and `UnFix` functions, as follows.

A monitor has *provably zero fixed cost* if enough of the solution is currently fixed (by calls to `KheMeetAssignFix` and `KheTaskAssignFix`) to allow KHE to prove that the monitor must have cost 0 while those fixes remain. For each kind of monitor, either a specific definition of when it has provably zero fixed cost is given below, or else it never has provably zero fixed cost.

When one of the fixing operations just listed is called, after doing the actual fixing KHE ensures that all monitors which did not have provably zero fixed cost before but now do are detached. When one of the corresponding unfix operations is called, after doing the actual unfixing it ensures that all monitors which had provably zero fixed cost before but now do not are attached. So there is no risk that detaching these monitors could lead to cost errors; as soon as unfixes make a non-zero cost possible, they are attached again.

## 6.4. Time ranges and sweep times

Function

```
bool KheMonitorTimeRange(KHE_MONITOR m,
  KHE_TIME *first_time, KHE_TIME *last_time);
```

determines the times `t` such that the cost of `m` is affected by what is happening at time `t`. If there are any such times, it sets `*first_time` to the first of them and `*last_time` to the last of them, and returns `true`. Otherwise it sets `*first_time` and `*last_time` to `NULL` and returns `false`.

When `m` is a resource monitor, `KheMonitorTimeRange` depends only on the instance; it just returns the first and last times referenced (explicitly or implicitly) by `m`. These values are cached in `m`, so requesting them a second time has virtually no cost.

When `m` is an event monitor, `KheMonitorTimeRange` depends on the times assigned to the meets monitored by `m`. When `m` is an event resource monitor, `KheMonitorTimeRange` depends on the times assigned to the meets containing the tasks monitored by `m`. In these cases the values are not cached; they are recalculated each time they are requested. Meets without an assigned time are skipped and contribute nothing to the result.

The main use for `KheMonitorTimeRange` is in handling sweep times, as explained now.

A *time sweep* algorithm is a resource assignment algorithm that sweeps through a solution, assigning tasks in increasing order of their assigned times. A *sweep time* is a time `t` representing the point that a time sweep has reached: assignments have been or are being made up to and including `t`, but not later. Or `t` may be `NULL`, meaning that the time sweep is just beginning and the entire cycle is beyond where it has reached.

Now consider a limit active intervals monitor which specifies a minimum limit of two on the number of consecutive busy days. Starting a new sequence of busy days during a time sweep violates the minimum limit, and not starting one does not, which is misleading. The same problem afflicts cluster busy times monitors: a minimum limit can encourage a resource to be busy at the start, when it could very well remain free until later.

So monitors should not report defects that could disappear later as the time sweep proceeds. We say that tasks and times beyond the sweep time are *open*: neither assigned nor unassigned. The cost reported by a monitor is the true cost when nothing relevant is open, and a lower bound on the true cost otherwise, treating the open parts as don't knows.

There is a connection here with the 'history after' values of some resource monitors. These count notional time groups that lie beyond the end of the cycle. They are open in exactly the same way that times beyond the sweep time are open. However, they can never be not open.

Function

```
void KheMonitorSetSweepTime(KHE_MONITOR m, KHE_TIME t);
```

tells `m` that the sweep time is `t` (possibly `NULL` as above). This may change the cost of `m`.

As an example of the use of `KheMonitorSetSweepTime`, suppose a time sweep algorithm assigns one day at a time. Initially it calls `KheMonitorSetSweepTime(m, NULL)`, and then, before assigning each day *d*, it calls `KheMonitorSetSweepTime(m, t)`, where `t` is *d*'s last time. On the last call, `t` is the last time of the cycle; the time sweep has ended and nothing is open.

Behind the scenes, limit busy times and limit workload monitors which need to know what resource `r` is doing during time group `tg` share access to a private object that keeps track of this information. When they receive a new sweep time they pass it on to this object. Giving different sweep times to monitors which share the same private object will not work, although it is safe to change the sweep times of those monitors one by one to a new common value, provided no affected costs are accessed until the last change has been made.

It can be slow to call `KheMonitorSetSweepTime` for every monitor and every time (or every day). A basis for speeding things up is provided by function

```
bool KheMonitorSweepTimeRange(KHE_MONITOR m,
  KHE_TIME *first_time, KHE_TIME *last_time);
```

If the cost of `m` could possibly change when the sweep time changes, this is the same as `KheMonitorTimeRange` above. If the cost of `m` can never change when the sweep time changes, this sets `*first_time` and `*last_time` to `NULL` and returns `false`. In making this determination, it is assumed that all tasks lying beyond the sweep time are unassigned.

Calling `KheMonitorSetSweepTime` with any time `t` preceding `*first_time` is the same as calling it with `t` set to `NULL`, and calling it with any time `t` following `*last_time` is the same as calling it with `t` set to `*last_time`. So `m` needs only an initial call with `t` set to `NULL`, then calls for times between `*first_time` and `*last_time`, and finally, to bring the monitor back to its usual state, a call with `t` set to `*last_time` or any later time. Or if `KheMonitorSweepTimeRange` returns `false`, no `KheMonitorSetSweepTime` calls are needed.

All monitors have their own private versions of the functions defined here, which these functions call on. However, not all monitors are affected by sweep times. For those which are not, setting the sweep time does nothing and the sweep time range function returns `false`.

Sweep times are used by `KheTimeSweepAssignResources` (Section 12.7.3) and have been added to support that solver. `KheDynamicResourceSolverSolve` (Section 12.6) utilizes similar ideas, but within a completely different and more thorough implementation.

## 6.5. Event monitors

An *event monitor* monitors one or more events. The set of monitors (attached or unattached) which monitor a given event may be found by calling

```
int KheSolnEventMonitorCount(KHE_SOLN soln, KHE_EVENT e);
KHE_MONITOR KheSolnEventMonitor(KHE_SOLN soln, KHE_EVENT e, int i);
```

These return the number of monitors that monitor `e` in `soln`, and the `i`th of these, as usual. The timetable monitor for event `e` (Section 6.8) is not visited by these functions; it may be retrieved by calling `KheEventTimetableMonitor`.

The total cost of these monitors measures how well `e` is timetabled. Functions

```
KHE_COST KheSolnEventCost(KHE_SOLN soln, KHE_EVENT e);
KHE_COST KheSolnEventMonitorCost(KHE_SOLN soln, KHE_EVENT e,
  KHE_MONITOR_TAG tag);
```

return the total cost of all the monitors monitoring `e`, and the total cost of all monitors monitoring `e` of a specific type, defined by `tag`. `KheSolnEventMonitorCost` returns 0 when `tag` does not specify one of the monitor types in the following subsections.

Each point of application of a spread events constraint or link events constraint is one event group, and a monitor of these kinds appears on the list of monitors of each of the events in its event group. Similarly, an order events monitor appears on the list of monitors of both of the events it monitors. If `KheSolnEventCost(soln, e)` is summed over all events, the cost of such monitors is counted repeatedly, and the total may exceed the total cost of all event monitors.

Event monitors are unaffected by sweep times. This is because sweep times are concerned with time sweep resource assignment, which assumes that assigned times are not changing.

The following subsections list the various kinds of event monitors and the details specific to each of them. Their types (`KHE_SPLIT_EVENTS_MONITOR` and so on) may be obtained by downcasting from `KHE_MONITOR` after checking the type tag.

### 6.5.1. Split events monitors

A split events monitor has tag `KHE_SPLIT_EVENTS_MONITOR_TAG` and monitors an event which is one point of application of one split events constraint. Functions

```
KHE_SPLIT_EVENTS_CONSTRAINT KheSplitEventsMonitorConstraint(
  KHE_SPLIT_EVENTS_MONITOR m);
KHE_EVENT KheSplitEventsMonitorEvent(KHE_SPLIT_EVENTS_MONITOR m);
```

return the split events constraint and event being monitored, and

```
void KheSplitEventsMonitorLimits(KHE_SPLIT_EVENTS_MONITOR m,
  int *min_duration, int *max_duration, int *min_amount, int *max_amount);
```

sets the four last variables to the corresponding attributes of the monitor's constraint. Function

```
void KheSplitEventsMonitorDebug(KHE_SPLIT_EVENTS_MONITOR m,
  int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

### 6.5.2. Distribute split events monitors

A distribute split events monitor has tag `KHE_DISTRIBUTE_SPLIT_EVENTS_MONITOR_TAG` and monitors one point of application of a distribute split events constraint (one event). Functions

```
KHE_DISTRIBUTE_SPLIT_EVENTS_CONSTRAINT
  KheDistributeSplitEventsMonitorConstraint(
  KHE_DISTRIBUTE_SPLIT_EVENTS_MONITOR m);
KHE_EVENT KheDistributeSplitEventsMonitorEvent(
  KHE_DISTRIBUTE_SPLIT_EVENTS_MONITOR m);
```

return the constraint and event being monitored, and

```
void KheDistributeEventsMonitorLimits(
  KHE_DISTRIBUTE_SPLIT_EVENTS_MONITOR m,
  int *duration, int *minimum, int *maximum, int *meet_count);
```

sets `*duration`, `*minimum`, and `*maximum` to the corresponding attributes of the monitor's constraint, and `*meet_count` to the number of meets derived from the monitored event whose duration is `*duration` (or to the total number of meets if `*duration` is `KHE_ANY_DURATION`). Function

```
void KheDistributeSplitEventsMonitorDebug(
  KHE_DISTRIBUTE_SPLIT_EVENTS_MONITOR m, int verbosity,
  int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

### 6.5.3. Assign time monitors

An assign time monitor has tag `KHE_ASSIGN_TIME_MONITOR_TAG` and monitors an event which is one point of application of one assign time constraint. Functions

```
KHE_ASSIGN_TIME_CONSTRAINT KheAssignTimeMonitorConstraint(
  KHE_ASSIGN_TIME_MONITOR m);
KHE_EVENT KheAssignTimeMonitorEvent(KHE_ASSIGN_TIME_MONITOR m);
```

return the assign time constraint and event being monitored. Function

```
void KheAssignTimeMonitorDebug(KHE_ASSIGN_TIME_MONITOR m,
  int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

An assign time monitor does not have provably zero fixed cost when `KheMeetAssignFix` has been called for each of the meets derived from the event it monitors and the monitor has cost 0 when attached, because the assignments may be to other meets whose assignments are not fixed. The full assignment paths leading out of the monitored meets would need to be fixed; but that would be awkward to implement and give no efficiency payoff, because then the monitor would never be updated anyway. So an assign time monitor never has provably zero cost.

### 6.5.4. Prefer times monitors

A prefer times monitor has tag `KHE_PREFER_TIMES_MONITOR_TAG` and monitors an event which is one point of application of one prefer times constraint. Functions

```
KHE_PREFER_TIMES_CONSTRAINT KhePreferTimesMonitorConstraint(
  KHE_PREFER_TIMES_MONITOR m);
KHE_EVENT KhePreferTimesMonitorEvent(KHE_PREFER_TIMES_MONITOR m);
```

return the prefer times constraint and event being monitored. Function

```
void KhePreferTimesMonitorDebug(KHE_PREFER_TIMES_MONITOR m,
  int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

### 6.5.5. Spread events monitors

A spread events monitor has tag `KHE_SPREAD_EVENTS_MONITOR_TAG` and monitors an event group which is one point of application of a spread events constraint. It appears in the list of monitors of all the events in its event group. Functions

```
KHE_SPREAD_EVENTS_CONSTRAINT KheSpreadEventsMonitorConstraint(
  KHE_SPREAD_EVENTS_MONITOR m);
KHE_EVENT_GROUP KheSpreadEventsMonitorEventGroup(
  KHE_SPREAD_EVENTS_MONITOR m);
```

return the spread events constraint and event group being monitored. There are also

```
int KheSpreadEventsMonitorTimeGroupCount(KHE_SPREAD_EVENTS_MONITOR m);
void KheSpreadEventsMonitorTimeGroup(KHE_SPREAD_EVENTS_MONITOR m, int i,
  KHE_TIME_GROUP *time_group, int *minimum, int *maximum, int *incidences);
```

The first returns the number of time groups (as in the corresponding constraint). The second returns the `i`'th time group and the minimum and maximum number of meets wanted there (again, as in the constraint), plus the current number of meets incident on that time group. If `*incidences` is less than `*minimum` or more than `*maximum`, a cost is incurred. Function

```
void KheSpreadEventsMonitorDebug(KHE_SPREAD_EVENTS_MONITOR m,
  int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

### 6.5.6. Link events monitors

A link events monitor has tag `KHE_LINK_EVENTS_MONITOR_TAG` and monitors an event group which is one point of application of a link events constraint. It appears in the list of monitors of all the events in its event group. Functions

```
KHE_LINK_EVENTS_CONSTRAINT KheLinkEventsMonitorConstraint(
  KHE_LINK_EVENTS_MONITOR m);
KHE_EVENT_GROUP KheLinkEventsMonitorEventGroup(
  KHE_LINK_EVENTS_MONITOR m);
```

return the link events constraint and event group being monitored. Function

```
void KheLinkEventsMonitorDebug(KHE_LINK_EVENTS_MONITOR m,
  int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

A link events monitor has provably zero fixed cost when following to the end the chains of fixed assignments out of the meets of the events it monitors produces the same result for each event: the same offsets and durations within the same final meets. `KheMeetAssignFix` and `KheMeetAssignUnFix` may detach and attach link events monitors.

Detaching link events monitors is the most important service provided by fixing. Keeping these monitors up to date is slow, despite the author's best efforts to optimize. When the times of a set of linked events change together, an attached link events monitor receives the changes one by one, forcing it through a tedious sequence of cost changes beginning and ending with 0.

### 6.5.7. Order events monitors

An order events monitor has tag `KHE_ORDER_EVENTS_MONITOR_TAG` and monitors two events which together constitute one point of application of an order events constraint. It appears in the list of monitors of both events. Functions

```
KHE_ORDER_EVENTS_CONSTRAINT KheOrderEventsMonitorConstraint(
  KHE_ORDER_EVENTS_MONITOR m);
KHE_EVENT KheOrderEventsMonitorFirstEvent(KHE_ORDER_EVENTS_MONITOR m);
KHE_EVENT KheOrderEventsMonitorSecondEvent(KHE_ORDER_EVENTS_MONITOR m);
int KheOrderEventsMonitorMinSeparation(KHE_ORDER_EVENTS_MONITOR m);
int KheOrderEventsMonitorMaxSeparation(KHE_ORDER_EVENTS_MONITOR m);
```

return the constraint being monitored and the four attributes of the monitor: the two events monitored, and the minimum and maximum separations. Function

```
void KheOrderEventsMonitorDebug(KHE_ORDER_EVENTS_MONITOR m,
  int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

An order events monitor has provably zero fixed cost when both of its events are broken into a single meet, following to the end the chains of fixed assignments out of those two meets leads to the same final meet, and their separation (the offset into the final meet of the second meet, minus the duration plus offset into the final meet of the first meet) is in the legal range. `KheMeetAssignFix` and `KheMeetAssignUnFix` may detach and attach order events monitors.

## 6.6. Event resource monitors

An *event resource monitor* monitors one or more event resources. The monitors (attached or unattached) which monitor a given event resource may be visited by

```
int KheSolnEventResourceMonitorCount(KHE_SOLN soln, KHE_EVENT_RESOURCE er);
KHE_MONITOR KheSolnEventResourceMonitor(KHE_SOLN soln,
  KHE_EVENT_RESOURCE er, int i);
```

The total cost of these monitors measures how well `er` is timetabled. Functions

```
KHE_COST KheSolnEventResourceCost(KHE_SOLN soln, KHE_EVENT_RESOURCE er);
KHE_COST KheSolnEventResourceMonitorCost(KHE_SOLN soln,
  KHE_EVENT_RESOURCE er, KHE_MONITOR_TAG tag);
```

return the total cost of all the monitors monitoring `er`, and the total cost of all monitors monitoring `er` of a specific type, defined by `tag`. `KheSolnEventResourceMonitorCost` returns 0 when `tag` does not specify one of the monitor types in the following subsections.

Each point of application of an avoid split assignments constraint is a whole set of event resources, and a monitor of this kind is attached to each of the event resources in its set. If `KheSolnEventResourceCost(soln, er)` is summed over all event resources, such a monitor is counted repeatedly, so the total may exceed the total cost of all event resource monitors.

Different event resource monitors are affected by sweep times in different ways. The details are given below for each event resource type.

The following subsections list the various kinds of event resource monitors and the details specific to each of them. Their types (KHE_ASSIGN_RESOURCE_MONITOR and so on) may be obtained by downcasting from KHE_MONITOR after checking the type tag.

### 6.6.1. Assign resource monitors

An assign resource monitor has tag KHE_ASSIGN_RESOURCE_MONITOR_TAG and monitors an event resource which is one point of application of one assign resource constraint. Functions

```
KHE_ASSIGN_RESOURCE_CONSTRAINT KheAssignResourceMonitorConstraint(
  KHE_ASSIGN_RESOURCE_MONITOR m);
KHE_EVENT_RESOURCE KheAssignResourceMonitorEventResource(
  KHE_ASSIGN_RESOURCE_MONITOR m)
```

return the assign resource constraint and event resource being monitored. Like assign time monitors, assign resource monitors are never considered to have provably zero fixed cost.

Assign resource monitors are affected by sweep times: an unassigned monitored task has a cost when it lies within the time sweep, but not when it lies beyond it. This is implemented at present by ensuring that every assign resource monitor m is detached when it monitors a task beyond the sweep time, and attached otherwise. This rough implementation is correct when there is one sweep time per day, the time range of m lies within one day, and m is attached initially.

Function

```
void KheAssignResourceMonitorDebug(KHE_ASSIGN_RESOURCE_MONITOR m,
  int verbosity, int indent, FILE *fp);
```

is like KheMonitorDebug, only specific to this type of monitor.

### 6.6.2. Prefer resources monitors

A prefer resources monitor has tag KHE_PREFER_RESOURCES_MONITOR_TAG and monitors an event resource which is one point of application of one prefer resources constraint. Functions

```
KHE_PREFER_RESOURCES_CONSTRAINT KhePreferResourcesMonitorConstraint(
  KHE_PREFER_RESOURCES_MONITOR m);
KHE_EVENT_RESOURCE KhePreferResourcesMonitorEventResource(
  KHE_PREFER_RESOURCES_MONITOR m);
```

return the prefer resources constraint and event resource being monitored. Functions

```
KHE_RESOURCE_GROUP KhePreferResourcesMonitorDomain(
  KHE_PREFER_RESOURCES_MONITOR m);
KHE_RESOURCE_GROUP KhePreferResourcesMonitorDomainComplement(
  KHE_PREFER_RESOURCES_MONITOR m);
```

return m's domain and the complement of its domain in its resource type.

As Section 7.6 explains, some prefer resources monitors do not have a corresponding constraint. For those monitors, KhePreferResourcesMonitorConstraint returns NULL. So

`KhePreferResourcesMonitorDomain` and `KhePreferResourcesMonitorDomainComplement` should be used to find domains, since calling `KhePreferResourcesConstraintDomain` and `KhePreferResourcesMonitorDomainComplement` will not work when there is no constraint.

Prefer resources monitors are not affected by sweep times, because tasks are assumed to be unassigned beyond the sweep time, yielding cost 0. So, when given a prefer resources monitor, `KheMonitorSetSweepTime` does nothing and `KheMonitorSweepTimeRange` returns `false`.

Function

```
void KhePreferResourcesMonitorDebug(KHE_PREFER_RESOURCES_MONITOR m,
  int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

### 6.6.3. Avoid split assignments monitors

The operations for building avoid split assignments constraints accept a role and event groups, as required when reading XML. However, they also accept a set of event resources, and these are what are actually used. Accordingly, one avoid split assignments monitor monitors a set of event resources, and appears in the list of monitors of each of those event resources. Functions

```
KHE_AVOID_SPLIT_ASSIGNMENTS_CONSTRAINT
  KheAvoidSplitAssignmentsMonitorConstraint(
  KHE_AVOID_SPLIT_ASSIGNMENTS_MONITOR m)
int KheAvoidSplitAssignmentsMonitorEventGroupIndex(
  KHE_AVOID_SPLIT_ASSIGNMENTS_MONITOR m)
```

return the constraint and the index of the set of event resources being monitored, suitable for passing to functions `KheAvoidSplitAssignmentsConstraintEventResourceCount` and `KheAvoidSplitAssignmentsConstraintEventResource` (Section 3.7.7). There are also

```
int KheAvoidSplitAssignmentsMonitorResourceCount(
  KHE_AVOID_SPLIT_ASSIGNMENTS_MONITOR m);
KHE_RESOURCE KheAvoidSplitAssignmentsMonitorResource(
  KHE_AVOID_SPLIT_ASSIGNMENTS_MONITOR m, int i);
int KheAvoidSplitAssignmentsMonitorResourceMultiplicity(
  KHE_AVOID_SPLIT_ASSIGNMENTS_MONITOR m, int i);
```

The first returns the number of distinct resources currently assigned to tasks monitored by `m`. If `m` is a defect this number will be at least 2. The second and third return the `i`th of these distinct resources (in an arbitrary order) and the number of tasks monitored by `m` to which that resource is currently assigned. The monitor does not record which tasks those are.

Avoid split assignments monitors are not affected by sweep times, because tasks are assumed to be unassigned beyond the sweep time, and avoid split assignments monitors take no notice of unassigned tasks. So, when given an avoid split assignments monitor, `KheMonitorSetSweepTime` does nothing and `KheMonitorSweepTimeRange` returns `false`.

Function

```
void KheAvoidSplitAssignmentsMonitorDebug(
  KHE_AVOID_SPLIT_ASSIGNMENTS_MONITOR m, int verbosity,
  int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

An avoid split assignments monitor has provably zero fixed cost when the paths of fixed assignments leading out of the tasks it monitors have the same endpoint. `KheTaskAssignFix` and `KheTaskAssignUnFix` may detach and attach avoid split assignments monitors. Similarly to link events monitors, the efficiency payoff is significant.

### 6.6.4. Limit resources monitors

The operations for building limit resources constraints accept event groups and roles, as needed when reading XML. However, what one limit resources monitor actually monitors is a set of event resources, and it appears in the lists of monitors of those event resources. Functions

```
KHE_LIMIT_RESOURCES_CONSTRAINT KheLimitResourcesMonitorConstraint(
  KHE_LIMIT_RESOURCES_MONITOR m);
int KheLimitResourcesMonitorEventGroupIndex(
  KHE_LIMIT_RESOURCES_MONITOR m);
```

return the constraint, and the index within it of the set of event resources being monitored, suitable for passing to functions `KheLimitResourcesConstraintEventResourceCount` and `KheLimitResourcesConstraintEventResource` (Section 3.7.18). These allow the user to visit the monitored event resources, and thence, using `KheEventResourceTaskCount` and `KheEventResourceTask`, the monitored tasks. There is also

```
void KheLimitResourcesMonitorActiveDuration(KHE_LIMIT_RESOURCES_MONITOR m,
  int *minimum, int *maximum, int *active_durn);
```

It returns `m`'s minimum limit (taken from the constraint; it will be 0 when there is no minimum limit), its maximum limit (also from the constraint; it will be `INT_MAX` when there is no maximum limit), and the *active duration*, which is the total duration of the tasks derived from the event resources being monitored which are assigned resources from the constraint. The deviation is the amount (if any) by which `*active_durn` exceeds `*maximum` or falls short of `*minimum`.

Limit resources monitors are affected by sweep times. As for assign resources monitors, this is implemented in a rough way at present by ensuring that every limit resources monitor `m` is detached when it monitors a task beyond the sweep time, and attached otherwise.

Function

```
void KheLimitResourcesMonitorDebug(KHE_LIMIT_RESOURCES_MONITOR m,
  int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

## 6.7. Resource monitors

A *resource monitor* monitors a resource. The set of monitors (attached or unattached) which monitor a given resource may be visited by calling

```
int KheSolnResourceMonitorCount(KHE_SOLN soln, KHE_RESOURCE r);
KHE_MONITOR KheSolnResourceMonitor(KHE_SOLN soln, KHE_RESOURCE r, int i);
```

The order is arbitrary and may be changed by calling

```
void KheSolnResourceMonitorSort(KHE_SOLN soln, KHE_RESOURCE r,
  int(*compar)(const void *, const void *));
```

to sort the monitors into increasing order of `compar`.

The total cost of these monitors measures how well `r` is timetabled. Functions

```
KHE_COST KheSolnResourceCost(KHE_SOLN soln, KHE_RESOURCE r);
KHE_COST KheSolnResourceMonitorCost(KHE_SOLN soln, KHE_RESOURCE r,
  KHE_MONITOR_TAG tag);
```

return the total cost of all the monitors monitoring `r`, and the total cost of all monitors monitoring `r` of a specific type, defined by `tag`. `KheSolnResourceMonitorCost` returns 0 when `tag` does not specify one of the monitor types in the following subsections.

Generally speaking, resource monitors are affected by sweep times. (The exceptions are avoid clashes and avoid unavailable times monitors, for which the usual assumption that tasks are unassigned beyond the sweep time implies cost 0.) The implementations have been done with care, and almost always yield the best possible lower bound on true cost.

The following subsections list the kinds of resource monitors and their features. Their types (`KHE_AVOID_CLASHES_MONITOR` etc.) may be obtained by downcasting from `KHE_MONITOR` after checking the type tag. Monitors of type `KHE_WORKLOAD_DEMAND_MONITOR`, defined in Section 7.3, are also visited by `KheSolnResourceMonitorCount` and `KheSolnResourceMonitor`. However, the timetable monitor for a resource is not visited by these functions; as explained in Section 6.8, it is retrieved by calling `KheResourceTimetableMonitor`.

### 6.7.1. Avoid clashes monitors

An avoid clashes monitor has tag `KHE_AVOID_CLASHES_MONITOR_TAG` and monitors a resource which is one point of application of one avoid clashes constraint. Functions

```
KHE_AVOID_CLASHES_CONSTRAINT KheAvoidClashesMonitorConstraint(
  KHE_AVOID_CLASHES_MONITOR m);
KHE_RESOURCE KheAvoidClashesMonitorResource(
  KHE_AVOID_CLASHES_MONITOR m);
```

return the avoid clashes constraint and resource being monitored. Function

```
void KheAvoidClashesMonitorDebug(KHE_AVOID_CLASHES_MONITOR m,
  int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

An avoid clashes monitor `m` may have non-zero `KheMonitorLowerBound(m)`. Let $t$ be the total duration of the events to which `m`'s resource is preassigned which either have preassigned times or are subject to an assign time constraint of weight greater than `m`'s weight. Then if $t$ exceeds the number of times in the cycle, the excess is a lower bound on the number of defects that `m` must have in any reasonable solution (one in which violations of `m` are preferred to violations of the more expensive assign time constraints). Converting this number of defects into a cost using `m`'s cost function in the usual way gives the lower bound.

### 6.7.2. Avoid unavailable times monitors

This monitor has tag `KHE_AVOID_UNAVAILABLE_TIMES_MONITOR_TAG` and monitors a resource which is one point of application of one avoid unavailable times constraint. Functions

```
KHE_AVOID_UNAVAILABLE_TIMES_CONSTRAINT
  KheAvoidUnavailableTimesMonitorConstraint(
  KHE_AVOID_UNAVAILABLE_TIMES_MONITOR m);
KHE_RESOURCE KheAvoidUnavailableTimesMonitorResource(
  KHE_AVOID_UNAVAILABLE_TIMES_MONITOR m);
```

return the avoid unavailable times constraint and resource being monitored. Function

```
void KheAvoidUnavailableTimesMonitorDebug(
  KHE_AVOID_UNAVAILABLE_TIMES_MONITOR m, int verbosity,
  int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

An avoid unavailable times monitor `m` may have non-zero `KheMonitorLowerBound(m)`. Suppose `m`'s resource is subject to an avoid clashes constraint of weight greater than `m`'s weight. Let $t_1$ be the total duration of the events to which `m`'s resource is preassigned which either have preassigned times or are subject to an assign time constraint of weight greater than `m`'s weight. Let $t_2$ be the number of times to be avoided according to `m`. Then if $t_1 + t_2$ exceeds the number of times in the cycle, the excess is a lower bound on the number of defects that `m` must have in any reasonable solution (one in which every meet is assigned a time, and violations of `m` are preferred to violations of the more expensive assign time and avoid clashes constraints). Converting this number of defects into a cost using `m`'s cost function in the usual way gives the lower bound.

### 6.7.3. Limit idle times monitors

A limit idle times monitor has tag `KHE_LIMIT_IDLE_TIMES_MONITOR_TAG` and monitors a resource which is one point of application of one limit idle times constraint. Functions

```
KHE_LIMIT_IDLE_TIMES_CONSTRAINT KheLimitIdleTimesMonitorConstraint(
  KHE_LIMIT_IDLE_TIMES_MONITOR m);
KHE_RESOURCE KheLimitIdleTimesMonitorResource(
  KHE_LIMIT_IDLE_TIMES_MONITOR m);
```

return the limit idle times constraint and resource being monitored, and

```
int KheLimitIdleTimesMonitorTimeGroupCount(
  KHE_LIMIT_IDLE_TIMES_MONITOR m);
KHE_TIME_GROUP KheLimitIdleTimesMonitorTimeGroup(
  KHE_LIMIT_IDLE_TIMES_MONITOR m, int i);
```

visit the time groups that `m` monitors, that is, the time groups from the constraint. There is also

```
KHE_TIME_GROUP KheLimitIdleTimesMonitorTimeGroupState(
  KHE_LIMIT_IDLE_TIMES_MONITOR m, int i, int *busy_count, int *idle_count,
  KHE_TIME extreme_busy_times[2], int *extreme_busy_times_count);
```

which, in addition to returning the `i`th time group, also reports its state, by setting `*busy_count` to its number of busy times, `*idle_count` to its number of idle times, and placing its first and last busy times into `extreme_busy_times[0 .. *extreme_busy_times_count - 1]`. If there are no busy times, `*extreme_busy_times_count` is 0; if there is one it is 1; otherwise it is 2. Function

```
void KheLimitIdleTimesMonitorDebug(KHE_LIMIT_IDLE_TIMES_MONITOR m,
  int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

### 6.7.4. Cluster busy times monitors

A cluster busy times monitor (tag `KHE_CLUSTER_BUSY_TIMES_MONITOR_TAG`) monitors a resource and offset making one point of application of a cluster busy times constraint. Functions

```
KHE_CLUSTER_BUSY_TIMES_CONSTRAINT KheClusterBusyTimesMonitorConstraint(
  KHE_CLUSTER_BUSY_TIMES_MONITOR m);
KHE_RESOURCE KheClusterBusyTimesMonitorResource(
  KHE_CLUSTER_BUSY_TIMES_MONITOR m);
```

return the cluster busy times constraint and the resource being monitored. Functions

```
int KheClusterBusyTimesMonitorHistoryBefore(
  KHE_CLUSTER_BUSY_TIMES_MONITOR m);
int KheClusterBusyTimesMonitorHistoryAfter(
  KHE_CLUSTER_BUSY_TIMES_MONITOR m);
int KheClusterBusyTimesMonitorHistory(
  KHE_CLUSTER_BUSY_TIMES_MONITOR m);
```

return the history before, history after, and history values from `m`'s constraint, or 0 if not present. In the high school model, these are always 0. Function

```
int KheClusterBusyTimesMonitorOffset(KHE_CLUSTER_BUSY_TIMES_MONITOR m);
```

returns the offset being monitored. In the high school model, and when the constraint has `NULL` for `applies_to_tg`, the offset is always 0, otherwise the offset is the difference in index between one useful time in `applies_to_tg` and the first time in `applies_to_tg`. Functions

```
int KheClusterBusyTimesMonitorTimeGroupCount(
  KHE_CLUSTER_BUSY_TIMES_MONITOR m);
KHE_TIME_GROUP KheClusterBusyTimesMonitorTimeGroup(
  KHE_CLUSTER_BUSY_TIMES_MONITOR m, int i, KHE_POLARITY *po);
```

return the time groups that `m` monitors (one for each time group in the cluster busy times constraint, adjusted using `KheTimeGroupNeighbour` by the offset), and their associated polarities.

Two functions report the current state of the monitor, as it varies during the solve. Function

```
void KheClusterBusyTimesMonitorActiveTimeGroupCount(
  KHE_CLUSTER_BUSY_TIMES_MONITOR m, int *active_group_count,
  *open_group_count, int *minimum, int *maximum, bool *allow_zero);
```

sets `*active_group_count` to the number of active time groups (busy positive time groups plus non-busy negative time groups), `*open_group_count` to the number of time groups not known to be either active or inactive (because `history_after` is non-zero, or because there is a non-trivial sweep time), and `*minimum`, `*maximum`, and `*allow_zero` to the values from the constraint. If `m` has non-zero cost, then either `*active_group_count + *open_group_count < *minimum` or `*active_group_count > *maximum`. Function

```
bool KheClusterBusyTimesMonitorTimeGroupIsActive(
  KHE_CLUSTER_BUSY_TIMES_MONITOR m, int i, KHE_TIME_GROUP *tg,
  KHE_POLARITY *po, int *busy_count);
```

returns `true` when the time group at index `i` is currently active. It also sets `*tg` and `*po` to the time group and polarity at index `i`, and `*busy_count` to the number of busy times in the time group. It returns `true` when `*tg` is active, that is, when

```
(*po == KHE_NEGATIVE) == (*busy_count == 0)
```

as the definition of the constraint specifies.

There may be value in obtaining advance warning that a constraint is close to being violated. For that there is function

```
int KheClusterBusyTimesMonitorAtMaxLimitCount(
  KHE_CLUSTER_BUSY_TIMES_MONITOR m);
```

It returns 1 if the monitor is not detecting a violation but the number of active time groups equals the maximum limit, and 0 otherwise. It returns an integer rather than a boolean for consistency with `KheLimitActiveIntervalsMonitorAtMaxLimitCount`.

There is a peculiar but apparently unavoidable asymmetry in the handling of cluster time groups at or after the sweep time: if they are busy they have a definite state, either active or inactive, but if they are not busy they are open. This can be mitigated by calling

```
void KheClusterBusyTimesMonitorSetNotBusyState(
  KHE_CLUSTER_BUSY_TIMES_MONITOR m, int i, bool active);
```

where `i` is the index of one of `m`'s time groups, call it `tg`. This informs `m` that when `tg` is at or after the sweep time and is not busy, it should be considered either active or inactive (depending on the `active` parameter) rather than open. No other cases are affected. There are no restrictions on when this function can be called, relative to setting sweep times or anything else. It may change the cost of `m`. Function

```
void KheClusterBusyTimesMonitorClearNotBusyState(
  KHE_CLUSTER_BUSY_TIMES_MONITOR m, int i);
```

returns `tg` to its default state. (In practice, there is no reason to call this function, because as the sweep time increases these effects become irrelevant anyway.)

The user can set the monitor's minimum limit:

```
int KheClusterBusyTimesMonitorMaximum(KHE_CLUSTER_BUSY_TIMES_MONITOR m);
int KheClusterBusyTimesMonitorMinimum(KHE_CLUSTER_BUSY_TIMES_MONITOR m);
bool KheClusterBusyTimesMonitorSetMinimum(
  KHE_CLUSTER_BUSY_TIMES_MONITOR m, int val);
void KheClusterBusyTimesMonitorResetMinimum(
  KHE_CLUSTER_BUSY_TIMES_MONITOR m);
```

`KheClusterBusyTimesMonitorMaximum` returns `m`'s maximum limit. This is always equal to the maximum limit of `m`'s constraint. `KheClusterBusyTimesMonitorMinimum` returns the current value of `m`'s minimum limit. Its default value is the minimum limit stored unchangeably in `m`'s constraint. But `KheClusterBusyTimesMonitorSetMinimum` can be called at any time to change the monitor's minimum limit. If the monitor is attached when this is done, the solution cost may change immediately. `KheClusterBusyTimesMonitorResetMinimum` resets the minimum limit to the value stored in the constraint. Alternatively, `KheSolnEnsureOfficialCost` (Section 6.3) can be called; it resets the minimums of all monitors to their original values.

Finally, function

```
void KheClusterBusyTimesMonitorDebug(KHE_CLUSTER_BUSY_TIMES_MONITOR m,
  int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

### 6.7.5. Limit busy times monitors

A limit busy times monitor (tag `KHE_LIMIT_BUSY_TIMES_MONITOR_TAG`) monitors a resource and offset which make up one point of application of a limit busy times constraint. Functions

```
KHE_LIMIT_BUSY_TIMES_CONSTRAINT KheLimitBusyTimesMonitorConstraint(
  KHE_LIMIT_BUSY_TIMES_MONITOR m);
KHE_RESOURCE KheLimitBusyTimesMonitorResource(
  KHE_LIMIT_BUSY_TIMES_MONITOR m);
int KheLimitBusyTimesMonitorOffset(KHE_LIMIT_BUSY_TIMES_MONITOR m);
```

return the limit busy times constraint and the resource and offset being monitored. In the high school model, and when the constraint has `NULL` for `applies_to_tg`, the offset is always 0, otherwise the offset is the difference in index between one useful time in `applies_to_tg` and the first time in `applies_to_tg`.

The monitored time groups (after applying the offset) are returned by

```
int KheLimitBusyTimesMonitorTimeGroupCount(
  KHE_LIMIT_BUSY_TIMES_MONITOR m);
KHE_TIME_GROUP KheLimitBusyTimesMonitorTimeGroup(
  KHE_LIMIT_BUSY_TIMES_MONITOR m, int i, int *busy_count);
```

with `*busy_count` set to the number of busy times in the time group. The time groups are those of the constraint, adjusted using `KheTimeGroupNeighbour` by the offset.

Functions

```
int KheLimitBusyTimesMonitorDefectiveTimeGroupCount(
  KHE_LIMIT_BUSY_TIMES_MONITOR m);
void KheLimitBusyTimesMonitorDefectiveTimeGroup(
  KHE_LIMIT_BUSY_TIMES_MONITOR m, int i, KHE_TIME_GROUP *tg,
  int *busy_count, int *open_count, int *minimum, int *maximum,
  bool *allow_zero);
```

visit the time groups monitored by `m` that are currently defective. The order in which they appear depends on their times. This means that if a time group changes from defective to non-defective then back to defective, it reappears in its previous position in the list. Without this, a traversal of the list trying to remove each defective time group in turn would not work properly.

For each `i`, `*tg` is set to one defective time group, `*busy_count` is set to the number of times `m`'s resource is busy during `*tg`, `*open_count` is set to the number of open times in `*tg` (determined by the sweep time), and `*minimum`, `*maximum`, and `*allow_zero` come from the constraint. So either the resource is overloaded during `*tg` and `*busy_count > *maximum`, or it is underloaded during `*tg` and `*busy_count + *open_count < *minimum`. Also,

```
int KheLimitBusyTimesMonitorDeviation(KHE_LIMIT_BUSY_TIMES_MONITOR m);
```

returns the deviation.

Limit busy times monitors contain a `ceiling` attribute, set and retrieved by

```
void KheLimitBusyTimesMonitorSetCeiling(KHE_LIMIT_BUSY_TIMES_MONITOR m,
  int ceiling);
int KheLimitBusyTimesMonitorCeiling(KHE_LIMIT_BUSY_TIMES_MONITOR m);
```

When `busy_count > ceiling`, the usual formula is overridden: the deviation is 0. For why this might be useful, consult Section 8.8.3. The default value of `ceiling` is `INT_MAX`, which effectively turns it off. If `m` is attached when `KheLimitBusyTimesMonitorSetCeiling` is called, it will be detached and reattached by the call.

Function

```
void KheLimitBusyTimesMonitorDebug(KHE_LIMIT_BUSY_TIMES_MONITOR m,
  int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

A limit busy times monitor `m` may have non-zero `KheMonitorLowerBound(m)`. Suppose `m`'s resource is subject to an avoid clashes constraint of weight greater than m's weight. Let $t_1$ be the total duration of the events to which m's resource is preassigned which either have preassigned times or are subject to an assign time constraint of weight greater than m's weight. Let $t_2$ be the number of times in the cycle minus the number of times in m's constraint's domain. Then at least $t_1 - t_2$ of the times of the events preassigned to m's resource must occur in time groups limited by m. If this exceeds the number of time groups in m's constraint times its Maximum, then the excess, converted into a cost in the usual way, gives the lower bound. Monitors are only created for offsets applicable to all times in the constraint, so this lower bound is the same for all offsets.

### 6.7.6. Limit workload monitors

A limit workload monitor has tag KHE_LIMIT_WORKLOAD_MONITOR and monitors a resource which is one point of application of one limit workload constraint. Functions

```
KHE_LIMIT_WORKLOAD_CONSTRAINT KheLimitWorkloadMonitorConstraint(
  KHE_LIMIT_WORKLOAD_MONITOR m);
KHE_RESOURCE KheLimitWorkloadMonitorResource(
  KHE_LIMIT_WORKLOAD_MONITOR m);
int KheLimitWorkloadMonitorOffset(KHE_LIMIT_WORKLOAD_MONITOR m);
```

return the limit workload constraint and the resource and offset being monitored. In the high school model, and when the constraint has NULL for applies_to_tg, the offset is always 0, otherwise the offset is the difference in index between one useful time in applies_to_tg and the first time in applies_to_tg.

The monitored time groups (after applying the offset) are returned by

```
int KheLimitWorkloadMonitorTimeGroupCount(
  KHE_LIMIT_WORKLOAD_MONITOR m);
KHE_TIME_GROUP KheLimitWorkloadMonitorTimeGroup(
  KHE_LIMIT_WORKLOAD_MONITOR m, int i, float *workload);
```

with *workload set to the current workload of the ith time group. The time groups are from the constraint, adjusted using KheTimeGroupNeighbour by the offset.

Functions

```
int KheLimitWorkloadMonitorDefectiveTimeGroupCount(
  KHE_LIMIT_WORKLOAD_MONITOR m);
void KheLimitWorkloadMonitorDefectiveTimeGroup(
  KHE_LIMIT_WORKLOAD_MONITOR m, int i, KHE_TIME_GROUP *tg,
  float *workload, int *open_count, int *minimum, int *maximum,
  bool *allow_zero);
```

visit the time groups monitored by m that are currently defective. The order in which they appear depends on their times. This means that if a time group changes from defective to non-defective then back to defective, it reappears in its previous position in the list. Without this, a traversal of the list trying to remove each defective time group in turn would not work properly.

For each i, *tg is set to one defective time group, *workload is set to the workload of m's resource during *tg, *open_count is the number of open times in *tg (determined by the sweep time), and `*minimum`, `*maximum`, and `*allow_zero` come from the constraint. So either the resource is underloaded during `*tg` and `*workload < *minimum`, or it is overloaded during `*tg` and `*workload > *maximum`. If `*open_count > 0`, sweep times are in effect, so the comparison against `*minimum` is adjusted to allow for the possibility that `*open_count` more times could be busy, each with workload per time up to `KheResourceTypeMaxWorkloadPerTime(rt)`, where `rt` is the type of `m`'s resource:

```
*workload + *open_count * KheResourceTypeMaxWorkloadPerTime(rt) < *minimum
```

See Section 3.5.1 for `KheResourceTypeMaxWorkloadPerTime`. Also,

```
int KheLimitWorkloadMonitorDeviation(KHE_LIMIT_WORKLOAD_MONITOR m);
```

returns the deviation of `m`.

Limit workload monitors contain a `ceiling` attribute, set and retrieved by

```
void KheLimitWorkloadMonitorSetCeiling(KHE_LIMIT_WORKLOAD_MONITOR m,
  int ceiling);
int KheLimitWorkloadMonitorCeiling(KHE_LIMIT_WORKLOAD_MONITOR m);
```

When `workload > ceiling`, the usual formula is overridden: the deviation is 0. For why this might be useful, consult Section 8.8.3. The default value of `ceiling` is `INT_MAX`, which effectively turns it off. If `m` is attached when `KheLimitWorkloadMonitorSetCeiling` is called, it will be detached and reattached by the call.

Function

```
void KheLimitWorkloadMonitorDebug(KHE_LIMIT_WORKLOAD_MONITOR m,
  int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

A limit workload monitor `m` may have non-zero `KheMonitorLowerBound(m)`. This is true in all cases, but at present KHE only calculates a potentially non-zero lower bound when `m` monitors the whole cycle. In that case, add up the workloads of the tasks to which `m`'s resource is preassigned. If this exceeds the maximum of the corresponding limit workload constraint, converting the excess into a cost using `m`'s cost function in the usual way gives the lower bound.

## 6.7.7. Limit active intervals monitors

A limit active intervals monitor has tag `KHE_LIMIT_ACTIVE_INTERVALS_MONITOR_TAG` and monitors a resource and offset which together make one point of application of one limit active intervals constraint. Limit active intervals constraints occur only in the employee scheduling model, so limit active intervals monitors also occur only in that model. Functions

```
KHE_LIMIT_ACTIVE_INTERVALS_CONSTRAINT
  KheLimitActiveIntervalsMonitorConstraint(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m);
KHE_RESOURCE KheLimitActiveIntervalsMonitorResource(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m);
```

return the limit active intervals constraint and the resource being monitored. Functions

```
int KheLimitActiveIntervalsMonitorMinimum(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m);
int KheLimitActiveIntervalsMonitorMaximum(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m);
```

return the minimum and maximum limits from the constraint.

```
int KheLimitActiveIntervalsMonitorHistoryBefore(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m);
int KheLimitActiveIntervalsMonitorHistoryAfter(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m);
int KheLimitActiveIntervalsMonitorHistory(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m);
```

return the history before, history after, and history values from `m`'s constraint, or 0 if not present. Function

```
int KheLimitActiveIntervalsMonitorOffset(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m);
```

returns the offset being monitored. When the constraint has `NULL` for `applies_to_tg`, the offset is 0, otherwise it is the difference in index between one useful time in `applies_to_tg` and the first time in `applies_to_tg`. Functions

```
int KheLimitActiveIntervalsMonitorTimeGroupCount(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m);
KHE_TIME_GROUP KheLimitActiveIntervalsMonitorTimeGroup(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m, int i, KHE_POLARITY *po);
```

return the time groups that `m` monitors (one for each time group in the limit active intervals constraint, adjusted using `KheTimeGroupNeighbour` by the offset), and their associated polarities.

There are also functions which report the state of the monitor during the solve. Function

```
bool KheLimitActiveIntervalsMonitorTimeGroupIsActive(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m, int i, KHE_TIME_GROUP *tg,
  KHE_POLARITY *po, int *busy_count);
```

returns `true` when the time group at index `i` is currently active. It sets `*tg` and `*po` to the time group and polarity at index `i`, and `*busy_count` to the number of busy times in the time group. It returns the value of the condition `(*po == KHE_NEGATIVE) == (*busy_count == 0)`, as the definition of the constraint specifies.

For visiting defective active intervals (active intervals whose length is less than the minimum limit or greater than the maximum limit from the constraint), functions

```
int KheLimitActiveIntervalsMonitorDefectiveIntervalCount(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m);
void KheLimitActiveIntervalsMonitorDefectiveInterval(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m, int i, int *history_before,
  int *first_index, int *last_index, int *history_after, bool *too_long);
```

return the number of defective active intervals and attributes of the `i`th defective active interval:

> `*history_before`. If the interval includes the first time group, the part of its length from before there (i.e. `KheLimitActiveIntervalsMonitorHistory(m)`), otherwise 0.
>
> `*first_index`. The index of the first time group in the interval, not including any history part, so always at least `0`.
>
> `*last_index`. The index of the last time group in the interval, not including any history part, so always at most `KheLimitActiveIntervalsMonitorTimeGroupCount(m) - 1`. The value could be negative, indicating that the interval lies entirely within history.
>
> `*history_after`. If the interval includes the last time group, the part of its length from after the last time group. This must be 0 when the the interval violates a maximum limit.
>
> `*too_long`. Since this is a defective interval, its length must either be too long or too short. This value is `true` if it is too long, and `false` if it is too short.

The value compared with the limits is

```
*history_before + (*last_index - *first_index + 1) + *history_after
```

See Jeff Kingston's paper on history for the rationale for this. All these definitions hold good (although their consequences are not quite obvious) when there is a sweep time.

In rare cases, `KheLimitActiveIntervalsMonitorDefectiveInterval` sets `*last_index` to `-1`. This indicates that there is a defective interval lying entirely within the history range. A solver can do nothing about this; it must check this condition and do nothing when it occurs.

`KheLimitActiveIntervalsMonitorDefectiveInterval` visits the defective intervals in increasing order of `*first_index`. This ensures that if, between calls to this function, the solution is changed, then changed back again to its previous state, a partially completed traversal of defective intervals using this function is not invalidated.

There may be value in obtaining advance warning that a constraint is close to being violated. For that there is function

```
int KheLimitActiveIntervalsMonitorAtMaxLimitCount(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m);
```

It returns the number of active intervals which do not violate any limits, but whose length equals the maximum limit. It has been considered most efficient to not maintain this value incrementally; instead, the list of non-violating intervals is scanned when this function is called.

There is a peculiar but apparently unavoidable asymmetry in the handling of time groups at or after the sweep index: if they are busy they have a definite state, either active or inactive, but if they are not busy they are open. This can be mitigated by calling

```
void KheLimitActiveIntervalsMonitorSetNotBusyState(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m, int i, bool active);
```

where `i` is the index of one of `m`'s time groups, call it `tg`. This informs `m` that when `tg` is at or after the sweep index and is not busy, it should be considered either active or inactive (depending on the `active` parameter) rather than open. No other cases are affected. There are no restrictions on when this function can be called, relative to setting the sweep index or anything else. It may change the cost of `m`. Function

```
void KheLimitActiveIntervalsMonitorClearNotBusyState(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m, int i);
```

returns `tg` to its default state. (In practice, there is no reason to call this function, because as the sweep index increases these effects become irrelevant anyway.)

For example, suppose that some resource `r` has requested that a certain time group `tg` be kept free. Suppose that `r` is subject to a limit active intervals monitor `m`, whose `i`th time group is a subset of `tg`. Then we can expect that time group to be free, and hence active if it is negative and inactive if it is positive. This expectation can be conveyed to `m` by calling `KheLimitActiveIntervalsMonitorSetNotBusyState`. This can make a significant difference to time sweep solvers when `m` has a non-trivial minimum limit (2 or more), by penalizing them for starting a new sequence of busy days just before a resource is due for some free time.

To support *tilting the plateau* (Section 11.5.2), KHE offers functions

```
void KheLimitActiveIntervalsMonitorSetTilt(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m);
void KheLimitActiveIntervalsMonitorClearTilt(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m);
bool KheLimitActiveIntervalsMonitorTilt(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m)
```

The tilt is a Boolean flag stored in `m` which changes the costs of defective intervals slightly when set. `KheLimitActiveIntervalsMonitorSetTilt` sets the flag to `true`, or does nothing if it is already `true`. `KheLimitActiveIntervalsMonitorClearTilt` sets the flag to `false`, or does nothing if it is already `false`. `KheLimitActiveIntervalsMonitorTilt` returns the current value of the flag. For why tilting is useful, and how the costs change, consult Section 11.5.2.

If `m` is attached when the flag's value is changed, it will be detached before the change and reattached afterwards. The cost after reattaching may differ from the cost before detaching.

Finally, function

```
void KheLimitActiveIntervalsMonitorDebug(
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m, int verbosity,
  int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

## 6.8. Timetable monitors

A *timetable* is a record of what is going on at each time. As part of monitoring cost, KHE monitors the timetable of each event and each resource.

### 6.8.1. Event timetable monitors

Function

```
KHE_EVENT_TIMETABLE_MONITOR KheEventTimetableMonitor(KHE_SOLN soln,
  KHE_EVENT e);
```

returns the event timetable monitor of event `e`. Type `KHE_EVENT_TIMETABLE_MONITOR` is a subtype of `KHE_MONITOR` with tag `KHE_EVENT_TIMETABLE_MONITOR_TAG`.

An event timetable monitor always has cost 0. When it is attached, a particular set of meets is known to it at any moment: the set of meets derived from `e` that are assigned a time. The monitor offers these operations, which report which meets are running at each time:

```
int KheEventTimetableMonitorTimeMeetCount(
  KHE_EVENT_TIMETABLE_MONITOR etm, KHE_TIME time);
KHE_MEET KheEventTimetableMonitorTimeMeet(
  KHE_EVENT_TIMETABLE_MONITOR etm, KHE_TIME time, int i);
```

`KheEventTimetableMonitorTimeMeetCount` returns the number of meets running at `time`, and `KheEventTimetableMonitorTimeMeet` returns the `i`th of these meets. Closely related is

```
bool KheEventTimetableMonitorTimeAvailable(
  KHE_EVENT_TIMETABLE_MONITOR etm, KHE_MEET meet, KHE_TIME time);
```

which returns `true` if moving `meet` within `etm`, or adding it to `etm`, so that its starting time is `time`, would neither place `meet` partly off the end of the timetable nor cause clashes.

An event timetable monitor offers no operations which report its set of meets directly. For that, call functions `KheEventMeetCount` and `KheEventMeet` from Section 4.2.7 to obtain the meets derived from a particular event; the timetabled meets are those with an assigned time.

As usual, event timetable monitors are created by `KheSolnMake` and exist for as long as the solution does. There is one for each event. Link events monitors (but not spread events monitors) depend on event timetable monitors.

Unlike most monitors, event timetable monitors are not attached initially. The event timetable monitor returned by `KheEventTimetableMonitor` may be unattached and so not up to date (it will be empty in that case). When a monitor is attached, any unattached timetable monitor(s) it depends on are also attached. When the last monitor that depends on some event timetable monitor is detached, that event timetable monitor is detached. Thus, unless the user chooses to attach an event timetable monitor explicitly, it will be attached only as needed by other monitors. Detaching an event timetable monitor does nothing unless no attached monitors depend on it. In practice, when using an event timetable monitor `etm`, it is best to call

```
if( !KheMonitorAttachedToSoln((KHE_MONITOR) etm) )
  KheMonitorAttachToSoln((KHE_MONITOR) etm);
```

beforehand, and

```
KheMonitorDetachFromSoln((KHE_MONITOR) etm);
```

afterwards, unless `etm` must be attached, because some monitor that depends on it is attached.

Although it would make sense to treat an event timetable monitor as a group monitor (Section 6.9), that option is not offered. The user who wants all the problems associated with a given event to be channelled through a single monitor must create a group monitor, separate from the event timetable monitor, and add the appropriate monitors to it in the usual way.

Event timetable monitors may be debugged by calling `KheEventTimetableMonitorDebug` (defined below) as usual. And

```
void KheEventTimetableMonitorPrintTimetable(
  KHE_EVENT_TIMETABLE_MONITOR etm, int cell_width, int indent, FILE *fp);
```

prints a conventional tabular timetable, using `Days` and possibly `Weeks` time groups from the instance to determine its shape. Parameter `cell_width` is the width of each cell, in characters.

The user may create an event timetable monitor by calling

```
KHE_EVENT_TIMETABLE_MONITOR KheEventTimetableMonitorMake(KHE_SOLN soln,
  KHE_EVENT_GROUP eg);
```

The result monitors the meets of `soln` derived from the events of `eg`, and thus offers a way to keep track of which events of `eg` are running at each time, something which is not otherwise available in KHE. It can be attached and detached at will in the usual way. Initially, it is detached, so in practice its creation would always be followed by a call to `KheMonitorAttachToSoln`.

To delete an event timetable monitor made in this way, call

```
KheEventTimetableMonitorDelete(KHE_EVENT_TIMETABLE_MONITOR etm);
```

This function begins by detaching `etm` if it is attached. Function

```
void KheEventTimetableMonitorDebug(KHE_EVENT_TIMETABLE_MONITOR etm,
  int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

### 6.8.2. Resource timetable monitors

Function

```
KHE_RESOURCE_TIMETABLE_MONITOR KheResourceTimetableMonitor(
  KHE_SOLN soln, KHE_RESOURCE r);
```

returns the resource timetable monitor of resource `r`. Type `KHE_RESOURCE_TIMETABLE_MONITOR` is a subtype of `KHE_MONITOR` with tag `KHE_RESOURCE_TIMETABLE_MONITOR_TAG`. Functions

```
KHE_SOLN KheResourceTimetableMonitorSoln(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm);
KHE_RESOURCE KheResourceTimetableMonitorResource(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm);
```

return `rtm`'s solution and resource attributes.

A resource timetable monitor always has cost 0. When it is attached, a particular set of tasks is known to it at any moment: those assigned the resource (either directly, or indirectly via other tasks) whose enclosing meet is assigned a time (either directly, or indirectly via other meets). The monitor offers these operations, which report which tasks are running at each time:

```
int KheResourceTimetableMonitorTimeTaskCount(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm, KHE_TIME time);
KHE_TASK KheResourceTimetableMonitorTimeTask(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm, KHE_TIME time, int i);
```

`KheResourceTimetableMonitorTimeTaskCount` returns the number of tasks running at `time`; `KheResourceTimetableMonitorTimeTask` returns the `i`th of these tasks.

Several functions work out whether `rtm`'s resource is free, or available. Function

```
bool KheResourceTimetableMonitorTimeAvailable(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm, KHE_MEET meet, KHE_TIME time);
```

returns `true` if moving `meet` within `rtm`, or adding it to `rtm`, so that its starting time is `time`, would neither place `meet` partly off the end of the timetable nor cause clashes.

Function

```
int KheResourceTimetableMonitorBusyTimesForTimeGroup(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm, KHE_TIME_GROUP tg);
```

returns the number of times during `tg` when `rtm`'s resource is busy attending one or more tasks.

Here are three functions for determining whether a resource is free at certain times:

```
bool KheResourceTimetableMonitorFreeForTime(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm, KHE_TIME t,
  KHE_FRAME frame, KHE_TASK ignore_task, bool ignore_nocost_tasks);
bool KheResourceTimetableMonitorFreeForTimeGroup(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm, KHE_TIME_GROUP tg,
  KHE_FRAME frame, KHE_TASK ignore_task, bool ignore_nocost_tasks);
bool KheResourceTimetableMonitorFreeForTask(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm, KHE_TASK task,
  KHE_FRAME frame, KHE_TASK ignore_task, bool ignore_nocost_tasks);
```

All three functions return `true` when `rtm`'s resource is free (that is, not assigned to any task) at a certain set of times $T$. For `KheResourceTimetableMonitorFreeForTime`, $T$ is the set of times containing just `t`. For `KheResourceTimetableMonitorFreeForTimeGroup`, $T$ is the set of times of `tg`. For `KheResourceTimetableMonitorFreeForTask`, $T$ is the set of times that `task` is running, including any tasks assigned to `task`, directly or indirectly.

For all three functions, if `frame != NULL` then the test is extended to include every time in every time group of `frame` that contains at least one time from *T*. This is useful when `rtm`'s resource can be assigned to at most one task per day.

For all three functions, if `ignore_task != NULL`, then that task is ignored if it is present in the timetable. This is useful when investigating whether a task swap is feasible.

For all three functions, if `ignore_nocost_tasks` is `true`, then every task `t` for which `KheTaskNeedsAssignment(t)` returns `false` is ignored. This is useful for deciding whether the resource would be free if it was unassigned from these tasks, something that incurs no cost.

Here are three similar functions that determine whether a resource is available (that is, not subject to an avoid unavailable times constraint) at certain times:

```
bool KheResourceTimetableMonitorAvailableForTime(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm, KHE_TIME t);
bool KheResourceTimetableMonitorAvailableForTimeGroup(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm, KHE_TIME_GROUP tg);
bool KheResourceTimetableMonitorAvailableForTask(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm, KHE_TASK task);
```

The same times *T* are used, but here the test is for whether there is an avoid unavailable times constraint at any of the times. This is why the other parameters are omitted. For example, there is no `frame` parameter because unavailable times are not influenced by days like busy times are.

Next we have

```
void KheResourceTimetableMonitorAddProperRootTasks(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm, KHE_TIME_GROUP tg,
  bool include_preassigned, KHE_TASK_SET ts);
```

This adds to existing task set `ts` the proper root tasks of the tasks of `rtm` that overlap with time group `tg`. Here `tg` may be `NULL`, in which case the time group containing every time of the cycle is used. It adds the tasks in the order that it discovers them, as it proceeds through `tg` in chronological order; so the tasks will appear in `ts` in chronological order, except for tasks that lie partly within `tg` and partly outside it. The function does not add tasks that are already present. If `include_preassigned` is `true`, preassigned tasks are included, otherwise they aren't. Omitting them makes sense when the tasks will be reassigned.

One can also get direct access to the times where the resource has clashes:

```
int KheResourceTimetableMonitorClashingTimeCount(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm);
KHE_TIME KheResourceTimetableMonitorClashingTime(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm, int i);
```

These work in the usual way, visiting each clashing time in turn. The times are kept in increasing time index order, which is handy when repairing clashes, because it means that removing a clash then reinstating it does not change the order that the times appear.

A resource timetable monitor offers no operations which report its set of tasks directly. For that, one can use `KheResourceAssignedTaskCount` and `KheResourceAssignedTask` from Section 4.6.1 to obtain all the tasks assigned the resource; the timetabled ones are just those whose enclosing meet has an assigned time.

As usual, resource timetable monitors are created by `KheSolnMake` and exist for as long as the solution does. There is one for each resource. All resource monitors (except possibly limit workload monitors) depend on resource timetable monitors.

Unlike most monitors, resource timetable monitors are not attached initially. The resource timetable monitor returned by `KheResourceTimetableMonitor` may be unattached and so not up to date (it will be empty in that case). When a monitor is attached, any unattached timetable monitor(s) it depends on are also attached. When the last monitor that depends on some resource timetable monitor is detached, that resource timetable monitor is detached. Thus, unless the user chooses to attach a resource timetable monitor explicitly, it will be attached only as needed by other monitors. Detaching a resource timetable monitor does nothing unless no attached monitors depend on it. So when using a resource timetable monitor `rtm`, it is best to call

```
if( !KheMonitorAttachedToSoln((KHE_MONITOR) rtm) )
  KheMonitorAttachToSoln((KHE_MONITOR) rtm);
```

beforehand, and

```
KheMonitorDetachFromSoln((KHE_MONITOR) rtm);
```

afterwards, unless `rtm` must be attached, because some monitor that depends on it is attached.

Although it would make sense to treat a resource timetable monitor as a group monitor (Section 6.9), that option is not offered. The user who wants all the problems associated with a given resource to be channelled through a single monitor must create a group monitor, separate from the resource timetable monitor, and add the appropriate monitors to it in the usual way.

Here are two functions created to support the needs of particular solvers. First,

```
int KheResourceTimetableMonitorAtMaxLimitCount(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm, KHE_TIME t);
```

returns the sum, over all cluster busy times and limit active intervals monitors that monitor `rtm`'s resource at time `t`, of the values returned by those monitors' `AtMaxLimitCount` functions. It is an efficient way to find out, during time sweep resource assignment, whether assignments at time `t` have brought any of these monitors to their maximum limits. Second,

```
void KheResourceTimetableMonitorAddRange(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm, int first_time_index,
  int last_time_index, KHE_GROUP_MONITOR gm);
```

adds to `gm` all cluster and limit busy times monitors which monitor `rtm`, are derived from constraints which apply to every resource of the type of `rtm`'s resource, and whose range (as given by `KheClusterBusyTimesMonitorRange` and `KheLimitBusyTimesMonitorRange`) lies between the times indexed by `first_time_index` and `last_time_index` inclusive. A monitor is not added if `KheGroupMonitorHasChildMonitor` reports that it is already there. This function is used by combinatorial grouping.

At present, all resource timetable monitors are created automatically when the solution is created. The KHE user is offered nothing equivalent to `KheEventTimetableMonitorMake`.

Function

```
void KheResourceTimetableMonitorDebug(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm, int verbosity,
  int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor. There is also

```
void KheResourceTimetableMonitorSetDebug(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm, KHE_TIME_GROUP tg, bool val);
```

If `val` is `true`, this marks the times of `tg` for debugging. Whenever `rtm`'s timetable changes at any of these times, a one-line debug print is produced on `stderr` giving the resource, the time, and a brief indication of the change. This is useful for working out why a resource is busy (or not) during a given time group. If `val` is `false` this debugging is turned off at the times of `tg`.

Finally, there is

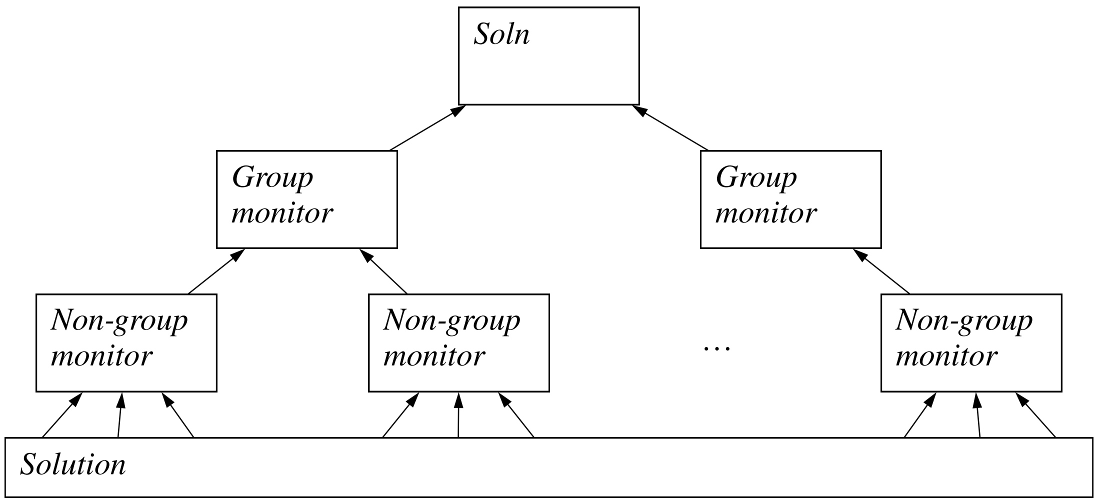
```
void KheResourceTimetableMonitorPrintTimetable(
  KHE_RESOURCE_TIMETABLE_MONITOR rtm, int cell_width, int indent, FILE *fp);
```

which prints a tabular timetable, using `Days` and possibly `Weeks` time groups from the instance to determine its shape. Parameter `cell_width` is the width of each cell, in characters.

## 6.9. Group monitors

Sometimes the cost of a *single* monitor is needed: for example, when reporting problems to the user. And the total cost of *all* monitors is always needed, since that is the cost of the solution.

Sometimes something in between these two extremes is needed: the cost of a set of related monitors. To support this, the monitors of a solution are organized into a directed acyclic graph, or *dag* for short, of arbitrary depth. Each monitor reports its cost to its parent monitors. The dag is often a tree, in which case the picture looks like this:



The leaves are the *non-group monitors*, the various monitors described previously which monitor the solution directly. The internal nodes are called *group monitors*, because they monitor a set of monitors (their children). The cost of a group monitor is the sum of the costs of its children.

The solution object itself is a group monitor (initially, the only one). It supports all the group monitor operations, plus the many other operations described earlier.

Group monitors have type `KHE_GROUP_MONITOR`, a concrete subtype of `KHE_MONITOR`, like `KHE_ASSIGN_TIME_MONITOR` etc. `KHE_GROUP_MONITOR` is a supertype of `KHE_SOLN`, so upcast

```
(KHE_GROUP_MONITOR) soln
```

is safe, although often unnecessary, since many operations on type `KHE_GROUP_MONITOR` have `KHE_SOLN` versions. For example, since `KHE_GROUP_MONITOR` is itself a subtype of `KHE_MONITOR`, the total cost of all monitors could be found by calling

```
KheMonitorCost((KHE_MONITOR) soln)
```

but of course the equivalent `KHE_SOLN` version, `KheSolnCost`, is easier to use.

When the solution changes at some point, the change is reported to the non-group monitors that monitor that point. Each updates its cost and reports any change to its parents, which update their cost and report to their parents, and so on until there are no parents. The dag usually has a single root, the solution object itself, but it does not have to be that way, because the links that join non-group and group monitors to their parent monitors can be added and deleted at will.

### 6.9.1. Basic operations on group monitors

Unlike other types of monitors, group monitors other than the solution object can be freely created and deleted. Function

```
KHE_GROUP_MONITOR KheGroupMonitorMake(KHE_SOLN soln, int sub_tag,
  char *sub_tag_label);
```

creates a new group monitor with no parents and no children. It is passed the solution as a parameter, and it remembers it, but it is not made a child of it. Functions

```
int KheGroupMonitorSubTag(KHE_GROUP_MONITOR gm);
char *KheGroupMonitorSubTagLabel(KHE_GROUP_MONITOR gm);
```

return the `sub_tag` and `sub_tag_label` attributes of `gm`. These are used to distinguish kinds of group monitors. If `sub_tag_label` is non-`NULL`, it is printed when debugging. The values of these attributes in solution objects are `-1` and `"Soln"`. The term ‘sub-tag’ is used because group monitors already have a tag attribute, whose value is `KHE_GROUP_MONITOR_TAG`.

A group monitor other than the solution object may be deleted by calling

```
void KheGroupMonitorDelete(KHE_GROUP_MONITOR gm);
```

Its children will no longer have it as a parent, and its parents will no longer have it as a child. For each parent of `gm`, the hole in the parent’s list of child monitors is plugged by moving the last child monitor to `gm`’s position. For each child of `gm`, the hole in the child’s list of parent monitors is plugged by moving the last parent monitor to `gm`’s position.

Every group monitor can have any number of child monitors, and every monitor (group or non-group) can have any number of parent monitors. Even the solution object can have parents, allowing monitoring of the total cost of a set of solutions. The operations for adding children to a group monitor and removing them are

```
void KheGroupMonitorAddChildMonitor(KHE_GROUP_MONITOR gm, KHE_MONITOR m);
void KheGroupMonitorDeleteChildMonitor(KHE_GROUP_MONITOR gm, KHE_MONITOR m);
```

Here `m` could be a non-group monitor or a group monitor. `KheGroupMonitorAddChildMonitor` makes `m` a child of `gm`, and `gm` a parent of `m`. It aborts if this would create a cycle in the dag (only possible when `m` is a group monitor). `KheGroupMonitorDeleteChildMonitor` removes `m` from `gm`, leaving `m` with one less parent and `gm` with one less child. The resulting holes are plugged as described above for deleting group monitors. It aborts if `m` is not a child of `gm`. There is also

```
bool KheGroupMonitorHasChildMonitor(KHE_GROUP_MONITOR gm, KHE_MONITOR m);
```

which returns `true` when `m` is a child of `gm`. It is useful when `m` may already be a child of `gm`:

```
if( !KheGroupMonitorHasChildMonitor(gm, m) )
  KheGroupMonitorAddChildMonitor(gm, m);
```

No-one is checking that one monitor does not become the child of another twice over; and if it does, its cost will be counted twice in the cost of its parent.

For group monitor `m`, `KheMonitorLowerBound(m)` sums the lower bounds of `m`'s children. It may increase when a descendant is added, and decrease when a descendant is removed.

Initially, all non-group monitors are made children of the solution object, and all of them except demand monitors are attached to the solution, so that `KheSolnCost` is the total cost of all non-demand monitors, which is indeed the cost of the solution. Care is needed when grouping not to inadvertently disconnect monitors from the solution, since then their costs will not be counted, or to connect them via multiple paths, since then their costs will be counted multiple times. It is usually best to make a new group monitor a child of the solution immediately:

```
gm = KheGroupMonitorMake(soln, sub_tag, sub_tag_label);
KheGroupMonitorAddChildMonitor((KHE_GROUP_MONITOR) soln,
  (KHE_MONITOR) gm);
```

And when deleting a group monitor, the best option may be helper function

```
void KheGroupMonitorBypassAndDelete(KHE_GROUP_MONITOR gm);
```

It calls `KheGroupMonitorDelete`, but first it makes `gm`'s children into children of `gm`'s parents, if any, thus keeping them linked in. There is also

```
void KheSolnBypassAndDeleteAllGroupMonitors(KHE_SOLN soln);
```

which applies `KheGroupMonitorBypassAndDelete` to every group monitor of `soln`.

Functions

```
int KheGroupMonitorChildMonitorCount(KHE_GROUP_MONITOR gm);
KHE_MONITOR KheGroupMonitorChildMonitor(KHE_GROUP_MONITOR gm, int i);
```

visit the child monitors of group monitor `gm` in the usual way. If `gm` is the solution object, these versions of the functions allow the user to avoid the upcast:

```
int KheSolnChildMonitorCount(KHE_SOLN soln);
KHE_MONITOR KheSolnChildMonitor(KHE_SOLN soln, int i);
```

Functions

```
int KheMonitorParentMonitorCount(KHE_MONITOR m);
KHE_GROUP_MONITOR KheMonitorParentMonitor(KHE_MONITOR m, int i);
```

visit the parent monitors of `m`. There is also

```
bool KheMonitorDescendant(KHE_MONITOR m1, KHE_MONITOR m2);
```

which returns `true` if `m1` is a descendant of `m2`, including when the two are equal, and

```
void KheMonitorParentMonitorsSort(KHE_MONITOR m);
```

which sorts the parent monitors of `m` so that traversal using `KheMonitorParentMonitorCount` and `KheMonitorParentMonitor` visits them in order of increasing solution index. And

```
void KheGroupMonitorDebug(KHE_GROUP_MONITOR gm,
  int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor. When C verbosity is 2 or more, it prints the child monitors and checks that their total cost is equal to the group monitor's cost, aborting if not.

A group monitor has the usual attach and detach operations, but they do nothing substantial; in particular, they do not change its cost. They just mark the monitor as attached or detached. Perhaps they should attach and detach it from its children, but they don't.

### 6.9.2. Defects

Informally, a defect is a specific problem with a solution. In KHE, the word has a formal meaning as well: a *defect* is a monitor whose cost is non-zero.

It can be helpful to target defects directly, rather than wasting time changing parts of the solution where there are no defects. This is especially the case near the end of the solve process, when there may be thousands of monitors but only a handful of defects. To support this, KHE offers fast access to those child monitors of a group monitor which are defects:

```
int KheGroupMonitorDefectCount(KHE_GROUP_MONITOR gm);
KHE_MONITOR KheGroupMonitorDefect(KHE_GROUP_MONITOR gm, int i);
```

When a monitor's cost changes from zero to non-zero, the monitor is added to its parents' defect lists; and when its cost changes from non-zero to zero it is removed. This takes a negligible amount of time. When the group monitor is the solution there are convenience versions:

```
int KheSolnDefectCount(KHE_SOLN soln);
KHE_MONITOR KheSolnDefect(KHE_SOLN soln, int i);
```

There is also

```
void KheGroupMonitorDefectDebug(KHE_GROUP_MONITOR gm,
  int verbosity, int indent, FILE *fp);
```

which is like `KheGroupMonitorDebug` applied to `gm`, except that it prints only defective children, and

```
void KheGroupMonitorDefectTypeDebug(KHE_GROUP_MONITOR gm,
  KHE_MONITOR_TAG tag, int verbosity, int indent, FILE *fp);
```

which is like `KheGroupMonitorDefectDebug` except that it prints only children of type `tag`.

If a solution is changed and then changed back again to its original state, its cost returns to its original value, but there are two ways in which its defects can be different. First, they may appear in a different order. Second, although the number of defects which are demand monitors (Chapter 7) must return to its original value, the demand monitors that make up that number may change. This is because there are many maximum matchings in general, and KHE does not guarantee to find any particular one of them.

In practice, one wants to traverse a list of defects and try to repair them. Quite commonly, an attempt to repair a defect will remove it temporarily and then reinstate it if the repair was not successful. This will cause the defect to be shifted to the end of the defect list. A simple traversal of the defects from first to last will visit some defects more than once, and others not at all. To handle this problem, it is necessary to make a copy of the defects and traverse the copy. Although every defect will have non-zero cost at the time it is copied, as the list is traversed, after the solution changes or if the list includes demand monitors, one cannot assume that every monitor on the copy list will have non-zero cost.

To find the total cost of all monitors of a given type in the descendants of `gm`, call

```
KHE_COST KheGroupMonitorCostByType(KHE_GROUP_MONITOR gm,
  KHE_MONITOR_TAG tag, int *defect_count);
```

It returns the number of defects, in `*defect_count`, as well as the cost. It traverses the whole sub-dag of monitors of `gm` (actually, just the defects), so it is slow: it is intended for reporting, not for solving. It returns `0` when `tag` is `KHE_GROUP_MONITOR_TAG`, because it attributes cost to the monitors that originally generated it. Version

```
KHE_COST KheSolnCostByType(KHE_SOLN soln, KHE_MONITOR_TAG tag,
  int *defect_count);
```

may be called when the group monitor is the solution object. The values returned by these functions are displayed in a convenient tabular form by functions

```
void KheGroupMonitorCostByTypeDebug(KHE_GROUP_MONITOR gm,
  int verbosity, int indent, FILE *fp);
void KheSolnCostByTypeDebug(KHE_SOLN soln,
  int verbosity, int indent, FILE *fp);
```

which print one line for each kind of monitor under `gm` or `soln` for which there are defects.

### 6.9.3. Tracing

Sometimes a solver needs to know which monitors have experienced a change in cost recently. Ejection chain solvers, for example, need this information, and *monitor tracing* provides it.

Tracing involves objects of type `KHE_TRACE`. To create one, call

```
KHE_TRACE KheTraceMake(KHE_GROUP_MONITOR gm);
```

where `gm` is the group monitor to be traced. The solution may be traced by upcasting it:

```
t = KheTraceMake((KHE_GROUP_MONITOR) soln);
```

The group monitor that a trace object is for can be found by calling

```
KHE_GROUP_MONITOR KheTraceGroupMonitor(KHE_TRACE t);
```

To delete a trace object, call

```
void KheTraceDelete(KHE_TRACE t);
```

This will call `KheTraceEnd(t)` below if needed. KHE keeps a free list of trace objects in the solution object, so many trace objects can be created and deleted at virtually no cost.

Actual tracing is initiated and ended by calling

```
void KheTraceBegin(KHE_TRACE t);
void KheTraceEnd(KHE_TRACE t);
```

These must be called in matching pairs. `KheTraceBegin` removes any information left over from any preceding trace, and attaches `t` to its group monitor so that it can record what happens. `KheTraceEnd` detaches `t` from its group monitor. Different trace objects may be attached and detached quite independently of each other, even when they have the same group monitor.

After the trace ends, the following functions may be called:

```
KHE_COST KheTraceInitCost(KHE_TRACE t);
int KheTraceMonitorCount(KHE_TRACE t);
KHE_MONITOR KheTraceMonitor(KHE_TRACE t, int i);
KHE_COST KheTraceMonitorInitCost(KHE_TRACE t, int i);
```

`KheTraceInitCost` returns the initial cost of `t`'s group monitor (at the time the trace began); `KheTraceMonitorCount` returns the number of child monitors of `t`'s group monitor whose cost changed during the trace; `KheTraceMonitor` returns the `i`th of these child monitors; and `KheTraceMonitorInitCost(t, i)` returns the initial cost of `KheTraceMonitor(t, i)`. Also,

```
KHE_COST KheTraceMonitorCostIncrease(KHE_TRACE t, int i);
```

returns `KheMonitorCost(KheTraceMonitor(t, i)) - KheTraceMonitorInitCost(t, i)`. It will be negative when the monitor's cost decreased.

The list of child monitors whose cost has changed never contains the same monitor `m` twice, no matter how many times `m`'s cost changes during the trace. This is desirable, but it means that when `m`'s cost changes, this list has to be searched to see if `m` is already present. So it is best to use tracing on group monitors that group only a small number of monitors; or if a large group monitor like the solution object is traced, to trace it for only small sequences of operations that are not likely to change the cost of a large number of monitors.

These functions may be called during a trace as well as after it, returning values as though the trace had just ended. While it is not an error to call `KheGroupMonitorAddChildMonitor` or `KheGroupMonitorDeleteChildMonitor` while tracing the group monitor concerned, it is not recommended. A solution cannot be copied while one of its group monitors is being traced.

For the convenience of ejection chain algorithms, function

```
void KheTraceReduceByCostIncrease(KHE_TRACE t, int max_num);
```

sorts the monitors by decreasing `KheTraceMonitorCostIncrease`, removes all monitors whose cost increase is zero or negative, then keeps removing monitors from the end until at most `max_num` remain. These may be accessed with `KheTraceMonitorCount`, `KheTraceMonitor`, and `KheTraceMonitorInitCost` as usual. The other monitors are gone and cannot be got back.

Finally, function

```
void KheTraceDebug(KHE_TRACE t, int verbosity, int indent, FILE *fp);
```

prints `t` onto `fp` with the given verbosity and indent, showing monitors whose cost changed.

# Chapter 7. Matchings and Evenness

Suppose a decision is made to run five Music meets simultaneously, when the school has only two Music teachers and two Music rooms. Clearly, when teachers and rooms are assigned later, there will be major problems, but until then the usual cost function will not reveal any problems.

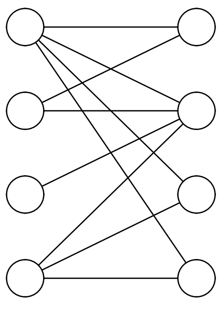
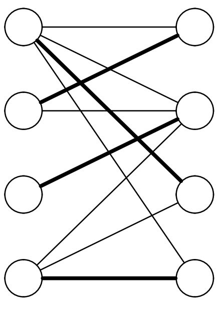
More subtly, suppose there are eight teachers, and that three of them teach English only, three teach History only, and two teach both. Suppose a decision is make to run five English meets and five History meets simultaneously. Then there are enough English teachers to teach the five English meets, and there are enough History teachers to teach the five History meets, but there are not enough English and History teachers, taken together, to teach the ten meets.

*Matchings* (officially, *unweighted bipartite matchings*) detect such problems. Although not compulsory, they are often helpful. This chapter describes them in general, how they apply to timetabling, and how to use them in KHE.

The functions defined here are everything that the KHE platform offers to support matching. They are used as is for some things, such as diagnosing failure to match, but for setting up a matching to begin with and deleting it later, in practice it is better to use the solver functions `KheMatchingBegin` and `KheMatchingEnd` (Section 8.8.1) since they call the functions defined here in just the right way.[1]

## 7.1. Introducing bipartite matching

A *bipartite graph* is an undirected graph whose nodes are divided into two sets, such that every edge connects a node of one set to a node of the other. A *matching* is a subset of the edges such that no two edges touch the same node. A *maximum matching* is a matching containing as many edges as possible. The *bipartite matching problem* is the problem of finding a maximum matching in a bipartite graph. For example, here is a bipartite graph (at left), and the same graph with a maximum matching shown in bold (at right):

There is a standard polynomial-time algorithm for this problem.

In timetabling, where bipartite matching has been used for many years [2, 4, 14], it is usual for one of the two sets of nodes to represent variables (slots, events, etc.) demanding something

[1]Prior to Version 2.10, most of the setup code was in the platform. The current arrangement is better in every way.

to be assigned to them, while the other set represents values (times, resources, etc.) which are available to supply these demands. So these sets are called the *demand nodes* and the *supply nodes* here. A maximum matching assigns supply nodes to as many demand nodes as possible, given that each demand node requires any one of the supply nodes it is connected to, and each supply node may be assigned to at most one demand node. Although the problem is formally symmetrical between the two kinds of nodes, in timetabling it is not symmetrical: it does not matter if some supply nodes are not matched, but it does matter if some demand nodes are not matched.

One does not usually want to make the assignments indicated by a maximum matching, because there are other constraints not modelled by it, and the aim is to find, not just any maximum matching, but one satisfying these other constraints. Instead, the matching helps to evaluate the current state. Because it is maximum, it indicates that there must be at least a certain number of problems, in the form of unassigned demand nodes, in any solution incorporating the decisions already made, and that is valuable information when evaluating those decisions.

Some applications of matching to timetabling utilize the idea of a *tixel*, the author's term for one resource at one time (the name recalls the *pixel* of computer graphics). For example, teacher Smith during the first time on Mondays is one tixel; it may be represented by the ordered pair

$$(\mathit{Smith}, \mathit{Mon1})$$

This is also called a *supply tixel*, because it can supply the demands of events for teachers. The events are said to contain *demand tixels*. For example, an event of duration 2 which requests student group 8*A*, one English teacher, and one room, is said to contain six demand tixels. This is shorthand for saying that it demands six supply tixels.

Underlying the high school timetabling problem is a matching that we will call the *global tixel matching*. Its supply nodes are the supply tixels, one for each resource of the instance at each time. Its demand nodes are the demand tixels of the events of the instance. Edges connect demand tixels to those supply tixels that suit them. For example, a demand for student group 8A would be connected to supply tixels whose resource is 8A; a demand for an English teacher at time *Mon1* would be connected to those supply tixels whose resource is an English teacher and whose time is *Mon1*. Each demand tixel wants to be assigned one supply tixel, and each supply tixel may only be assigned to one demand tixel (otherwise there would be a timetable clash). So a matching is indeed required, and a maximum matching will have the fewest problems.

As decisions are made, in the form of assignments of times to meets or resources to tasks (or domain reductions, for example from all qualified resources to a smaller set of preferred resources), the demand tixels affected by these decisions become connected to fewer supply tixels. When the maximum matching is recalculated (there is an efficient algorithm for doing this incrementally as the graph changes) there may be more unmatched nodes than before, suggesting that the decisions made may have been poor ones, and that alternatives should be explored.

The global tixel matching is useful for evaluating instances before solving begins. It can reveal, for example, that the supply of computer laboratories is insufficient to cover the demand, and other problems of that kind. It turns out to be very powerful late in the solve process, when resources are being assigned after times have been assigned, provided it is enhanced with tixels expressing resource unavailabilities and workload limits (Section 7.3). However, it is quite weak before times are assigned, because it does not understand that the supply tixels assigned to events must be correlated in time: it does not perceive the contradiction in assigning, say, the two supply tixels (*Smith*,*Mon1*) and (*Lab6*, *Wed5*) to an event of duration 1.

An example given earlier, of scheduling five Music events simultaneously when there are only two Music teachers and two Music rooms, shows that useful checks can be made when deciding to run events simultaneously, even though their actual time is not fixed. Whatever time is ultimately assigned to such events, each resource can supply at most one tixel to satisfy their demands. So the demand tixels for one time of the events concerned may be matched with a set of supply nodes, one for each resource. This will be called *local tixel matching*. The tixels are rather different: they share a common generic time rather than holding a variety of true times.

## 7.2. Basic operations

By default, a solution contains no matching. To add one, and later to delete it, call

```
void KheSolnMatchingBegin(KHE_SOLN soln);
void KheSolnMatchingEnd(KHE_SOLN soln);
```

(As already remarked, solver functions `KheMatchingBegin` and `KheMatchingEnd` from Section 8.8.1 will usually be better in practice.) The matching is also deleted when its solution is deleted, since a matching without a solution makes no sense. `KheSolnMatchingBegin` does not add any demand nodes, and `KheSolnMatchingEnd` requires all demand nodes that have been added (by calls that we'll come to later) since the preceding `KheSolnMatchingBegin` to have been deleted before it is called.

A solution can have at most one matching, and KHE will abort if `KheSolnMatchingBegin` is called twice without an intervening `KheSolnMatchingEnd`. When present, the matching is kept up to date automatically as the solution changes. A lazy implementation is used: no matching is done until a query is received (for example, a request for the current number of unmatched demand nodes). This allows the time spent matching to be amortized over all operations carried out since the previous query. There is no way for the user to observe the laziness. The key operation, of bringing the matching up to date (making it maximum) runs in time roughly proportional to the number of unmatched nodes in the graph when it is called.

Function

```
bool KheSolnHasMatching(KHE_SOLN soln);
```

returns `true` when `soln` has a matching. Most of the operations of this chapter assume that the matching is present. If it isn't, some may abort, while others may do nothing.

A demand node is a kind of monitor; we use the terms *demand node* and *demand monitor* interchangeably. Demand monitors may be attached and detached separately as usual. Detaching a demand monitor removes its node from the matching graph.

In the usual way, a demand monitor contributes a cost to the solution when it is attached to the solution and linked in as a descendant of the solution object (considered as a group monitor). The cost is 0 when the node is matched, and some non-negative value when it is unmatched. This value, the cost to report when the node is unmatched, is set and retrieved by functions

```
void KheSolnMatchingSetWeight(KHE_SOLN soln, KHE_COST weight);
KHE_COST KheSolnMatchingWeight(KHE_SOLN soln);
```

The value is the same for all demand nodes, because this is unweighted bipartite matching. Any change in weight is reflected immediately in the costs of all demand monitors.

The matching has a *type* that may be changed at any moment:

```
void KheSolnMatchingSetType(KHE_SOLN soln, KHE_MATCHING_TYPE mt);
KHE_MATCHING_TYPE KheSolnMatchingType(KHE_SOLN soln);
```

KHE_MATCHING_TYPE is the enumerated type

```
typedef enum {
  KHE_MATCHING_TYPE_EVAL_INITIAL,
  KHE_MATCHING_TYPE_EVAL_TIMES,
  KHE_MATCHING_TYPE_EVAL_RESOURCES,
  KHE_MATCHING_TYPE_SOLVE
} KHE_MATCHING_TYPE;
```

A full explanation of these values is given in the following section. Just briefly, however, KHE_MATCHING_TYPE_SOLVE implements a kind of local tixel matching and is the best choice when solving; it is also the default value. The others are variants of global tixel matching. A change of type is reflected immediately in the costs of all demand monitors.

For the most part, matchings work quietly behind the scenes without attention from the user. However, there is an important optimization that only the user can invoke. Suppose that some changes are made to the solution as an experiment, then either retained or undone. Then KHE will run faster if that part of the program is bracketed by calls to these functions:

```
void KheSolnMatchingMarkBegin(KHE_SOLN soln);
void KheSolnMatchingMarkEnd(KHE_SOLN soln, bool undo);
```

Calls to these operations must occur in matching pairs, possibly nested. If undo is true, then KheSolnMatchingMarkEnd assumes without checking that all changes to soln since the corresponding call to KheSolnMatchingMarkBegin have been undone. It uses this information to bring the matching up to date more quickly than could be done without it. To encourage their use, both functions are well-defined even when there is no matching: in that case, they do nothing.

As an aid to debugging, function

```
void KheSolnMatchingDebug(KHE_SOLN soln, int verbosity,
  int indent, FILE *fp);
```

ensures that the matching is up to date, then prints its current state onto fp. Verbosity 1 prints just the number of unmatched demand monitors, verbosity 2 prints those monitors, and verbosity 3 prints all demand monitors and the supply nodes they are matched with.

## 7.3. Supply nodes and demand nodes

Supply nodes are created automatically behind the scenes, and are not accessible to the user. There is one supply node for each resource at each time of each meet `m` that is not assigned to another meet. When such an assignment is made, the supply nodes of `m` are deleted, since the two meets then run simultaneously.

The rest of this section deals with demand nodes. These are of two kinds: *ordinary demand nodes* and *workload demand nodes*. To create and delete an ordinary demand node, call

```
KHE_ORDINARY_DEMAND_MONITOR KheOrdinaryDemandMonitorMake(
  KHE_TASK task, int offset, KHE_MONITOR orig_m);
void KheOrdinaryDemandMonitorDelete(KHE_ORDINARY_DEMAND_MONITOR odm);
```

An ordinary demand node represents a demand for one resource at one time made by `task` at `offset` (between 0 inclusive and the duration of the task exclusive). The originating monitor, `orig_m`, is the monitor that led to the creation of this demand monitor. It would probably be an assign resource monitor for `task`, although there is no strict rule; it may be `NULL`.

The usual monitor operations (attach and detach, etc.) may be obtained by upcasting from `KHE_ORDINARY_DEMAND_MONITOR` to `KHE_MONITOR` as usual. There are also these operations specific to ordinary demand monitors:

```
KHE_TASK KheOrdinaryDemandMonitorTask(KHE_ORDINARY_DEMAND_MONITOR odm);
int KheOrdinaryDemandMonitorOffset(KHE_ORDINARY_DEMAND_MONITOR odm);
KHE_MONITOR KheOrdinaryDemandMonitorOriginatingMonitor(
  KHE_ORDINARY_DEMAND_MONITOR odm);
void KheOrdinaryDemandMonitorDebug(KHE_ORDINARY_DEMAND_MONITOR odm,
  int verbosity, int indent, FILE *fp);
```

Functions

```
int KheTaskDemandMonitorCount(KHE_TASK task);
KHE_ORDINARY_DEMAND_MONITOR KheTaskDemandMonitor(KHE_TASK task, int i);
```

visit the ordinary demand monitors associated with `task`, attached or detached. In practice there will be one of these for each legal offset, although that is not an absolute requirement.

We turn now to workload demand nodes. These originate in constraints on the availability of resources. For example, if resource `r` is unavailable at time `Mon1`, we might want to remove supply node `(r, Mon1)`, but there is no way to do that, so instead we add a workload demand node that matches only with that supply node. To create and delete such a node, call

```
KHE_WORKLOAD_DEMAND_MONITOR KheWorkloadDemandMonitorMake(
  KHE_SOLN soln, KHE_RESOURCE r, KHE_TIME_GROUP tg, KHE_MONITOR orig_m);
void KheWorkloadDemandMonitorDelete(KHE_WORKLOAD_DEMAND_MONITOR wdm);
```

`KheWorkloadDemandMonitorMake` creates one tixel of demand which matches with all supply nodes whose resource is `r` and whose time is an element of time group `tg`. In our example, `tg` would be the singleton time group containing time `Mon1`. As usual, `orig_m` is the monitor that originates this demand (in our example it would be an avoid unavailable times monitor), and the usual monitor operations are available by upcasting. There are also these operations specific to workload demand monitors:

```
KHE_RESOURCE KheWorkloadDemandMonitorResource(
  KHE_WORKLOAD_DEMAND_MONITOR wdm);
KHE_TIME_GROUP KheWorkloadDemandMonitorTimeGroup(
  KHE_WORKLOAD_DEMAND_MONITOR wdm);
KHE_MONITOR KheWorkloadDemandMonitorOriginatingMonitor(
  KHE_WORKLOAD_DEMAND_MONITOR wdm);
void KheWorkloadDemandMonitorDebug(KHE_WORKLOAD_DEMAND_MONITOR wdm,
  int verbosity, int indent, FILE *fp);
```

There are no functions specifically for visiting workload demand monitors. They are classed as resource monitors and can be visited along with the other resource monitors for a given resource `r` by calling `KheSolnResourceMonitorCount` and `KheSolnResourceMonitor` (Section 6.7).

Ordinary and workload demand monitors are created in the detached state. Users who create these monitors are free to attach each of them immediately after it is created, if they wish.

Before leaving demand monitors we need to explain matching types in detail. An ordinary demand node's *own meet* is the meet its task lies in. Its *root meet* is the meet reached by following the chain of assignments (possibly empty) out of its own meet to a meet that contains no assignment. Its *own offset* is its offset in its own meet, and its *root offset* is its offset in its root meet (the sum of its own offset and the offsets along the assignment path).

When linking an ordinary demand node to supply nodes, there are at least two ways to take time into account:

A. Link it only to ordinary supply nodes lying in cycle meets at offsets that represent the times of the time domain of its own meet, shifted by its own offset.

B. Link it only to ordinary supply nodes lying in its root meet at its root offset.

Informally, (A) evaluates the initial state of time assignment, whereas (B) evaluates its current state in a way that ensures that simultaneous demands compete for the same supply nodes, as in local tixel matching. And there are at least two ways to take resources into account:

1. Link it to supply nodes representing the resources of its task's domain.

2. Link it to supply nodes representing the resources of its task's root task's domain. If the root task is a cycle task, this will link only to supply nodes representing that resource.

Informally, (1) evaluates the initial state of resource assignment, whereas (2) evaluates the current state. The four matching types produce the four conjunctions of these conditions:

| | A | B |
|---|---|---|
| 1 | `KHE_MATCHING_TYPE_EVAL_INITIAL` | `KHE_MATCHING_TYPE_EVAL_TIMES` |
| 2 | `KHE_MATCHING_TYPE_EVAL_RESOURCES` | `KHE_MATCHING_TYPE_SOLVE` |

Type (B2) is suited to solving; the others are suited to evaluation before or after solving.

## 7.4. Diagnosing failure to match

KHE's usual methods of organizing monitors, such as grouping and tracing, may be applied to demand monitors. This section offers three other ways to visit unmatched demand monitors.

### 7.4.1. Visiting unmatched demand nodes

The unmatched demand nodes may be visited by functions

```
int KheSolnMatchingDefectCount(KHE_SOLN soln);
KHE_MONITOR KheSolnMatchingDefect(KHE_SOLN soln, int i);
```

Each monitor is either an ordinary demand monitor or a workload demand monitor; a call to `KheMonitorTag` followed by a downcast will produce the specific type. Then functions defined earlier give access to the part of the solution being monitored by these monitors.

Unmatched demand nodes with higher indexes tend to have become unmatched more recently than demand nodes with lower indexes. When the number of unmatched demand nodes increases, it is reasonable to take the last unmatched demand node as an indication of what went wrong. However, it will usually be better to use grouping and tracing to localize problems.

### 7.4.2. Hall sets

*Hall sets* are the definitive method of diagnosing failure to match. They are fine for occasional use, such as for generating a report to the user, but too slow for repeated use during solving.

Suppose there is a set $D$ of demand nodes, whose outgoing edges all lead to nodes in some set $S$ of supply nodes. Then every node in $D$ must be matched with a node in $S$, or not matched at all. If $|D| > |S|$, then at least $|D| - |S|$ nodes of $D$ will be unmatched in any maximum matching.

It turns out that every case of an unmatched node can be explained in this way, often utilizing sets $D$ and $S$ that are small enough to understand in user terms: they might represent the demand and supply of Science laboratories, for example. Such a $D$ and $S$, with every edge out of $D$ leading to $S$, and $|D| > |S|$, is called a *Hall set* after the mathematician Philip Hall. Given a maximum matching, every unmatched demand node lies in a Hall set.

The following functions examine the Hall sets of a matching. They all begin by building the Hall sets if the ones currently stored are not up to date. This means that any change to the solution invalidates everything returned by all previous calls to these functions.

The number of Hall sets is returned by

```
int KheSolnMatchingHallSetCount(KHE_SOLN soln);
```

This is not usually the same as the number of unmatched demand nodes, since there may be several of those in one Hall set. No node lies in two Hall sets. The number of supply and demand nodes in the `i`'th Hall set may be found by calling

```
int KheSolnMatchingHallSetSupplyNodeCount(KHE_SOLN soln, int i);
int KheSolnMatchingHallSetDemandNodeCount(KHE_SOLN soln, int i);
```

By the way that Hall sets are defined, `KheSolnMatchingHallSetDemandNodeCount(soln, i)` must be larger than `KheSolnMatchingHallSetSupplyNodeCount(soln, i)`.

The `j`'th supply node of the `i`'th Hall set can only be an ordinary supply node, but, in case other kinds of supply nodes are added in future, the following function is used to find the meet it lies in, its offset within that meet, and the resource it represents:

```
bool KheSolnMatchingHallSetSupplyNodeIsOrdinary(KHE_SOLN soln,
  int i, int j, MEET *meet, int *meet_offset, KHE_RESOURCE *r);
```

At present this always returns `true`. A report to the user should distinguish the cases when `*meet` is and is not a cycle meet. The `j`'th demand node of the `i`'th Hall set is returned by

```
KHE_MONITOR KheSolnMatchingHallSetDemandNode(KHE_SOLN soln,
  int i, int j);
```

It will be either an ordinary demand node or a workload demand node as usual. Finally,

```
void KheSolnMatchingHallSetsDebug(KHE_SOLN soln,
  int verbosity, int indent, FILE *fp);
```

prints the Hall sets of `m`'s matching onto `fp` with the given verbosity and indent. The verbosity must be at least 1 but otherwise does not affect what is printed.

### 7.4.3. Finding competitors

Given an unmatched demand monitor `m` returned by `KheSolnMatchingHallSetDemandNode` or `KheSolnMatchingDefect`, a *competitor* of that monitor is either `m` itself or a monitor whose removal would allow `m` to match. Competitors are similar to the demand nodes of Hall sets, except that they relate to a single unmatched demand node. They are themselves always matched. Finding competitors amounts to redoing the search for an augmenting path for the failed node and noting the demand nodes that are visited along the way.

Functions

```
void KheSolnMatchingSetCompetitors(KHE_SOLN soln, KHE_MONITOR m);
int KheSolnMatchingCompetitorCount(KHE_SOLN soln);
KHE_MONITOR KheSolnMatchingCompetitor(KHE_SOLN soln, int i);
```

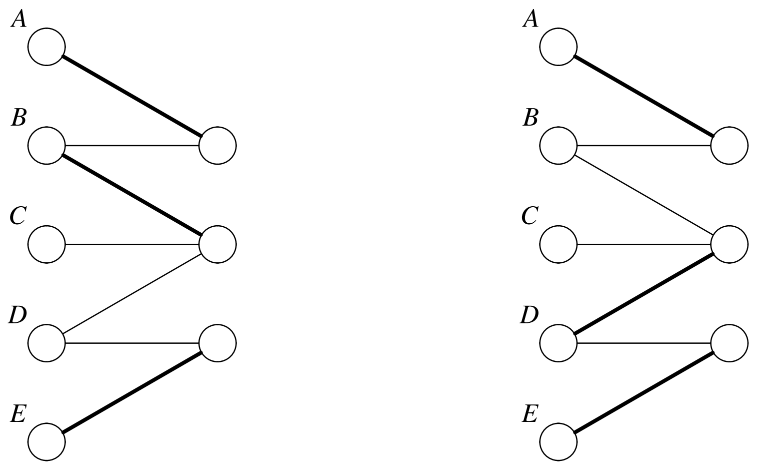
may be used together to visit the competitors of unmatched demand monitor `m`:

```
KheSolnMatchingSetCompetitors(soln, m);
for( i = 0;  i < KheSolnMatchingCompetitorCount(soln);  i++ )
{
  competitor_m = KheSolnMatchingCompetitor(soln, i);
  ... visit competitor_m ...
}
```

The competitors are visited in breadth-first order, beginning with `m` (which the user may choose to skip by initializing `i` in the loop above to `1` rather than `0`). There may be any number of competitors other than `m`, including none, and they may be ordinary demand monitors and workload demand monitors.

The solution contains one set of competitors which remains constant except when reset by a call to `KheSolnMatchingSetCompetitors`. If the solution changes, this set of competitors remains well-defined as a set of monitors, but becomes out of date as a set of competitors.

Competitors are useful because they can be found quickly, but they are not definitive in the way that Hall sets are: in unusual cases, a given unmatched monitor may have different competitors in different maximum matchings. For example, consider these two matchings:



Both are maximum, since all three supply nodes are matched in each; but the competitors of $C$ in the first matching are $A$ and $B$, while the competitors of $C$ in the second are $D$ and $E$.

It is important not to change the solution when traversing competitors. Even if it is changed back again, the unmatched demand nodes may be different afterwards. In the usual case where the aim is to move one meet that is competing for some scarce resources, the right approach is to use the loop to find those meets, store them, and move them after it ends.

In applications such as ejection chains it is important to understand what the defect really is. In the case of unmatched demand nodes, the true defect is the Hall set. This may be approximated in practice by the set of competitors. Thus, a repair should operate on the set of competitors independently of their order or which one happens to be the unmatched one.

## 7.5. Evenness monitoring

Suppose that a school has seven Mathematics teachers, and that at some time there are seven Mathematics lessons running simultaneously. All seven teachers must be utilized at that time, which, although feasible, will restrict the options for resource assignment later.

Unless the teachers are very overworked, there must be other times when few Mathematics lessons are running. The Mathematics lessons are distributed unevenly through the cycle.

KHE offers a kind of monitor, of type `KHE_EVENNESS_MONITOR`, which monitors this kind of evenness. These work similarly to demand monitors; they are created and removed by

```
void KheSolnEvennessBegin(KHE_SOLN soln);
void KheSolnEvennessEnd(KHE_SOLN soln);
```

although the call to `KheSolnEvennessEnd` may be omitted when evenness monitoring is wanted for the lifetime of the solution. Evenness monitors are created by `KheSolnEvennessBegin` but not attached initially. There is one evenness monitor for each resource partition of the instance and each time of the cycle, which keeps track of how many tasks whose domains lie within that partition (as determined by `KheResourceGroupPartition`) are running at that time. The monitor reports a deviation when this number exceeds some limit, which is usually set at one less than the number of resources in the partition. Thus, the deviation would be zero when six Mathematics teachers are needed, and one when seven are needed. Function

```
bool KheSolnHasEvenness(KHE_SOLN soln);
```

returns `true` when evennness monitors are present.

Like demand monitoring, evenness monitoring depends on the resources demanded at each time. Unlike demand monitoring, however, domains that cross partition boundaries are not taken into account, and evenness is only monitored at the root level of the layer tree. Despite these simplifications, evenness monitoring is potentially useful for giving early warning of demand problems, especially when used in conjunction with domain tightening (Section 11.3).

When present, evenness monitors may be found in the list of all monitors kept in the solution, and attached and detached in the usual way. More useful in practice are functions

```
void KheSolnAttachAllEvennessMonitors(KHE_SOLN soln);
void KheSolnDetachAllEvennessMonitors(KHE_SOLN soln);
```

which visit each evenness monitor and ensure that it is attached or detached. The usual operations on monitors may be carried out by upcasting to type `KHE_MONITOR` as usual. There are also operations specific to evenness monitors:

```
KHE_RESOURCE_GROUP KheEvennessMonitorPartition(KHE_EVENNESS_MONITOR m);
KHE_TIME KheEvennessMonitorTime(KHE_EVENNESS_MONITOR m);
int KheEvennessMonitorCount(KHE_EVENNESS_MONITOR m);
```

These return the partition being monitored, the time being monitored, and the number of tasks whose domains lie within that partition that are currently running at that time (or 0 if `m` is unattached). It would be useful to be able to retrieve the specific tasks that go to make up this count, but that information is not kept. If it is needed, it is necessary to search the cycle meet overlapping the time, and all the meets assigned to that cycle meet directly or indirectly, to find the tasks running at the monitored time whose domains lie within the monitored partition.

Each evenness monitor also contains a limit, such that when the count goes above that limit a deviation is reported. This limit can be retrieved and changed at any time by calling

```
int KheEvennessMonitorLimit(KHE_EVENNESS_MONITOR m);
void KheEvennessMonitorSetLimit(KHE_EVENNESS_MONITOR m, int limit);
```

Its default value is the number of resources in the partition, minus this same number divided by six and rounded down. Thus, when there are less than six resources, the value is the number of resources; when there are between six and eleven resources, the value is one less than the number of resources; and so on. This seems to work reasonably well in practice. Another way to ignore unevenness in small partitions would be to detach their monitors.

The deviation is `KheEvennessMonitorCount(m) - KheEvennessMonitorLimit(m)`, or 0 if this number is negative. This is converted into a cost by multiplying by a weight (there is no choice of cost function). The weight may be retrieved, and set at any time, by

```
KHE_COST KheEvennessMonitorWeight(KHE_EVENNESS_MONITOR m);
void KheEvennessMonitorSetWeight(KHE_EVENNESS_MONITOR m, KHE_COST weight);
```

The default weight is the smallest non-zero weight, `KheCost(0, 1)`. Helper function

```
void KheSolnSetAllEvennessMonitorWeights(KHE_SOLN soln, KHE_COST weight);
```

sets the weights of all evenness monitors at once. Finally, function

```
void KheEvennessMonitorDebug(KHE_EVENNESS_MONITOR m,
  int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

Evenness is not monitored in the current version of `KheGeneralSolve` (Section 8.4), because tests run by the author showed run time increases of about 20%, for little or no gain. Although it has some beneficial effects, evenness monitoring tends to disrupt node regularity and reduce diversity, since it causes very similar solutions to have slightly different costs.

## 7.6. Redundancy monitoring

In nurse rostering it is common for a shift to require, not a specific number of nurses, but rather a number in some range, for example between 3 and 5 nurses. This is expressed in KHE by 3 tasks with assign resource monitors plus 2 tasks without either an assign resource monitor (requiring assignment) or a prefer resources monitor with an empty domain (requiring non-assignment). These last two tasks have event resource monitor cost 0 whether they are assigned a resource or not. We call them *redundant tasks*, although they are not completely useless, because assigning a resource to a redundant task may help that resource to satisfy its resource constraints.

A redundant task may have a prefer resources monitor with a non-empty set of resources, saying in effect that it is open for assignment by certain resources but not others. Such constraints do not affect its status as a redundant task.

When workload is tight, assigning a resource to a redundant task can have a cost. Not an overt cost in the sense of a monitor with non-zero cost, but a hidden cost which comes out in workload overloads. In such cases it may be worthwhile to associate a cost with assigning a resource to a redundant task, to discourage solvers from doing it, and also to give repair methods like ejection chains a reason to try to remove these unnecessary assignments.

The same idea is at work in the ‘complete weekends’ constraints that many nurse rostering instances have, requiring a nurse to work both days of a weekend or neither. Arguably, the true constraint is a limit on the number of busy weekends (where the nurse works on one or both days), but requiring complete weekends helps to reduce the total number of busy weekends.

What is needed is one prefer resources monitor with an empty domain for each redundant task. However, the instance does not contain prefer resources constraints for these tasks’ event resources, so we need some other way to add these prefer resources monitors during solving.

KHE offers such an alternative. It follows the pattern set by evenness monitoring, except that it installs monitors that we are already familiar with: prefer resources monitors with empty domains. These are just like ordinary prefer resources monitors (Section 6.6.2) except that they are not derived from any constraint, so `KhePreferResourcesMonitorConstraint` returns `NULL`. `KhePreferResourcesMonitorDomain` works as usual and returns an empty resource group.

We call these monitors *redundancy monitors*. To begin and end monitoring, the calls are:

```
void KheSolnRedundancyBegin(KHE_SOLN soln, KHE_RESOURCE_TYPE rt,
  KHE_COST_FUNCTION cf, KHE_COST combined_weight);
void KheSolnRedundancyEnd(KHE_SOLN soln);
```

`KheSolnRedundancyBegin` makes one prefer resources monitor with empty domain for each task of type `rt` for which there are no assign resource monitors and no prefer resources monitors with empty domains, assigning cost function `cf` and weight `combined_weight` to it. `KheSolnRedundancyEnd` deletes these monitors.

As for evenness monitoring, there are functions

```
bool KheSolnHasRedundancy(KHE_SOLN soln)
```

which returns `true` when `soln` is currently monitoring unnecessary assignments, and

```
void KheSolnAttachAllRedundancyMonitors(KHE_SOLN soln);
void KheSolnDetachAllRedundancyMonitors(KHE_SOLN soln);
```

to attach all redundancy monitors that are not already attached, and to detach all redundancy monitors that are not already detached.

Redundancy monitors appear in the same places that ordinary prefer resources monitors do. They are distinguishable from ordinary prefer resources monitors by the `NULL` value returned by `KhePreferResourcesMonitorConstraint`, and by a somewhat different monitor Id.

# Part B

# Solvers

# Chapter 8. Introducing Solvers

A *solver* is a function that finds a solution, or changes a solution, or potentially changes one. This chapter introduces solvers, defines interfaces for them, and presents a few. It has also evolved into a repository for features called on by solvers: time limits, options, statistics, and so on.

Solvers operate at a high level and should not be cluttered with implementation details: their source files will include `khe_platform.h` as usual, but should not include header file `khe_interns.h` which gives access to KHE's internals. Thus, the user of KHE is as well equipped to write a solver as its author.

Many solvers are packaged with KHE. They are the subject of this part of the manual, all of which is implemented using `khe_platform.h` but not `khe_interns.h`. To gain access to these solvers, simply include header file `khe_solvers.h`, which lies in subdirectory `src_solvers` of the KHE distribution. It includes header file `khe_platform.h`, so you can omit it.

## 8.1. Keeping track of running time

Before we can get to grips with solvers, we need to describe (in this section) KHE's functions for handling time (real time, that is), and (in the following section) how the detailed behaviour of solvers can be controlled using *options*.

For the sake of compilations that do not have the Unix system functions that report time, file `khe_solvers.h` has a `KHE_USE_TIMING` preprocessor flag. Its default value is 1; changing it to 0 will turn off all calls to Unix timing system functions. If that is done, the functions documented in this section will still compile and run without error, but they will return placeholder values.

First up we have

```
char *KheDateToday(void);
```

This returns the current date as a string in static memory, or `"?"` if `KHE_USE_TIMING` is 0.

KHE offers *timer* objects, of type `KHE_TIMER`, which keep track of running time. A timer object stores its *start time*, the time that it was most recently created or reset. It may also store a *time limit*, in which case it can report whether that much time has passed since its start time. Storing a time limit does not magically stop the program at the time limit; it is up to solvers to check the time limit periodically and stop themselves when it is reached. Function `KheOptionsTimeLimitReached` (Section 8.2) is the right way to check, as we'll eventually see.

Timers represent a time as a floating point number of seconds. `KHE_NO_TIME`, a synonym for `-1.0`, means 'no time'. Function

```
float KheTimeFromString(char *str);
```

converts a string into a floating point time. If `str` is `"-"`, it returns `KHE_NO_TIME`, otherwise it returns the number of seconds represented by `str`, in the format `secs`, or `mins:secs`, or `hrs:mins:secs`. For example, `0.5` is 0.5 seconds, and `5:0` is 5 minutes. Conversely,

```
char *KheTimeShow(float secs, char buff[20]);
```

returns `secs` in string form, using `buff` for scratch memory. It writes in a more readable format than the input format, for example `"2.5 secs"` or `"5.0 mins"`.

To make a new timer object in arena `a`, call

```
KHE_TIMER KheTimerMake(char *tag, float limit_in_seconds, HA_ARENA a);
```

The `tag` parameter, which must be non-`NULL`, identifies the timer, and also appears in debug output. The `limit_in_seconds` parameter is the time limit; it may be `KHE_NO_TIME`. The timer's start time is set to the time that `KheTimerMake` is called. Also,

```
KHE_TIMER KheTimerCopy(KHE_TIMER timer, HA_ARENA a);
```

returns a copy of `timer` in arena `a`. Nothing is reset.

To retrieve the attributes of a timer, call

```
char *KheTimerTag(KHE_TIMER timer);
float KheTimerTimeLimit(KHE_TIMER timer);
```

`KheTimerTimeLimit` may return `KHE_NO_TIME`. To change them, call

```
void KheTimerResetStartTime(KHE_TIMER timer);
void KheTimerResetTimeLimit(KHE_TIMER timer, float limit_in_seconds);
```

`KheTimerResetStartTime` resets `timer`'s start time to the time that it is called. `KheTimerResetTimeLimit` resets `timer`'s time limit to `limit_in_seconds`, which may be `KHE_NO_TIME` as usual.

These functions give access to elapsed time:

```
float KheTimerElapsedTime(KHE_TIMER timer);
float KheTimerRemainingTime(KHE_TIMER timer);
bool KheTimerTimeLimitReached(KHE_TIMER timer);
```

`KheTimerElapsedTime` returns the amount of time that has elapsed since the most recent call to `KheTimerMake` or `KheTimerResetStartTime` for the timer.

`KheTimerRemainingTime` returns the amount of time remaining until the time limit is reached: the time limit minus the elapsed time. However, there are two special cases. First, if the time limit is passed it returns 0.0 rather than a negative result. And second, if `timer` has no time limit the result is logically infinite, but what is actually returned is `KHE_NO_TIME`.

`KheTimerTimeLimitReached` returns `true` when the elapsed time is equal to or greater than the time limit (always false when the time limit is `KHE_NO_TIME`). Finally,

```
void KheTimerDebug(KHE_TIMER timer, int verbosity, int indent, FILE *fp);
```

produces a debug print of `timer` onto `fp` with the given verbosity and indent.

Complex solvers may want to keep track of several time limits simultaneously, for example a global limit plus a limit on the running time of one phase. For this there are objects of type `KHE_TIMER_SET`, representing sets of timers. To create a new, empty timer set in arena `a`, call

```
KHE_TIMER_SET KheTimerSetMake(HA_ARENA a);
```

To make a copy of a timer set, call

```
KHE_TIMER_SET KheTimerSetCopy(KHE_TIMER_SET timer_set, HA_ARENA a);
```

To add and delete timers, call

```
void KheTimerSetAddTimer(KHE_TIMER_SET timer_set, KHE_TIMER timer);
void KheTimerSetDeleteTimer(KHE_TIMER_SET timer_set, KHE_TIMER timer);
```

`KheTimerSetDeleteTimer` aborts if `timer` is not present in `timer_set`. There is also

```
bool KheTimerSetContainsTimer(KHE_TIMER_SET timer_set, char *tag,
  KHE_TIMER *timer);
```

which seaches for a timer with the given `tag` in `timer_set`. If there is one, it sets `*timer` to one such timer and returns `true`, otherwise it returns `false`.

This time around there are only two functions for elapsed time:

```
float KheTimerSetRemainingTime(KHE_TIMER_SET timer_set);
bool KheTimerSetTimeLimitReached(KHE_TIMER_SET timer_set);
```

A `KheTimerSetElapsedTime` function would make no sense, since different timers may have different start times. `KheTimerSetRemainingTime` returns the minimum of the remaining times of the timer set's timers that have time limits (never negative as usual), or `KHE_NO_TIME` if there are no such timers, or if time limit consistency checking is active as described below. `KheTimerSetTimeLimitReached` returns `true` if at least one of the timers of `timer_set` has reached its limit—the logical moment to stop if several time limits are present.

When there are time limits, slight variations in running time can cause runs with identical options to produce different results. This is inconvenient when testing, and it can mask uninitialized variable bugs. So timer sets offer these functions:

```
void KheTimerSetTimeLimitConsistencyBegin(KHE_TIMER_SET timer_set,
  KHE_SOLN soln);
void KheTimerSetTimeLimitConsistencyEnd(KHE_TIMER_SET timer_set,
  KHE_SOLN soln);
```

The idea is to call `KheTimerSetTimeLimitConsistencyBegin` near the start of the solve that `timer_set` supplies time limits for, and `KheTimerSetTimeLimitConsistencyEnd` near the end. Then when subsequent runs call on `KheTimerSetTimeLimitReached` they will get the same results as this run, ensuring that any difference in their solutions is not due to time limits.

The timer set caches the result of each call to `KheTimerSetTimeLimitReached`. At the end of the run, `KheTimerSetTimeLimitConsistencyEnd` writes these results to a file. At the start of the next run, `KheTimerSetTimeLimitConsistencyBegin` reads the file and takes control of `KheTimerSetTimeLimitReached`, ensuring that each call returns what it returned last time.

This means that a timer set's time limit consistency aspect can be in one of three states: it can be *absent* (when `KheTimerSetTimeLimitConsistencyBegin` is not called), or *starting* (when `KheTimerSetTimeLimitConsistencyBegin` is called but there is no file to read), or *active* (when `KheTimerSetTimeLimitConsistencyBegin` is called and there is a file to read). In the active state, the timers' time limits are bypassed and the contents of the file are used to determine when to tell callers that time has run out.

The file name is `khe_<instanceid>_<diversifier>.tlc`. The user must ensure that when a file with this name is present, the next run has identical options to the run that wrote the file (because if options differ, time limit consistency usually makes no sense). That is, the user must delete this file if this condition does not hold. Although time limit consistency is packaged up conveniently for the user in option `gs_time_limit_consistency` (Section 8.4), that option reads any `.tlc` file with the right name; it has no way to check whether it really should.

Finally,

```
void KheTimerSetDebug(KHE_TIMER_SET timer_set, int verbosity,
  int indent, FILE *fp)
```

produces a debug print of `timer_set` onto `fp` with the given verbosity and indent.

The usual way to keep track of running time is by calling the timer functions of options objects (Section 8.2). These just delegate to a timer set object stored within the options object.

## 8.2. Options, running time, and time limits

Solvers have an `options` parameter of type `KHE_OPTIONS`, holding options that influence their behaviour. This type is similar to a Unix environment: it is a symbol table with strings for its keys and values. The KHE main program allows options to be passed in via the command line.

To create a new options object containing the empty set of options, call

```
KHE_OPTIONS KheOptionsMake(HA_ARENA a);
```

It is created in arena `a`, which it remembers and returns in

```
HA_ARENA KheOptionsArena(KHE_OPTIONS options);
```

There is no operation to delete an options object; instead, delete its arena.

Options can be changed at any time, so when solving in parallel it is important for different options objects to be passed to each solve. These can be created by copying using

```
KHE_OPTIONS KheOptionsCopy(KHE_OPTIONS options, HA_ARENA a);
```

The copy is stored in arena `a`. `KheArchiveParallelSolve` and `KheInstanceParallelSolve` (Section 8.5) do this.

To set an option, and to retrieve the previously set value, the calls are

```
void KheOptionsSet(KHE_OPTIONS options, char *key, char *value);
char *KheOptionsGet(KHE_OPTIONS options, char *key, char *dft);
```

`KheOptionsGet` returns the value associated with `key` in the most recent call to `KheOptionsSet` with that key. If there is no such call, it returns `dft`, reflecting the principle that solvers should not rely on their options being set, but rather should be able to choose a suitable value when they are absent—a value that may depend upon circumstances, not necessarily a fixed default value.

By convention, when an option represents a Boolean, its legal values are `"false"` and `"true"`. On the KHE command line, omitting the option omits it from the options object, which usually means that its value is intended to be `false`, while including it, either in the full form `"option=true"` or the short form `"option"`, gives it value `"true"`. Functions

```
void KheOptionsSetBool(KHE_OPTIONS options, char *key, bool value);
bool KheOptionsGetBool(KHE_OPTIONS options, char *key, bool dft);
```

make it easy to handle Boolean options. `KheOptionsSetBool` calls `KheOptionsSet`, with value `"true"` or `"false"` depending on `value`. `KheOptionsGetBool` calls `KheOptionsGet`, returning an actual Boolean rather than a string. It aborts if the value is not `"false"` or `"true"`. If there is no value it returns `dft`, which, as explained above, would usually be `false`.

Another common case is when an option represents an integer. Convenience functions

```
void KheOptionsSetInt(KHE_OPTIONS options, char *key, int value);
int KheOptionsGetInt(KHE_OPTIONS options, char *key, int dft);
```

make this case easy. `KheOptionsSetInt` calls `KheOptionsSet`, with value equal to `value` in string form. `KheOptionsGetInt` calls `KheOptionsGet`, then returns the value converted to an integer. It aborts if the value is not an integer. If there is no value it returns `dft`. Functions

```
void KheOptionsSetFloat(KHE_OPTIONS options, char *key, float value);
float KheOptionsGetFloat(KHE_OPTIONS options, char *key, float dft);
```

work in the same way for floating-point options.

It is also possible to associate an arbitrary pointer with a key, by calling functions

```
void KheOptionsSetObject(KHE_OPTIONS options, char *key, void *value);
void *KheOptionsGetObject(KHE_OPTIONS options, char *key, void *dft);
```

These work in much the same way as the other functions.

When `KheOptionsCopy` is called, by `KheArchiveParallelSolve` for example, object options are shared between the copies. Care is needed, since sharing mutable objects between threads is not safe. The KHE solvers avoid problems here by not adding any object options until after the copying has been done: only single-threaded solve functions add them.

Options can be roughly classified into two kinds. One kind is for end users, to allow them to try out different possibilities. Options of this kind are not set by KHE's solvers, only used. The other kind is for KHE's solvers, to allow them to vary the behaviour of other solvers that they call. These are set by KHE's solvers, so it is usually futile for the end user to set them.

Throughout this Guide, options are described along with the solvers they influence. As an aid to managing option names, there is a convention for beginning option names with a three-character prefix:

| | |
|---|---|
| `gs_` | Options set or consulted by general solvers |
| `ps_` | Options set or consulted by parallel solvers |
| `ss_` | Options set or consulted by structural solvers |
| `ts_` | Options set or consulted by time solvers |
| `rs_` | Options set or consulted by resource solvers |
| `es_` | Options set or consulted by ejection chain solvers |

Some options are set by one kind of solver and consulted by another; such options are hard to classify. The sole option consulted by the KHE main program has no prefix. It is:

`no_print`
If this Boolean option appears in the first list of options on the `khe -s` or `khe -r` command line, then solving will proceed as usual but the result archive will not be printed.

The default values of all Boolean options consulted by KHE code are always `false`; for the other options, a default value is always given as part of the description of the option.

Options objects are passed around through solvers, making them a good, if not entirely natural, place to keep other things which are not options, strictly speaking. In particular, each options object contains a timer set (Section 8.1) which may be used to keep track of running time and impose time limits. The relevant functions are

```
KHE_TIMER KheOptionsAddTimer(KHE_OPTIONS options, char *tag,
  float limit_in_seconds);
void KheOptionsDeleteTimer(KHE_OPTIONS options, KHE_TIMER timer);
bool KheOptionsContainsTimer(KHE_OPTIONS options, char *tag,
  KHE_TIMER *timer);
float KheOptionsRemainingTime(KHE_OPTIONS options);
bool KheOptionsTimeLimitReached(KHE_OPTIONS options);
void KheOptionsTimeLimitConsistencyBegin(KHE_OPTIONS options,
  KHE_SOLN soln);
void KheOptionsTimeLimitConsistencyEnd(KHE_OPTIONS options,
  KHE_SOLN soln);
int KheOptionsTimeLimitReachedQueryCount(KHE_OPTIONS options);
void KheOptionsTimerSetDebug(KHE_OPTIONS options, int verbosity,
  int indent, FILE *fp);
```

All these functions simply delegate their work to the corresponding function of the option object's timer set. See Section 8.1 to find out what they do.

Finally, there is one stray function,

```
KHE_FRAME KheOptionsFrame(KHE_OPTIONS options, char *key, KHE_SOLN soln);
```

This returns a shared common frame for use by solvers, as described in Section 5.10.

## 8.3. Do-it-yourself solving

KHE offers many solvers, opening up several questions: are they all useful? which order should they be called in? how much running time should each get? KHE makes it easy to investigate these questions empirically, by means of *do-it-yourself solving*, which means using options to determine which solvers get called, in what order, and how much time to give to each.

Some solvers do something, then run an arbitrary solver, then finish by doing something else (typically undoing some or all of what they did initially). For example, they might change the weights of some monitors, run an arbitrary solver, then change the weights back again. Another possibility is to make some assignments and fix them, run an arbitrary solver, then remove the fixes. We call such solvers *enclosing solvers* here. Other solvers just do something. For example, a time sweep assignment algorithm just assigns resources to tasks; it does not do something else later. Such solvers we call *non-enclosing solvers.*

There are three do-it-yourself solving options: `gs` for general solving, `ts` for time solving, and `rs` for resource solving. The values of these three options all follow this little grammar:

```
<solver>  ::=  <name> [ <solver> ]
          ::=  "(" <item> "," <item> { "," <item> } ")"
<name>    ::=  <id> [ "!" <id> ]
<item>    ::=  [ <int> ":" ] <solver>
```

Here `{` and `}` mean zero or more of what they enclose, `[` and `]` mean that what they enclose is optional, `<id>` is an identifier from a fixed set of short names for solvers defined in this and following chapters, `<int>` is an integer concerned with time limits, as explained below, and values in quotes appear literally. White space is allowed between tokens but not within them.

The first line of the grammar calls a solver defined by `<name>`. For each identifier from the fixed set, do-it-yourself solving knows whether that identifier names an enclosing solver or an unenclosing solver, and it chooses

```
<name> <solver>
```

when the solver is enclosing, and

```
<name>
```

when it is non-enclosing. If `<name>` denotes an enclosing solver, then that solver is called, then `<solver>` is run, then `<name>`'s finishing action is called. If `<name>` denotes a non-enclosing solver, then that solver (only) is called.

If `<name>` has the form `id1!id2`, then `id1` is called when the model is high school timetabling, and `id2` is called when it is nurse rostering. In this case, `do` and `skip` make good values for `id1` and `id2`; the solve will do something for one model and nothing for the other.

The second line of the grammar causes two or more solvers to be called sequentially, in the order they appear. In this case the grammar allows you to specify how running time is to be apportioned among the solvers, if you wish. This is done in two steps.

First, set an overall time limit. For example,

```
gs_time_limit="10:0"
```

sets an overall time limit of 10 minutes. There are also `ts_time_limit` and `rs_time_limit` options which limit the time given to the time assignment and resource assignment phases (options `ts` and `rs`), in conjunction with (not replacing) any `gs_time_limit` value. None of these time limits is required; omitting one means that there is no time limit.

The second step is to prefix an integer *time weight* to each solver in a sequence of two or more solvers. Each solver gets an amount of time proportional to its weight. For example, the hypothetical solver

```
"(3: s1, 1: s2, 1: s3)"
```

apportions 60% of the available time to `s1`, 20% to `s2`, and 20% to `s3`.

A time weight must be a non-negative integer. Omitted time weights default to 1. Time weight 0 means that the item's solver will not be called, even when there is no time limit.

Each solver is expected to return promptly when its share of available time is up. It can and should find this out, without caring what its time limit is or who imposed it, by periodically calling `KheOptionsTimeLimitReached` (Section 8.2). Do-it-yourself solving sets timers which influence `KheOptionsTimeLimitReached`, but it cannot compel a solver to return on time. If a solver returns late, the time available to following solvers is reduced, possibly to zero, in which case they are not called. If a solver returns early, the time available to following solvers is increased. Either way, the proportions given by the following time weights are preserved.

One function, `KheDoItYourselfSolverParseAndRun`, is exported by the source file that implements do-it-yourself solving. It parses an option value and runs it. It is easy to use, but it is not intended for end users. It is called by the three documented do-it-yourself solver functions.

## 8.4. General solving

We have defined a solver to be a function that finds a solution, or changes a solution, or potentially changes one. A *general solver* solves an instance completely, unlike, say, a *time solver* which only finds time assignments, or a *resource solver* which only finds resource assignments. A general solver may split meets, build layer trees and task trees, assign times and resources, and so on without restriction.

The recommended interface for a general solver, defined in `khe.h`, is

```
typedef KHE_SOLN (*KHE_GENERAL_SOLVER)(KHE_SOLN soln, KHE_OPTIONS options);
```

It will usually return the solution it is given, but it may return a different solution to the same instance, in which case it should delete the solution it is given. Its second parameter, `options`, is a set of options (Section 8.2) which may be used to vary the behaviour of the solver.

The main general solver distributed with KHE is

```
KHE_SOLN KheGeneralSolve2024(KHE_SOLN soln, KHE_OPTIONS options);
```

This single-threaded solver works by calling functions defined elsewhere in this guide. It returns the solution it is given. The name includes the year it was completed and will change from time to time. In publications and solution group names it is referred to as KHE20, KHE24, etc.

If memory runs out while `KheGeneralSolve2024` is solving `soln`, by default it will abort.

This can be changed by calling `HaArenaSetJmpEnvBegin` (Appendix A.1.2) on `soln`'s arena set. `KheArchiveParallelSolve` and `KheInstanceParallelSolve` (Section 8.5) do this.

`KheGeneralSolve2024` assumes that `soln` is as returned by `KheSolnMake`, so it carries on from there. To begin with, it checks `options` for the options listed below, and handles them as described there. Then it calls a few functions that are basically unavoidable when starting a solve: `KheSolnSplitCycleMeet` (Section 4.5.3), `KheSolnMakeCompleteRepresentation` (Section 4.3), `KheLayerTreeMake` (Section 9.1), and `KheTaskTreeMake` (Section 11.3). It also sets option `es_split_moves`, to `true` if the instance contains soft split assignments constraints, and to `false` otherwise, for the convenience of other solvers which might be wondering whether there is any point in splitting or merging meets.

`KheGeneralSolve2024` then switches to do-it-yourself solving, using option `gs` (described in detail below) as its guide. Then it calls a few functions that again are basically unavoidable when ending a solve: `KheSolnEnsureOfficialCost` (Section 6.3), `KheMergeMeets` (Section 9.7.2), `KheSolnTryTaskUnAssignments` (Section 12.14), and `KheSolnTryMeetUnAssignments` (Section 10.4). It also sets the running time field of the solution to the wall clock time since it began. This is not necessarily what the caller wants, but it can easily be reset after the function returns if not. Finally it returns the solution object it was given, highly modified.

By convention, options set or consulted directly by `KheGeneralSolve2024` have names beginning with `gs`. Here is the full list:

`gs`

A string option containing a do-it-yourself solver which determines most of what `KheGeneralSolve2024` does. See below for futher details.

`gs_diversifier`

An integer option which, when set, causes `KheGeneralSolve2024` to set the diversifier of the solution it is given to the given value. When omitted, the diversifier retains the value it has when `KheGeneralSolve2024` is called.

`gs_time_limit`

A string option defining a soft time limit for each call to `KheGeneralSolve2024`. The format is as for function `KheTimeFromString` described above (Section 8.1): either `"-"`, meaning no time limit (the default value), or `secs`, or `mins:secs`, or `hrs:mins:secs`. For example, `10` is 10 seconds, and `5:0` is 5 minutes. Enforcement is up to particular solvers; this option merely calls `KheOptionsAddTimer` (Section 8.2).

`gs_time_limit_consistency`

A Boolean option whose default value is `false`. When its value is changed to `true`, `KheOptionsTimeLimitConsistencyBegin` (Section 8.2) is called near the start of the solve, and `KheOptionsTimeLimitConsistencyEnd` is called near the end. As Section 8.1 explains, this ensures that successive runs with the same options do not produce different results owing to small differences in running time.

This option works with multiple instances and parallel solves, but it does not reinitialize when options change, making it quite error-prone. It is best used only occasionally, to verify that annoying variations in results really are due to small differences in running time. This option writes and reads files with `.tlc` suffixes, and these files need to be deleted by the user after finishing with this option, or when changing any other options.

`gs_unassignment_off`
: A Boolean option which, when `"true"`, instructs `KheGeneralSolve2024` to omit the calls to `KheSolnTryTaskUnAssignments` (Section 12.14) and `KheSolnTryMeetUnAssignments` (Section 10.4) at the end.

`gs_debug_monitor_id`
: This option is a string identifying a monitor. It has two or more fields, separated by slashes. The first field is a constraint Id; the others identify a point of application of the constraint. For example, `"Constraint:5/Nurse3/27"` is the monitor for constraint `"Constraint:5"` at point of application `Nurse3`, offset `27`. This option is used by `KheGeneralSolve2024` to define option `gs_debug_monitor`, as explained next. The conversion from string to monitor is carried out by function `KheSolnRetrieveMonitor` (Section 6.2).

`gs_debug_monitor`
: This option is set at the start of `KheGeneralSolve2024`, when `gs_debug_monitor_id` is present, to the monitor identified by `gs_debug_monitor_id`. Any solver can reference it and use it as a hint to produce debug output relevant to that monitor. At present only ejectors do this: they produce debug output focussed on answering the question 'Why is the defect represented by this monitor not removed by the ejection chain algorithm?'.

`gs_debug_rtm`
: This option is a string whose format is `resource:timegroup`. When present it causes `KheGeneralSolve2024` to call `KheResourceTimetableMonitorSetDebug` (Section 6.8.2) to set up debugging of the resource timetable monitor of `resource` at the times of `timegroup`. This will abort if macro `DEBUG_CELL` in file `khe_resource_timetable_monitor.c` does not have value 1.

`KheGeneralSolve2024` is affected indirectly by many other options, via the solvers it calls.

Option `gs` is a do-it-yourself solver (Section 8.3) which determines most of what `KheGeneralSolve2024` does, as described earlier. The solvers it is able to call are these:

| `<item>` | Meaning |
|---|---|
| `do <solver>` | Run `<solver>`. Useful with `!`. |
| `skip <solver>` | Do nothing (don't run `<solver>`). Useful with `!`. |
| `empty` | Do nothing (non-enclosing syntax). Useful with `!`. |
| `ts` | Call `KheCombinedTimeAssign` (Section 10.9). |
| `rs` | Call `KheCombinedResourceAssign` (Section 12.15). |
| `gdl <solver>` | Call `KheDetachLowCostMonitors` (Section 8.6.2), then run `<solver>`, then undo what `KheDetachLowCostMonitors` did. |
| `gts <solver>` | Install a separate matching, run `<solver>`, then uninstall it. 'Separate' means that unmatched demand nodes can be accessed but do not contribute to the cost of the solution (Section 8.8.1). |
| `gti <solver>` | Install an integrated matching, run `<solver>`, then uninstall it. 'Integrated' means that unmatched demand nodes contribute to the cost of the solution (Section 8.8.1). |
| `gem <solver>` | Install evenness monitoring (Section 7.5), run `<solver>`, then uninstall it. |
| `gtp <solver>` | Call `KheTiltPlateau` (Section 11.5.2), run `<solver>`, then remove the tilt. |
| `gpu` | Call `KhePropagateUnavailableTimes` (Section 11.5.3). |

Actually everything callable from any of the `gs`, `ts`, and `rs` options is callable from all of them. But for sanity it seems best to partition solvers into general, time, and resource kinds.

It is wrong to call a solver from within itself. Also `gti` should not be called from within `gts`, and `gts` should not be called from within `gti`. The result in such cases is unpredictable.

The default value of `gs` is

```
gs="(gpu, ts, rs)"
```

Calls on `gts` and `gti` appear within the default values of `ts` and `rs`. The default values do not call `gdl` or `gem`, because the author has not found them to be useful. However they are available.

Function

```
void KheSolveDebug(KHE_SOLN soln, KHE_OPTIONS options, char *fmt, ...);
```

produces a one-line debug of the current state of a solve. For conciseness it always prints onto `stderr` with indent 2. The print contains `soln`'s instance name, diversifier, cost, and running time (if `options` contains a timer called `"gs_time_limit"`; if not, the running time is omitted), and ends with whatever `fprintf(stderr, fmt, ...)` would produce, followed by a newline.

## 8.5. Parallel solving

Function

```
void KheArchiveParallelSolve(KHE_ARCHIVE archive,
  KHE_GENERAL_SOLVER solver, KHE_OPTIONS options,
  KHE_SOLN_TYPE soln_type, HA_ARENA_SET as);
```

solves the instances of `archive` in parallel, optionally adding the solutions it finds to `archive`. Any existing instances and solution groups of `archive` remain as they were, although if the `ps_cache` option (see below) is active they may be replaced by new versions of themselves.

Each individual solve is carried out by `solver`, which is passed a fresh solution and a copy of `options`. The fresh solution is as returned by `KheSolnMake` except that the diversifier is set, as explained below.

If solutions are saved (see options `ps_soln_group` and `ps_first_soln_group` below), parameter `soln_type` determines whether they are left as they are or reduced to placeholders (Section 4.2.6). `KHE_SOLN_WRITABLE_PLACEHOLDER` is recommended because it recycles a large amount of memory, while still permitting the solutions to be written. Only if further processing of the solutions is intended would they be left as they are, by passing `KHE_SOLN_ORDINARY`.

Each call to `KheSolnMake` is passed an arena set. There is one arena set per thread, with `as` (which must be non-`NULL`) serving one thread and freshly created arena sets serving the others. At the end, any solutions needing to be kept are copied into arenas of `as`, and the other arena sets are merged back into `as`, so no memory is lost. If further parallel solving of these solutions is attempted, it will be necessary to copy them into arenas from distinct arena sets first.

Before each call to `solver`, `KheArchiveParallelSolve` calls `HaArenaSetJmpEnvBegin` (Appendix A.1.2) on the arena set of the solution passed to `solver`. This ensures that if memory runs out while `solver` is running, `KheArchiveParallelSolve` can take back control and handle the problem. What it does is to take the solution object passed to `solver` (which contains a valid value, the value just before `solver` ran out of memory) and use that as the final solution, recording in its description metadata field that memory ran out.

There is also

```
KHE_SOLN KheInstanceParallelSolve(KHE_INSTANCE ins,
  KHE_GENERAL_SOLVER solver, KHE_OPTIONS options,
  KHE_SOLN_TYPE soln_type, HA_ARENA_SET as);
```

Behind the scenes it is the same, but it solves a single instance rather than an entire archive, and it returns any one best solution rather than storing solutions in a solution group.

All objects created by these two functions, except for solutions that are kept, are deleted before they return. This includes all copies of `options`, and all freshly created arena sets.

Options consulted by parallel solvers have names beginning with `ps_`. Here is the full list:

`ps_threads`
: The number of threads used for solving. This includes the initial thread, the one that called `KheArchiveParallelSolve` or `KheInstanceParallelSolve`, so the value must be at least 1. If `ps_threads` is absent, or present but KHE has been compiled with multi-threading off, its value is taken to be 1.

`ps_make`
: The number of solutions `KheArchiveParallelSolve` and `KheInstanceParallelSolve`

are supposed to make per instance. If `ps_make` is absent, its value is taken to be 1. The number actually made will be less than this in two cases: when time runs out before all the solves are started, and when an evidently optimal solution is found before all the solves are started. A solution is evidently optimal when its cost equals the lower bound returned by `KheMonitorLowerBound((KHE_MONITOR) soln)`. This value is usually 0.

`ps_no_diversify`

For each instance, the solutions passed to `solver` are identical except that the diversifier of the first is 0, the diversifier of the second is 1, and so on. The solver may use these values to create diverse solutions. Boolean option `ps_no_diversify`, when `"true"`, gives the same diversifier (namely 0) to all solutions. All solutions should then turn out the same, except when there are time limits: they can cut off solving at slightly different moments.

`ps_keep`

The maximum number of solutions that `KheArchiveParallelSolve` keeps (stores in `ps_soln_group` below) per instance. If `ps_keep` is absent, its value is taken to be 1. The best `ps_keep` solutions are kept. `KheInstanceParallelSolve` does not consult this option; it always keeps (in fact, returns) one solution, the best it found.

`ps_soln_group`

A string option, which, if present, causes a solution group to be added to `archive` holding the best `ps_keep` solutions to each instance. The value of the string is the name of the solution group. If there is already a solution group in `archive` with that name, `KheArchiveParallelSolve` aborts with an error message.

If `ps_soln_group` is omitted, no solution group is made. When solutions have been found but they are not in the result archive, this is the usual reason. Or the solution group may have been added to the archive in memory, but the archive has not been written to a file.

`ps_first_soln_group`

Like `ps_soln_group` except that the solution group holds one solution for each instance, the one whose solve was started first. This solution will thus be added to two solution groups if `ps_soln_group` and `ps_first_soln_group` are both present and the solution is one of the `ps_keep` best for its instance. (Actually, in that case the solution is copied, owing to the possible need to store different running times in the two versions, as explained just below under option `ps_time_measure`.) The author has at times used

```
ps_first_soln_group=KHE20 ps_soln_group=KHE20x8
```

to get the results of a single run and of a best of 8 run, while producing only eight (not nine) solutions. If present, `ps_first_soln_group` will precede the other in the archive.

`ps_time_measure`

Measuring running time is awkward for parallel solving. This option says how to do it.

If `ps_time_measure` is `"omit"`, the parallel solver does not set the solutions' running times. They have the values given to them by `solver`. If `solver` is `KheGeneralSolve2024`, for example, each holds the wall clock time from when `KheGeneralSolve2024` was called to when it returns. This is useful when all solutions are kept, for showing how running times vary. It is misleading when `ps_threads` exceeds the number of processors.

If `ps_time_measure` is `"shared"`, each instance monopolizes all threads while its solutions are being constructed. There is some idle time for some threads while they wait for others to finish off the current instance, making the total wall clock time of the solve somewhat larger than for `"omit"`. Then the running times of all solutions for one instance are set to the same value: the wall clock time from when the first solve of their instance began until the last solve ended. This is useful when only the best, or the few best, solutions are being kept, because it records in those solutions how long it really takes to find them, given that all the solutions have to be found, albeit in parallel, before the few can be chosen.

If `ps_time_measure` is `"auto"` (the default value), then the behaviour is as for `"omit"` when `ps_keep >= ps_make`, and as for `"shared"` when `ps_keep < ps_make`.

This option only affects the solutions stored in `ps_soln_group`. The solutions stored in `ps_first_soln_group` have the running times given to them by `solver`.

There are at least two other effects that can influence the integrity of reported running times. One is that arena memory allocated for one solve is recycled for the next solve by the same thread, which could change the running time for solving an instance depending on whether it is the thread's first solve. The other is that parallel solves are not completely independent of one another: they may contend for resources such as `malloc`, memory caches, and memory buses. On the author's nominally 12-core computer, one run of 12 parallel runs proceeds about 20% more slowly than one run with no competing parallel runs.

`ps_time_limit`

A string option defining a soft time limit for solving each instance. The parallel solver will stop initiating solves of an instance once the wall clock time since it initiated the first solve of that instance exceeds this limit, even if the requested `ps_make` solves have not all begun. The format is as for function `KheTimeFromString` described above (Section 8.1): either `"-"`, meaning no time limit (which is the default value), or `secs`, or `mins:secs`, or `hrs:mins:secs`. For example, `10` is 10 seconds, and `5:0` is 5 minutes. What `ps_time_limit` does is quite weak; in practice, setting `gs_time_limit` (Section 8.4) will be more effective.

`ps_avail_mem`

An integer option that informs the solve how much memory is available, measured in bytes. When present, the value is divided by `ps_threads` and the result passed as memory limit to each thread's arena set, ensuring that no thread can monopolize memory. Special value `sysinfo` causes the value to be taken from the Linux `sysinfo` system function, provided the `USE_SYSINFO` flag, defined at the top of `khe_solvers.h`, has value 1. Value `sysinfo` is the default when available, otherwise the default behaviour is to install no memory limits.

`ps_use_cache`

This Boolean option, when `true`, makes `KheArchiveParallelSolve` store intermediate results in a file, allowing a long solve to be interrupted and resumed later. This works as follows. If the file exists when `KheArchiveParallelSolve` is called, it is read as an archive and its instances and solution groups replace the instances and solution groups initially present in the `archive` parameter. Then, before starting to solve each instance `ins`, the archive is checked to see whether it already contains any solutions for `ins`. If so, then `ins` is not solved. Finally, after solving each instance, if at least one solution was added to the archive, the whole archive is written to the file, overwriting any existing value.

This only works for `KheArchiveParallelSolve`, not for `KheInstanceParallelSolve`. There must be a `ps_soln_group` or `ps_first_soln_group` option.

If the Id of the original archive is `X`, then the name of the cache file is `tmp_X.xml`. This ensures that unrelated archives never become confused and that original archive files are never overwritten (although they may be unexpectedly bypassed in favour of a cache file). To run from scratch, remove the cache file or set `ps_use_cache` to `false`.

Parallelism is obtained via functions `pthread_create` and `pthread_join` from the Posix threads library. KHE has been carefully designed to ensure that operations carried out in parallel on distinct solutions cannot interfere with each other. If you do not have Posix threads, a simple workaround documented in KHE's makefile will allow you to compile KHE without it. The only difference is that `KheArchiveParallelSolve` and `KheInstanceParallelSolve` will find their solutions sequentially rather than in parallel.

## 8.6. Solution adjustments

To *adjust* a solution is to change it in some way, with the intention of changing it back again later. This section presents the *solution adjuster* type, which supports many kinds of adjustments. It accepts a mixture of different kinds of adjustments, which can be useful, but its main value is that it makes it easy to remove the adjustments when they are no longer wanted. (We can't use marks and paths for this, because there may be other operations on the path that we don't want to remove.) One application of adjustment is documented in this section. Other sections of this Guide contain other applications. There are also sections which would naturally use adjusters but currently do not. This is for historical reasons only; in time, hopefully, all places where adjustments are made and subsequently removed will use adjusters.

### 8.6.1. The solution adjuster

When we adjust a solution we want to remember what we have done, so that we can undo it later. KHE offers *solution adjusters* for this. They are created and deleted by

```
KHE_SOLN_ADJUSTER KheSolnAdjusterMake(KHE_SOLN soln);
void KheSolnAdjusterDelete(KHE_SOLN_ADJUSTER sa);
```

Function

```
KHE_SOLN KheSolnAdjusterSoln(KHE_SOLN_ADJUSTER sa);
```

returns `sa`'s `soln` attribute.

Between the calls to `KheSolnAdjusterMake` and `KheSolnAdjusterDelete`, any number of adjustments may be made to `soln`. These are both applied to `soln` and recorded within the adjuster. They are undone (in reverse order) when `KheSolnAdjusterDelete` is called.

The adjustments to monitors currently supported by solution adjusters are

```
void KheSolnAdjusterMonitorEnsureAttached(KHE_SOLN_ADJUSTER sa,
  KHE_MONITOR m);
void KheSolnAdjusterMonitorEnsureDetached(KHE_SOLN_ADJUSTER sa,
  KHE_MONITOR m);
void KheSolnAdjusterMonitorChangeWeight(KHE_SOLN_ADJUSTER sa,
  KHE_MONITOR m, KHE_COST combined_weight);
void KheSolnAdjusterMonitorSetTilt(KHE_SOLN_ADJUSTER sa,
  KHE_LIMIT_ACTIVE_INTERVALS_MONITOR laim);
```

KheSolnAdjusterMonitorEnsureAttached attaches m, or if m is already attached it does nothing. KheSolnAdjusterMonitorEnsureDetached detaches m, or if m is already detached it does nothing. KheSolnAdjusterMonitorChangeWeight changes the combined weight of m to combined_weight. KheSolnAdjusterMonitorSetTilt sets the tilt flag of laim, or does nothing if it is already set. (It does not change the weight; that may be done separately.)

The adjustments to tasks currently supported by solution adjusters are

```
bool KheSolnAdjusterTaskMove(KHE_SOLN_ADJUSTER sa,
  KHE_TASK task, KHE_TASK target_task);
bool KheSolnAdjusterTaskAssignResource(KHE_SOLN_ADJUSTER sa,
  KHE_TASK task, KHE_RESOURCE r);
void KheSolnAdjusterTaskEnsureFixed(KHE_SOLN_ADJUSTER sa,
  KHE_TASK task);
void KheSolnAdjusterTaskEnsureUnFixed(KHE_SOLN_ADJUSTER sa,
  KHE_TASK task);
```

KheSolnAdjusterTaskMove calls KheTaskMove(task, target_task) and returns what it returns; KheSolnAdjusterTaskAssignResource calls KheTaskAssignResource(task, r) and returns what it returns. KheSolnAdjusterTaskEnsureFixed fixes the assignment of task, or does nothing if it is already fixed. KheSolnAdjusterTaskEnsureUnFixed unfixes the assignment of task, or does nothing if it is already not fixed.

For all these operations, the adjuster remembers whatever is needed about the initial state, so that it can undo the adjustment correctly later. Other adjustments may be added in the future.

It is safe to call these operations when sa is NULL. In that case, the adjustment is carried out but not remembered anywhere. This can be convenient when writing code that remembers the adjustments it makes, but only if requested.

There is one more adjustment that we have saved to last because it is more complicated:

```
bool KheSolnAdjusterTaskGroup(KHE_SOLN_ADJUSTER sa,
  KHE_TASK task, KHE_TASK leader_task);
```

This groups task with leader_task; undoing it takes that grouping away. Concretely, it moves task to leader_task, exactly like KheSolnAdjusterTaskMove. But undoing is different: instead of moving task back to its original assignment, it moves it to whatever leader_task is assigned to at the moment of undo (possibly NULL). This removes the grouping without removing any assignment that has appeared since the grouping was done.

KheSolnAdjusterTaskGroup requires task to be non-NULL and unassigned, and it requires `leader_task` to be unassigned. It fixes the assignment of `task` to `leader_task`, by calling `KheTaskAssignFix` after making it, and unfixes it when undoing. This is for sanity: it says, more or less, that the solution adjuster owns this assignment and only it can take it away again.

### 8.6.2. Detaching low-cost monitors

On difficult instances it might make sense to forget about monitors whose violations cost very little, and concentrate on monitors whose violations cost a lot. This idea is implemented by

```
void KheDetachLowCostMonitors(KHE_SOLN soln, KHE_COST min_weight,
  KHE_SOLN_ADJUSTER sa);
```

It detaches all attached monitors of `soln` whose combined weight is less than `min_weight`. If `sa != NULL` it records what it has done in solution adjuster `sa` (Section 8.6.1) so that it can be undone later. For example,

```
sa = KheSolnAdjusterMake(soln);
KheDetachLowCostMonitors(soln, sa, KheCost(1, 0));
do_something;
KheSolnAdjusterDelete(sa);
```

detaches monitors for soft constraints while `do_something` is running, then reattaches them. One could pass `NULL` for `sa`, but then undoing the detaches would be awkward, because it would not be clear which of all the detached monitors were detached by this function.

## 8.7. Monitor grouping

To *group* two or more monitors means to make them children of a common parent group monitor. This is a kind of solution adjustment, but for historical reasons it is treated separately.

There are several reasons why monitor grouping might be needed, the main ones being *correlation grouping*, where monitors are grouped because they monitor the same thing, and *focus grouping*, where the parent group monitor is a focus for some operation.

### 8.7.1. Introduction

We'll start by making a few points about monitor grouping in general.

Solutions often contain structural constraints: nodes, restricted domains, fixed assignments, and so on. A solver is expected to respect such constraints, unless its specification explicitly states otherwise. They are part of the solution, and every solver should be able to deal with them. In the same way, a solver may find that some monitors have been deliberately detached before it starts running. For example, all monitors of soft constraints may have been detached, because the caller wants the solver to concentrate on hard constraints. A solver should not change the attachments of monitors to the solution, unless its specification explicitly states otherwise. Its aim is to minimize `KheSolnCost(soln)`, however that is defined by `soln`'s monitor attachments.

There are two ways to exclude a monitor from contributing to the solution cost: by detaching it using `KheMonitorDetachFromSoln`, and by ensuring that there is no path from it to the solution group monitor. The first way is usually best, because it is the efficient way.

Some solvers need specific monitor groupings. The Kempe meet move (Section 10.2.2) is an example: its precondition specifies that a particular group monitor must be present. This is permissible, and as with all preconditions it imposes a requirement on the caller of the operation to ensure that the precondition is satisfied when the operation is called. But such requirements should not prohibit the presence of other group monitors. For example, the implementation of the Kempe meet move operation begins with a tiny search for the group monitor it requires. If other group monitors are present nearby, that is not a problem. If this example is followed, multiple requirements for group monitors will not conflict.

There is a danger that group monitors will multiply, slowing down the solve and confusing its logic. It is best if each function that creates a group monitor takes responsibility for deleting it later, even if this means creating the same group monitors several times over. Timing tests conducted by the author show that adding and deleting the group monitors used by the various solvers in this guide takes an insignificant amount of time.

It is convenient to have standard values for the sub-tags and sub-tag labels of the group monitors created by grouping functions, both correlation and focus. So KHE defines type

```
typedef enum {
  KHE_SUBTAG_SPLIT_EVENTS,            /* "SplitEventsGroupMonitor"           */
  KHE_SUBTAG_DISTRIBUTE_SPLIT_EVENTS, /* "DistributeSplitEventsGroupMonitor" */
  KHE_SUBTAG_ASSIGN_TIME,             /* "AssignTimeGroupMonitor"            */
  KHE_SUBTAG_PREFER_TIMES,            /* "PreferTimesGroupMonitor"           */
  KHE_SUBTAG_SPREAD_EVENTS,           /* "SpreadEventsGroupMonitor"          */
  KHE_SUBTAG_LINK_EVENTS,             /* "LinkEventsGroupMonitor"            */
  KHE_SUBTAG_ORDER_EVENTS,            /* "OrderEventsGroupMonitor"           */
  KHE_SUBTAG_ASSIGN_RESOURCE,         /* "AssignResourceGroupMonitor"        */
  KHE_SUBTAG_PREFER_RESOURCES,        /* "PreferResourcesGroupMonitor"       */
  KHE_SUBTAG_AVOID_SPLIT_ASSIGNMENTS, /* "AvoidSplitAssignmentsGroupMonitor" */
  KHE_SUBTAG_AVOID_CLASHES,           /* "AvoidClashesGroupMonitor"          */
  KHE_SUBTAG_AVOID_UNAVAILABLE_TIMES, /* "AvoidUnavailableTimesGroupMonitor" */
  KHE_SUBTAG_LIMIT_IDLE_TIMES,        /* "LimitIdleTimesGroupMonitor"        */
  KHE_SUBTAG_CLUSTER_BUSY_TIMES,      /* "ClusterBusyTimesGroupMonitor"      */
  KHE_SUBTAG_LIMIT_BUSY_TIMES,        /* "LimitBusyTimesGroupMonitor"        */
  KHE_SUBTAG_LIMIT_WORKLOAD,          /* "LimitWorkloadGroupMonitor"         */
  KHE_SUBTAG_LIMIT_ACTIVE_INTERVALS,  /* "LimitActiveIntervalsGroupMonitor"  */
  KHE_SUBTAG_LIMIT_RESOURCES,         /* "LimitResourcesGroupMonitor"        */
  KHE_SUBTAG_ORDINARY_DEMAND,         /* "OrdinaryDemandGroupMonitor"        */
  KHE_SUBTAG_WORKLOAD_DEMAND,         /* "WorkloadDemandGroupMonitor"        */
  KHE_SUBTAG_KEMPE_DEMAND,            /* "KempeDemandGroupMonitor"           */
  KHE_SUBTAG_NODE_TIME_REPAIR,        /* "NodeTimeRepairGroupMonitor"        */
  KHE_SUBTAG_LAYER_TIME_REPAIR,       /* "LayerTimeRepairGroupMonitor"       */
  KHE_SUBTAG_TASKING,                 /* "TaskingGroupMonitor"               */
  KHE_SUBTAG_ALL_DEMAND               /* "AllDemandGroupMonitor"             */
} KHE_SUBTAG_STANDARD_TYPE;
```

for the sub-tags, and the strings in comments, obtainable by calling

```
char *KheSubTagLabel(KHE_SUBTAG_STANDARD_TYPE sub_tag);
```

for the corresponding sub-tag labels. There is also

```
KHE_SUBTAG_STANDARD_TYPE KheSubTagFromTag(KHE_MONITOR_TAG tag);
```

which returns the appropriate sub-tag for a group monitor whose children have the given `tag`.

When adding and deleting groupings, these public functions are often helpful:

```
void KheMonitorAddSelfOrParent(KHE_MONITOR m, int sub_tag,
  KHE_GROUP_MONITOR gm);
bool KheMonitorHasParent(KHE_MONITOR m, int sub_tag,
  KHE_GROUP_MONITOR *res_gm);
```

Consult the documentation in source file `khe_ss_grouping.c` to find out what they do.

### 8.7.2. Focus groupings

*Focus groupings* are monitor groupings that are not correlation groupings (that is, they do not group monitors which monitor the same thing). They group uncorrelated monitors for particular purposes, such as efficient access to defects.

Focus groupings are often built on correlation groupings: if a monitor that a focus grouping handles is a child of a correlation group monitor, the correlation group monitor goes into the focus grouping, replacing the individual monitors which are its children.

A focus grouping makes one group monitor, called a *focus group monitor*, not many. The focus group monitor is not made a child of the solution object, nor are its children unlinked from any other parents that they may have. So it does not disturb existing calculations in any way; rather, it adds a separate calculation on the side. It follows that a focus grouping can be removed by passing the focus group monitor to `KheGroupMonitorDelete`.

Functions for creating focus groupings appear elsewhere in this guide. They include `KheKempeDemandGroupMonitorMake`, needed by Kempe and ejecting meet moves (Section 10.2.2), and several functions used by ejection chain repair algorithms (Section 13.6.2).

### 8.7.3. Correlation groupings

Two monitors are *correlated* when they monitor the same thing, not necessarily formally, but in reality. For example, if two events are joined by a link events constraint, and one is fixed to the other, then two spread events monitors, one for each event, will be correlated.

A *defect* is a specific point of imperfection in a solution, represented concretely by a monitor with non-zero cost. *Correlated defects* are a problem for some solvers, notably for ejection chains. The cost of each defect separately might not be large enough to end the chain if removed, causing an ejection chain to terminate in failure, whereas if it was clear that there was really only one problem, the chain might be able to repair it and continue.

We solve this problem as follows. Suppose we know that monitors `m1` and `m2` are correlated. We introduce a group monitor `g`, make `m1` and `m2` children of `g`, and ensure that when handling defects we use `g`, not `m1` and `m2`. For example, ejection chains would ensure that focus groupings `start_gm` and `continue_gm` have `g` among their children, not `m1` and `m2`. When `m1` and `m2` signal a defect, the signal is received as the single defect `g`, not as the two defects `m1` and `m2`.

A *correlation grouping* is a set of group monitors, each of which groups some correlated monitors. Installing a correlation grouping should not disturb existing groupings, or change the solution cost. So a function which creates a correlation grouping works as follows.

Monitors not relevant to the grouping remain untouched. Relevant monitors are partitioned into sets $S_i$ of monitors that monitor the same thing, and so could be grouped; and then each $S_i$ is partitioned again into sets $S_{ij}$ of monitors that have the same parents. For each $S_{ij}$ of cardinality at least 2, create a group monitor $g$, make each $s \in S_{ij}$ a child of $g$, make $g$ a child of each of the common parents, and unlink each $s$ from its other parents.

Group monitors that are part of a correlation grouping are given sub-tags that identify them as such. These are used in two ways. Whenever a focus grouping wants to add a certain monitor, it should check first whether that monitor has a parent which is part of a correlation grouping. In that case it should add the parent, not the original monitor, taking care to not add the same parent twice. (Notice that the same result is then produced whether the focus grouping is added first or the correlation grouping is added first.) Second, a function which deletes a correlation grouping can do so by visiting all monitors relevant to the grouping and deleting those parents whose sub-tag identifies them as part of the correlation grouping. The deleting is done by calls to `KheGroupMonitorBypassAndDelete`. Clearly, this will return the monitor structure to its state before the grouping was added.

### 8.7.4. Functions for correlation grouping

One correlation grouping that does everything turns out to be best. These functions provide it:

```
void KheGroupCorrelatedMonitors(KHE_SOLN soln);
void KheUnGroupCorrelatedMonitors(KHE_SOLN soln);
```

`KheGroupCorrelatedMonitors` adds the grouping; `KheUnGroupCorrelatedMonitors` removes it.

The rest of this section explains `KheGroupCorrelatedMonitors` in detail. Here are three points that apply throughout what follows. They won't be mentioned again because it would get too tedious. First, grouping does not care whether a monitor is attached or detached, and so it groups both kinds. It is quite safe to attach and detach monitors while they are grouped. Second, the groupings described below are the larger ones, denoted $S_i$ in Section 8.7.3. The further division into the $S_{ij}$ based on equal parents is always done as well. Third, if an $S_{ij}$ contains just one monitor, adding a group monitor would accomplish nothing useful and is not done.

***Grouping correlated event monitors.*** `KheGroupCorrelatedMonitors` begins by partitioning the events of `soln`'s instance into classes, placing events into the same class when following the fixed assignment paths out of their meets proves that their meets must run at the same times. It then groups event monitors as follows.

*Split events and distribute split events monitors.* For each class, it groups together the split events and distribute split events monitors that monitor the events of that class. It gives sub-tag `KHE_SUBTAG_SPLIT_EVENTS` to any group monitors it creates. There is also a `KHE_SUBTAG_DISTRIBUTE_SPLIT_EVENTS` subtag, but it is not used.

*Assign time monitors.* For each class, it groups the assign time monitors that monitor the events of that class, giving sub-tag `KHE_SUBTAG_ASSIGN_TIME` to any group monitors.

*Prefer times monitors.* Within each class, it groups those prefer times monitors that monitor events of that class whose constraints request the same set of times, giving sub-tag `KHE_SUBTAG_PREFER_TIMES` to any group monitors.

*Spread events monitors.* For each spread events monitor, it finds the set of classes that hold the events it monitors. It groups spread events monitors whose sets of classes are equal, giving sub-tag `KHE_SUBTAG_SPREAD_EVENTS` to any group monitors. Strictly speaking, only monitors whose constraints request the same time groups with the same limits should be grouped, but that check is not currently being made.

*Link events monitors.* It does nothing with these monitors, because they are usually handled structurally, by `KheLayerTreeMake` (Section 9.1).

*Order events monitors.* For each order events monitor, it finds the sequence of two classes that hold the two events it monitors. (If these classes are the same, there is probably a conflict between an order events monitor and a link events monitor. The order events monitor remains ungrouped in that case.) It groups order events monitors whose sequences of classes are equal, giving sub-tag `KHE_SUBTAG_ORDER_EVENTS` to any group monitors. Strictly speaking, only monitors whose constraints request the same event separations should be grouped, but that check is not currently being made.

***Grouping correlated event resource monitors.*** Next, `KheGroupCorrelatedMonitors` partitions the event resources of `soln`'s instance into classes, placing event resources into the same class when following the fixed assignment paths out of their tasks proves that they must be assigned the same resources. It then groups event resource monitors as follows.

*Assign resource monitors.* For each class, it groups the assign resource monitors of that class's event resources, giving sub-tag `KHE_SUBTAG_ASSIGN_RESOURCE` to any group monitors.

*Prefer resources monitors.* Within each class, it groups those prefer resources monitors that monitor the event resources of that class whose constraints request the same set of resources, giving sub-tag `KHE_SUBTAG_PREFER_RESOURCES` to any group monitors.

*Avoid split assignments monitors.* There seems to be no useful correlation grouping of these monitors, so nothing is done with them. They may be handled structurally, in which case they will have provably zero fixed cost and will be already detached.

*Limit resources monitors.* These occur in nurse rostering instances and in practice place limits on the number of nurses (of a particular type, perhaps) assigned to a given shift. There seems to be no useful correlation grouping of such monitors, so nothing is done with them.

***Grouping correlated resource monitors.*** Students who follow the same curriculum have the same timetable. So for each resource type `rt` whose resources are all preassigned, according to `KheResourceTypeDemandIsAllPreassigned(rt)` (Section 3.5.1), the resource monitors of `rt`'s resources are grouped by `KheGroupCorrelatedMonitors` as follows.

*Avoid clashes monitors.* It groups those avoid clashes monitors derived from the same constraint whose resources attend the same events, giving sub-tag `KHE_SUBTAG_AVOID_CLASHES` to any group monitors.

*Avoid unavailable times monitors.* It groups avoid unavailable times monitors derived from the same constraint whose resources attend the same events, giving any group monitors sub-tag `KHE_SUBTAG_AVOID_UNAVAILABLE_TIMES`.

*Limit idle times monitors.* It groups limit idle times monitors derived from the same constraint whose resources attend the same events, giving sub-tag `KHE_SUBTAG_LIMIT_IDLE_TIMES` to any group monitors.

*Cluster busy times monitors.* It groups cluster busy times monitors derived from the same constraint whose resources attend the same events, giving any group monitors sub-tag `KHE_SUBTAG_CLUSTER_BUSY_TIMES`.

*Limit busy times monitors.* IKheGroupCorrelatedMonitorst groups limit busy times monitors derived from the same constraint whose resources attend the same events, giving sub-tag `KHE_SUBTAG_LIMIT_BUSY_TIMES` to any group monitors.

*Limit workload monitors.* It groups limit workload monitors derived from the same constraint whose resources attend the same events, giving sub-tag `KHE_SUBTAG_LIMIT_WORKLOAD` to any group monitors.

*Limit active intervals monitors.* It groups limit active intervals monitors derived from the same constraint whose resources attend the same events, giving sub-tag `KHE_SUBTAG_LIMIT_ACTIVE_INTERVALS` to any group monitors.

Fixed assignments between meets are taken into account when deciding whether two resources attend the same events. As far as resource monitors are concerned, it is when the resource is busy that matters, not which meets it attends.

***Grouping correlated demand monitors.*** `KheGroupCorrelatedMonitors` groups demand monitors as follows. A limit monitor holding them can be created separately (Section 8.7.2).

*Ordinary demand monitors.* For each set of meets such that the fixed assignment paths out of those meets end at the same meet, it groups the demand monitors of those meets' tasks, giving sub-tag `KHE_SUBTAG_ORDINARY_DEMAND` to any group monitors. The reasoning is that the only practical way to repair an ordinary demand defect is to change the assignment of its meet (or some other clashing meet), which will affect all the demand monitors grouped with it here.

Actually the situation is a little more nuanced. What has just been said is correct for time assignment, when moves of unfixed meets are the only repairs available. During resource assignment it is usual to use the global tixel matching only as a limit monitor (or not at all), making other groupings of demand monitors irrelevant. This leaves undetermined the grouping to use if repairs that combine changes to both time and resource assignments are ever tried.

*Workload demand monitors.* These remain ungrouped. Workload demand defects appear only indirectly, as competitors of ordinary demand defects.

## 8.8. Matchings

The KHE platform implements bipartite matching (Chapter 7), but leaves the details of setting it up in practice to the user. This section fills in those details.

### 8.8.1. Introduction

Creating and deleting the matching is a simple as calling

```
void KheMatchingBegin(KHE_SOLN soln, bool integrated);
void KheMatchingEnd(KHE_SOLN soln);
```

`KheMatchingBegin` calls `KheSolnMatchingBegin`, then adds suitable ordinary and workload demand nodes (Section 8.8.2), as well as detaching or otherwise adjusting some monitors to avoid double counting (Section 8.8.3). `KheMatchingEnd` undoes what `KheMatchingBegin` does, including calling `KheSolnMatchingEnd`.

Setting `integrated` to `false` produces a *separate matching*, which means that the matching is kept separate from the regular calculation of solution cost. This is achieved by attaching all demand monitors (so that they are included in the matching graph and kept up to date as the solution changes) but not making them descendants of the solution (so that they don't contribute to its cost). Do-it-yourself solvers (Section 8.3) have item `gts` for this. Solvers can still take the matching into account despite the separateness, by calling `KheSolnMatchingDefectCount` and `KheSolnMatchingDefect` (Section 7.4), which return the unmatched demand nodes.

An example of separate matching is the *resource assignment invariant*, which states that as resource assignment proceeds the number of unmatched demand nodes must not increase. The details don't concern us here (see Section 12.2); our point is that this approach allows the matching to influence the solve while keeping it separate from the regular calculation of solution cost.

Setting parameter `integrated` to `true` produces an *integrated matching*, which means that the cost of the matching is included in the cost of the solution. This is achieved by attaching all demand monitors as for separate matching, but also making them descendants of the solution. Each unmatched demand node contributes a cost to the solution cost; there is no need to call `KheSolnMatchingDefectCount`. Do-it-yourself solvers (Section 8.3) have item `gti` for this.

The main advantage of integrated matching is that it allows any solver to benefit from the matching without accessing it explicitly. But there are problems. One is that the weight of each demand node, considered as a monitor, suddenly matters. Finding a suitable weight can be awkward, especially considering that every demand node has to have the same weight. `KheMatchingBegin` chooses `KheCost(1, 0)` (that is, hard cost 1) as the weight, although the implementation has been written to allow this value to be passed as a parameter if that turns out to be needed in the future. We will refer to this important weight as $W$ from here on.

The other main problem is *double counting*, where the matching reports some defect and some other monitor reports it too, effectively doubling its weight, making it more important than it really is. This may not matter much when $W$ is a hard weight, but anyway `KheMatchingBegin` tries to eliminate double counting, as explained in Section 8.8.3.

Double counting resembles correlated monitors (Section 8.7.3), in involving two monitors that monitor the same thing. But there is a vital difference. Two correlated monitors are part of the official cost of the solution; both have a right to be there. An integrated matching is not part of the official cost, so the aim changes from merely recognizing the correlation to minimizing the distance from the official cost while retaining the advantages of detecting matching defects.

### 8.8.2. Demand nodes

Apart from calling `KheSolnMatchingBegin`, the main task of `KheMatchingBegin` is to create an appropriate set of demand nodes. This section explains how this is done.

`KheMatchingBegin` ignores detached monitors and monitors of weight less than $W$: it behaves as though they are not there at all. This is important for integrated matching, which replaces some of the monitors it does not ignore with demand monitors of weight $W$. It is less important for separate matching, but even there we do not want to place some minor monitor with soft weight 1 on the same footing as an important monitor with weight *W*.

An ordinary demand node represents a demand for one resource at one offset of one task. A first thought is that `KheMatchingBegin` should create one of these for each offset of each task. So it does, but with three exceptions. The first two never occur in practice; the third does.

First, if some resource does not have an attached avoid clashes monitor with weight at least *W*, then part of the rationale for the matching is absent, the part that says that each supply node may match with at most one demand node. So `KheMatchingBegin` omits demand nodes for tasks of the same resource type as any resource that does not have such an avoid clashes monitor.

Second, if some meet does not have either a preassigned time or an attached assign time monitor with weight at least *W*, then that meet does not have to be assigned a time at all, and by omitting to assign a time one can avoid all clashes involving that meet. So `KheMatchingBegin` omits all demand nodes for tasks from meets of this kind.

Third, if some task does not have either a preassigned resource or an attached assign resource monitor of weight at least *W*, it does not have to be assigned a resource at all, and so `KheMatchingBegin` omits its demand nodes. This case arises in nurse rostering, when a shift needs at least *a* and at most *b* nurses. The last $b - a$ tasks have no assign resource monitors. Or if assignment is desirable but not essential, there may be attached assign resource monitors of weight less than *W*, which are ignored as usual.

In addition to ordinary demand nodes, `KheMatchingBegin` adds workload demand nodes, used to take account of avoid unavailable times monitors, limit busy times monitors, and limit workload monitors, collectively called *workload demand monitors* here. Again, only attached monitors of weight *W* or more are noticed.

For example, suppose that the cycle has 40 times, and teacher *Smith* has a workload limit of 30 times and is unavailable at time *Mon1*. Then ten workload demand nodes are wanted, one demanding supply tixel (*Smith*, *Mon1*), and the other nine demanding *Smith* at one unconstrained time. In this way, 10 of Smith's 40 supply nodes have to match with workload demand nodes, leaving 30 available for matching with ordinary demand nodes; and one of the workload demand nodes also has to match with *Mon1*, leaving *Smith* unavailable for regular work at that time.

It is important to include workload demand nodes, since otherwise the problems reported by the matching will be unrealistically few. They are the same for all matching types, and the same for separate and integrated matching, although when setting up for integrated matching we also have to avoid double counting, as discussed below (Section 8.8.3).

For the purposes of matchings only, a *workload requirement* is a requirement imposed on a resource saying that it should be occupied attending meets for at most a given number of the times of some time group. We can represent a workload requirement concisely by the pair $n\ T$, where the resource is implicit, $n$ is the number of times, and $T$ is the time group. For example, if `r` is unavailable at time `Mon1`, this would give rise to workload requirement $0\ \{Mon1\}$.

To make the workload demand nodes for a given resource *r*, `KheMatchingBegin` first makes a set of workload requirements derived from the avoid unavailable times, limit busy times, and limit workload monitors of `r` (attached and of weight at least *W*, as usual). It places these requirements into a tree structure that we will present shortly, and then generates `r`'s workload demand nodes in the course of a postorder traversal of this tree.

If m is an avoid unavailable times monitor, or a limit busy times monitor whose Maximum attribute is 0, then for each time $t$ in m's constraint's domain, KheMatchingBegin generates workload requirement 0 $\{t\}$. If m is a limit busy times monitor with Maximum greater than 0, then for each time group $T$ in m's constraint, KheMatchingBegin generates workload requirement $n$ $T$, where $n$ is Maximum. The Minimum attribute is ignored.

A limit workload monitor is like a limit busy times monitor whose time group contains all the times of the cycle, so one workload requirement is generated, with the entire cycle as time group. The number of this requirement requires careful calculation, involving the workloads of all events. The remainder of this section explains this calculation.

Let $k$ be the integer eventually given to the workload requirement. Initialize $k$ to the Maximum attribute of the limit workload constraint. For each event resource $er$, let $d(er)$ be its duration and $w(er)$ be its workload. The basic idea is that if r is assigned to $er$, then $d(er) - w(er)$ should be added to $k$. For example, a resource with workload limit 30 that is assigned to an event resource with duration 3 and workload 2 needs a workload requirement of 31, not 30. And if r is assigned to an event with duration 6 but workload 12, then $k$ needs to be decreased by 6.

In some cases, preassignments or domain restrictions make it certain that r will be assigned to some event, and in those cases the adjustment can be done safely in advance. For example, if every staff member attends a weekly meeting with duration 1 and workload 0, then their workload requirements can all be increased by 1 to compensate. Similarly, if r will definitely not be assigned to some event, then the event's duration and workload have no effect on r.

The residual problem cases are those event resources $er$ whose workload and duration differ, which r may be assigned to but not necessarily. In these cases, an inexact model is used which preserves the guarantee that the number of unmatched nodes is a lower bound on the final number, but the number is weaker (that is, smaller) than the ideal.

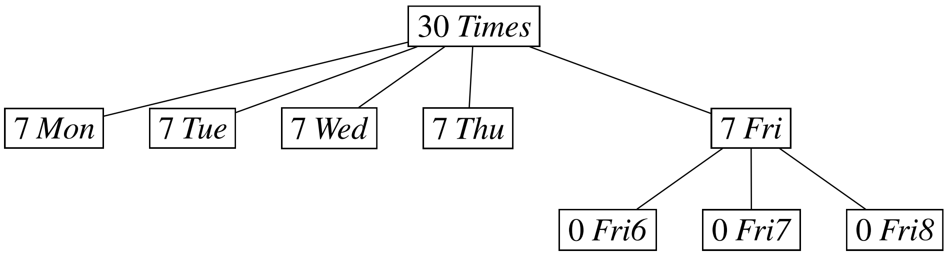
If $w(er) > d(er)$, then $er$ is ignored. This case can only make the problem harder, so ignoring it means that the number returned will be smaller than the ideal. If $w(er) < d(er)$, then $d(er) - w(er)$ is added to $k$, just as though r was assigned to $er$. If r is ultimately assigned to $er$, then this will be exact; if it is not, then again it will weaken the bound, by overestimating r's available workload.

These tests are actually applied to clusters of events known to be running simultaneously, because of required link events constraints or preassignments and other time domain restrictions. Each resource can be assigned to at most one of the event resources of the events of a cluster, so only one of the events, the one whose modelling is least exact, needs to be taken account of.

Once a complete set of workload requirements for resource r has been assembled, the next step is to convert these requirements into workload demand nodes. The following explanation of how this is done is adapted from [9].

When converting workload requirements into workload demand nodes, the relationships between the requirements' time groups affect the outcome. In general, an exact conversion seems to be possible only when these time groups satisfy the *subset tree condition*: each pair of time groups is either disjoint, or else one is a subset of the other.

For example, suppose the cycle has five days of eight times each, and resource $r$ is required to be occupied for at most thirty times altogether and at most seven on any one day, and to be unavailable at times *Fri6*, *Fri7*, and *Fri8*. These requirements form a tree (in general, a forest):



A postorder traversal of this tree may be used to deduce that workload demand nodes for *r* are needed for one *Mon* time, one *Tue* time, one *Wed* time, one *Thu* time, one *Fri6* time, one *Fri7* time, one *Fri8* time, and three arbitrary times. In general, each tree node contributes a number of demand nodes equal to the size of its time group minus its number minus the number of demand nodes contributed by its proper descendants, or none if this number is negative.

The tree is built by inserting the workload requirements in decreasing order of the weight of the monitors that led to them, ignoring requirements that fail the subset tree condition. For example, a failure would occur if, in addition to the above requirements, there were limits on the number of morning and afternoon times. The constraints which give rise to such requirements are still monitored by other monitors, but their omission from the matching causes it to report fewer unmatchable nodes than the ideal. Fortunately, such overlapping requirements do not seem to occur in practice, at least, not as hard constraints.

### 8.8.3. Avoiding double counting

When an integrated matching is installed, several cases of double counting arise which can bias a solver. Since it is important to preserve the matching, so that problems such as six simultaneous Science classes and only five Science laboratories are detected, `KheMatchingBegin` solves this problem by detaching the non-demand monitors that double count with the demand monitors, as far as possible. Here now are the details.

The work described here is only done for integrated matchings. As usual, only attached monitors of weight at least *W* are taken into account. They double count with demand monitors of weight *W*, so detaching them produces solution costs which are roughly equal to the regular costs. When *W* is `Khe(1, 0)` (the value used at present) and all monitors with hard costs have cost `Khe(1, 0)` (which is usual, many instances choosing to not distinguish different degrees of hardness), the resulting costs are exactly equal to the regular costs, except that they include the advance warnings that matchings give—a very desirable state of affairs.

*Attached avoid clashes monitors of weight at least W.* We simply detach these monitors, since each clash also appears as one unmatched demand node.

*Attached avoid unavailable times monitors of weight at least W*. Again we detach these monitors, since each violation appears as one unmatched demand node.

*Attached limit busy times monitors that give rise to workload demand monitors.* As far as upper limits are concerned we could again detach, but we have to consider lower limits as well.

The obvious solution is to break each limit busy times monitor into two, an underload monitor and an overload monitor, and detach the overload monitor but not the underload monitor. KHE expresses this idea in a different way, chosen because it also solves the problem presented by limit workload monitors, to be discussed in their turn.

Limit busy times monitors have two attributes, `Minimum` and `Maximum`, such that less than `Minimum` busy times is an underload, and more than `Maximum` is an overload. Add a third attribute, `Ceiling`, such that `Ceiling >= Maximum`, and specify that, with higher priority than the usual rule, when the number of busy times exceeds `Ceiling` the deviation is 0.

Function `KheLimitBusyTimesMonitorSetCeiling` (Section 6.7.5) may be called to set the ceiling. Setting it to `INT_MAX` (the default value) produces the usual rule. Setting it to `Maximum` is equivalent to detaching overload monitoring.

*Attached limit workload monitors that give rise to workload demand monitors.* Limit workload monitors are similar to limit busy times monitors whose set of times is the entire cycle. However, the demand monitors derived from a limit workload monitor do not necessarily model even the upper limit exactly. This problem can be solved as follows.

Consider a resource with a limit workload monitor and some workload demand monitors derived from it, and suppose that all of these monitors are attached. As the resource's workload increases, it crosses from a 'white region' of zero cost into a 'grey region' where the limit workload monitor has non-zero cost but the workload demand monitors do not, and then into a 'black region' where both the limit workload monitor and the workload demand monitors have non-zero cost. This black region is the problem.

The problem is solved by adding a `Ceiling` attribute to limit workload monitors, as for limit busy times monitors. Function `KheLimitWorkloadMonitorSetCeiling` (Section 6.7.6) sets the ceiling. As before, the default value is `INT_MAX`. The appropriate alternative value is not `Maximum`, but rather a value which marks the start of the black region, so that the limit workload monitor's cost drops to zero as the workload crosses from the grey region to the black region. In this way, all workload overloads are reported, but by only one kind of monitor at any one time.

There is one anomaly in this arrangement: a repair that reduces workload from the black region to the grey region does not always decrease cost. This is a pity but it is very much a second-order problem, given that the costs involved are all hard costs, so that in practice repairs are wanted that reduce them to zero. What actually happens is that one repair puts a resource above the white zone, and this stimulates a choice of next repair which returns it to the white zone. Repairs which move between the grey and black zones are possible but are not likely, so it does not matter much if their handling is imperfect.

The appropriate value for `Ceiling` is the number of times in the cycle minus the total number of workload demand monitors for the resource in question, regardless of their origin. When the resource's workload exceeds this value, there will be at least one demand defect, and it is time for the limit workload monitor to bow out.

## 8.9. Generating files of tables and graphs

KHE offers a module for generating files containing tables and graphs. Any number of files may be generated simultaneously, even in parallel, although an individual file cannot be generated in parallel. One file may contain any number of tables and graphs, and these can be generated simultaneously, although not in parallel.

To begin and end a file, call

```
KHE_FILE KheFileBegin(char *file_name, KHE_FILE_FORMAT fmt);
void KheFileEnd(KHE_FILE kf);
```

This writes a file called `file_name` in sub-directory `stats` (which the user must have created previously) of the current directory. The file is opened by `KheFileBegin` and closed by `KheFileEnd`. `KheFileEnd` also reclaims all memory (taken from a specially created arena) used by all tables and graphs of that file. Three file formats are supported:

```
typedef enum {
  KHE_FILE_PLAIN,
  KHE_FILE_LOUT,
  KHE_FILE_LOUT_STANDALONE,
  KHE_FILE_LATEX
} KHE_FILE_FORMAT;
```

These represent plain text, Lout, standalone Lout (i.e. ready for converstion to Encapsulated PostScript) and LaTeX. Only the two Lout values support graphs. To generate the actual tables and graphs, see the following subsections.

## 8.9.1. Tables

To generate tables, make matching pairs of calls to the following functions in between the calls to `KheFileBegin` and `KheFileEnd`:

```
KHE_TABLE KheTableBegin(KHE_FILE kf, int col_width, char *corner,
  bool with_average_row, bool with_total_row, bool highlight_cost_minima,
  bool highlight_time_minima, bool highlight_int_minima);
void KheTableEnd(KHE_TABLE kt);
```

The table is begun by `KheTableBegin`, and finished, including being written out to file `kf`, by `KheTableEnd`. Where the file format permits, a label will be associated with the table: the file name for the first table, the file name followed by an underscore and 2 for the second table, and so on. The value of the table is created in between these two calls, by calling functions to be presented shortly. Because the entire table is saved in memory until `KheTableEnd` is called, these other calls may occur in any order. In particular it is equally acceptable to generate a table row by row or column by column.

Parameter `col_width` determines the width in characters of each column when the format is `KHE_FILE_PLAIN`; it is ignored by the other formats. Parameter `corner` is printed in the top left-hand corner of the table. It must be non-`NULL`, but it can be the empty string.

Each entry in the table has a type, which may be either *string*, *cost*, *time* (really just an arbitrary `float`), or *int*. If `with_average_row` is `true`, the table ends with an extra row. Each entry in this row contains the average of the non-blank, non-string entries above it, if they all have the same type; otherwise the entry is blank. If `with_total_row` is `true`, the effect is the same except that totals are printed, not averages.

If `highlight_cost_minima` is `true`, the minimum values of type *cost* in each row appear in bold font, or marked by an asterisk in plain text. Parameters `highlight_time_minima` and `highlight_int_minima` are the same except that they highlight values of type *time* or *int*.

A caption can be added by calling

```
void KheTableCaptionAdd(KHE_TABLE kt, char *fmt, ...);
```

at any time between `KheTableBegin` and `KheTableEnd`, as often as desired. This does what `printf` would do with the arguments after `file_name`. The results of all calls are saved and printed as a caption by `KheTableEnd`.

In any given table, each row except the first (header) row must be declared, by calling

```
void KheTableRowAdd(KHE_TABLE kt, char *row_label, bool rule_below);
```

The rows appear in the order of the calls. Parameter `row_label` both identifies the row and appears in the first (header) column of the table. If `rule_below` is `true`, the row will have a rule below it. The header row always has a rule below it, and there is always a rule below the last row (not counting any average or total row).

In the same way, non-header columns are declared, in order, by calls to

```
void KheTableColAdd(KHE_TABLE kt, char *col_label, bool rule_after);
```

where `col_label` both identifies the column and appears in the first (header) row of the table, and setting `rule_after` to `true` causes a rule to be printed after the column.

To add an entry to the table, call any one of these functions:

```
void KheTableEntryAddString(KHE_TABLE kt, char *row_label,
  char *col_label, char *str);
void KheTableEntryAddCost(KHE_TABLE kt, char *row_label,
  char *col_label, KHE_COST cost);
void KheTableEntryAddTime(KHE_TABLE kt, char *row_label,
  char *col_label, float time);
void KheTableEntryAddInt(KHE_TABLE kt, char *row_label,
  char *col_label, int val);
```

These add an entry to `kt` at row `row_label` and column `col_label`, aborting if these are unknown or an entry has already been added there. If no entry is ever added at some position, the table will be blank there. The entry's format depends on the call. For example,

```
KheTableEntryAddCost(file_name, row_label, col_label, KheSolnCost(soln));
```

adds a solution cost to the table which will be formatted in the standard way.

Strings passed to these functions are copied where required, so mutating strings are not a concern. There is no locking, so calls which create and add to tables should be single-threaded.

### 8.9.2. Graphs

To generate graphs in Lout format, make matching pairs of calls to the following functions in between the calls to `KheFileBegin` and `KheFileEnd`:

```
KHE_GRAPH KheGraphBegin(KHE_FILE kf);
void KheGraphEnd(KHE_GRAPH kg);
```

To set options which control the overall appearance of the graph, call

```
void KheGraphSetWidth(KHE_GRAPH kg, float width);
void KheGraphSetHeight(KHE_GRAPH kg, float height);
void KheGraphSetXMax(KHE_GRAPH kg, float xmax);
void KheGraphSetYMax(KHE_GRAPH kg, float ymax);
void KheGraphSetAboveCaption(KHE_GRAPH kg, char *val);
void KheGraphSetBelowCaption(KHE_GRAPH kg, char *val);
void KheGraphSetLeftCaptionAndGap(KHE_GRAPH kg, char *val, char *gap);
void KheGraphSetRightCaptionAndGap(KHE_GRAPH kg, char *val, char *gap);
```

These determine the width and height of the graph (in centimetres), the maximum x and y values, and the small captions above, below, to the left of, and to the right of the graph. If calls to these functions are not made, the options remain unspecified, causing Lout's graph package to substitute default values for them in its usual way. The caption values must be valid Lout source.

`KheGraphSetLeftCaptionAndGap` and `KheGraphSetRightCaptionAndGap` have the extra `gap` parameter. This controls the gap between the caption and the graph. For example,

```
KheGraphSetLeftCaptionAndGap(kg, "Caption", "0c");
```

produces the minimum gap (0 cm), but a larger value is usually needed, to avoid unsightly overstriking. The value of `gap` can also be `NULL`, in which case Lout's default value is used.

There is also

```
void KheGraphSetKeyLabel(KHE_GRAPH kg, char *val);
```

which sets the 'key label' of the graph. This is the first line of the graph's key, described below. Omitting to call this function is fine; it just means that this first line is omitted.

Any number of *datasets* may be displayed on one graph; each dataset is a sequence of points. Often there is just one dataset. To create a dataset, call

```
KHE_DATASET KheDataSetAdd(KHE_GRAPH kg, KHE_DATASET_POINTS_TYPE points_type,
  KHE_DATATSET_PAIRS_TYPE pairs_type, char *label);
```

where `points_type` has type

```
typedef enum {
  KHE_DATASET_POINTS_NONE,
  KHE_DATASET_POINTS_CROSS,
  KHE_DATASET_POINTS_SQUARE,
  KHE_DATASET_POINTS_DIAMOND,
  KHE_DATASET_POINTS_CIRCLE,
  KHE_DATASET_POINTS_TRIANGLE,
  KHE_DATASET_POINTS_PLUS,
  KHE_DATASET_POINTS_FILLED_SQUARE,
  KHE_DATASET_POINTS_FILLED_DIAMOND,
  KHE_DATASET_POINTS_FILLED_CIRCLE,
  KHE_DATASET_POINTS_FILLED_TRIANGLE
} KHE_DATASET_POINTS_TYPE;
```

and says what to print at each data point (nothing, or a cross, etc.), and `pairs_type` has type

```
typedef enum {
  KHE_DATATSET_PAIRS_NONE,
  KHE_DATATSET_PAIRS_SOLID,
  KHE_DATATSET_PAIRS_DASHED,
  KHE_DATATSET_PAIRS_DOTTED,
  KHE_DATATSET_PAIRS_DOT_DASHED,
  KHE_DATATSET_PAIRS_DOT_DOT_DASHED,
  KHE_DATATSET_PAIRS_DOT_DOT_DOT_DASHED,
  KHE_DATASET_PAIRS_YHISTO,
  KHE_DATASET_PAIRS_SURFACE_YHISTO,
  KHE_DATASET_PAIRS_FILLED_YHISTO,
  KHE_DATASET_PAIRS_XHISTO,
  KHE_DATASET_PAIRS_SURFACE_XHISTO,
  KHE_DATASET_PAIRS_FILLED_XHISTO
} KHE_DATASET_PAIRS_TYPE;
```

and says what connects each successive pair of points (nothing, a solid line, a dashed line, a histogram, etc.). These are converted into values of the `points` and `pairs` options of the `@Data` symbol of Lout's Graph package. The Lout User's Guide has examples of what is produced.

When the `label` parameter of `KheDataSetAdd` is non-`NULL`, one line is added to the *key* of the graph, a small area in the top left-hand corner which indicates what each data set represents. The line shows two points and what they are separated by, followed by the label. The first line of the key may be set separately, by calling `KheGraphSetKeyLabel` as described above.

Function

```
void KhePointAdd(KHE_DATASET kd, float x, float y);
```

adds a point to a dataset. The points are generated in the order received, so in practice, successive calls to `KhePointAdd` on the same dataset should have increasing `x` values.

Several datasets can be built simultaneously. This can be useful for recording several quantities as a solver proceeds.

## 8.10. Exponential backoff

One strategy for making solvers faster is to do a lot of what is useful, and not much of what isn't useful. When something is always useful, it is best to simply do it. When something might be useful but wastes a lot of time when it isn't, it is best to try it, observe whether it is useful, and do more or less of it accordingly. Solvers that do this are said to be *adaptive*.

For example, suppose there is a choice of two or more methods of doing something. In that case, information can be kept about how successful each method has been recently, and the choice can be weighted towards recently successful methods.

However, this section is concerned with a different situation, involving just one method. Suppose there is a sequence of *opportunities* to apply this method, and that as each opportunity arrives, the solver can choose to apply the method or not. Typically, the method will be a repair method: repair is optional. If the solver *accepts* the opportunity, the method is then run and either *succeeds* (does something useful) or *fails* (does nothing useful). Otherwise, the solver *declines* the opportunity. So opportunities are classified as successful, failed, or declined.

*Exponential backoff* from computer network implementation is a form of adaptation suited to this situation. It works as follows. If the solver applies the method and it is successful, then it forgets all history and will accept the next opportunity. But if the solver applies the method and it fails, then it remembers the total number of failed opportunities $F$ (including this one) since the last successful opportunity, and does not accept another opportunity until after it has declined $2^{F-1}$ opportunities. Declined opportunities do not count as failures.

Here are some examples. Each character is one opportunity; `S` is a successful opportunity (or the start of the sequence), `F` is a failed one, and `.` is a declined one. Each successful opportunity makes a fresh start, so the examples all begin with `S` and contain only `F` and `.` thereafter:

```
S
SF.
SF.F..
SF.F..F....
SF.F..F....F........
```

and so on. Every complete trace of exponential backoff can be broken at each `S` into sub-traces like these. Methods that always succeed are tried at every opportunity. Methods that always fail are tried only about $\log_2 n$ times, where $n$ is the total number of opportunities.

Other rules for which opportunities to accept could be used, rather than waiting until $2^{F-1}$ opportunities have been declined. For example, every opportunity could be accepted, which amounts to having no backoff at all. The principles are the same, only the rule changes.

KHE offers three operations which together implement exponential backoff:

```
KHE_BACKOFF KheBackoffBegin(KHE_BACKOFF_TYPE backoff_type, HA_ARENA a);
bool KheBackoffAcceptOpportunity(KHE_BACKOFF bk);
void KheBackoffResult(KHE_BACKOFF bk, bool success);
```

`KheBackoffBegin` creates a new backoff object in arena `a`, passing a `backoff_type` value of type

```
typedef enum {
  KHE_BACKOFF_NONE,
  KHE_BACKOFF_EXPONENTIAL
} KHE_BACKOFF_TYPE;
```

which determines which rule is used: none or exponential. `KheBackoffAcceptOpportunity` is called when an opportunity arises, and returns `true` if that opportunity should be accepted. In that case, the next call must be to `KheBackoffResult`, reporting whether or not the method was successful. As usual, the backoff object's memory is reclaimed when the arena is deleted.

Suppose that the program pattern without exponential backoff is

```
while( ... )
{
  ...
  if( opportunity_has_arisen )
    success = try_repair_method(soln);
  ...
}
```

Then the modified pattern for including exponential backoff is

```
bk = KheBackoffBegin(KHE_BACKOFF_EXPONENTIAL);
while( ... )
{
  ...
  if( opportunity_has_arisen && KheBackoffAcceptOpportunity(bk) )
  {
    success = try_repair_method(soln);
    KheBackoffResult(bk, success);
  }
  ...
}
```

Each successful `KheBackoffAcceptOpportunity` is followed by a call to `KheBackoffResult`.

All backoff objects hold a few statistics, kept only for printing by `KheBackoffDebug` below, and a boolean flag which is `true` if the next call must be to `KheBackoffResult`. When exponential backoff is requested, a backoff object also maintains two integers, $C$ and $M$. $C$ is the number of declines since the last accept (or since the backoff object was created). $M$ is the maximum number of opprtunities that may be declined, defined by

$$M = \begin{cases} 0 & \text{if } F = 0 \\ 2^{F-1} & \text{if } F \geq 1 \end{cases}$$

where $F$ is the number of failures since the last success (or since the backoff object was created). The next call to `KheBackoffAcceptOpportunity` will return `true` if $C \geq M$. The implementation will not increase $M$ if that would cause an overflow. Overflow is very unlikely, since an enormous number of opportunities would have to occur first.

Function

```
char *KheBackoffShowNextDecision(KHE_BACKOFF bk);
```

returns "ACCEPT" when the next call to `KheBackoffAcceptOpportunity` will return `true`, and "DECLINE" when it will return `false`. There is also

```
void KheBackoffDebug(KHE_BACKOFF bk, int verbosity, int indent, FILE *fp);
```

Verbosity 1 prints the current state, including a '`!`' when the flag is set, on one line. Verbosity 2 prints some statistics: the number of opportunities so far, and how many are successful, failed, and declined, in a multi-line format. There is also

```
void KheBackoffTest(FILE *fp);
```

which may be called to test this module.

## 8.11. Thread-safe random numbers

Incredibly, C has no standard thread-safe way to generate random numbers. There is `rand_r`, but that is obsolete; there is `random_r`, but that is a nonstandard glibc extension; there is `drand48`, but that is not thread-safe; and there is `drand48_r`, but that is a GNU extension and is not portable. This is according to the manual entries on the author's machine.

So KHE offers the `KHE_RANDOM_GENERATOR` type, representing a random number generator. It does not use heap memory. To declare a random number generator, do this:

```
KHE_RANDOM_GENERATOR rgen;
```

If this is local to some function, then each call on that function (including calls in different threads) will have its own independent generator. To initialize it, passing a seed, call

```
void KheRandomGeneratorInit(KHE_RANDOM_GENERATOR *rgen, uint32_t seed);
```

It would make sense to pass a solution's diversifier as the seed:

```
KheRandomGeneratorInit(&rgen, KheSolnDiversifier(soln));
```

so that different solutions get different random numbers. To obtain one random number, call

```
uint32_t KheRandomGeneratorNext(KHE_RANDOM_GENERATOR *rgen);
```

It returns a fairly random unsigned 32-bit integer, good enough for solvers, but not cryptography. It may be more convenient to call

```
int KheRandomGeneratorNextRange(KHE_RANDOM_GENERATOR *rgen,
  int first, int last);
```

This uses a call to `KheRandomGeneratorNext` to find a random integer between `first` and `last` inclusive. There is also

```
bool KheRandomGeneratorNextBool(KHE_RANDOM_GENERATOR *rgen);
```

which uses a call to `KheRandomGeneratorNext` to find a random Boolean value. Finally,

```
void KheRandomGeneratorTest(uint32_t seed, int count,
  int first, int last, int indent, FILE *fp);
```

initializes a random number generator using `seed`, then prints out `count` random numbers in the range `first` to `last` inclusive, onto file `fp`, indented `indent` spaces.

## 8.12. Intervals

Include file `khe_solvers.h` includes this type definition:

```
typedef struct khe_interval_rec {
  int first;
  int last;
} KHE_INTERVAL;
```

It represents an *interval*, that is, a sequence of consecutive integers. The first member is `first`, and the last is `last`.

The usual use for intervals is to represent a set of consecutive time groups from the common frame. Just their indexes are stored; the frame itself is held elsewhere. However other uses for intervals are quite acceptable.

The condition `last >= first - 1` must hold. Values of `first` and `last` that violate it cause an abort. But `last == first - 1` is acceptable and denotes an empty interval.

When an interval is empty, `last` is determined by `first`, but `first` is free to take on any value. This raises the question of whether two empty intervals with different values for `first` are equal or not. The answer to this question varies with the operation, as follows.

Most of the time, empty intervals are treated like empty sets of integers. For example, all empty intervals are considered to be equal, even when their `first` attributes differ. The few exceptions are flagged by the statement 'empty intervals are not empty sets'. They mainly handle cases where we want to start with an empty interval and grow it, and `first` is significant because it denotes the point that the interval grows from.

Function

```
KHE_INTERVAL KheIntervalMake(int first, int last);
```

makes and returns a new interval with the given `first` and `last` attributes. It checks the condition `last >= first - 1` and aborts if it does not hold. No arena is needed, because `KHE_INTERVAL` is not a pointer type. Functions

```
int KheIntervalFirst(KHE_INTERVAL in);
int KheIntervalLast(KHE_INTERVAL in);
```

return the `first` and `last` attributes of `in`. Two empty intervals can return different values for `first` and also for `last`. These two functions are implemented by macros in `khe_solvers.h`.

Function

```
int KheIntervalLength(KHE_INTERVAL in);
```

returns the length of `in`, possibly 0 when the interval is empty. Function

```
bool KheIntervalEmpty(KHE_INTERVAL in);
```

returns `true` when `in` is empty. Function

```
bool KheIntervalContains(KHE_INTERVAL in, int index);
```

returns `true` when `in` contains `index`. Functions

```
bool KheIntervalEqual(KHE_INTERVAL in1, KHE_INTERVAL in2);
bool KheIntervalSubset(KHE_INTERVAL in1, KHE_INTERVAL in2);
bool KheIntervalDisjoint(KHE_INTERVAL in1, KHE_INTERVAL in2);
```

return `true` when `in1` and `in2` are equal, when `in1` is a subset of `in2`, and when `in1` and `in2` are disjoint (have an empty intersection). Functions

```
KHE_INTERVAL KheIntervalUnion(KHE_INTERVAL in1, KHE_INTERVAL in2);
KHE_INTERVAL KheIntervalIntersection(KHE_INTERVAL in1, KHE_INTERVAL in2);
```

return the union and intersection of `in1` and `in2`. If either of `in1` and `in2` is empty, the union is the other; otherwise the union includes all integers from `min(in1.first, in2.first)` to `max(in1.last, in2.last)` inclusive. Some of these integers may not lie in either of `in1` and `in2`, in which case the result is not a true set union. If `in1` and `in2` are disjoint, the intersection is empty; otherwise the intersection includes all integers from `max(in1.first, in2.first)` to `min(in1.last, in2.last)` inclusive. In either case the result is a true set intersection.

Function

```
int KheIntervalTypedCmp(KHE_INTERVAL in1, KHE_INTERVAL in2);
```

compares `in1` and `in2`, returning `-1` if `in1` is lexicographically less than `in2`, `1` if it is lexicographically greater, and 0 if `in1` and `in2` are equal. Empty intervals are not empty sets. Function

```
int KheIntervalCmp(const void *t1, const void *t2);
```

is an untyped version of `KheIntervalTypedCmp`, suited to `qsort` and `HaArraySort`. Finally,

```
char *KheIntervalShow(KHE_INTERVAL in, KHE_FRAME frame)
```

returns a display of `in` in static memory. If `frame != NULL`, then `in` is considered to be a set of indexes into `frame`, and time group Ids are printed rather than integers.

Finally, here are some functions that return intervals of interest to solvers:

```
KHE_INTERVAL KheTimeInterval(KHE_TIME t, KHE_FRAME frame);
KHE_INTERVAL KheTimeGroupInterval(KHE_TIME_GROUP tg, KHE_FRAME frame);
KHE_INTERVAL KheTaskInterval(KHE_TASK task, KHE_FRAME frame);
KHE_INTERVAL KheTaskSetInterval(KHE_TASK_SET ts, KHE_FRAME frame);
```

`KheTimeInterval` returns the interval of `frame` that covers `t`. This will always have length 1. `KheTimeGroupInterval` returns the interval of `frame` that covers `tg`. If `tg` is empty, this will be an empty interval. `KheTaskInterval` returns the interval of `frame` that covers `task`, including tasks assigned directly or indirectly to `task`. If none of these tasks lie in a meet with an assigned time, this will be an empty interval. `KheTaskSetInterval` returns the interval of `frame` that covers the tasks of `ts`. This will be empty if `ts` is empty or none of its tasks lie in a meet with an assigned time.

# Chapter 9. Time-Structural Solvers

This chapter documents the solvers packaged with KHE that modify the time structure of a solution: split and merge its meets, add nodes and layers, and so on. These solvers may alter time and resource assignments, but they only do so occasionally and incidentally to their structural work.

## 9.1. Layer tree construction

KHE offers a solver for building a layer tree holding the meets of a given solution:

```
KHE_NODE KheLayerTreeMake(KHE_SOLN soln);
```

The root node of the tree, holding the cycle meets, is returned. The function has no special access to data behind the scenes. Instead, it works by calling basic operations and helper functions:

- It calls `KheMeetSplit` to satisfy split events constraints and other influences on the number and duration of meets, as far as possible. It is usual to call `KheLayerTreeMake` when each event is represented in `soln` by a single meet of the full duration (that is, after `KheSolnMake` and `KheSolnMakeCompleteRepresentation`), but some meets may be already split. In any case, `KheLayerTreeMake` does not create, delete, or merge meets.

- It calls `KheMeetBoundMake` with a `NULL` meet bound group to set the time domains of meets to satisfy preassigned times, prefer times constraints, and other influences on time domains, as far as possible. For each meet, one call to `KheMeetBoundMake` is made for each possible duration. It is usual to call `KheLayerTreeMake` at a moment when the time domains of the meets are not restricted by meet bounds, but some meets may already have bounds. In any case, `KheLayerTreeMake` only adds bounds, never removes them, so it either leaves a domain unchanged, or reduces it to a subset of its initial value.

- It calls `KheMeetAssign` in trivial cases where there is no doubt that the assignments will be final. Precisely, if there are two events of equal duration linked by a link events constraint and split into meets of equal durations, and the algorithm places one in a parent node and the other in a child of that parent, then, provided the child node itself has no children (which would render the case non-trivial), the meets of the child node will be assigned to meets of the parent node, and the child node will be deleted in accordance with the convention given in Chapter 10, that meets whose assignments will never change should not lie in nodes.

- It calls `KheMeetAssignFix` to fix all the assignments it makes (as defined immediately above). These can be unfixed afterwards if desired.

- It calls `KheNodeMake` and `KheNodeAddMeet` to ensure that for each event there is one node holding the meets of that event, unless these meets receive the trivial assignments just described. There is also a node (the root node returned by `KheLayerTreeMake`, also accessible as `KheSolnNode(soln, 0)`) holding the cycle meets. Any other meets (usually none) are not placed into nodes. `KheLayerTreeMake` requires `soln` to contain no nodes initially.

- It calls `KheNodeAddParent` to reflect link events constraints (even between events whose durations differ), as far as possible, and the need to ultimately assign every meet to a cycle meet. When `KheLayerTreeMake` returns, every node is a descendant of the root node.

- Some instances contain events which have already been split, with the fragments presented as distinct events. It is best if the nodes holding the meets derived from these fragments are merged. So for each pair of distinct events which appear to be part of one course because they share a spread events constraint or avoid split assignments constraint, if certain other conditions (Section 9.1.5) are satisfied, the nodes holding the meets of those two events are merged by a call to `KheNodeMerge`.

These elements interact in ways that make most of them impossible to separate. For example, the splitting of an event into meets needs to be influenced not just by the event's own split events constraints and distribute split events constraints, but also by the constraints of the events that it is linked to by link events constraints.

Logically, order events constraints should also affect the construction of layer trees. In the version of KHE documented here they are not consulted, but this will change.

Although `KheLayerTreeMake` does not call `KheLayerMake`, resource layers (sets of events that share a common preassigned resource which has a hard avoid clashes constraint) strongly influence its behaviour. It ensures that the events of each layer are split into meets which can be packed into the cycle meets without overlapping in time, except in the unlikely case where the total duration of the events of the layer exceeds the total number of times in the cycle.

For each `meet` with a pre-existing assignment to some `target_meet`, `KheLayerTreeMake` tries to place `meet` into a child node of `target_meet`'s node. In exceptional circumstances, this may not be possible, and then the pre-existing assignment is removed by `KheLayerTreeMake`. Suppose there is an event with two meets, both assigned to other meets. If those two other meets are both derived from the same event, or if they are both cycle meets, then all is well; but if not, one of the original meets will be unassigned. This is done because `KheLayerTreeMake` tracks relations between events, not meets, and cannot cope with the idea of one event being assigned partly to one event and partly to another. A meet will also be unassigned when there is a cycle of assignments, but that should never occur in practice.

The above attempts to be a complete specification of `KheLayerTreeMake`, sufficient for using it. For the record, the following subsections explain how it works in detail.

### 9.1.1. Overview

`KheLayerTreeMake` uses a constructive heuristic which runs quickly. It works by examining the relevant constraints and taking actions to satisfy them, giving priority to those with higher weight. It does not search through a large space of possible solutions to find the best. This is appropriate, because in practice good solutions are easy to find. The problem is more about giving due weight to the many influences on the solution than about real solving.

`KheLayerTreeMake` begins by unassigning meets to remove cases where two meets derived from a single event are assigned to meets not both derived from the same event or both cycle meets, and splitting meets whose duration exceeds the number of times in the instance into meets of duration within that bound. This allows the remainder of the algorithm to assume that each event is preassigned to at most one other event, and that there are no oversize meets.

In practice, it is likely that the constraints of an instance will cooperate harmoniously, but for completeness it is necessary to handle cases where they do not. For example, there is nothing to prevent a link events constraint from linking two events, one of which is required by a split events constraint to split into three meets, while the other is required to split into one.

There is a data structure, described in the following sections, which embodies all the requirements that the final layer tree must satisfy, including how events are to be split into meets, and how meets are to be grouped into nodes. It is an invariant that at least one layer tree must satisfy all these requirements. Initially, the data structure embodies no requirements at all. A long series of *jobs* is then applied to it, each inspired by some constraint or other feature of the instance to request that the data structure add some new requirements to the ones it currently embodies. If no layer trees would satisfy both the old and new requirements, the job is *rejected* (it is ignored); otherwise, it is *accepted* (its requirements are added). There are also cases in which some of the requirements of a job are accepted but others have to be rejected. The jobs are sorted by decreasing priority, which is usually the combined weight of the constraint that inspired the job. In this way, contradictory requests are resolved by giving preference to requests of higher priority.

Here is the full list of job types, with brief descriptions. How each job modifies the data structure will be explained later. The jobs not derived from constraints have high priority.

*Pre-existing splits.* Each already split event *e* generates a job requiring the meets that *e* is ultimately split into to be packable into (created by further splitting of) the pre-existing meets.

*Preassigned times.* XHSTT specifies that a meet derived from an event with a preassigned time must be assigned that time. Several simultaneous meets derived from one event are unlikely to be wanted, so this job requests that a preassigned event be not split further than its pre-existing splits, and that the meets' time domains be set to singleton domains.

*Pre-existing assignments and link events constraints.* These are interpreted as requests to create parent-child links between nodes.

*Avoid clashes constraints.* Each resource subject to a required avoid clashes constraint gives rise to a job which requests that the layer tree recognize that the events to which the resource is preassigned cannot overlap in time.

*Split events constraints and distribute split events constraints.* These request restrictions on the number of meets that an event may be split into, and their durations.

*Spread events constraints.* If the events of an event group of a spread events constraint are split into too many or too few meets, then a non-zero number of deviations of the constraint becomes inevitable. The job tries to tighten the requirements on the number of meets of the events concerned, to the point where this problem cannot arise.

*Prefer times constraints.* This kind of job requests that the time domain of the meets of an event which have a certain duration be reduced to satisfy a prefer times constraint. This may lead to an empty domain for meets of that duration; if so, then there can be no meets of that duration at all, which may prevent the job from being accepted.

After all jobs have been applied, the data structure is traversed and a layer tree is built. Finally, `KheLayerTreeMake` examines each pair of events connected by a spread events or avoid split assignments constraint, and if those events' nodes satisfy the conditions given in Section 9.1.5, it merges them by calling `KheNodeMerge`.

### 9.1.2. Linking

The data structure used by `KheLayerTreeMake` must be close enough to the layer tree to make it straightforward to derive an actual layer tree at the end. In fact, it needs to represent the set of layer trees that satisfy the requirements of all the jobs accepted so far. This section explains how this is done for linking, and later sections explain the parts that handle splitting and layering.

If meet $s_1$ can be assigned to meet $s_2$ at offset $o_1$, and $s_2$ can be assigned to $s_3$ at offset $o_2$, then it is always possible to assign $s_1$ directly to $s_3$ at offset $o_1 + o_2$. Thus, the relation of assignability between meets is transitive. Although it is not safe to assign a meet to itself, it does no harm to pretend here that assignability is reflexive as well.

In some cases, two meets are assignable to each other. They must have equal durations and time domains, but that is not unusual. By a well-known fact about reflexive and transitive relations, two-way assignability is an equivalence relation between meets.

Similar relations can be defined between events. Let $A(e_1, e_2)$ hold when the meets of $e_1$ can be assigned to the meets of $e_2$ at non-overlapping offsets. Define

$$S(e_1, e_2) = A(e_1, e_2) \wedge A(e_2, e_1)$$

Again, $A$ is reflexive and transitive, and $S$ is an equivalence relation.

The data structure used for linking events includes a representation of relations $A$ and $S$. The equivalence classes defined by $S$ are represented by nodes of a graph, containing the events of the class and connected to other equivalence classes by directed edges representing $A$. $A$ could be an arbitrary directed acyclic graph, but in fact it is limited to a tree: each equivalence class is recorded as assignable to at most one other equivalence class. Relational nodes will always be called classes, to avoid confusion with layer tree nodes.

The child classes of each equivalence class are organized into layers. That additional structure is not needed for linking, however, so its description will be deferred to Section 9.1.4.

Initially, each event lies in its own class, plus there is one class with no events, representing the cycle meets. Every event class is a child of the cycle meets class. Thus, initially relation $S$ is empty, and relation $A$ records only the basic fact that every event is assignable to the cycle meets to begin with. This is quite true, since, at this initial stage, before any jobs are accepted, the data structure believes that each event's domain is the entire cycle, that each event is free to split into meets of duration 1, and that there are no layers.

Basing the data structure on events, rather than on meets, seems to be right, but it does cause differences between the meets of one event to be overlooked. For example, the data structure believes that all meets derived from the same event have the same time domain.

Jobs that link events together do so by proposing elements of $A$ and $S$ to the data structure, which accepts them when it can. An $S$ proposal is a request to merge the equivalence classes containing its two events into one (if they are not already the same); an *A* proposal is a request to replace one parent link by another (which must still imply the first by transitivity). A proposal could be rejected for various reasons: it might lead to a directed acyclic graph which is not a tree, or cause events from the same layer to overlap in time, or lead to unacceptable restrictions on how events are to be split (as in the example at the start of this chapter), and so on.

Pre-existing assignments are proposed first as elements of *S*, and if that fails as elements of *A*. The second proposal at least cannot fail to be accepted, because these jobs have maximum priority and do not contradict each other. A link events constraint job first proposes all pairs of linked events of equal duration as elements of *S*, and then all pairs regardless of duration as elements of *A*. In general, an *A* proposal could require that the whole set of classes lying on a cycle of *A* links be evaluated for merging, but this particular way of making proposals ensures that, in fact, only pairwise merges need to be evaluated.

Each equivalence class has a *class leader*, one of its own events. When an equivalence class is created, its leader is the sole event it initially contains, and when two classes are merged, one of the two leaders is chosen to be the leader of the merged class. For convenience, we pretend that the cycle meets are derived from a single *cycle event* which is the leader of their class.

If class *C* contains an event *e* which is assigned to an event outside *C*, then the event *e* is assigned to lies in the parent class of *C*. There may not be two such events in *C* unless they are assigned to the same event at the same offset. The leader must be one of these events. The data structure only becomes aware of assignments when the jobs representing them are accepted.

If *C* does not contain an event which is assigned to another event outside the class, then it must contain at least one event which is not assigned at all, since otherwise there would be a cycle of assignments within the class. Any such unassigned event may be the leader.

These conditions are trivially satisfied when a class is created, by making its sole event the leader. When two classes are merged, there are various possibilities, including failure to merge when the two leaders are assigned to distinct events outside both classes.

When constructing the final layer tree, all the unassigned events of each class except the leader are placed in layer tree nodes which are made children of the node containing the leader. Similarly, the nodes containing the leaders of child classes become children of the node containing the leader of the parent class. In reality, of course, it is the meets derived from these events by the splitting algorithm to be described next that are placed into these nodes.

## 9.1.3. Splitting

Given an event *e* of duration *d*, any mathematical partition of *d* is a possible outcome of splitting *e*. For example, if *e* has duration 6, the possible outcomes are the eleven partitions

| | | | | | |
|---|---|---|---|---|---|
| 6 | 4 2 | 3 3 | 3 1 1 1 | 2 2 1 1 | 1 1 1 1 1 1 |
| 5 1 | 4 1 1 | 3 2 1 | 2 2 2 | 2 1 1 1 1 | |

One element of a partition is called a *part*, and is the duration of one meet.

Any condition that limits how an event is split defines a subset of this set of partitions. For example, if a split events constraint states that an event of duration 6 should be split into exactly four meets, that is equivalent to requiring the partition to be either 3 1 1 1 or 2 2 1 1.

Each equivalence class holds a set of events of equal duration that are assignable to each other. These will eventually be partitioned into meets in the same way. In addition to the events, the class holds the requirements that the final partition must satisfy. These define a subset of the set of all partitions of the duration, but it is not possible to store the subset directly, because for large durations it may be very large. One partition *is* stored, however: the lexically minimum one satisfying the requirements. (A lexically minimum partition has minimum largest part, and so on recursively. For example, 1 1 1 1 1 1 is the lexically minimum partition of 6.) It is an invariant that the set of partitions satisfying the requirements may not be empty.

In the special case of the equivalence class that represents the cycle meets, the requirements are fixed to allow exactly one partition: the one representing the durations of the cycle meets.

The requirements on partitions are of two kinds. First, there are the *local requirements*. These are mainly lower and upper bounds on the total number of parts, and on the number of parts of each possible duration, modelled on the corresponding fields of the split events and distribute split events constraints. Another kind of local requirement arises when a pre-existing split job is accepted: if an event of duration 6 is already split into meets of duration 4 and 2, say, when the algorithm begins, then, to be acceptable, a partition must be packable into partition 4 2. One partition is *packable* into another if splitting some parts of the second partition and discarding others can produce the first. For example, 2 1 1 is packable into 2 2 2, but neither of 3 1 1 1 and 2 2 1 1 is packable into the other.

Second, there are the *structural requirements*. Each parent class has an arbitrary number of child classes, whose events will eventually be assigned to the parent class's events. So the lexically minimum partition of each child class must be packable into the parent class. In these calculations the constraint always flows upwards: the child's lexically minimum partition is taken as given, and the parent's minimum partition is adjusted (if possible) to ensure that the child's is packable into it. When a child class's minimum partition changes, the parent's requirements must be re-tested. In this way, a change to a partition propagates upwards through the structure until it either dies out or causes some class to have no legal partitions. In the second case, the job which originated the changes must be rejected.

Some of the child classes may be organized into layers. In that case, each layer's classes, taken together, must be packable into the parent class. Each layer is represented by a split layer object, as explained in detail in the next section. That object contains a minimum partition which must be packable into the parent class, just like the minimum partitions of child classes.

Deciding whether any partitions satisfy even the local requirements is non-trivial: is it safe to place two events into one class, when one is already split into partition 4 2 and the other is already split into partition 3 2 1? Some simple checks are made, then a full generate-and-test enumeration is begun and interrupted at the first success. The enumeration produces the lexically minimum acceptable partition first, which is then stored and propagated upwards. Fortunately, packability can be tested very quickly in practice, despite being an NP-complete bin packing problem, because event durations are usually small.

At the end, after the last job is processed, each event of each class is split into meets whose durations form the lexically minimum partition of that class.

### 9.1.4. Layering

The relation between meets and layers (sets of events that share a common preassigned resource with a required avoid clashes constraint) is a many-to-many relation: a layer may contain any number of meets, and a meet may lie in any number of layers.

Suppose that meet $s_1$ lies in layer $l$ and is assigned to meet $s_2$. KHE enforces the rule that any assignment of $s_2$ may not be such as to cause $s_1$ to overlap in time with any other meet of $l$. In a sense, $s_2$ (actually, that part of it assigned $s_1$) becomes a member of $l$ while $s_1$ is assigned to it. We say that $s_1$ lies *directly* in $l$, and $s_2$ lies *indirectly* in $l$.

An event lies directly in a layer if any of its meets lie directly in the layer. An equivalence class lies directly in a layer if any of its events lie directly in the layer, and it lies indirectly in the layer if any of its child classes lie in the layer, either directly or indirectly. This is because the events of child classes will eventually be assigned to the events of the class.

The layering aspect of `KheLayerTreeMake` is based on an object called a *split layer*, which represents one element of the many-to-many relation between equivalence classes and layers. In other words, there is one split layer object for each case of an equivalence class lying in a layer, directly or indirectly. Its attributes are the class, the resource defining the layer, the set of all child classes of the class that lie in the layer, and a partition, whose value will be defined shortly.

When an equivalence class lies directly in a layer (when it contains an event that lies directly in the layer), none of its child classes can lie in the layer, since that would mean that two events of the same layer overlap in time. So in that case the set of child classes must be empty. To keep it that way, the partition contains as many 1's as the duration of the class. This makes it clear that there is no room for any child classes in the layer, without constraining the division of the class's events into sub-events in any way.

When an equivalence class lies indirectly in a layer, some of its child classes lie in the layer. Their total duration must not exceed the duration of the class, and their meets, taken together, must be packable into the class, since they are disjoint in time. So in this case the set of child classes may be (in fact, must be) non-empty, and the partition holds the multiset union of the lexically minimum partitions of the child classes.

The job which adds a layer to the data structure adds its events one by one. In the unlikely event that the duration of the layer exceeds the number of times in the cycle, or bin packing problems prevent an event being added, the job rejects the event, which amounts to ignoring the presence of the preassigned resource in that event.

Adding an event to a layer means that the event's class and all its ancestors must get split layer objects for the layer. For all these classes, moving upwards until either there are no more ancestors or a class already has a split layer object for the layer, either add a new split layer object holding just the current child class, or add the child class to an existing split layer object.

While the upward propagation adds new split layer objects, there is no possibility of failure, since a layer containing a single event is no more constraining than the event alone (the event is already present, only its membership of a layer is changing). But if an existing split layer object is reached, the class must be added to it, and so its partition grows, possibly leading to an empty set of acceptable partitions in the parent, causing rejection of the request.

### 9.1.5. Merging

As mentioned earlier, when instances contain events which have already been split, it is best to merge the nodes containing those events. The advantages include ensuring that how the instance is presented does not affect the way it is solved, exposing symmetries which could be expensive if left hidden, and taking a step towards regularity.

Node merging is carried out after the main part of the layer tree construction algorithm is complete and a layer tree is present. For each pair of events that share a spread events or avoid split assignments constraint, the first meet of each event is found and the chain of fixed assignments is followed to the first unfixed meet and from there to the node. The two nodes thus found are candidates for merging. If they both exist, and they are distinct, and the first meet in each contains the same preassigned resources (counting resources in meets assigned to the meet, directly or indirectly, as well as resources in the meet itself), then the nodes are merged.

Only nodes which share at least one preassigned resource are merged. This ensures that it is right to assign non-overlapping times to the meets of a node, which is what solvers usually do.

Requiring the same preassigned resources turns out to be important, because of the way that layers are built from nodes, not from meets. If some of the meets of a node contain a resource but others do not, then when the nodes containing that resource are formed into a layer later, the layer's duration may be longer than the cycle length, making it awkward to timetable.

## 9.2. Time-equivalence

Two sets of meets are *time-equivalent* if it can be shown, by following fixed meet assignments, that each set of meets must occupy the same set of times as the other while fixed assignments remain in place. This may be true even when none of the meets is assigned a time.

Two events are time-equivalent if their sets of meets are time-equivalent. Usually, this is because they are joined by a link events constraint which is being handled structurally, for example by `KheLayerTreeMake` (Section 9.1).

Two resources are time-equivalent if they have the same resource type (call it `rt`), `KheResourceTypeDemandIsAllPreassigned(rt)` (Section 3.5.1) is `true`, and the sets of meets containing their preassigned tasks are time-equivalent. Time-equivalent resources are busy at the same times. They are usually students who choose the same courses.

It is clear that time-equivalence between sets of meets is an equivalence relation, as is time-equivalence between events and between resources. So the events and resources of an instance can be partitioned into time-equivalence classes. These classes are calculated by a *time-equivalence solver*, which can be created and deleted by calling

```
KHE_TIME_EQUIV KheTimeEquivMake(void);
void KheTimeEquivDelete(KHE_TIME_EQUIV te);
```

To perform the calculation for a particular `soln`, call

```
void KheTimeEquivSolve(KHE_TIME_EQUIV te, KHE_SOLN soln);
```

However, the usual way to obtain a time-equivalence object is by calling

```
KHE_TIME_EQUIV KheTimeEquivOption(KHE_OPTIONS options,
  char *key, KHE_SOLN soln);
```

with key `"ss_time_equiv"`. This returns a solved time equivalence object stored in `options` under `key`; if it is not present, it creates one, solves it, and adds it to `options` before returning it.

The equivalence classes of events are event groups which can be visited by

```
int KheTimeEquivEventGroupCount(KHE_TIME_EQUIV te);
KHE_EVENT_GROUP KheTimeEquivEventGroup(KHE_TIME_EQUIV te, int i);
```

in the usual way. The equivalence class for a given event is returned efficiently by

```
KHE_EVENT_GROUP KheTimeEquivEventEventGroup(KHE_TIME_EQUIV te,
  KHE_EVENT e);
```

If `e` is not time-equivalent to any other event, a singleton event group containing `e` is returned. There is also

```
int KheTimeEquivEventEventGroupIndex(KHE_TIME_EQUIV te, KHE_EVENT e);
```

which returns the value `i` such that `KheTimeEquivEventGroup(te, i)` contains `e`.

Similarly, the equivalence classes of resources are resource groups which can be visited by

```
int KheTimeEquivResourceGroupCount(KHE_TIME_EQUIV te);
KHE_RESOURCE_GROUP KheTimeEquivResourceGroup(KHE_TIME_EQUIV te, int i);
```

in the usual way. The equivalence class for a given resource is returned efficiently by

```
KHE_RESOURCE_GROUP KheTimeEquivResourceResourceGroup(KHE_TIME_EQUIV te,
  KHE_RESOURCE r);
```

If `r` is not time-equivalent to any other resource, including the case when its resource type is not all preassigned, a singleton group containing `r` is returned. Again,

```
int KheTimeEquivResourceResourceGroupIndex(KHE_TIME_EQUIV te,
  KHE_RESOURCE r);
```

returns the value `i` such that `KheTimeEquivResourceGroup(te, i)` contains `r`.

All of these results reflect the state of the solution at the time of the most recent call to `KheTimeEquivSolve(te)`; they are not updated as the solution changes.

## 9.3. Layers

Layers were introduced in Section 5.3, but no easy way to build a set of layers was provided. This section remedies that deficiency and adds some useful aids to solving with layers.

### 9.3.1. Layer construction

The usual rationale for the existence of a layer is that its nodes' meets must not overlap in time because they contain preassignments of a common resource. Function

```
KHE_LAYER KheLayerMakeFromResource(KHE_NODE parent_node,
  KHE_RESOURCE r);
```

builds a layer of this kind. It calls `KheLayerMake` to make a new child layer of `parent_node`, and `KheLayerAddResource` to add `r` to this layer. Then, each child node of `parent_node` which contains a meet preassigned `r` (either directly within the node, indirectly within descendant nodes, or in meets assigned, directly or indirectly, to those meets) is added to the layer.

The *layering* of node `parent_node` is a particular set of layers which is useful when assigning times to the child nodes of `parent_node`, created by calling function

```
void KheNodeChildLayersMake(KHE_NODE parent_node);
```

This will delete any existing child layers of `parent_node` and add the layers of the layering.

The layering is built as follows. First, for each resource of the instance that possesses a required avoid clashes constraint, one layer is built by calling `KheLayerMakeFromResource` above. If it turns out to be empty, it is immediately deleted again. Each pair of these layers such that one's node set is a subset of the other's is merged with `KheLayerMerge`. Finally, each child of `parent_node` not in any layer goes into a layer (with no resources) by itself.

The layers emerge from `KheNodeChildLayersMake` in whatever order they happen to be. The user will probably need to sort them, by calling `KheNodeChildLayersSort` (Section 5.3), passing it a user-defined comparison function. Section 10.8.2 has an example of a comparison function that seems to work well in practice.

After sorting, there may be value in calling

```
void KheNodeChildLayersReduce(KHE_NODE parent_node);
```

This merges some layers of marginal utility into others, as follows. Suppose there is a layer $L$ whose nodes all appear in earlier layers. Then if the meets of the nodes are assigned layer by layer, $L$'s nodes will all be assigned before time assignment reaches $L$. Arguably, $L$ could be deleted without harm. However, it does contain one piece of useful information: it knows that the meets to which its resources are preassigned will all be assigned times after $L$ is assigned. If this information is to be preserved, $L$'s resources need to be moved forwards to the first earlier layer that is true of. For each node $N$ of $L$, find the minimum over all layers containing $N$ of the index of the layer. This is the index of the layer during whose time assignment $N$ will be assigned. Then find the maximum, over all nodes $N$ of $L$, of these minima. This is index of the layer whose assignment will complete the assignment of all the nodes of $L$. If this is smaller than $L$'s index, `KheNodeChildLayersReduce` deletes $L$ and moves its resources to this earlier layer.

Two important facts about layers and layerings must be borne in mind. First, they reflect the state of the layer tree at a particular moment. If, after they are built, the tree is restructured (if nodes are moved, etc.) they become out of date and useless. Second, building a layering is slow and should not be done within the inner loops of a solver.

Altogether, it seems best to regard layers as temporary structures, created when required by `KheChildLayersMake` and destroyed by `KheChildLayersDelete`. In between these two calls, nodes may be merged and split, but it is best not to move them. A useful convention, supported by several of KHE's solvers that use layers, is to assume that if child layers are present, then they are up to date. Such solvers begin by calling `KheChildLayersMake` if there are no layers, and end by calling `KheChildLayersDelete`, but only if they called `KheChildLayersMake`.

### 9.3.2. Layer coordination

High schools usually contain *forms* or *years*, which are sets of students of the same age who follow the same curriculum, at least approximately. These students may be grouped into classes, each represented by one student group resource. At some times, the student group resources of one form might attend the same events, or linked events. For example, they might all attend a common Sport event, or they might all attend Mathematics at the same times so that they can be regrouped by ability at Mathematics. At other times, they might attend quite different events, but over the course of the cycle they all attend the same amount of each different kind of event: so many times of English, so many of Science, so many of a shared elective, and so on.

As an aid to producing a regular timetable, it might be helpful to *coordinate* the timetables of student groups from the same form: run all the form's English classes simultaneously, all its Mathematics classes simultaneously, and so on. Where resources are insufficient to support this, changes can be made later. In this way, a regular timetable is produced to begin with, and irregularities are introduced only where necessary.

The XML format does not explicitly identify forms, or even say which resource type contains the student group resources. This is in fact an advantage, because it forces us to look for structure that aids regularity. We then coordinate the timetabling of resources that possess the useful structure, without knowing or caring whether they are in fact student group resources.

Coordination will only work when the chosen resources attend similar events. This was the rule when inferring resource partitions (Section 3.5.6), so we take the resource partition as the structural equivalent of the form. The events should occupy all or most of the times of the cycle, otherwise coordination eliminates too many options for spreading them in time. 'Forms' of teachers and rooms are rarely useful, just because they do not satisfy these conditions.

After `KheLayerTreeMake` returns, it is the nodes lying directly below the root node that need to be coordinated, not events or meets. Two child nodes may be coordinated by moving one of them so that it is a child node of the other. KHE offers solver function

```
void KheCoordinateLayers(KHE_NODE parent_node, bool with_domination);
```

which carries out such moves on some of the children of `parent_node`, as follows.

`KheCoordinateLayers` is only interested in resources whose layers have duration at least 90% of the duration of `parent_node`. For each pair of such resources lying in the same resource partition, it checks whether their two layers are similar by building the layers with `KheLayerMakeFromResource` and calling `KheLayerSimilar` (Section 9.3). If so, it uses `KheNodeMove` (Section 9.5.3) to make each node of the second layer a child of the corresponding node of the first, unless the two nodes are the same, forcing these nodes to be simultaneous. It does not assign meets, or remove them from nodes. Finally, it removes the two layers it made.

If `with_domination` is `false`, the behaviour is as described. If `with_domination` is `true`, a slight generalization is used. Suppose that one of the two layers has duration equal to the duration of `parent_node`, and all but one of its nodes is similar to some node in the other layer. Then the dissimilar nodes of the other layer (possibly none) might as well be made children of the one dissimilar node of that layer, since if the other nodes are coordinated they must run simultaneously with it anyway. (The durations of their meets may be incompatible; that is not checked at present, although it should be.) So that is done.

In unusual cases the duration of a layer can be larger after coordinating than before. At the end, if any layers have duration larger than the parent node's duration, `KheCoordinateLayers` tries to reduce the duration of those layers to the parent node's duration, by finding cases where one node of a layer can be safely moved to below another.

## 9.4. Runarounds

Layer coordination can lead to problems assigning resources. For example, suppose that the five student groups of the Year 7 form each attend one Music event, and that the school has two Music teachers and two Music rooms. Each event is easily accommodated individually, but when the Year 7 layers are coordinated, they run simultaneously and exceed resource limits.

These problems do not arise in large faculties with sufficient resources to accommodate an entire form at once. Thus they do not invalidate the basic idea of node layer coordination. What is needed is a local fix for these problems. This is what *runarounds* provide: a way to spread the events concerned through the times they need, without abandoning coordination altogether.

### 9.4.1. Minimum runaround duration

Consider the case above where there are not enough Music resources to run the Year 7 Music events simultaneously. If these events lie in nodes that are children of a common parent (one may lie in the parent itself), it is easy to detect this problem: carry out a time assignment at the parent, and see whether the cost of the solution increases. This is assuming that the matching monitors, which detect unsatisfiable resource demands, are attached.

More generally, we can ask how large the duration of the parent node has to be in order to ensure that there is no cost increase. This quantity is called the *minimum runaround duration* of the node. It will be equal to the duration when there is no problem, and larger when there is a problem. It can be calculated as follows. While a time assignment of the child nodes produces a state of higher cost than the unassigned state, add new meets to the parent node. The duration of the parent node when this process ends is its minimum runaround duration. Function

```
bool KheMinimumRunaroundDuration(KHE_NODE parent_node,
   KHE_TIME_SOLVER time_solver, KHE_TIME_OPTIONS options,
   int *duration);
```

sets `*duration` to the minimum runaround duration of `parent_node` and returns `true`, except in an unlikely case, documented below, when it returns `false` with `*duration` undefined.

`KheMinimumRunaroundDuration` first unassigns all the child meets and saves the unassigned cost. It then carries out the time assignment trials just described. For each trial after the first it adds one fresh meet to `parent_node` for each of its original meets, utilizing their durations and time domains, but with no event resources. So the result's duration must be a multiple of the duration of `parent_node`. Before returning, it unassigns all the children and removes the meets it added, leaving the tree in its initial state, unless some child meets were assigned to begin with.

Parameter `time_solver` is a time assignment solver which is called to carry out each trial. A simple solver, such as `KheSimpleAssignTimes` from Section 10.4, should be sufficient here.

Increasing the duration at each trial by the full duration of the node may seem excessive, and there are cases where fewer additional meets would be enough. However, those cases require the child nodes' assignments to overlap in ways that do not work out well in practice, because they may lead to split assignments in the tasks affected.

How many trials are needed? In reasonable instances, each child node's duration should be no greater than the parent node's duration. Thus, after as many trials as there are child nodes plus one, there should be enough room in the parent node to assign every child meet at an offset which does not overlap with any other, or with the original parent meets. This is the number of trials that `KheMinimumRunaroundDuration` carries out. It stops early if one succeeds with cost no greater than the unassigned cost. It returns `false` only when each trial either did not assign all the child meets (that is, the call on `time_solver` returned `false`) or did assign them all, but at a higher cost than the unassigned cost.

### 9.4.2. Building runarounds

Nodes may be classified into three types. A *fixed node* has no child nodes. There is no possibility of spreading the events of a fixed node and its descendants through more times than the node's duration. A *problem node* has minimum runaround duration larger than its duration, like the node of Music events used as an example above. It must have child nodes, and timetabling them simultaneously is known to be inferior to spreading them out further. The remaining nodes are *free nodes*: they have child nodes which may run simultaneously, or not, as convenient.

Using `KheNodeMerge` to merge problem nodes with other problem nodes and free nodes can eliminate problem nodes without greatly disrupting regularity. For example, merging a Music problem node of duration 2 and minimum runaround duration 6 with a free node of duration 4 produces a merged node of duration 6 which can usually be timetabled without problems.

If a merged node can be timetabled without the cost of the solution increasing, it may be kept, and is then called a *runaround node*. (The term *runaround* is used by manual timetablers known to the author to describe this kind of timetable, where events like the Music events are 'run around' with other events.) Otherwise it must be split up again and some other merging tried instead. It only remains, then, to decide which sets of nodes to try to merge.

Regularity is easier to attain when nodes have the same duration, so if there are already many nodes of a certain duration, it is helpful if a merged node also has that duration. Nevertheless, a node should not be added to a merge merely to make up some duration: merging limits the choices open to later phases of the solve, so it should be done only when necessary.

A minimum runaround duration could be very large, close to the duration of the whole cycle. For example, suppose there is a single teacher, the school chaplain, who gives each of the five Year 7 student groups 6 times of religious instruction per week. Those events have a minimum runaround duration of 30. When the minimum runaround duration of a node is larger than a certain value, the algorithm given below ignores the node: its events will be awkward to timetable, but runarounds as defined here are not the answer.

To build runaround nodes from the child nodes of `parent_node`, call

```
void KheBuildRunarounds(KHE_NODE parent_node,
  KHE_NODE_TIME_SOLVER mrd_solver, KHE_TIME_OPTIONS mrd_options,
  KHE_NODE_TIME_SOLVER runaround_solver,
  KHE_TIME_OPTIONS runaround_options);
```

where `mrd_solver` and `mrd_options` are passed to `KheMinimumRunaroundDuration` when minimum runaround durations need to be calculated, and `runaround_solver` and `runaround_options` are used to timetable merged nodes. `KheSimpleAssignTimes` is sufficient for `mrd_solver`, and `KheRunaroundNodeAssignTimes` works well as `runaround_solver`. All nodes are unassigned afterwards.

It would not do to merge (for example) a node that includes both Year 7 and Year 8 events with a node that includes only Year 7 ones. So `KheBuildRunarounds` first works out which resources are preassigned to events in or below which nodes (taking account only of preassigned resources which have required avoid clashes constraints, and whose events occupy at least 90% of the duration of `parent_node`), and partitions the child nodes of `parent_node` into disjoint subsets, such the nodes in each subset have the same preassigned resources.

For each disjoint subset independently, `KheBuildRunarounds` tries to build a merged node around each of the subset's problem nodes in turn, largest minimum runaround duration first. When doing this, it prefers to build a node of a particular duration $u$, and it prefers to use other problem nodes (again, largest minimum runaround duration first), but it will also use free nodes (minimum duration first). It is heuristic, but it usually works well. It is not limited to sequences of pairwise mergings, as clustering algorithms often are. Here is the algorithm in detail:

1. The input is a set of nodes $N$ (one disjoint subset as above), plus $u$, a desirable duration for a merged node, and $v$, a maximum duration for a merged node. The output is $M$, the final set of nodes. Write $d(n)$ for the duration of node $n$, $r(n)$ for its minimum runaround duration, and $d(X)$ for the total duration of the set of nodes $X$.

2. Initialize $M$ to empty. Sort $N$ to put free nodes first, in decreasing duration order, problem nodes next, in increasing minimum runaround duration order, and fixed nodes last.

3. If $N$ is empty, stop. Otherwise delete the last element of $N$ and call it $n$.

4. If $n$ is fixed, problem with $r(n) \geq v$, or free, move it to $M$ and return to Step 3.

5. Here $n$ must be a problem node satisfying $r(n) < v$. Within each of the following cases, some non-empty subsets $X$ of $N$ are defined. In each case, $r(n) \leq d(n) + d(X)$, so a merged node consisting of $n$ merged with $X$ is likely to work well. For each case in turn, and for each set $X$ defined within each case in turn, remove $X$ from $N$, merge $n$ and $X$, and timetable the resulting merged node. If that is successful (all events timetabled with no increase in solution cost), add the merged node to $M$ and return to Step 3. If it fails, split the merged node up again, return the nodes of $X$ to their former places in $N$, and try the next set $X$; or if there are no more sets, add $n$ to $M$ and return to Step 3.

   Case 1. For each $x \in N$ from last to first such that $r(n) \leq d(n) + d(x) = u \leq v$, let $X = \{x\}$.

   Case 2. For each $i$ from 1 to $|N|$ such that $X_i$, the last $i$ elements of $N$, satisfies the condition $r(n) \leq d(n) + d(X_i) \leq v$, let $X = X_i$.

`KheBuildRunarounds` calls `KheMinimumRunaroundDuration` to find minimum runaround durations, passing `mrd_solver` to it. It calls `KheNodeMerge` to merge nodes, `runaround_solver` to timetable merged nodes, and `KheNodeSplit` to undo failed merges. It uses one-fifth of the duration of `parent_node` for $v$. For $u$, it builds a frequency table of the durations of child nodes of `parent_node`. It then chooses the duration for which the frequency times the duration is maximum. This weights the choice away from small durations, which are not very useful.

## 9.5. Rearranging nodes

Earlier sections of this chapter contain the major solvers which work with nodes. This section contains a miscellany of smaller helper funtions which rearrange nodes.

### 9.5.1. Node merging

Two nodes may be merged by calling

```
bool KheNodeMergeCheck(KHE_NODE node1, KHE_NODE node2);
bool KheNodeMerge(KHE_NODE node1, KHE_NODE node2, KHE_NODE *res);
```

The nodes may be merged if they have the same parent node, possibly `NULL`.

The meets of the result, `*res`, are the meets of `node1` followed by the meets of `node2`, and the child nodes of `*res` are the child nodes of `node1` followed by the child nodes of `node2`. The two nodes must either lie in the same layers and have the same parent, or have no parent, otherwise `KheNodeMerge` aborts. This implies that node merging cannot violate the cycle rule, or any rule. As usual with merging, `node1` and `node2` are undefined afterwards (actually, `node1` is recycled as `*res` and `node2` is freed), but one may write, for example,

```
KheNodeMerge(node1, node2, &node1);
```

to re-use variable `node1` to hold the result.

Merging permits the meets of the child nodes of the two nodes to be assigned to the meets of either node, rather than to just one as before. For example, suppose the layer tree rooted at `node1` contains the Science events of several groups of Year 7 students, and the layer tree rooted at `node2` contains the Music events of the same groups of students. Then originally the Science events must be simultaneous and the Music events must be simultaneous, but afterwards the two kinds of events may intermingle. This may be useful if there are few Music teachers and Music rooms, so that the Music events must be spread out in time. This kind of arrangement is well known to manual timetablers; it has various names, including *runaround*.

There is no operation to split a node into two nodes. However, `KheNodeMerge` may be undone using marks and paths as usual.

### 9.5.2. Node meet splitting and merging

Node meet splitting and merging (not to be confused with node merging above) split the meets of a node as much as possible, and merge them together as much as possible:

```
void KheNodeMeetSplit(KHE_NODE node, bool recursive);
void KheNodeMeetMerge(KHE_NODE node, bool recursive);
```

Both operations always succeed, although they may do nothing.

For every offset of every meet of `node`, `KheNodeMeetSplit` calls `KheMeetSplit`, passing it the `recursive` parameter. In this way, the meets become as split up as possible.

`KheNodeMeetMerge` sorts the meets so that meets assigned to the same target meets are adjacent, with their target offsets in increasing order, using `KheMeetIncreasingAsstCmp` from Section 5.2. Unassigned meets go at the end. It then tries to merge each pair of adjacent meets. Any calls to `KheMeetMerge` it makes are passed the `recursive` parameter.

### 9.5.3. Node moving

A node may be made the child of `parent_node`, instead of its current parent, by calling

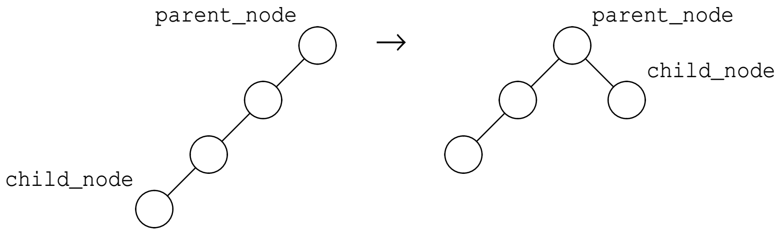
```
bool KheNodeMoveCheck(KHE_NODE child_node, KHE_NODE parent_node);
bool KheNodeMove(KHE_NODE child_node, KHE_NODE parent_node);
```

This does the same as the sequence

```
KheNodeDeleteParent(child_node);
KheNodeAddParent(child_node, parent_node);
```

except that this sequence will fail if any of `child_node`'s meets are assigned initially, whereas `KheNodeMove` deals with such assignments and can fail only the cycle rule.

In most cases, `KheNodeMove` begins by deassigning those meets of `child_node` that are assigned. However, there is one interesting exception. Suppose that `child_node`'s new parent node is an ancestor of `child_node`'s current parent node:



In each case where a complete chain of assignments reaches from a meet `meet` of `child_node` to a meet of `parent_node`, `meet` will be assigned afterwards, to the meet at the end of the chain, with offset equal to the sum of the offsets along the chain. This is valid (it does not change the timetable). Where there is no complete chain, `meet` will be unassigned afterwards.

For example, suppose node `p` has accumulated children to make the timetable regular, but now the children's original freedom to be assigned elsewhere needs to be restored:

```
while( KheNodeChildCount(p) > 0 )
  KheNodeMove(KheNodeChild(p, 0), KheNodeParent(p));
```

`KheNodeMove` preserves the current timetable during these relinkings.

### 9.5.4. Vizier nodes

A *vizier* (Arabic *wazir*) is a senior official, the one who actually runs the country while the nominal ruler gets the adulation. In a similar way, a *vizier node* sits below another node and does what that other node nominally does: act as the common parent of the subordinate nodes, and hold the meets that those nodes' meets assign themselves to.

Any node can have a vizier, but only the cycle node really has a use for one. By connecting everything to the cycle node indirectly via a vizier, it becomes trivial to try time repairs in which the meets of the vizier node change their assignments, effecting global alterations such as swapping everything on Tuesday morning with everything on Wednesday morning. Function

```
KHE_NODE KheNodeVizierMake(KHE_NODE parent_node);
```

inserts a new vizier node directly below `parent_node`. Afterwards, `parent_node` has exactly one child node, the vizier; it may be accessed using `KheNodeChild(parent_node, 0)` as usual, and it is also the return value. For every meet `pm` of the parent node, the vizier has one meet `vm` with the same duration as `pm` and assigned to `pm` at offset 0. The domain of `vm` is `NULL`; its assignment is not fixed. Each child node of `parent_node` becomes a child of the vizier; each child layer of `parent_node` becomes a child layer of the vizier; each meet assigned to a meet of the parent node becomes assigned to the corresponding meet of the vizier. If `parent_node` has zones, the vizier is given new corresponding zones, and the parent node's zones are removed.

All this leaves the timetable unchanged, including constraints imposed by domains and zones. The vizier takes over without affecting anyone's existing rights and privileges. A vizier node is not different from any other node; only its role is special.

`KheNodeSwapChildNodesAndLayers` (Section 5.2) is used to move the child nodes and layers to the vizier node, so they are the exact same objects after the call as before. But although the zones added to the vizier correspond exactly with the original zones, they are new objects.

To remove a vizier node, call

```
void KheNodeVizierDelete(KHE_NODE parent_node);
```

Here `parent_node` must have no child layers, no zones, and exactly one child node, assumed to be the vizier. It calls `KheNodeSwapChildNodesAndLayers` again, to make the child nodes of the vizier into child nodes of `parent_node`, and the child layers of the vizier into child layers of `parent_node`. Any assignments to meets in the child nodes of the vizier must be to meets in the vizier, and they are converted into assignments to meets in `parent_node` where possible (when the target meet in the vizier is itself assigned). New zones are created in `parent_node` based on the zones and meet assignments in the vizier. Finally the vizier and its meets are deleted.

Zones are not preserved across calls to `KheNodeVizierMake` and `KheNodeVizierDelete` in the exact way that child nodes and child layers are. The zones added to the vizier node by `KheNodeVizierMake` are new objects, although they do correspond exactly with the zones in `parent_node`. The zones added to `parent_node` by `KheNodeVizierDelete` are also new, and there will be a zone in a given parent meet at a given offset only if there was a meet in the vizier which was assigned that parent meet and was running (with a zone) at that offset. If vizier meets overlap in time (not actually prohibited), that will further confuse the reassignment of zones. It may be best to follow `KheNodeVizierDelete` by a call to some function which ensures that every offset of every parent meet has a zone, for example `KheNodeExtendZones` (Section 9.6).

Function `KheNodeMeetSplit` (Section 9.5.2) is useful with vizier nodes. Splitting a vizier's meets non-recursively opens the way to fine-grained swaps, between half-mornings instead of full mornings, and so on. A wild idea, that the author has not tried, is to have an unsplit vizier with its own split vizier. Then the larger swaps and the smaller ones are available together.

### 9.5.5. Flattening

Although layer coordination and runaround building are useful for promoting regularity, there may come a point where these kinds of voluntary restrictions prevent assignments which satisfy more important constraints, and so they must be removed.

What is needed is to flatten the layer tree. Two functions are provided for this. The first is

```
void KheNodeBypass(KHE_NODE node);
```

This requires `node` to have a parent, and it moves the children of `node` so that they are children of that parent. The second is

```
void KheNodeFlatten(KHE_NODE parent_node);
```

It moves nodes as required to ensure that all the proper descendants of `parent_node` initially are children of `parent_node` on return.

Both functions use `KheNodeMove` to move nodes. They cannot fail, because `KheNodeMove` fails only when there is a problem with the cycle rule, which cannot occur here. Both functions are ‘interesting exceptions’ (Section 9.5.3) where assignments are preserved. By convention (Chapter 10), meets with fixed, final assignments should not lie in nodes. If that convention is followed, these functions do not affect such meets.

## 9.6. Adding zones

Suppose a layer of child nodes of node *n* has its meets assigned to the meets of *n* at various offsets. Define one zone for each child node *c* of the layer, whose meet-offsets are the ones at which *c*’s meets are running. Helper function

```
void KheLayerInstallZonesInParent(KHE_LAYER layer);
```

installs these zones, first deleting any existing zones of the parent node of `layer`, then installing one zone for each child node of `layer` containing at least one assigned meet. Such zones form an image of how one child layer (the first to be assigned, usually) is assigned. An algorithm can use them as a template when assigning the other child layers, or when repairing the assignments of any child layers, including the first layer.

`KheLayerInstallZonesInParent` installs zones representing the assignments of one layer into the layer’s parent node. If the duration of the parent node exceeds the duration of the layer, some offsets in some parent node meets will not be assigned any zone. This seems likely to be a problem, or at least a lost opportunity. What to do about it is not clear.

Arguably, zones should be derived from all layers, not just one, in a way that gives every offset a zone. But that is not easy to do, even heuristically. Anyway, there are advantages in using zones derived from a good assignment of some layer, since the assignment proves that those zones work well. This suggests taking the zones installed by `KheLayerInstallZonesInParent` and extending them until every offset has a zone. Accordingly, function

```
void KheNodeExtendZones(KHE_NODE node);
```

ensures that every offset of every meet of `node` has a zone, by assigning one of `node`’s existing zones to each offset in each meet of `node` that does not have a zone—unless `node` has no zones to begin with, in which case it does nothing.

For each (zone, meet) pair where the meet has at least one offset without a zone, the algorithm finds one option for adding some of the zone to the meet (how much to add, and where), and assigns a priority to the option. Then it selects an option of minimum priority, carries it out, and repeats. It runs out of options only when every offset in every meet has a zone.

An option for adding some of a given zone to a given meet is found as follows. If the zone is already present in the meet, it is best to add it at offsets adjacent to the offsets it already occupies, if possible. If the zone is not already present, it is best to add it adjacent to existing offsets or the ends of the meet, in a continuous run, to avoid fragmentation of the offsets it occupies as well as the offsets it doesn't occupy. Constraints on zone durations arise either way. Within the limits imposed by them, it is best to aim for an ideal zone duration, which in a completely unoccupied meet is the meet duration divided by the total number of zones, but which is adjusted to take account of existing zone durations, with higher being a better option than lower. As the option is decided on, it is assigned a priority based on whether it utilizes an underutilized zone, avoids fragmentation, and approximates to the ideal zone duration.

## 9.7. Meet splitting and merging

This section presents features which modify the meet splits made by layer tree construction.

### 9.7.1. Analysing split defects

Given a defect (a monitor of non-zero cost), it is usually easy to see what needs to be done to repair it: if there is a clash, move one of the clashing meets away; if there is a split assignment, try to find a resource to assign to all the tasks; and so on.

*Split defects*, that is, split events and distribute split events monitors of non-zero cost, are awkward to analyse in this way, partly because split events monitors monitor both the number of meets and their durations, and partly because several split events and distribute split events monitors may cooperate in constraining how a given event is split into meets.

KHE offers a *split analyser* which analyses the split events and distribute split events monitors of a given event, and comes up with a sequence of suggestions as to how any defects among those monitors could be repaired using splits or merges (or both: for example, if there are too few meets of a given duration, that could be corrected by splitting larger meets or by merging smaller ones). To create and subsequently delete a split analyser object, call

```
KHE_SPLIT_ANALYSER KheSplitAnalyserMake(KHE_SOLN soln);
void KheSplitAnalyserDelete(KHE_SPLIT_ANALYSER sa);
```

In practice, it is better to obtain a split analyser object from the `"ss_split_analyser"` option, which can be done by a call to

```
KHE_SPLIT_ANALYSER KheSplitAnalyserOption(KHE_OPTIONS options,
  char *key, KHE_SOLN soln);
```

with key `"ss_split_analyser"`. This creates a split analyser and stores it in `options` if it is not already present. The option name is conventional; any name could have been chosen.

To carry out the analysis for a particular event, call

```
void KheSplitAnalyserAnalyse(KHE_SPLIT_ANALYSER sa, KHE_EVENT e);
```

After doing this, the sequence of suggestions for `e` which are splits may be retrieved by calling

```
int KheSplitAnalyserSplitSuggestionCount(KHE_SPLIT_ANALYSER sa);
void KheSplitAnalyserSplitSuggestion(KHE_SPLIT_ANALYSER sa, int i,
  int *merged_durn, int *split1_durn);
```

for `i` between `0` and `KheSplitAnalyserSplitSuggestionCount(sa) - 1` as usual. Each split suggestion suggests splitting any meet of duration `*merged_durn` into two fragments, one with duration `*split1_durn`. Similarly, the sequence of merge suggestions may be retrieved by

```
int KheSplitAnalyserMergeSuggestionCount(KHE_SPLIT_ANALYSER sa);
void KheSplitAnalyserMergeSuggestion(KHE_SPLIT_ANALYSER sa, int i,
  int *split1_durn, int *split2_durn);
```

Each suggests merging any two meets with durations `*split1_durn` and `*split2_durn`.

Each suggestion is distinct from the others. No notice is taken of constraint weights, except that constraints of weight zero are ignored. The suggestions are updated only by calls to `KheSplitAnalyserAnalyse`; they are unaffected by later changes to the solution. So they go out of date after a split or merge, but become up to date again if that split or merge is undone.

Function

```
void KheSplitAnalyserDebug(KHE_SPLIT_ANALYSER sa, int verbosity,
  int indent, FILE *fp);
```

places a debug print of `sa` onto `fp` with the given verbosity and indent, including suggestions.

### 9.7.2. Merging adjacent meets

It sometimes happens that at the end of a solve, two meets derived from the same event are adjacent in time and not separated by a break. If the same resources are assigned to both, they can be merged, which may remove a spread defect and thus reduce the overall cost. Function

```
void KheMergeMeets(KHE_SOLN soln);
```

unfixes meet splits in all meets derived from events and carries out all merges that reduce solution cost. For each event `e`, it takes the meets derived from `e` that have assigned times and sorts them chronologically. Then, for each pair of adjacent meets in the sorted order, it tries `KheMeetMerge`, keeping the merge if it succeeds and reduces cost.

`KheMergeMeets` can be called at any time. The best time to call it is probably at the very end of solving, or possibly after time assignment.

# Chapter 10. Time Solvers

A *time solver* assigns times to meets, or changes their assignments. This chapter presents a specification of time solvers, and describes the time solvers packaged with KHE.

## 10.1. Specification

If time solvers share a specification, where possible, it is easy to replace one by another, pass one as a parameter to another, and so on. This section recommends such a specification.

In hierarchical timetabling, 'time assignment' means the assignment of the meets of child nodes to the meets of a parent node, so the recommended interface is

```
typedef bool (*KHE_NODE_TIME_SOLVER)(KHE_NODE parent_node,
  KHE_OPTIONS options);
```

This typedef appears in `khe_solvers.h`. The intended meaning is that such a *node time solver* should assign or reassign some or all of the meets of the proper descendants of `parent_node`: it might assign the unassigned meets of the child nodes of `parent_node`, or reassign the meets of proper descendants of `parent_node`, and so on. It is free to reorganize the tree below `parent_node`, provided that every descendant of `parent_node` remains a descendant. It must not change anything in or above `parent_node`. In the tree below `parent_node` it may add, delete, split, and merge meets. Some solvers (e.g. ejection chains) do actually do this, so the caller must take care to avoid the error (very easily made, as the author can testify) of assuming that the set of meets after a time solver is called is the same as before. The `options` parameter is as in Section 8.2; by convention, options consulted by time solvers have names beginning with `ts_`.

A solver should return `true` when it has changed the solution (usually for the better, but not necessarily), and when it is not sure whether it did or not. It should return `false` when it did not change the solution. The caller may use this information to evaluate the helpfulness of the solver, or to decide whether to follow it with a repair step, and so on.

A second time solver type is defined in `khe_solvers.h`:

```
typedef bool (*KHE_LAYER_TIME_SOLVER)(KHE_LAYER layer,
  KHE_OPTIONS options);
```

Instead of assigning or reassigning meets in the proper descendants of some parent node, a *layer time solver* assigns or reassigns meets in the nodes of `layer` and their descendants, like a node time solver for the parent node of `layer`, but limited to `layer`. The solver is free to reorganize the layer tree below the nodes of `layer` (but not to alter the nodes of `layer`), provided every descendant of each node of `layer` remains a descendant of that node.

If all time solvers follow these rules, then meets that do not lie in nodes will never be visited by them. The recommended convention is that meets should not lie in nodes if and only if they already have assignments that should never be changed.

Time assignment solvers (and solvers generally) are free to use the back pointers of the solution entities they target. However, since there is potential for conflict here when one solver calls another, the following conventions are recommended.

If solver `S` does not use back pointers (if it never sets any), then this should be documented, and solvers that call `S` may assume that back pointers will be unaffected by it. If `S` uses back pointers (if it sets at least one), then this should be documented, and solvers that call `S` must assume that back pointers in the solution objects targeted by `S` will not be preserved. As a safety measure, solvers should set the back pointers that they have used to `NULL` before returning.

## 10.2. Helper functions

The functions presented in this section assign and unassign meets, but are not complete time solvers in themselves. Instead, they are helper functions that time solvers might find useful.

### 10.2.1. Node assignment functions

This section presents several functions which affect the assignments of the meets of one node.

These functions swap the assignments of the meets of two nodes:

```
bool KheNodeMeetSwapCheck(KHE_NODE node1, KHE_NODE node2);
bool KheNodeMeetSwap(KHE_NODE node1, KHE_NODE node2);
```

Both `node1` and `node2` must be non-`NULL`. Both functions return `true` if the nodes have the same number of meets, and a sequence of `KheMeetSwap` operations applied to corresponding meets would succeed. `KheNodeMeetSwapCheck` just makes the check, while `KheNodeMeetSwap` performs the meet swaps as well. If `node1` and `node2` are the identical same node, `false` is returned. As usual when swapping, the code fragment

```
if( KheNodeMeetSwap(node1, node2) )
  KheNodeMeetSwap(node1, node2);
```

is guaranteed to change nothing, whether the first swap succeeds or not.

To maximize the chances of success it is naturally best to sort the meets before calling these functions, probably like this:

```
KheNodeMeetSort(node1, &KheMeetDecreasingDurationCmp);
KheNodeMeetSort(node2, &KheMeetDecreasingDurationCmp);
```

This sorting has been omitted from `KheNodeMeetSwapCheck` and `KheNodeMeetSwap` for efficiency, since each node's meets need to be sorted only once, yet the node may be swapped many times. The user is expected to sort the meets of every relevant node, perhaps like this:

```
for( i = 0;  i < KheSolnNodeCount(soln);  i++ )
  KheNodeMeetSort(KheSolnNode(soln, i), &KheMeetDecreasingDurationCmp);
```

before any swapping begins. Some other functions, for example `KheNodeRegular` (Section 5.2), also sort meets, so care is needed.

These functions propagate one node's assignments to another:

```
bool KheNodeMeetRegularAssignCheck(KHE_NODE node, KHE_NODE sibling_node);
bool KheNodeMeetRegularAssign(KHE_NODE node, KHE_NODE sibling_node);
```

`KheNodeMeetRegularAssignCheck` calls `KheNodeMeetRegular` (Section 5.2) to check that the two nodes are regular, and if they are, it goes on to check that each meet in `sibling_node` is assigned, and that each meet of `node` is either already assigned to the same meet and offset that the corresponding meet of `sibling_node` is assigned to, or else may be assigned to that meet and offset. `KheNodeMeetRegularAssign` makes all these checks too, and then carries out the assignments if the checks all pass.

To unassign all the meets of `node`, call

```
void KheNodeMeetUnAssign(KHE_NODE node);
```

Even preassigned meets are unassigned, so some care is needed here.

### 10.2.2. Kempe and ejecting meet moves

The *Kempe meet move* is a well-known generalization of moves and swaps. It originates as a move of one meet, say from time $t_1$ to time $t_2$ (in reality, from one meet and offset to another meet and offset). If this initial move creates clashes with other meets, then they are moved from $t_2$ to $t_1$. If that in turn creates clashes with other meets, then they are moved from $t_1$ to $t_2$, and so on until all clashes are removed. The result is usually a move or swap, but it can be more complex.

The Kempe meet move is not unlike an ejection chain algorithm. Instead of removing a single defect at each step, it removes an arbitrary number, but it tries only one repair: moving to $t_2$ on odd-numbered steps and to $t_1$ on even-numbered steps.

Suppose the original meet $m_1$ has duration $d_1$. Usually, the Kempe meet move only moves meets of duration $d_1$, and only from $t_1$ to $t_2$ (on odd-numbered steps) and from $t_2$ to $t_1$ (on even-numbered steps). However, when $m_1$ is being moved to a different offset in the same target meet, the Kempe meet move does not commit itself to this until it has examined the first meet, call it $m_2$, which has to be moved on the second step. If $m_2$ was immediately adjacent to $m_1$ in time before $m_1$ was moved on the first step, it is acceptable for $m_2$ to have a duration $d_2$ which is different from $d_1$. In that case, all meets moved on odd-numbered steps must have duration $d_1$, and all meets moved on even-numbered steps must have duration $d_2$, and each meet is moved to the opposite end of the block of adjacent times that $m_1$ and $m_2$ were together assigned to originally.

Kempe meet moves need to know what clashes they have caused. Clashes occur between preassigned tasks. So the first step is to search the meet being moved, and if necessary the meets assigned to that meet (and so on recursively) for the first *preassigned task*: a task derived from a preassigned event resource. If there are no preassigned tasks, there can be no clashes. In that case, the Kempe meet move operation does exactly what an ordinary meet move would do.

If there is a first preassigned task, then clashes are possible and must be detected. This is done via the matching, partly because it is the fastest way, and partly because it works at any level of the layer tree, unlike avoid clashes monitors, which work only at the root. Accordingly, the matching must be present, as witnessed by the presence of a first demand monitor in the first preassigned task of the meet to be moved. If this demand monitor is not present, a Kempe move is not possible, and the operation returns `false`.

Furthermore, preassigned demand monitors must be attached, and grouped (directly or indirectly) under a group monitor with sub-tag `KHE_SUBTAG_KEMPE_DEMAND`, by calling

```
KHE_GROUP_MONITOR KheKempeDemandGroupMonitorMake(KHE_SOLN soln);
```

before making any Kempe meet moves. This is a focus grouping, as defined in Section 8.7.2. The group monitor's children are the ordinary demand monitors of the preassigned tasks of `soln`. No primary groupings are relevant here so primary group monitors never replace the ordinary demand monitors. The operation will abort if it cannot find a group monitor with this sub-tag among the parents of the first demand monitor of the first preassigned task.

Use of the matching raises the question of whether Kempe meet moves should try to remove demand defects other than *simple clashes*: clashes involving a resource which possesses a hard avoid clashes constraint which is preassigned to two meets which are running at the same time. The author's view is that it should not. When there is a simple clash caused by one meet moving to a time, the only possible resolution is for the other to move away. With demand defects in general, there may be multi-way clashes which can be resolved by moving one of several meets away, and that is not what the Kempe meet move is about.

Assuming that the grouping is done correctly, then, a call to

```
bool KheKempeMeetMove(KHE_MEET meet, KHE_MEET target_meet,
  int offset, bool preserve_regularity, int *demand, bool *basic,
  KHE_KEMPE_STATS kempe_stats);
```

will make a Kempe meet move. It is similar to `KheMeetMove` in moving the current assignment of `meet` to `target_meet` at `offset`, but it requires `meet` to be already assigned so that it knows where to move clashing meets back to. It does not use back pointers or visit numbers. It sets `*demand` to the total demand of the meets it moves, to give the caller some idea of the disruption it caused, and it sets `*basic` to `true` if it did not find any meets that needed to be moved back the other way, so that what it did was just a basic meet move. The `kempe_stats` parameter is used for collecting statistics about Kempe meet moves, as described below; it may be `NULL` if statistics are not wanted. There is also

```
bool KheKempeMeetMoveTime(KHE_MEET meet, KHE_TIME t,
  bool preserve_regularity, int *demand, bool *basic,
  KHE_KEMPE_STATS kempe_stats);
```

which moves `meet` to the cycle meet and offset representing time `t`.

If `preserve_regularity` is `false`, these functions ignore zones. One way to take zones into account is to call `KheMeetMovePreservesZones` (Section 5.4) first. In theory this is inadequate when meets of different durations are moved, but the inadequacy will virtually never arise in practice. The other way is to set `preserve_regularity` to `true`, and then the functions will use `KheNodeIrregularity` (Section 5.4) to measure the irregularity of the nodes affected, before and after; the operation will fail if the total irregularity of the nodes affected has increased.

`KheKempeMeetMove` succeeds, returning `true`, if it moves `meet` to `target_meet` at `offset`, possibly moving other meets as well, to ensure that the final state has no new simple clashes and no new cases of a preassigned resource attending a meet at a time when it is unavailable. It fails, returning `false`, in these cases:

- The matching is not present.
- Some call to `KheMeetMove`, which is used to make the individual moves, returns `false`. This includes the case where `meet` is already assigned to `target_meet` at `offset`, which, as previously documented, is defined to fail for the practical reason that the move accomplishes nothing and pursuing it can only waste time.
- Moving some meet makes some preassigned resource busy when it is unavailable.
- A meet which needs to be moved is not currently assigned to the expected target meet (either `meet`'s original target meet or `target_meet`, depending on whether the current step is odd or even), or has the wrong duration or offset. This prevents the changes from spreading beyond the expected area of the solution.
- `preserve_regularity` is `true` but the operation increases irregularity (discussed above).
- Some meet needs to be moved, but it has already moved during this operation, indicating that the classical graph colouring reason for failure has occurred.

If `KheKempeMeetMove` fails, it leaves the solution in the state it was in at the failure point. In practice, it must be enclosed in `KheMarkBegin` and `KheMarkEnd` (Section 4.8), so that undoing can be used to clean up the mess. This could easily have been incorporated into `KheKempeMeetMove`, producing a version that left the solution unchanged if it failed. However, the caller will probably want to enclose the operation in `KheMarkBegin` and `KheMarkEnd` anyway, since it may need to be undone for other reasons, so cleanup is left to the caller.

The `kempe_stats` parameter is an object (the usual pointer to a private record) used to record statistics about Kempe meet moves. If statistics are wanted, then to create and delete a Kempe stats object, call

```
KHE_KEMPE_STATS KheKempeStatsMake(HA_ARENA a);
void KheKempeStatsDelete(KHE_KEMPE_STATS kempe_stats);
```

Actually the usual way to obtain a `KHE_KEMPE_STATS` object is from the `ts_kempe_stats` option, via a call to

```
KHE_KEMPE_STATS KheKempeStatsOption(KHE_OPTIONS options, char *key);
```

with key `"ts_kempe_stats"`. This returns the Kempe stats object stored under `key`, first creating it with `KheKempeStatsMake` and adding it to the options object if it is not present.

Each time a Kempe stats object is passed to a successful call to `KheKempeMeetMove` or `KheKempeMeetMoveTime`, its statistics are updated. They can be retrieved at any time using the following functions.

A *step* of a Kempe meet move is a move of one meet. The statistics include a histogram of the number of successful Kempe meet moves with `step_count` steps, for each `step_count`, retrievable by calling

```
int KheKempeStatsStepHistoMax(KHE_KEMPE_STATS kempe_stats);
int KheKempeStatsStepHistoFrequency(KHE_KEMPE_STATS kempe_stats,
  int step_count);
int KheKempeStatsStepHistoTotal(KHE_KEMPE_STATS kempe_stats);
float KheKempeStatsStepHistoAverage(KHE_KEMPE_STATS kempe_stats);
```

These return the maximum `step_count` for which there is at least one Kempe meet move, or `0` if none; the number of Kempe meet moves with `step_count` steps; the total number of steps over all Kempe meet moves; and the average number of steps. This last is only safe to call if `KheKempeStatsStepHistoTotal > 0`.

A *phase* of a Kempe meet move is a move of one or more meets in one direction. For example, a Kempe move that turns out to be an ordinary move has one phase; one that turns out to move one meet in one direction, then two in the other, has two phases; and so on. The statistics include a histogram of the number of successful Kempe meet moves with `phase_count` phases, for each `phase_count`, retrievable by calling

```
int KheKempeStatsPhaseHistoMax(KHE_KEMPE_STATS kempe_stats);
int KheKempeStatsPhaseHistoFrequency(KHE_KEMPE_STATS kempe_stats,
  int phase_count);
int KheKempeStatsPhaseHistoTotal(KHE_KEMPE_STATS kempe_stats);
float KheKempeStatsPhaseHistoAverage(KHE_KEMPE_STATS kempe_stats);
```

These return the maximum `phase_count` for which there is at least one Kempe meet move, or `0` if none; the number of Kempe meet moves with `phase_count` phases; the total number of phases over all Kempe meet moves; and the average number of phases. This last is only safe to call if `KheKempeStatsPhaseHistoTotal > 0`.

Functions

```
bool KheEjectingMeetMove(KHE_MEET meet, KHE_MEET target_meet, int offset,
  bool allow_eject, bool preserve_regularity, int *demand, bool *basic);
bool KheEjectingMeetMoveTime(KHE_MEET meet, KHE_TIME t,
  bool allow_eject, bool preserve_regularity, int *demand, bool *basic);
```

offer a variant of the Kempe meet move called the *ejecting meet move*. This begins by moving `meet` to `target_meet` at `offset`, and then finds the meets that need to be moved back the other way exactly as for Kempe meet moves (using the same group monitor), but instead of moving them, it unassigns them and stops. This is when `allow_eject` is `true`; when `allow_eject` is `false`, if any meets need to be ejected, instead of doing that the function returns `false`. `KheEjectingMeetMove` does not require `meet` to be assigned initially (the move may be an assignment), not does it carry out any checking of the durations and offsets of the meets it unassigns. All other details are as for Kempe meet moves. Similarly,

```
bool KheBasicMeetMove(KHE_MEET meet, KHE_MEET target_meet,
  int offset, bool preserve_regularity, int *demand);
bool KheBasicMeetMoveTime(KHE_MEET meet, KHE_TIME t,
  bool preserve_regularity, int *demand);
```

are variants in which even the unassignments are omitted. They are the same as `KheMeetMove` and `KheMeetMoveTime` as far as changing the solution goes, differing from them only in optionally preserving regularity, and in reporting demand. No group monitor is needed.

Finally, functions

```
bool KheTypedMeetMove(KHE_MEET meet, KHE_MEET target_meet, int offset,
  KHE_MOVE_TYPE mt, bool preserve_regularity, int *demand, bool *basic,
  KHE_KEMPE_STATS kempe_stats);
bool KheTypedMeetMoveTime(KHE_MEET meet, KHE_TIME t,
  KHE_MOVE_TYPE mt, bool preserve_regularity, int *demand, bool *basic,
  KHE_KEMPE_STATS kempe_stats);
```

allow the type of move (unchecked, checked, ejecting, or Kempe) to be selected on the fly, using parameter `mt`, which has type

```
typedef enum {
  KHE_MOVE_UNCHECKED,
  KHE_MOVE_CHECKED,
  KHE_MOVE_EJECTING,
  KHE_MOVE_KEMPE,
} KHE_MOVE_TYPE;
```

Unchecked means basic, checked means ejecting with `false` for `allow_eject`, ejecting means ejecting with `true` for `allow_eject`, and Kempe means Kempe. These functions switch on `mt` and call the appropriate variant. The `kempe_stats` parameter is only passed to Kempe moves.

The rest of this section describes `KheKempeMeetMove`'s implementation. It is an important operation, so its implementation must be robust, and must squeeze every drop of utility out of the basic idea. `KheEjectingMeetMove` is just a cut-down version of `KheKempeMeetMove`.

A *frame* (nothing to do with type `KHE_FRAME`) is a set of adjacent positions in a target meet, defined by the target meet, a start offset into the target meet, and a stop offset, which may equal the duration of the target meet, but be no larger. The set of positions runs from the start offset inclusive to the stop offset exclusive. A meet *lies in* a frame when it is assigned to that frame's target meet, and the set of positions it occupies in that target meet is a subset of the set of positions defined by the frame.

The Kempe meet move operation defines four frames. On odd-numbered steps, including the move of the original meet, every move is of a meet lying in a frame called the *odd-from frame* to a frame called the *odd-to frame*. Similarly, every meet move on even-numbered steps is from the *even-from frame* to the *even-to frame*.

The odd-from frame and the odd-to frame have the same duration, and the even-from frame and the even-to frame have the same duration. When a meet is moved, its new target meet is the target meet of the to frame of its step, and its offset in that target meet is defined by requiring its offset in its to frame to equal its former offset in its from frame. This completely determines where the meet is moved to, and ensures that the timetable of moved meets is replicated in the to frame exactly as it was in the from frame.

The implementation will now be described, assuming that the four frames are given. How they are defined will be described later.

First, if there are no preassigned tasks within `meet` or within meets assigned to `meet`, directly or indirectly, then `KheKempeMeetMove` calls `KheMeetMove` and returns its result. Otherwise, it finds the group monitor it needs as described above and begins to trace it. It then carries out a sequence of steps. As each step begins, there is a given set of meets to move, and the step tries to move them. An empty set signals success.

On odd-numbered steps, `KheKempeMeetMove` moves the given set of meets from their offsets in the odd-from frame to the same offsets in the odd-to frame. This will fail if any of the meets do not lie entirely within the odd-from frame, and if any call to `KheMeetMove` returns `false`. Even-numbered steps are the same, using the even-from frame and even-to frame.

The set of meets to move on the first step contains just `meet`. At the end of each step, the set of meets for the next step is found, as follows. The monitor trace is used to find the preassigned demand monitors whose cost increased during the current step. For each of these monitors, `KheMonitorFirstCompetitor` and `KheMonitorNextCompetitor` (Section 7.4.3) are used to find the demand monitors competing with them for supply. These can be of four kinds:

1. A workload demand monitor derived from an avoid unavailable times monitor signals that a preassigned resource has moved to an unavailable time, so fail.

2. Any other workload demand monitor signals a workload overload other than an unavailable time, so ignore it. At a higher level, this defect might cause failure, but, as explained above, the Kempe meet move itself only takes notice of simple clashes and unavailabilities.

3. A demand monitor derived from an unpreassigned task does not signal a simple clash, so ignore it, on the same reasoning as the previous item.

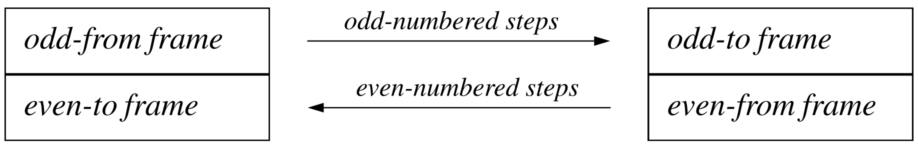
4. A demand monitor derived from a preassigned task signals a simple clash. The appropriate enclosing meet of the task (the one on the chain of assignments leading out of the task's meet just before the expected target meet) is found. If there is no such meet, or it was moved on a previous step, fail. If it was moved on the current step, or is already scheduled to move on the next step, ignore it. Otherwise schedule it to be moved on the next step.

A task is taken to be preassigned when a call to `KheTaskIsPreassigned` (Section 4.6.3), with `as_in_event_resource` set to `false`, returns `true`.

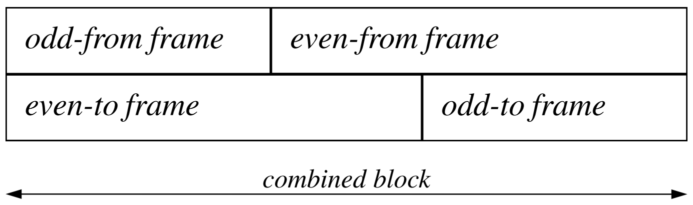
It remains to explain how the four frames are defined.

Given the call `KheKempeMeetMove(meet, target_meet, offset, ...)`, the target meet of the odd-from frame and the even-to frame is `KheMeetAsst(meet)`, and the target meet of the even-from frame and the odd-to frame is `target_meet`. These may be equal, or not.

The odd frames have the same duration, and the even frames have the same duration. Usually, all frames have the same duration, the odd-from frame and the even-to frame are equal, and the even-from frame and the odd-to frame are equal. This is the *separate case*:



But there is another possibility, the *combined case*. Suppose the odd-from frame and the even-from frame are adjacent in time (suppose they have the same target meet, and the start offset of either equals the stop offset of the other). Call the union of their two sets of offsets the *combined block*. In that case, the durations of the odd-from frame and the even-from frame may differ. The odd-to frame occupies the opposite end of the combined block from the odd-from frame, and the even-to frame occupies the opposite end from the even-from frame:



Four diagrams could be drawn here, showing cases where the odd-from frame has shorter and longer duration than the even-from frame, and where it appears to the left and right of the even-from frame. But in all these cases, meets move between the frames in the same way.

To find these frames, first make the initial move of `meet` to `target_meet` at `offset`. This is an odd-numbered move, so it moves a meet from the odd-from frame to the odd-to frame. But it is defined by the caller, so no frames are needed. If it fails, then fail. Otherwise, find the resulting clashing meets. This may cause failure in various cases, as explained above; if successful, all the clashing meets will currently be assigned to `target_meet` at various offsets. If there are no clashing meets, the initial move suffices, so return success. Otherwise, let the *initial clash frame* be the smallest frame enclosing the clashing meets. The even-from frame will be a superset of this frame, to allow all the clashing meets to move legally on the second step.

Next, see whether the separate case applies, as follows. The initial meet must lie inside the odd-to frame after it moves. Since the even-from frame must equal the odd-to frame in the separate case, let the even-from frame be the initial clash frame, enlarged as little as possible to include the initial meet after it moves. Then the odd-from frame is defined completely by the requirements that its duration must equal the duration of the even-from frame, and that the offset of the initial meet in the odd-from frame before it moves must equal its offset in the odd-to frame, and so in the even-from frame, after it moves. Once the odd-from frame is defined in this way, check that it does not protrude out either end of its target meet, nor overlap with the even-from frame. If it passes this check, set the odd-to frame equal to the even-from frame, and set the even-to frame equal to the odd-from frame. The separate case applies.

Otherwise, see whether the combined case applies, as follows. If the initial meet's original target meet is not `target_meet`, or its original position overlaps the initial clash frame, then the combined case does not apply, and so the entire operation fails. Otherwise, set the even-from frame to the initial clash frame, and set the odd-from frame to the smallest frame which both includes the initial meet's original position and also abuts the even-from frame. This frame must exist; no further checks are needed. Set the odd-to frame to occupy the opposite end of the combined block from the the odd-from frame, and set the even-to frame to occupy the opposite end of the combined block from the even-from frame. The combined case applies.

## 10.3. Meet bound groups and domain reduction

The functions described in this section do not assign meets. Instead, they reduce meet domains.

### 10.3.1. Meet bound groups

Meet domains are reduced by adding meet bound objects to meets (Section 4.5.4). Frequently, meet bound objects need to be stored somewhere where they can be found and deleted later. The required data structure is trivial—just an array of meet bounds—but it is convenient to have a standard for it, so KHE defines a type `KHE_MEET_BOUND_GROUP` with suitable operations.

To create a meet bound group, call

```
KHE_MEET_BOUND_GROUP KheMeetBoundGroupMake(KHE_SOLN soln);
```

To add a meet bound to a meet bound group, call

```
void KheMeetBoundGroupAddMeetBound(KHE_MEET_BOUND_GROUP mbg,
  KHE_MEET_BOUND mb);
```

To visit the meet bounds of a meet bound group, call

```
int KheMeetBoundGroupMeetBoundCount(KHE_MEET_BOUND_GROUP mbg);
KHE_MEET_BOUND KheMeetBoundGroupMeetBound(KHE_MEET_BOUND_GROUP mbg, int i);
```

To delete a meet bound group, including deleting all the meet bounds in it, call

```
bool KheMeetBoundGroupDelete(KHE_MEET_BOUND_GROUP mbg);
```

This function returns `true` when every call it makes to `KheMeetBoundDelete` returns `true`.

### 10.3.2. Exposing resource unavailability

If a meet contains a preassigned resource with some unavailable times, run times will be reduced if those times are removed from the meet's domain, since then futile time assignments will be ruled out quickly. This idea is implemented by

```
void KheMeetAddUnavailableBound(KHE_MEET meet, KHE_COST min_weight,
  KHE_MEET_BOUND_GROUP mbg);
```

This makes a meet bound based on the available times of the resources preassigned to `meet` and to meets with fixed assignments to `meet`, directly or indirectly. It adds this bound to `meet`, and to `mbg` if `mbg` is non-`NULL`.

The meet bound is an occupancy bound whose default time group is the full cycle minus `KheAvoidUnavailableTimesConstraintUnavailableTimes(c)` for each avoid unavailable times constraint `c` for the relevant resources whose combined weight is at least `min_weight`. For example, setting `min_weight` to `0` includes all constraints; setting it to `KheCost(1, 0)` includes hard constraints only. Each time group is adjusted for the offset in `meet` of the meet containing the preassigned resource. If the resulting time group is the entire cycle, as it will be, for example, when `meet`'s preassigned resources are always available, then no meet bound is made.

There is also

```
void KheSolnAddUnavailableBounds(KHE_SOLN soln, KHE_COST min_weight,
  KHE_MEET_BOUND_GROUP mbg);
```

which calls `KheMeetAddUnavailableBound` for each non-cycle meet in `soln` whose assignment is not fixed, taking care to visit the meets in a safe order (parents before children).

### 10.3.3. Preventing cluster busy times and limit idle times defects

This section presents a function which reduces the cost of cluster busy times and limit idle times monitors, by reducing heuristically the domains of the meets to which the monitors' resources are preassigned, before time assignment begins. For example, suppose teacher Jones is limited by a cluster busy times constraint to attend for at most three of the five days of the week. Choose any three days and reduce the time domains of the meets that Jones is preassigned to to those three days. Then those meets cannot cause a cluster busy times defect for Jones.

But first, we need to consider the alternatives. One is to do nothing special during the initial time assignment, and repair any defects later. But there are likely to be many defects then, casting doubt on the value of the initial assignment, since repairing cluster busy times defects is time-consuming and difficult. Repairing limit idle times defects is easier, but it still takes time.

A second alternative is to take these monitors into account as part of the usual method of constructing an initial assignment of times to meets. The usual method is to group the meets into layers (sets of meets which must be disjoint in time, because they share preassigned resources) and assign the layers in turn. Some monitors are handled during layer assignment, including demand and spread events monitors. Cluster busy times monitors can be too, as follows.

Suppose there is a cluster busy times monitor for resource *r* requiring that *r* be busy on at most four of the five days of the cycle. Create a meet with duration equal to the number of times in one day, whose domain is the set of first times on all days. Add a task preassigned *r* to this meet. Then, in the course of assigning *r*'s layer, this meet will be assigned a time, and if there are no clashes, the other meets preassigned *r* will be limited to at most four days as required. At the author's university, this method is used to give most students two half-days off.

There are a few detailed problems: a whole-day meet may not be assignable to any cycle meet, and the author's best method of assigning the meets of one layer (Section 10.6) works best when there are several meets of each duration, whereas here there may be only one whole-day meet. These problems can be surmounted by reducing the domains of the other meets instead of adding a new meet. But there are other problems—problems that may be called fundamental, because they arise from handling clustering one layer at a time.

A resource is *lightly loaded* when it is preassigned to meets whose total duration is much less than the cycle's duration. Cluster busy times monitors naturally apply to lightly loaded resources, because heavily loaded ones don't have the free time that makes clustering desirable. In university problems, each layer is a set of meets preassigned just one resource: a lightly loaded student. The layers are fairly independent, being mutually constrained only by the capacities of class sections. Under these conditions, handling clustering one layer at a time works well.

But now consider the situation, common in high schools, where each meet contains two preassigned resources, one student group resource and one teacher resource. Suppose the student group resources are heavily loaded, and the teacher resources are lightly loaded and subject to cluster busy times constraints. It is best to timetable the meets one student group layer at a time, because the student group resources are heavily loaded, but this leaves no place to handle the teachers' cluster busy times monitors. Even if the meets were assigned in teacher layers, those layers are often not independent: electives, for example, have several simultaneous meets, requiring several teachers to have common available times.

This brings us to the third alternative, the subject of this section. Before time assignment begins, reduce the domains of meets subject to cluster busy times and limit idle times monitors to guarantee that the monitors have low (or zero) cost, whatever times are assigned later. Use the global tixel matching to avoid mistakes which would make meets unassignable. Function

```
void KheSolnClusterAndLimitMeetDomains(KHE_SOLN soln,
  KHE_COST min_cluster_weight, KHE_COST min_idle_weight,
  float slack, KHE_MEET_BOUND_GROUP mbg, KHE_OPTIONS options);
```

does this. It adds meet bounds to meets, and to `mbg` if `mbg` is non-`NULL`, based on cluster busy times monitors with combined weight at least `min_cluster_weight`, and on limit idle times monitors with combined weight at least `min_idle_weight`. `Minimum` limits are ignored. See below for precisely which monitors are included. If `KheOptionsDiversify(options)` is `true`, the result is diversified by varying the order in which domain reductions for limit idle times monitors are tried.

Carrying out all possible domain reductions is almost certainly too extreme; it gives other solvers no room to move. Parameter `slack` is offered to avoid this problem. For each resource $r$, function `KheSolnClusterAndLimitMeetDomains` keeps track of $p(r)$, the total duration of the events preassigned $r$, and $a(r)$, the total duration of the times available to these events, given the reductions made so far. Clearly, it is important for the function to ensure $a(r) \geq p(r)$, since otherwise these events will not have room to be assigned. But, letting $s$ be the value of `slack`, the function actually ensures $a(r) \geq s \cdot p(r)$, or rather, it does not apply any reduction that makes this condition `false`. The minimum acceptable value of `slack` is `1.0`, which is almost certainly too small. A value around `1.5` seems more reasonable.

The remainder of this section describes the issues involved in reducing domains, and how `KheSolnClusterAndLimitMeetDomains` works in detail.

A set of resources may be *time-equivalent*: sure to be busy at the same times. There would be no change in cost if all the cluster busy times and limit idle times monitors of a set of time-equivalent resources applied to just one of them: their costs depend only on when their resource is busy. So although for simplicity the following discussion speaks of individual resources, in fact `KheSolnClusterAndLimitMeetDomains` deals with sets of time-equivalent resources, taken from the `time_equiv` option of its `options` parameter. It obtains this by calling `KheTimeEquivOption` (Section 9.2), which creates the option if it is not already present.

A cluster busy times monitor for a resource `r` is included when its combined weight is at least `min_cluster_weight`, its `Maximum` limit is less than its number of time groups, and each time group is either disjoint from or equal to each time group of each previously included monitor for `r`. A limit idle times monitor for a resource `r` of type `rt` is included when its combined weight is at least `min_idle_weight`, `rt` satisfies `KheResourceTypeDemandIsAllPreassigned(rt)`, its time groups are disjoint from each other, and each time group is either disjoint from or equal to each time group of each previously included monitor for that resource. The time groups are usually days, so the disjoint-or-equal requirement is usually no impediment.

An *exclusion operation*, or just *exclusion*, is the addition of an occupancy meet bound (Section 4.5.4) to each meet preassigned a given resource, ensuring that those meets do not overlap a given set of times. An exclusion is *successful* if its calls to `KheMeetAddMeetBound` succeed and do not increase the number of unmatched demand tixels in the global tixel matching. `KheSolnClusterAndLimitMeetDomains` keeps only successful exclusions; unsuccessful ones are tried, then undone. It repeatedly tries exclusions until for each monitor, either a guarantee of sufficiently low cost is obtained, or no further successful exclusions are available. Exclusions based on cluster busy times monitors are tried first, since they are most important. After they have all been tried, the algorithm switches to exclusions based on limit idle times monitors.

Build a graph with one vertex for each resource. For each resource, the aim is to exclude some of its cluster busy times monitors' time groups from its meets, enough to satisfy those monitors' `Maximum` limits. Thinking of each time group as a colour, the aim is to assign a given number of distinct colours from a given set to each vertex.

If some meet (or set of linked meets) has several preassigned resources, those resources should exclude some of the same time groups, to leave others available. Linked meets with preassigned teachers $a$, $b$, $c$, $d$, and $e$ must not be excluded from Mondays by $a$, from Tuesdays by $b$, and so on. The global tixel matching test prevents this extreme example, but we also need to avoid even approaching it. So when two resources share meets, this evidence that they should have similar exclusions is recorded by connecting their vertices by a *positive edge* whose cost is the total duration of the meets they share.

Even when two resources share no meets, they may still influence each other's exclusions, when there is an intermediate resource which shares meets with both of them. Two teachers who teach the same student group are an example of this. If some time group is excluded by one of the teachers, it would be better if it was not excluded by the other, since that again limits choice. In this case the two resources' vertices are joined by a *negative edge* whose cost is the total duration of the meets they share with the intermediate resource. If there are several intermediate resources, the maximum of their costs is used.

Negative edges produce a soft graph colouring problem: a good result gives overlapping sets of colours to vertices connected by positive edges, and disjoint sets of colours to vertices connected by negative edges. This connection with graph colouring rules out finding an optimum solution quickly, but it also suggests a simple heuristic which is likely to work well, since it is based on the successful saturation degree heuristic for graph colouring.

A vertex is *open* when $a(v) > s \cdot p(v)$ (as explained above), and it has at least one untried exclusion with at least one cluster busy times monitor which would benefit from that exclusion. If there are no open vertices, the procedure ends. Otherwise an open vertex is chosen for colouring whose total cost of edges (positive and negative) going to partly or completely coloured vertices is maximum, with ties broken in favour of vertices of larger degree.

Once an open vertex is chosen, the cost of each of its untried colours is found, and the untried colours are tried in order of increasing cost until one of them succeeds or all have been tried. The cost of a colour $c$ is the total cost of outgoing negative edges to vertices containing $c$, minus the total cost of outgoing positive edges to vertices containing $c$.

The numbers used by the heuristic are adjusted to take account of the idea that one vertex requiring several colours is similar to several vertices, each requiring one colour, and connected in a clique by strongly negative edges. In particular, being partly coloured increases a vertex's chance of being chosen for colouring, as does requiring more than one more colour.

Saturation degree heuristics are often initialized by finding and colouring a large clique, but nothing of that kind is attempted here. A time group which is a subset of the unavailable times of its resource should always be excluded. This is done, wherever applicable, at the start, after which there may be several partly coloured vertices.

When handling limit idle times monitors, individual times are excluded instead of entire time groups. The time groups of limit idle times monitors are compact, and the excluded times lie at the start or end of one of these time groups. Exclusions which remove a last unexcluded time are tried first, followed by exclusions which remove a first unexcluded time.

Whether an idle exclusion is needed depends on the following calculation. As above, let the *preassigned duration* $p(v)$ of a vertex $v$ be the total duration of the meets that $v$'s resource is preassigned to. Let the *availability* $a(v)$ of vertex $v$ be the number of times that these same meets may occupy. Initially this is the number of times in the cycle, but as time groups are excluded during the cluster busy times phase it shrinks, and then as individual times are excluded during the limit idle times monitor phase it shrinks further.

As explained above, when an exclusion would cause $a(v) \geq s \cdot p(v)$ to become `false`, it is prevented. Assuming this obstacle is not present, consider limit idle times monitor $m$ within $v$. A worst-case estimate of its number of deviations $d(m)$ can be found as follows.

Let $a(m)$, the *availability* of $m$, be the total number of unexcluded times in $m$'s time groups. Since time groups are disjoint, $a(m) \leq a(v)$. The worst case for $m$ occurs when as many meets as possible are assigned times outside its time groups, leaving many unassigned and potentially idle times inside. The maximum duration of meets that can be assigned outside $m$'s time groups is $a(v) - a(m)$, leaving a minimum duration of

$$MD(m) = \max(0, p(v) - (a(v) - a(m)))$$

to be assigned within $m$'s time groups. This assignment leaves $a(m) - MD(m)$ of $m$'s available places unfilled. A little algebra shows that this difference is non-negative, given $a(v) \geq p(v)$.

Let $M(m)$ be $m$'s `Maximum` attribute. The worst-case deviation $d(m)$ is the amount by which the number of unfilled places exceeds $M(m)$, that is,

$$d(m) = \max(0, a(m) - MD(m) - M(m))$$

If $d(m)$ is positive, an exclusion which reduces $a(m)$ further may be tried, and multiplying $d(m)$ by $w(m)$, the combined weight of $m$'s constraint, gives a priority for trying such an exclusion.

Limit idle times monitors are tried in decreasing $d(m)w(m)$ order, updated dynamically, and modified by propagating exclusions across positive edges. Negative edges are not used.

## 10.4. Some basic time solvers

This section presents some basic time solvers. The simplest are

```
bool KheNodeSimpleAssignTimes(KHE_NODE parent_node, KHE_OPTIONS options);
bool KheLayerSimpleAssignTimes(KHE_LAYER layer, KHE_OPTIONS options);
```

They assign those meets of the child nodes of `parent_node` (or of the nodes of `layer`) that are not already assigned. For each such meet, in decreasing duration order, they try all offsets in all meets of the parent node. If `KheMeetAssignCheck` permits at least one of these, the best is made, measuring badness by calling `KheSolnCost`; otherwise the meet remains unassigned, and the result returned will be `false`. These functions do not use options or back pointers.

There is one wrinkle. When assigning a meet which is derived from an event `e`, these functions will not assign the meet to a meet which is already the target of an assignment of some other meet derived from `e`. This is because if two meets from the same event are assigned to the same meet, they are locked into being adjacent, or almost adjacent, in time, undermining the only possible motive for splitting them apart.

These functions are not intended for serious timetabling. They are useful for simple tasks: assigning nodes whose children are known to be trivially assignable, finding minimum runaround durations (Section 9.4.1), and so on.

The logical order to assign times to the nodes of a layer tree is postorder (from the bottom up), since until a node's children are assigned to it, its resource demands are not clear. Function

```
bool KheNodeRecursiveAssignTimes(KHE_NODE parent_node,
  KHE_NODE_TIME_SOLVER solver, KHE_OPTIONS options);
```

applies `solver` to all the nodes in the subtree rooted at `parent_node`, in postorder. It returns `true` when every call it makes on `solver` returns `true`. It uses options and back pointers if and only if `solver` uses them. For example,

```
KheNodeRecursiveAssignTimes(parent_node, &KheNodeSimpleAssignTimes, NULL);
```

carries out a simple assignment at each node, and

```
KheNodeRecursiveAssignTimes(parent_node, &KheNodeUnAssignTimes, NULL);
```

unassigns all meets in all proper descendants of `parent_node`.

Functions

```
bool KheNodeUnAssignTimes(KHE_NODE parent_node, KHE_OPTIONS options);
bool KheLayerUnAssignTimes(KHE_LAYER layer, KHE_OPTIONS options);
```

unassign any assigned meets of `parent_node`'s child nodes (or of `layer`'s nodes). They do not use options or back pointers. Also,

```
bool KheNodeAllChildMeetsAssigned(KHE_NODE parent_node);
bool KheLayerAllChildMeetsAssigned(KHE_LAYER layer);
```

return `true` when the meets of the child nodes of `parent_node` (or of `layer`) are all assigned.

Preassigned meets could be assigned separately first, then left out of nodes so that they are not visited by time assignment algorithms. The problem with this is that a few times may be preassigned to obtain various effects, such as Mathematics first in the day, and this should not affect the way that forms are coordinated. Accordingly, the author favours handling preassigned meets along with other meets, as far as possible.

However, when coordination is complete and real time assignment begins, it seems best to assign preassigned meets first, for two reasons. First, preassignments are special because they have effectively infinite weight. There is no point in searching for alternatives. Second, preassignments cannot be handled by algorithms that are guided by total cost, because they have no assign time constraints, so there is no reduction in cost when they are assigned. Functions

```
bool KheNodePreassignedAssignTimes(KHE_NODE root_node,
  KHE_OPTIONS options);
bool KheLayerPreassignedAssignTimes(KHE_LAYER layer,
  KHE_OPTIONS options);
```

search the child nodes of `root_node`, which must be the overall root node, or the nodes of `layer`, whose parent must be the overall root node, for unassigned meets whose time domains contain exactly one element. `KheMeetAssignTime` is called on each such meet to attempt to assign that one time to the meet, and the result is `true` when all of these calls return `true`. These functions do not use options or back pointers.

KHE's solvers assume that it is always a good thing to assign a time to a meet. However, occasionally there are cases where cost can be reduced by unassigning a meet, because the cost of the resulting assign time defect is less than the total cost of the defects introduced by the assignment. As some acknowledgement of these anomalous cases, KHE offers

```
bool KheSolnTryMeetUnAssignments(KHE_SOLN soln);
```

for use at the end. It tries unassigning each meet of `soln` in turn. If any unassignment reduces the cost of `soln`, it is not reassigned. The result is `true` if any unassignments were kept.

## 10.5. A time solver for runarounds

Time solver

```
bool KheRunaroundNodeAssignTimes(KHE_NODE parent_node,
  KHE_OPTIONS options);
```

assigns times to the unassigned meets of the child nodes of `parent_node`, using an algorithm specialized for runarounds. It tries to spread similar nodes out through `parent_node` as much as possible. By definition, some resources are scarce in runaround nodes, so it is good to spread demands for similar resources as widely as possible. It works well on symmetrical runarounds, but it can fail in more complex cases. If that happens, it undoes its work and makes a call to `KheNodeLayeredAssignTimes(parent_node, false)` from Section 10.8.2. This is not a very appropriate alternative, but any assignment is better than none.

`KheRunaroundNodeAssignTimes` begins by finding the child layers of `parent_node` using `KheNodeChildLayersMake` (Section 9.3.1), and placing similar nodes at corresponding indexes in the layers, using `KheLayerSimilar` (Section 5.3). It then assigns the unassigned meets of these nodes. Its first priority is to not increase solution cost; its second is to avoid assigning two child meets to the same parent meet (this would prevent them from spreading out in time); and its third is to prevent corresponding meets in different layers from overlapping in time.

The algorithm is based on a procedure (let's call it `Solve`) which accepts a set of child layers, each accompanied by a set of triples of the form

```
    (parent_meet, offset, duration)
```

meaning that `parent_meet` is open to assignment by a child meet of the layer, at the given offset and duration. The task of `Solve` is to assign all the unassigned meets of the nodes of its layers.

The initial call to `Solve` is passed all the child layers. Each layer's triples usually contain one triple for each parent meet, with offset 0 and the duration of the parent meet for duration, indicating that the parent meets are completely open for assignment. If any meets are assigned already, the triples are modified accordingly to record the smaller amount of open space.

`Solve` begins by finding the maximum duration, `md`, of an unassigned meet in any of its layers. It assigns all meets with this duration in all layers itself, and then makes recursive calls to assign the meets of smaller duration. For each layer, it takes the meets of duration `md` in the order they appear in the layer and its nodes. It assigns these meets to consecutive suitable positions through the layer, shifting the starting point of the search for suitable positions by one place in the parent layer as it begins each layer. It never makes an assignment which increases the cost of the solution, and it makes an assignment which causes two child meets to be assigned to the same parent meet only as a last resort. If some meet fails to assign, the whole algorithm fails and the problem is passed on to `KheNodeChildLayersAssignTimes` as described above.

As meets are assigned, the offsets and durations of the triples change to reflect the fact that the parent meets are more occupied. After all assignments of meets of duration `md` are complete, the layers are sorted to bring layers with equal triples together. Each set of layers with equal triples is then passed to a recursive call to `Solve`, which assigns its meets of smaller duration.

The purpose of handling sets of layers with equal triples together in this way can be seen in an example. Suppose the parent node has two doubles and each child node has one double. Then there are two ways to assign the child's double; half the child layers will get one of these ways, the other half will get the other way. The layers in each half have identical assignments so far, undesirably but inevitably. By bringing them together we maximize the chance that the recursive call which assigns the singles will find a way to vary the remaining assignments.

## 10.6. Extended layer matching with Elm

A good way to assign times to meets is to group the meets into nodes, group the nodes into layers, and assign times to the meets layer by layer. The advantage of doing it this way is that the meets of one layer strongly constrain each other, because they share preassigned resources so must be disjoint in time. Assigning times to the meets of one layer, then, is a key step.

Any initial assignment of times to the meets of one layer will probably require repair. But repair is time-consuming, and it will help if the initial assignment has few defects—as a first priority, few demand defects, but also few defects of other kinds. The method presented in this section, called *extended layer matching*, or *Elm* for short, is the author's best method of finding an initial assignment of times to the meets of one layer.

If all meets have duration 1 and minimizing ordinary demand defects is the sole aim, the problem can be solved efficiently using weighted bipartite matching. Make each meet a node and each time a node, and connect each meet to each time it may be assigned, by an edge whose cost is the number of demand defects that assignment causes. Among all matchings with the maximum number of edges, choose one of minimum cost and make the indicated assignments.

Elm is based on this kind of weighted bipartite matching, called *layer matching* by the author, making it good at minimizing demand defects. It is *extended* with ideas that heuristically reduce other defects. Layer matching was called *meta-matching* in the author's early work, because it operates above another matching, the global tixel matching.

Elm can be used without understanding it in detail, by calling

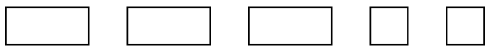
```
bool KheElmLayerAssign(KHE_LAYER layer,
  KHE_SPREAD_EVENTS_CONSTRAINT sec, KHE_OPTIONS options);
```

`KheElmLayerAssign` finds an initial assignment of the meets of the child nodes of `layer` to the meets of the parent node of `layer`, leaving any existing assignments unchanged, and returning `true` if every meet of `layer` is assigned afterwards. It works well with the reduced meet domains installed by solvers such as `KheSolnClusterAndLimitMeetDomains` (Section 10.3.3) for minimizing cluster busy times and limit idle times defects. It tries to minimize demand defects, and if `layer`'s parent node has zones, it also tries to make its assignments meet and node regular with those zones, which should help to minimize spread events defects. If the `diversify` option of `options` (Section 8.2) is `true`, it consults the solution's diversifier, and its results may vary with the diversifier. It does not repair its assignment, leaving that to other functions.

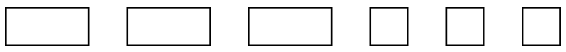
Parameter `sec` is optional (may be `NULL`); a simple choice for it would be any spread events constraint whose number of points of application is maximal. If `sec` is present, the algorithm tries to assign the same number of meets to each of `sec`'s time groups. To see why, consider an example of the opposite. Suppose the events are to spread through the days, and the Wednesday times are assigned eight singles, while the Friday times are assigned four doubles. It's likely that some events will end up meeting twice on Wednesdays and not at all on Fridays. The `sec` parameter acts only with low priority. It is mainly useful on the first layer, when there are no zones and the segmentation is more or less arbitrary.

### 10.6.1. Introducing layer matching

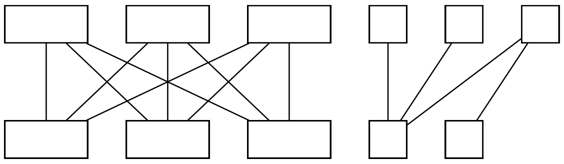
This section introduces layer matching. Later sections describe the implementation. Suppose some layer has three meets of duration 2 and two meets of duration 1, like this:



These *child meets* have to be assigned to non-overlapping offsets in the meets of the parent node (the *parent meets*). Suppose there are three parent meets of duration 2 and three of duration 1:



and suppose (for the moment) that assignments are only possible between meets of the same duration. Then a bipartite graph can represent all the possibilities:



The child meets (the bottom row) are the demand nodes, and the parent meets (the top row) are the supply nodes. Each edge represents one potential assignment of one child meet. Not all edges are present: some are missing because of unequal durations, others because of preassignments and other domain restrictions. For example, the last child meet above appears to be preassigned.

When one of the potential assignments is made, there is a change in solution cost. Each edge may be labelled by this change in cost. Suppose that a matching of maximum size (number of edges) is found whose cost (total cost of selected edges) is minimum. There is a reasonably efficient algorithm for doing this. This matching is the *layer matching*; it defines a legal assignment for some (usually all) child meets, and its cost is a lower bound on the change in solution cost when these meets are assigned to parent meets without any overlapping, as is required since the child meets share a layer and thus presumably share preassigned resources.

The lower bound is only exact if each assignment changes the solution cost independently of the others. This is true for many kinds of monitors, but not all, and it is one reason why the lower bound produced by the matching is not exact. In fact, costs contributed by limit idle times, cluster busy times, and limit busy times monitors only confuse layer matching. So for each resource of the layer, any attached monitors of these kinds are detached at the beginning of `KheElmLayerAssign` and re-attached at the end.

Parent meets usually have larger durations than child meets, allowing choices in packing the children into the parents. The parent node typically represents the week, so it might have, say, 10 meets each of duration 4 (representing 5 mornings and 5 afternoons), whereas the child meets typically represent individual lessons, so they might have durations 1 and 2. A *segment* of parent meet `target_meet` is a triple

```
(target_meet, offset, durn)
```

such that it is legal to assign a child meet of duration `durn` to `target_meet` at `offset`. A *segmentation* of the parent meets is a set of non-overlapping segments that covers all offsets of all parent meets. It is the segments of a segmentation, not the parent meets themselves, that are used as supply nodes. There may be many segmentations, but the layer matching uses only one. This is the other reason why the lower bound is not exact.

A *layer matching graph* is a bipartite graph with one demand node for each meet of a given layer, and one supply node for each segment of some segmentation of the meets of the layer's parent node. For each unassigned child meet `meet`, there is one edge to each parent segment whose duration equals the duration of `meet` and to which `meet` is assignable according to `KheMeetAssignCheck`. The cost of the edge is the cost of the solution when the assignment is made, found by making the assignment, calling `KheSolnCost`, then unassigning again. (Using the solution cost rather than the change in cost ensures that edge costs are always non-negative, as required behind the scenes.) For each assigned child meet `meet`, a parent segment with `meet`'s target meet, offset, and duration is the only possible supply node that the meet can be connected to; if present, the edge cost is 0.

A *layer matching* is a set of edges from the graph such that no node is an endpoint of two or more of the selected edges. A *best matching* is a layer matching of minimum *cost* (sum of edge costs) among all matchings of maximum *size* (number of edges).

The layer giving rise to the demand nodes consists of nodes, each of which typically contains a set of meets for one course. This set of meets will typically want to be spread through the cycle, not bunched together. Each meet generates a demand node, and a set of demand nodes whose meets are related in this way is called a *demand node group.*

There is also a natural grouping of supply nodes, with each *supply node group* consisting of those supply nodes which originated from the same parent meet. Thus, the supply nodes of one group are adjacent in time.

It would be good to enforce the following rule: two demand nodes from the same demand node group may not match with two supply nodes from the same supply node group (because if they did, all chance of spreading out the demand nodes in time would be lost). There is no hope of guaranteeing this rule, because there are cases where it must be violated, and because minimizing cost while guaranteeing it appears to be an NP-complete problem. However, Elm encourages it. When finding a minimum-cost matching, it adds an artificial increment to the cost of each augmenting path that would violate it, thus making those paths relatively uncompetitive and unlikely to be applied. The approach is purely heuristic, but it usually works well.

The overall structure of the layer matching graph is now clear. There are demand nodes, each representing one meet of the layer, grouped into demand node groups representing courses. There are supply nodes, each representing one segment of one meet of the parent node, grouped into supply node groups representing the meets of the parent node. Edges between supply nodes and demand nodes are not defined explicitly; they are determined by the durations and assignability of the meets and segments.

### 10.6.2. The core module

This section describes the *core module*, which implements the layer matching graph, including maintaining a best matching. Elm also has *helper modules*, described in following sections. They have no behind-the-scenes access to the graph; they use only the operations described here.

The core module follows the previous description closely, except that it uses ‘demand’ for ‘demand node’, ‘demand group’ for ‘demand node group’, and so on—for brevity, and so that ‘node’ always means an object of type `KHE_NODE`. This Guide will do this too from now on.

Elm’s types and functions (apart from `KheElmLayerAssign`) are declared in a header file of their own, called `khe_elm.h`. So to access the functions described from here on,

```
#include "khe_solvers.h"
#include "khe_elm.h"
```

must be placed at the start of the source file.

We begin with the operations on type `KHE_ELM`, representing one elm. An elm for a given layer is created by

```
KHE_ELM KheElmMake(KHE_LAYER layer, KHE_OPTIONS options, HA_ARENA a);
```

and deleted by deleting or recycling `a`. If the `diversify` option of `options` is `true`, then the layer's solution's diversifier is used to diversify the elm. In addition to the elm itself, `KheElmMake` creates one demand group for each child node of `layer`, containing one demand for each meet of the child node. It also creates one supply group for each meet of the layer's parent node, containing one supply representing the entire meet. The sets of meets in the parent and child nodes should not change during the elm's lifetime, although the state of one meet (its assignment, domain, etc.) may change.

The layer and options may be accessed by

```
KHE_LAYER KheElmLayer(KHE_ELM elm);
KHE_OPTIONS KheElmOptions(KHE_ELM elm);
```

To access the demand groups, call

```
int KheElmDemandGroupCount(KHE_ELM elm);
KHE_ELM_DEMAND_GROUP KheElmDemandGroup(KHE_ELM elm, int i);
```

in the usual way. To access the supply groups, call

```
int KheElmSupplyGroupCount(KHE_ELM elm);
KHE_ELM_SUPPLY_GROUP KheElmSupplyGroup(KHE_ELM elm, int i);
```

An elm also holds a best matching as defined above. The functions related to it are

```
int KheElmBestUnmatched(KHE_ELM elm);
KHE_COST KheElmBestCost(KHE_ELM elm);
bool KheElmBestAssignMeets(KHE_ELM elm);
```

`KheElmBestUnmatched` returns the number of unmatched demands in the best matching. `KheElmBestCost` returns its cost—not a solution cost, but a sum of edge costs, each of which is a solution cost. `KheElmDemandBestSupply`, defined below, reports which supply a given demand is matched with. To assign the unassigned meets of `elm`'s layer according to the best matching, call `KheElmBestAssignMeets`; it returns `true` if every meet is assigned afterwards. Elm updates the best matching only when one of these four functions is called, for efficiency.

Elm has a 'special node' which is begun and ended by calling

```
void KheElmSpecialModeBegin(KHE_ELM elm);
void KheElmSpecialModeEnd(KHE_ELM elm);
```

While the special mode is in effect, Elm assumes that edges can change their presence in the layer matching graph but not their cost. So when updating edges in special mode, Elm only needs to find whether each edge is present or not, which is much faster than finding costs as well.

To support splitting supplies so that their numbers in each time group of a spread events constraint are approximately equal, these functions are offered:

```
void KheElmUnevennessTimeGroupAdd(KHE_ELM elm, KHE_TIME_GROUP tg);
int KheElmUnevenness(KHE_ELM elm);
```

`KheElmUnevennessTimeGroupAdd` instructs `elm` to keep track of the number of supplies whose starting times lie within `tg`. `KheElmUnevenness` returns the sum over all these time groups of a quantity related to the square of this number. For a given set of supplies, this will be smaller when they are distributed evenly among the time groups than when they are not.

Function

```
void KheElmDebug(KHE_ELM elm, int verbosity, int indent, FILE *fp);
```

produces a debug print of `elm` onto `fp` with the given verbosity and indent. Demands are represented by their meets, and supplies are represented by their meets, offsets, and durations. If `verbosity >= 2`, the print includes the best matching. Function

```
void KheElmDebugSegmentation(KHE_ELM elm, int verbosity,
  int indent, FILE *fp);
```

is similar except that it concentrates on `elm`'s segmentation.

Demand groups have type `KHE_ELM_DEMAND_GROUP`. To access their attributes, call

```
KHE_ELM KheElmDemandGroupElm(KHE_ELM_DEMAND_GROUP dg);
KHE_NODE KheElmDemandGroupNode(KHE_ELM_DEMAND_GROUP dg);
int KheElmDemandGroupDemandCount(KHE_ELM_DEMAND_GROUP dg);
KHE_ELM_DEMAND KheElmDemandGroupDemand(KHE_ELM_DEMAND_GROUP dg, int i);
```

These return `dg`'s enclosing elm, the child node of the original layer that gave rise to `dg`, `dg`'s number of demands, and its `i`th demand.

Elm maintains edges between demands and supplies automatically. But if a demand's meet changes in some way (for example, if its domain changes), Elm has no way of knowing that this has occurred. When the meets of the demands of a demand group change, the user must call

```
void KheElmDemandGroupHasChanged(KHE_ELM_DEMAND_GROUP dg);
```

to inform Elm that the edges touching the demands of `dg` must be remade before being used.

A demand group may contain any number of zones. If there are none, then zones have no effect. If there is at least one zone, then the demand group's demands may match only with supplies that begin in one of its zones. The value `NULL` counts as a zone. Functions

```
void KheElmDemandGroupAddZone(KHE_ELM_DEMAND_GROUP dg, KHE_ZONE zone);
void KheElmDemandGroupDeleteZone(KHE_ELM_DEMAND_GROUP dg, KHE_ZONE zone);
```

add and delete a zone from `dg`, including calling `KheElmDemandGroupHasChanged`. The value of `zone` may be `NULL`. To check whether `dg` contains a given zone, call

```
bool KheElmDemandGroupContainsZone(KHE_ELM_DEMAND_GROUP dg, KHE_ZONE zone);
```

To visit the zones of a demand group, call

```
int KheElmDemandGroupZoneCount(KHE_ELM_DEMAND_GROUP dg);
KHE_ZONE KheElmDemandGroupZone(KHE_ELM_DEMAND_GROUP dg, int i);
```

Function

```
void KheElmDemandGroupDebug(KHE_ELM_DEMAND_GROUP dg,
  int verbosity, int indent, FILE *fp);
```

sends a debug print of `dg` with the given verbosity and indent to `fp`.

Demands have type `KHE_ELM_DEMAND`. To access their attributes, call

```
KHE_ELM_DEMAND_GROUP KheElmDemandDemandGroup(KHE_ELM_DEMAND d);
KHE_MEET KheElmDemandMeet(KHE_ELM_DEMAND d);
```

These return the enclosing demand group, and the meet that gave rise to the demand.

As explained above, when a demand's meet changes in some way that affects the demand's edges, Elm must be informed. For a single demand, this is done by calling

```
void KheElmDemandHasChanged(KHE_ELM_DEMAND d);
```

This is called by `KheElmDemandGroupHasChanged` for each demand in its demand group. To find out which supply `d` is matched with in the best matching, call

```
bool KheElmDemandBestSupply(KHE_ELM_DEMAND d,
  KHE_ELM_SUPPLY *s, KHE_COST *cost);
```

If `d` is matched with a supply in the best matching, `KheElmDemandBestSupply` sets `*s` to that supply and `*cost` to the cost of the edge, and returns `true`; otherwise it returns `false`. And

```
void KheElmDemandDebug(KHE_ELM_DEMAND d, int verbosity,
  int indent, FILE *fp);
```

sends a debug print of `d` with the given verbosity and indent to `fp`.

Supply groups have type `KHE_ELM_SUPPLY_GROUP`. To access their attributes, call

```
KHE_ELM KheElmSupplyGroupElm(KHE_ELM_SUPPLY_GROUP sg);
KHE_MEET KheElmSupplyGroupMeet(KHE_ELM_SUPPLY_GROUP sg);
int KheElmSupplyGroupSupplyCount(KHE_ELM_SUPPLY_GROUP sg);
KHE_ELM_SUPPLY KheElmSupplyGroupSupply(KHE_ELM_SUPPLY_GROUP sg, int i);
```

These return `sg`'s enclosing elm, the meet of the layer's parent node that gave rise to it, its number of supplies (segments), and its `i`th supply. And

```
void KheElmSupplyGroupDebug(KHE_ELM_SUPPLY_GROUP sg,
  int verbosity, int indent, FILE *fp);
```

sends a debug print of `sg` with the given verbosity and indent to `fp`.

Supplies have type `KHE_ELM_SUPPLY`. To access their attributes, call

```
KHE_ELM_SUPPLY_GROUP KheElmSupplySupplyGroup(KHE_ELM_SUPPLY s);
KHE_MEET KheElmSupplyMeet(KHE_ELM_SUPPLY s);
int KheElmSupplyOffset(KHE_ELM_SUPPLY s);
int KheElmSupplyDuration(KHE_ELM_SUPPLY s);
```

`KheElmSupplySupplyGroup` is the enclosing supply group, `KheElmSupplyMeet` is the enclosing supply group's meet, and KheElmSupplyOffset and KheElmSupplyDuration return an offset and duration within that meet, defining one segment.

To facilitate calculations with zones, each supply maintains the set of distinct zones that its offsets lie in. These may be accessed by calling

```
int KheElmSupplyZoneCount(KHE_ELM_SUPPLY s);
KHE_ZONE KheElmSupplyZone(KHE_ELM_SUPPLY s, int i);
```

A NULL zone counts as a zone, so KheElmSupplyZoneCount is always at least 1.

To facilitate the handling of preassigned and previously assigned demands, Elm offers

```
void KheElmSupplySetFixedDemand(KHE_ELM_SUPPLY s, KHE_ELM_DEMAND d);
KHE_ELM_DEMAND KheElmSupplyFixedDemand(KHE_ELM_SUPPLY s);
```

KheElmSupplySetFixedDemand informs elm that d is the only demand suitable for matching with s, or if d is NULL (the default), that there is no restriction of that kind. If d != NULL, d's duration must equal the duration of s. A call to KheElmDemandHasChanged(d) is included. KheElmSupplyFixedDemand returns s's current fixed demand, possibly NULL.

To facilitate the handling of irregular monitors, a supply can be temporarily removed from the graph (so that it does not match any demand) and subsequently restored:

```
void KheElmSupplyRemove(KHE_ELM_SUPPLY s);
void KheElmSupplyUnRemove(KHE_ELM_SUPPLY s);
```

KheElmSupplyRemove aborts if s has a fixed demand. A removed supply merely becomes unmatchabled, it does not get deleted from node lists and so on. Function

```
bool KheElmSupplyIsRemoved(KHE_ELM_SUPPLY s);
```

reports whether s is currently removed.

When KheElmMake returns, there is one demand group for each child node, one demand for each child meet, one supply group for each parent meet, and one supply for each supply group, with offset 0 and duration equal to the duration of the meet. All this is fixed except that supplies may be split and merged by calling

```
bool KheElmSupplySplitCheck(KHE_ELM_SUPPLY s, int offset, int durn,
  int *count);
bool KheElmSupplySplit(KHE_ELM_SUPPLY s, int offset, int durn,
  int *count, KHE_ELM_SUPPLY *ls, KHE_ELM_SUPPLY *rs);
void KheElmSupplyMerge(KHE_ELM_SUPPLY ls, KHE_ELM_SUPPLY s,
  KHE_ELM_SUPPLY rs);
```

KheElmSupplySplitCheck returns true when s may be split so that one of the fragments has the given offset and duration. If so, it sets *count to the total number of fragments that would be produced, either 1, 2, or 3. KheElmSupplySplit is the same except that it actually performs the split when possible, leaving s with the given offset and duration. Splitting is possible when

```
KheElmSupplyFixedDemand(s) == NULL &&
KheElmSupplyOffset(s) <= offset &&
offset + durn <= KheElmSupplyOffset(s) + KheElmSupplyDuration(s)
```

This says that `s` is not fixed to some demand, and that `offset` and `durn` define a set of offsets lying within the set of offsets currently covered by `s`. Otherwise it returns `false`.

If `KheElmSupplyOffset(s) < offset`, then a supply `*ls` is split off `s` at left, holding the offsets from `KheElmSupplyOffset(s)` inclusive to `offset` exclusive; otherwise `*ls` is set to `NULL`. If `offset + durn < KheElmSupplyOffset(s) + KheElmSupplyDuration(s)`, then a supply `*rs` is split off `s` at right, holding the offsets from `offset + durn` inclusive to `KheElmSupplyOffset(s) + KheElmSupplyDuration(s)` exclusive; otherwise `*rs` is set to `NULL`. The original `s` is left with offsets from `offset` inclusive to `offset + durn` exclusive.

`KheElmSupplyMerge` undoes the corresponding `KheElmSupplySplit`. Either or both of `ls` and `rs` may be `NULL`. No meet splitting or merging is carried out by these operations.

Finally,

```
void KheElmSupplyDebug(KHE_ELM_SUPPLY s, int verbosity,
  int indent, FILE *fp);
```

sends a debug print of `s` with the given verbosity and indent to `fp`.

### 10.6.3. Splitting supplies

The elm returned by `KheElmMake` has only a trivial segmentation, with one segment per parent meet. Few or no demands will match with these supplies, because only demands and supplies of equal duration match. So the initial supplies have to be split using `KheElmSupplySplit`.

Elm has a helper module which splits supplies heuristically. It offers just one function:

```
void KheElmSplitSupplies(KHE_ELM elm, KHE_SPREAD_EVENTS_CONSTRAINT sec);
```

If the `diversify` option of `elm`'s `options` attribute is `true`, its result varies depending on the layer's solution's diversifier. The `sec` parameter of `KheElmSplitSupplies` may be `NULL`. If non-`NULL`, `KheElmSplitSupplies` tries to find a segmentation in which each time group of `sec` covers the same number of segments, as explained for `KheElmLayerAssign` above.

`KheElmSplitSupplies` works as follows. Begin by handling demands whose meets are preassigned or already assigned. For each such demand, split a supply to ensure that exactly the right segment is present, and use `KheElmSupplySetFixedDemand` to fix the supply to the demand. If the required split cannot be made, the demand remains permanently unmatched.

Sort the remaining demands by increasing size of their meets' domains (in practice this also sorts by decreasing duration), breaking ties by decreasing demand. Use `KheMeetAssignFix` to ensure that these meets cannot be assigned. This removes them from the matching to begin with (strictly speaking, it prevents them from having any outgoing edges in the matching graph).

For each demand in turn, unfix its meet and observe the effect of this on the best matching. If the size of the best matching increases by one, proceed to the next demand. Otherwise, the demand has failed to match, and this must be corrected (if possible) by splitting segments of larger duration into smaller segments that it can match with. For each supply whose duration is larger than the duration of the demand, try splitting the supply in all possible ways into two or three smaller segments such that at least one of the fragments has the same duration as the demand. If there was at least one successful split, redo the best of them.

The best split is determined by an evaluation with five components:

1. The split must be *successful*: it must increase the size of the best matching by one. Only successful splits are eligible for use; if there are none, the demand remains unmatched.

2. It is better to split a segment into two fragments than into three. For example, when splitting a double from a meet of duration 4, it is better to take the first two times or the last two, rather than the middle two, since the latter leaves fewer choices for future splits.

3. If the parent node has zones, it is desirable to use a segment overlapping only one zone, to produce meet regularity (Section 5.4) with the layer used to create the zones.

4. The split should produce a best matching whose cost is as small as possible.

5. If `sec != NULL`, the split should encourage the evenness that `sec`'s presence requests.

These are combined lexicographically: later criteria only apply when earlier ones are equal. Meet regularity has higher priority than cost because cost can often be improved later, whereas meet regularity once lost is lost forever.

After all demands are processed, if any supplies have duration larger than the duration of all demands, split them into smaller pieces, preferably supplies regular with the zones, if any. This adds more edges, and so may reduce the cost of the best matching, at no risk to its size. It is important when timetabling layers of small duration, such as layers containing staff meetings.

### 10.6.4. Improving node regularity

When the parent node has zones, `KheElmSplitSupplies` produces good meet regularity but does nothing to promote node regularity. This can be done by following it with a call to

```
void KheElmImproveNodeRegularity(KHE_ELM elm);
```

implemented by another Elm helper module. It does nothing when there are no zones. When there are, it removes edges from the matching graph to improve the node regularity of the edges with respect to the zones. If requested by the `diversify` option of `elm`'s `options` attribute, it consults the solution's diversifier, and the edges it removes vary with the diversifier.

The problem of removing edges from a layer matching graph to maximize node regularity with zones while keeping the matching cost low may seem obscure, but it is one of the keys to effective time assignment in high school timetabling. Bin packing is reducible to this problem, so it is NP-complete. Even the small instances (up to ten nodes in each layer, say) that occur in practice seem hard to solve to optimality. The author tried a tree search which would have produced an optimal result, but could not make it efficient, even with several pruning rules. So `KheElmImproveNodeRegularity` is heuristic.

Although many kinds of defects contribute to the edge costs that make up the matching cost, in practice the cost is dominated by demand cost (including the cost of avoid clashes and avoid unavailable times defects). Every unit of demand cost incurred when assigning a time represents an unassignable resource at that time, implying that either the final solution will have a significant defect, or else that the time assignment will have to be changed later.

However, not all demand costs are equally important. When the cost is incurred by a child node with no children, all of the meets of that node at that time will have to be moved later, which is very disruptive. An assignment scarcely deserves to be called node-regular if that is going to happen. But when the cost is incurred by a child node with children, after flattening it is often possible to remove the defect by moving just one meet, disrupting node regularity only slightly. So it is important to give priority to nodes with no children.

This is done in two ways. First, the cost of edges leading out of meets whose nodes have no children is multiplied by 10. Second, when evaluating alternatives while improving node regularity, the cost of the best matching is divided into two parts: the total cost of edges leading out of meets in nodes with no children (the *without-children cost*) and the total cost of the remaining edges (the *with-children cost*), and without-children cost takes priority.

The heuristic sorts the child nodes by decreasing duration. Nodes with equal duration are sorted by increasing number of children. Although it is important to minimize without-children cost, even at the expense of with-children cost, it would be wrong to maximize without-children node regularity at the expense of with-children node regularity. Node regularity is generally harder to achieve for nodes of longer duration, so they are handled first.

For each child node in sorted order, the heuristic generates a sequence of sets of zones. For each set of zones, it reduces the matching edges leading out of the meets of the child node so that they go only to segments whose times overlap with the times of the zones. A best set of zones is chosen, the limitation of the child node's meets to those nodes is fixed, and the heuristic proceeds to the next child node.

The best set is the first one with a lexicographically minimum value of the triple

```
(without_children_cost, zones_cost, with_children_cost)
```

The `without_children_cost` and `with_children_cost` components are as defined above. The `zones_cost` component measures the badness of the set of zones. It is the number of zones in the set (we are trying to minimize this number, after all), adjusted to favour zones of smaller duration and zones already present in sets fixed on previously, to encourage the algorithm to use up zones completely wherever possible.

The algorithm for generating sets of zones generates all sets of cardinality 1, then all sets of cardinality 2, then one set containing every zone that the current best matching touches. This last set is included to ensure that at least one set leading to a reasonable matching cost is tried. A few optimizations are implemented, including skipping sets of insufficient duration, and skipping zones known to be fully utilized already.

### 10.6.5. Handling irregular monitors

Each edge of the layer matching graph is assigned a cost by making one meet assignment and measuring the solution cost afterwards. This amounts to assuming that the cost of each edge is independent of which other edges are present in the best matching. Costs come from monitors, and the truth of this assumption varies with the monitor type, as follows.

*Assign time and prefer times costs.* Independent when the cost function is `Linear`, which it always is in practice for these kinds of monitors.

*Split events and distribute split events costs.* Not changed by meet assignments.

*Spread events costs.* Non-independent. Previous sections have addressed this problem, by varying path costs to discourage two demands from one demand group from matching with two supplies from one supply group, and by improving node regularity.

*Link events costs.* Not changed by meet assignments when handled structurally, which they always are in practice.

*Order events costs.* Non-independent when both events lie in the current layer.

*Assign resource, prefer resources, and avoid split assignments costs.* Not changed by meet assignments.

*Avoid clashes costs.* Independent, because clashes are never introduced within one layer.

*Avoid unavailable times costs.* Independent when the cost function is `Linear`.

*Limit idle times, cluster busy times, and limit busy times costs.* Non-independent when present (when resources subject to them are preassigned in the layer's meets).

*Limit workload costs.* Not changed by meet assignments.

*Demand costs.* Independent when they monitor clashes and unavailable times. More subtle interactions can be non-independent, but most layer matchings are built when the timetable is incomplete and subtle demand overloads are unlikely.

Order events, limit idle times, cluster busy times, and limit busy times monitors stand out as needing attention. These will be called *irregular monitors.*

At present, the author has no experience with order events monitors, so Elm does nothing with them. The irregular monitors handled by Elm are those limit idle times, cluster busy times, and limit busy times monitors of the resources of the layer match's layer which are attached at the time the elm is created. The Elm core module stores these monitors in an array, accessible via

```
int KheElmIrregularMonitorCount(KHE_ELM elm);
KHE_MONITOR KheElmIrregularMonitor(KHE_ELM elm, int i);
void KheElmSortIrregularMonitors(KHE_ELM elm,
  int(*compar)(const void *, const void *));
```

`KheElmIrregularMonitorCount` and `KheElmIrregularMonitor` visit them in the usual way. `KheElmSortIrregularMonitors` sorts them; `compar` is a function suited to passing to `qsort` when sorting an array of monitors. Core function

```
bool KheElmIrregularMonitorsAttached(KHE_ELM elm);
```

returns `true` if all irregular monitors are currently attached. By definition, this is true initially.

As a first step in handling the irregular monitors of its layer, Elm offers functions

```
void KheElmDetachIrregularMonitors(KHE_ELM elm);
void KheElmAttachIrregularMonitors(KHE_ELM elm);
```

to detach any irregular monitors that are not already detached, and attach any that are not already attached. `KheElmLayerAssign` uses them to detach irregular monitors at the start and reattach them at the end. This ensures that the best matching never takes them into account. It would only cause confusion if it did.

For improving its performance when irregular monitors are present, Elm offers

```
void KheElmReduceIrregularMonitors(KHE_ELM elm);
```

If irregular monitors are attached, it detaches them. It installs the best matching's assignments, attaches irregular monitors, and remembers the solution cost. Then for each supply *s*, it detaches irregular monitors, removes *s* from the graph, installs the best matching's assignments, attaches irregular monitors, remembers the solution cost, and restores *s* to the graph. If none of the removals improves cost, it returns irregular monitors to their original state of attachment and terminates. Otherwise, it permanently removes the supply that produced the best cost and repeats from the start.

Some optimizations avoid futile work. If removing *s* would reduce the total duration of supply nodes to below the total duration of demand nodes, or reduce the number of supplies of *s*'s duration to below the number of demands of *s*'s duration, the removal of *s* is not tried. And the function returns immediately if the layer has no irregular monitors.

`KheElmReduceIrregularMonitors` is a plausible way to attack limit idle times and limit busy times defects, but it is not radical enough for cluster busy times defects. These are better handled by other means, such as `KheSolnClusterAndLimitMeetDomains` (Section 10.3.3).

## 10.7. Time repair

This section presents the time solvers packaged with KHE that take an existing time assignment and repair it (that is, attempt to improve it). However carefully an initial time assignment is made, it must proceed in steps, and it can never incorporate enough forward-looking information to ensure that each step does not create problems for later steps. So a repair phase after the initial assignment is complete seems to be a practical necessity.

### 10.7.1. Node-regular time repair using layer node matching

Suppose we have a time assignment with good node regularity, but with some spread and demand defects. Repairs that move meets arbitrarily might fix some defects, but the resulting loss of node regularity might have serious consequences later, during resource assignment. This section offers one idea for repairing time assignments without sacrificing node regularity.

One useful idea is to make repairs which are *node swaps*: swaps of the assignments of (the meets of) entire nodes. The available swaps are quite limited, because the nodes concerned must lie in the same layers and have the same number of meets with the same durations.

For any parent node, take any set of child nodes lying in the same layers whose meets are all assigned. Build a bipartite graph in which each of these child nodes is one demand node, and the set of assignments of its meets is one supply node. An assignment is a triple of the form

```
(target_meet, offset, durn)
```

as for layer matchings (Section 10.6), but here a supply node is a set of triples, not one triple.

For each case where a child node can be assigned to a set of triples, because the number of triples and their durations match the node's number of meets and durations, add an edge to the graph labelled by the change in solution cost when the corresponding set of assignments is made. Find a maximum matching of minimum cost in this graph and reassign the child nodes in accordance with it. The existing assignment is one maximum matching, so this will either reproduce that or find something which has a good chance of being better. Function

```
bool KheLayerNodeMatchingNodeRepairTimes(KHE_NODE parent_node,
  KHE_OPTIONS options);
```

applies these ideas to the child nodes of `parent_node`, returning `true` if it considers its work to have been useful, as is usual for time repair solvers. First, if `parent_node` has no child layers it calls `KheNodeChildLayersMake` to build them. Then it partitions the child nodes so that the nodes of each partition lie in the same set of layers. Then, for each partition in turn, it builds the weighted bipartite graph and carries out the corresponding reassignments. If the solution cost does not decrease, the reassignments are undone. It continues cycling around the partitions until *n* reassignments have occurred without a cost decrease, where *n* is the number of partitions. Finally, if it made layers to begin with it removes them. A related function is

```
bool KheLayerNodeMatchingLayerRepairTimes(KHE_LAYER layer,
  KHE_OPTIONS options);
```

It starts with the child nodes of `layer` rather than all the child nodes of its parent.

On a real instance, `KheLayerNodeMatchingNodeRepairTimes` found no improvements at all after all layers were assigned. Applied after each layer after the first was assigned, it found one improvement, which reduced the number of unassignable tixels by 1 or 2. This improvement was carried through to the final solution: the median number of unassigned tixels when solving 16 instances was reduced from about 9 to about 7, and there were modest reductions in spread defects and split assignment defects as well. The extra run time was about 0.6 seconds.

### 10.7.2. Ejection chain time repair

Time solvers

```
bool KheEjectionChainNodeRepairTimes(KHE_NODE parent_node,
  KHE_OPTIONS options);
bool KheEjectionChainLayerRepairTimes(KHE_LAYER layer,
  KHE_OPTIONS options);
```

use ejection chains (Chapter 13) to repair the assignments of the meets of the descendants of the child nodes of `parent_node`, or the assignments of the meets of the descendants of the child nodes of `layer`. For full details of these functions, consult Section 13.6.1.

### 10.7.3. Tree search layer time repair

Very large-scale neighbourhood (VLSN) search [1, 12] deassigns a relatively large chunk of the solution, then reassigns it in a hopefully better way. If the chunk is chosen carefully, it may be possible to find an optimal reassignment in a moderate amount of time.

One well-known VLSN neighbourhood is the set of meets of one layer (a set of meets which must be disjoint in time, usually because they have a resource in common). For example, finding a timetable for one university student is a kind of layer reassignment, with the choices of times for the meets determined by when sections of the student's courses are running. Function

```
bool KheTreeSearchLayerRepairTimes(KHE_SOLN soln, KHE_RESOURCE r);
```

reassigns the meets of `soln` currently assigned resource `r`, using a tree search. Once the number of nodes explored reaches a fixed limit, it switches to a simple heuristic, giving up the guarantee of optimality to ensure that running time remains moderate. Function

```
bool KheTreeSearchRepairTimes(KHE_SOLN soln, KHE_RESOURCE_TYPE rt,
  bool with_defects);
```

calls `KheTreeSearchLayerRepairTimes` for each resource in `soln`'s instance (or each of type `rt`, if `rt` is non-`NULL`). If `with_defects` is `true`, these calls are only made for resources with at least one resource defect, otherwise they are made for all resources. The rest of this section describes `KheTreeSearchLayerRepairTimes` in detail.

If a tree search is given a high standard to reach, it will run quickly because many paths will fail the standard and get pruned, and so it is quite likely to run to completion and reach that high standard if it is reachable at all. If it is given a low standard, it will run more slowly and quite possibly not run to completion. Either approach is legitimate, but a choice has to be made.

Because VLSN search is relatively slow, it seems best to use it near the end of a solve, when there are few defects left to target. `KheTreeSearchLayerRepairTimes` is intended to be used as a last resort in this way, when there is likely to be just one or two defects related to the layer being targeted. Accordingly, it aims high, for an assignment with no defects at all. It prunes paths whenever it can see that there is a defect that cannot be corrected by further assignments.

The meets are first sorted into decreasing duration order and unassigned. Each is given a *current domain*, which is initially its usual domain minus any starting times that would cause the meet to overlap a time when any of its resources are unavailable. Then a traditional tree search is carried out, which at each node of level *i* assigns a time from its current domain to the *i*th meet in the sorted list. The best leaf is remembered and replaces the original set of assignments if its solution cost is smaller. Three rules are used for pruning the tree.

First, any assignment which returns `false` or causes the number of unmatched demand tixels to exceed its value in the initial solution is rejected.

Second, after a fixed number of nodes is reached, new nodes are still explored, but only the first assignment that does not increase the number of unmatched demand tixels is tried therein.

Third, a form of forward checking is used. Let $m_1$ and $m_2$ be meets of the layer, and let $t_1$ and $t_2$ be times. At the start, a set of *exclusions* is built, each of the form

$$(m_1, t_1) \Rightarrow \neg(m_2, t_2)$$

This means that if $m_1$ is assigned starting time $t_1$, then $m_2$ may not be assigned starting time $t_2$. While the search is running, when $m_1$ is assigned $t_1$ this exclusion is applied, removing $t_2$ from the domain of $m_2$. When $m_1$ is unassigned later, the exclusion is removed ($m_2$ must come later in the list of meets to be assigned than $m_1$, so that at the moment $m_1$ is assigned, $m_2$ is not assigned).

Following is a list of true statements about various situations:

- Since the meets all share a resource, no two of the meets may overlap in time.
- Two meets linked by a spread events constraint cannot be assigned within the same time group of that constraint, when that time group has a `Maximum` attribute of 1.
- Two meets linked by an order events constraint must be assigned in a certain chronological order, possibly with a given separation.
- Given two meets with the same duration and the same resources, and monitored by the same event monitors, it is safe (and useful for avoiding symmetrical searches) to arbitrarily insist that the first one in the assignment list should appear earlier in the cycle than the second.

Each statement gives rise to exclusions, and all these exclusions are added, except that at present a couple of shortcuts are being used: order events constraints are not currently taken into account, and the symmetry breaking idea of the last point is being applied to a different set of pairs of meets, namely those which are linked by a spread events constraint and have the same duration.

Exclusions are used in two ways. First, when a meet's turn comes to be assigned, only times in its current domain (its initial domain minus any exclusions) are tried. Second, each meet keeps a count of the number of times in its current domain. If this number ever drops to 0, the assignment that caused that to happen is rejected immediately.

On instance IT-I4-96, with limit 10000, this method improved the timetables of four resources, reducing final cost from 0.00397 to 0.00390, and adding about 2 seconds to total run time. There was wide variation in the completeness of the search: for some resources, every possible timetable was tried; for others, there was only time to try timetables that assigned the first meet to the first time. It did not reduce the 0.00067 cost of the best of 8 solutions, nor find any improvements when solving instance AU-BG-98. A run with limit 1000000 improved a fifth resource in IT-I4-96, and showed that many searches do reach even this quite large limit.

### 10.7.4. Meet set time repair and the fuzzy meet move

Another VLSN idea is to use a tree search to repair the assignments of an arbitrary (but small) set of meets. Given a set of meets, build the set of all target meets they are assigned to, and for each target meet, the set of offsets within it that they are running. The aim is to reassign the meets optimally within these same target meets and offsets. The only pruning rule is that the number of unmatched demand tixels may not exceed its initial value.

The functions that implement this idea are

```
KHE_MEET_SET_SOLVER KheMeetSetSolveBegin(KHE_SOLN soln, int max_meets);
void KheMeetSetSolveAddMeet(KHE_MEET_SET_SOLVER mss, KHE_MEET meet);
bool KheMeetSetSolveEnd(KHE_MEET_SET_SOLVER mss);
```

`KheMeetSetSolveBegin` makes a meet-set solver object which coordinates the operation. `KheMeetSetSolveAddMeet` adds one meet to the solver, and may be called any number of times, building up a set of meets. If the number of meets added reaches the `max_meets` parameter of

KheMeetSetSolveBegin, further calls to KheMeetSetSolveAddMeet are allowed but ignored. Finally, KheMeetSetSolveEnd uses a tree search to find an optimal reassignment of the meets to (collectively) their original target meets and offsets, returning true if it reduced the cost of the solution, and frees the memory used by the solver object. If the number of nodes in the search tree exceeds a given fixed limit, the search switches to a simple linear heuristic at each remaining tree node, losing the guarantee of optimality but ensuring that run times remain moderate.

As a first application of these functions, KHE offers

```
bool KheFuzzyMeetMove(KHE_MEET meet, KHE_MEET target_meet, int offset,
  int width, int depth, int max_meets);
```

This may move meet to target_meet at offset, but not necessarily. Instead, it selects a set of meets likely to be affected by that move, including meet, and passes them all to the meet set solver above for (hopefully) optimal reassignment. It returns true if and only if it changed the solution, which will be if and only if it reduced the cost of the solution.

The point of KheFuzzyMeetMove is that if the caller has identified this move as likely to be useful, but with some uncertainty about its consequences, it allows the move to be tried, but with adjustments in the neighbourhood to get the most out of it. These adjustments are not unlike those made by Kempe meet moves, only more general and more costly in run time.

Two sets of meets are selected. To be in the first set, a meet has to be assigned to the same target meet as meet, at an offset lying between meet's current offset minus width, and meet's current offset plus width. Furthermore, if depth is 1 (the smallest reasonable value), a selected meet has to share a resource (assigned or preassigned) with meet. If depth is 2, a selected meet has to share a resource with a meet that would be selected when the depth is 1, and so on: the depth signifies the maximum length of a chain of shared resources that must connect a selected meet to meet. The second set of meets is the same as the first, only defined using target_meet and offset instead of meet's current target meet and offset.

As for meet set time repair, at most max_meets meets will be selected. If width and depth are small, it is reasonable for max_meets to be INT_MAX.

## 10.8. Layered time assignment

The heart of time assignment when layer trees are used is to assign the meets of the child nodes of a given parent node to the meets of the parent node. A *layered time assignment* is one which groups the child nodes into layers and assigns them layer by layer. This is a good way to do it, since the nodes of each layer strongly constrain each other (they must be disjoint in time).

KheElmLayerAssign (Section 10.6) is KHE's main solver for assigning the meets of the child nodes of one layer. But there is work to be done to prepare the way for calling this function, beyond the structural work of building the layer tree. This section presents KHE's functions for carrying out this preparatory work and calling KheElmLayerAssign.

### 10.8.1. Layer assignments

When assigning layers it is useful to be able to record an assignment of the meets of a layer, for undoing and redoing later. Marks and paths could do this, but they record every step. A layer assignment algorithm could be very long and wandering, so it is better to record just its result.

Accordingly, KHE offers the *layer assignment* object, with type `KHE_LAYER_ASST`:

```
KHE_LAYER_ASST KheLayerAsstMake(KHE_SOLN soln);
void KheLayerAsstDelete(KHE_LAYER_ASST layer_asst);
void KheLayerAsstBegin(KHE_LAYER_ASST layer_asst, KHE_LAYER layer);
void KheLayerAsstEnd(KHE_LAYER_ASST layer_asst);
void KheLayerAsstUndo(KHE_LAYER_ASST layer_asst);
void KheLayerAsstRedo(KHE_LAYER_ASST layer_asst);
void KheLayerAsstDebug(KHE_LAYER_ASST layer_asst, int verbosity,
  int indent, FILE *fp);
```

`KheLayerAsstMake` and `KheLayerAsstDelete` make and delete one. `KheLayerAsstBegin` is called before some algorithm for assigning `layer` is run. It records which of `layer`'s meets are unassigned then. `KheLayerAsstEnd` is called after the algorithm ends. For each meet recorded by `KheLayerAsstBegin`, it records the assignment of that meet. `KheLayerAsstUndo` undoes the recorded assignments, and `KheLayerAsstRedo` redoes them. `KheLayerAsstDebug` produces a debug print of `layer_asst` onto `fp`.

### 10.8.2. A solver for layered time assignment

Time solver

```
bool KheNodeLayeredAssignTimes(KHE_NODE parent_node, KHE_OPTIONS options);
```

assigns the meets of the child nodes of `parent_node` to the meets of `parent_node`, calling `KheElmLayerAssign` (Section 10.6) to assign them layer by layer. Existing assignments of the meets affected may change. The implementation is described at the end of this section.

If `parent_node` is the cycle node, `KheNodePreassignedAssignTimes` should be called first, to give priority to demands made by preassigned meets.

`KheNodeLayeredAssignTimes` is influenced by three options:

`ts_no_node_regularity`
A Boolean option which, when `true`, instructs `KheNodeLayeredAssignTimes` , as well as `KheEjectionChainNodeRepairTimes` and `KheEjectionChainLayerRepairTimes` (Section 13.6.1), to not try to make the assignments node-regular (Section 5.4). Node regularity will usually be appropriate for the cycle node, but not for other nodes, since in practice they are runaround nodes, and irregularity is wanted in them rather than regularity.

`ts_layer_swap`
`KheNodeLayeredAssignTimes` usually assigns each layer in turn, in a heuristically chosen order. But if the Boolean `ts_layer_swap` option is `true`, it does something more interesting. For each layer $i$ other than the first and last, it (a) tries assigning and repairing layer $i$ followed by layer $i + 1$, then (b) tries assigning and repairing layer $i + 1$ followed by layer $i$. If the solution cost after (a) is less than after (b), it leaves (a)'s assignment of layer $i$ in place and proceeds to the next layer; otherwise it leaves (b)'s assignment of layer $i + 1$ in place and proceeds to the next layer. So one layer is assigned on each iteration, as usual, but it could be either the usual one or the next one.

`ts_layer_repair`
: An option which instructs `KheNodeLayeredAssignTimes` which of its layers to repair after assignment. It has three values, `"none"` meaning repair no layers, `"all"` meaning repair all layers, and `"exp"` meaning use exponential backoff to decide which layers to repair. When the option is absent its value is taken to be `"all"`.

`ts_layer_time_limit`
: A string option defining a soft time limit for assigning a layer. The format is that accepted by `KheTimeFromString` (Section 8.1): `secs`, or `mins:secs`, or `hrs:mins:secs`. There is also the special value `-`, meaning 'set no limit', and this is the default value.

The rest of this section describes the implementation of `KheNodeLayeredAssignTimes`.

If `parent_node` has no layers, `KheNodeLayeredAssignTimes` first makes them, by calling `KheNodeChildLayersMake` (Section 9.3.1). It then sorts the layers, assigns and optionally repairs them, and ends with `KheNodeChildLayersDelete` if it called `KheNodeChildLayersMake`.

When sorting the layers, the first priority is to ensure that already assigned layers come first. These are marked by assigning visit number 1 to them. Among unvisited layers, a heuristic rule is used: decreasing value of the sum of the duration and the duration of meets that have already been assigned, minus the number of meets. The reasoning here is that layers with larger durations are harder to assign, and they become even harder when many of their meets' assignments are already decided on (since the algorithm does not change them); but, on the other hand, the more meets there are, the smaller their durations must be for a given overall duration, making assignment easier. Here is the layer comparison function; it may be called separately:

```
int KheNodeLayeredLayerCmp(const void *t1, const void *t2)
{
  KHE_LAYER layer1 = * (KHE_LAYER *) t1;
  KHE_LAYER layer2 = * (KHE_LAYER *) t2;
  int value1, value2, demand1, demand2;
  if( KheLayerVisitNum(layer1) != KheLayerVisitNum(layer2) )
    return KheLayerVisitNum(layer2) - KheLayerVisitNum(layer1);
  value1 = KheLayerDuration(layer1) - KheLayerMeetCount(layer1) +
    KheLayerAssignedDuration(layer1);
  value2 = KheLayerDuration(layer2) - KheLayerMeetCount(layer2) +
    KheLayerAssignedDuration(layer2);
  if( value1 != value2 )
    return value2 - value1;
  demand1 = KheLayerDemand(layer1);
  demand2 = KheLayerDemand(layer2);
  if( demand1 != demand2 )
    return demand2 - demand1;
  return KheLayerParentNodeIndex(layer1) -
    KheLayerParentNodeIndex(layer2);
}
```

As a last resort it compares total demand, then layer indexes, to give a non-zero result in all cases: `qsort`'s specification is non-deterministic, which is best avoided, if the result is zero.

KheNodeLayeredAssignTimes sets the time_vizier_node option to false before making the call that repairs the first layer, and resets it to its original value afterwards. It's a small point, but a vizier node would be redundant when repairing the first layer.

Let the *whole-timetable monitors* be the limit idle times, cluster busy times, and limit busy times monitors. These depend on the whole timetable of their resource, or large parts of it. The other resource monitors either depend on local parts of the timetable (avoid clashes and avoid unavailable times monitors) or are independent of the timetable (limit workload monitors).

In practice, evaluating a whole-timetable monitor before its resource's layer is assigned is problematical, since it depends on the whole timetable, which does not exist then. For example, a partial timetable may have idle times which could well be filled later when its resource's other meets are assigned times. Accordingly, KheNodeLayeredAssignTimes begins by detaching all whole-timetable monitors of all resources in all its layers. Just before assigning each layer, it attaches the whole-timetable monitors of the resources of the layer.

This detachment of whole-timetable monitors is similar to the detachment of irregular monitors during the assignment of one layer by Elm (Section 10.6.5). Both detachments are done because the monitors in question would not produce useful cost information if attached. However, in the case of Elm that is because of the particular algorithm employed, whereas here it is because of something more fundamental: the fact that only a partial timetable is present.

The remainder of this section describes the three extra things that are done when the time_node_regularity option of options is true.

First, when a meet from another layer is already assigned (because it is preassigned, usually), it is good to make that same assignment to a meet of the same duration in the first layer, for regularity between the two meets. Such an assignment to a meet of the first layer is called a *parallel assignment*. If there is a node from another layer containing two or more assigned meets, then it is good to make the corresponding parallel assignments within one node of the first layer, for regularity between the nodes; and if two nodes from one layer contain assigned meets, it is good to make the corresponding parallel assignments to distinct nodes of the first layer. The layer solver that makes these parallel assignments to the meets of the first layer is called only when time_node_regularity is true, but it is also available separately:

```
bool KheLayerParallelAssignTimes(KHE_LAYER layer, KHE_OPTIONS options);
```

It makes parallel assignments to layer heuristically, returning true if every assigned meet in every sibling layer of layer has a parallel assignment afterwards. It uses no options.

Second, KheElmLayerAssign takes a spread events constraint as an optional parameter. When time_node_regularity is true, KheNodeLayeredAssignTimes searches the instance for a spread events constraint with as many points of application as possible, and passes this constraint (if any) to KheElmLayerAssign.

Third, and most important, when time_node_regularity is true, after the first layer has been assigned and optionally repaired, KheNodeLayeredAssignTimes uses the first layer's assignments to define zones in the parent node, by calling KheLayerInstallZonesInParent (Section 5.4) and KheNodeExtendZones (Section 9.6). These zones encourage the following calls to KheElmLayerAssign and KheEjectionChainLayerRepairTimes to find and preserve zone-regular assignments.

## 10.9. Putting it all together

There is a `ts` option which allows you to define a do-it-yourself time solver (Section 8.3) built from the pieces presented in this and the preceding chapter. The default value of this option defines a reasonable solver which we will come to later.

The value of `ts` must be a `<solver>` as defined in Section 8.3. Here is the current list of valid items and their meanings:

| `<item>` | Meaning |
|---|---|
| `tcl` | Call `KheCoordinateLayers` (Section 9.3.2). |
| `tbr` | Call `KheBuildRunarounds` (Section 9.4.2). |
| `trt` | Call `KheNodeRecursiveAssignTimes` (Section 10.4) on each child of the cycle node, passing `KheRunaroundNodeAssignTimes` (Section 10.5) as the assignment function. |
| `tpa` | Call `KheNodePreassignedAssignTimes` (Section 10.4). |
| `tnp <solver>` | If not all events have preassigned times, call `<solver>`. If all events have preassigned times, do nothing. |
| `ttp <solver>` | Call `KheTaskingTightenToPartition` (Section 11.4.4) on each tasking, run `<solver>`, then undo the tightening. |
| `tmd <solver>` | Call `KheSolnClusterAndLimitMeetDomains` (Section 10.3.3), run `<solver>`, then undo what the clustering and limiting did. |
| `tnl` | Call `KheNodeLayeredAssignTimes` (Section 10.8.2). |
| `tec` | Call `KheEjectionChainNodeRepairTimes` (Section 10.7.2). |
| `tnf` | Call `KheNodeFlatten` (Section 9.5.5). |
| `tdz` | Call `KheNodeDeleteZones` (Section 5.4). |

`KheSolnTryMeetUnAssignments` has been omitted because the general solver calls it.

The remarks about time limits in Section 8.3 apply to time solving. There is a `ts_time_limit` option for placing a time limit on time assignment. For example,

```
ts_time_limit="2:0"
```

sets an overall time limit for time assignment (including repair) of 2 minutes. Time weights may be used to apportion the available time among the various solvers as usual.

All of this is carried out by function

```
bool KheCombinedTimeAssign(KHE_NODE cycle_node, KHE_OPTIONS options);
```

This sets the `ts_time_limit` time limit if that option is present, then assigns times using the value of option `ts` as its guide, and finally deletes the time limit if it sets it. The easiest way to call it is to include `ts` in the `gs` option (Section 8.4), causing the general solver to call it for you.

The `ts` option has default value

```
gti(tcl, tbr, trt, tpa, tnp ttp(tnl, tec, tnf, tdz, tec))
```

An integrated global tixel matching is installed throughout. This seems to be essential, since otherwise some time assignment solver will fail to recognize that assigning the same time to six Science meets will not work when there are only five Science laboratories, and will produce a time assignment that is of no use to anyone. `KheEjectionChainNodeRepairTimes` is called twice, the second time in a more permissive context. All this represents the author's current idea of how best to use the various time solvers. It will change as his ideas change.

# Chapter 11. Resource-Structural Solvers

This chapter documents the solvers packaged with KHE that modify the resource structure of a solution: group and ungroup tasks, and so on. These solvers may alter resource assignments, but they only do so occasionally and incidentally to their structural work.

## 11.1. Task bound groups

Task domains are reduced by adding task bound objects to tasks (Section 4.6.3). Frequently, task bound objects need to be stored somewhere where they can be found and deleted later. The required data structure is trivial—just an array of task bounds—but it is convenient to have a standard for it, so KHE defines a type `KHE_TASK_BOUND_GROUP` with suitable operations.

To create a task bound group, call

```
KHE_TASK_BOUND_GROUP KheTaskBoundGroupMake(KHE_SOLN soln);
```

To add a task bound to a task bound group, call

```
void KheTaskBoundGroupAddTaskBound(KHE_TASK_BOUND_GROUP tbg,
  KHE_TASK_BOUND tb);
```

To visit the task bounds of a task bound group, call

```
int KheTaskBoundGroupTaskBoundCount(KHE_TASK_BOUND_GROUP tbg);
KHE_TASK_BOUND KheTaskBoundGroupTaskBound(KHE_TASK_BOUND_GROUP tbg, int i);
```

To delete a task bound group, including deleting all the task bounds in it, call

```
bool KheTaskBoundGroupDelete(KHE_TASK_BOUND_GROUP tbg);
```

This function returns `true` when every call it makes to `KheTaskBoundDelete` returns `true`.

## 11.2. Task trees

What meets do for time, tasks do for resources. A meet has a time domain and assignment; a task has a resource domain and assignment. Link events constraints cause meets to be assigned to other meets; avoid split assignments constraints cause tasks to be assigned to other tasks.

There are differences. Tasks lie in meets, but meets do not lie in tasks. Task assignments do not have offsets, because there is no ordering of resources like chronological order for times.

Since the layer tree is successful in structuring meets for time assignment, let us see what an analogous tree for structuring tasks for resource assignment would look like. A layer tree is a tree, whose nodes each contain a set of meets. The root node contains the cycle meets. A meet's assignment, if present, lies in the parent of its node. By convention, meets lying outside nodes have fixed assignments to meets lying inside nodes, and those assignments do not change.

A *task tree*, then, is a tree whose nodes each contain a set of tasks. The root node contains the cycle tasks (or there might be several root nodes, one for each resource type). A task's assignment, if present, lies in the parent of its node. By convention, tasks lying outside nodes have fixed assignments to tasks lying inside nodes, and those assignments do not change.

Type `KHE_TASKING` is KHE's nearest equivalent to a task tree node. It holds an arbitrary set of tasks, but there is no support for organizing taskings into a tree structure, since that does not seem to be needed. It is useful, however, to look at how tasks are structured in practice, and to relate this to task trees, even though they are not explicitly supported by KHE.

A task is assigned to a non-cycle task and fixed, to implement an avoid split assignments constraint. Such tasks would therefore lie outside nodes (if there were any). When a solver assigns a task to a cycle task, the task would have to lie in a child node of a node containing the cycle tasks (again, if there were any). So there are three levels: a first level of nodes containing the cycle tasks; a second level of nodes containing unfixed tasks wanting to be assigned resources; and a third level of fixed, assigned tasks that do not lie in nodes.

This shows that the three-way classification of tasks presented in Section 4.6.1, into cycle tasks, unfixed tasks, and fixed tasks, is a proxy for the missing task tree structure. Cycle tasks are first-level tasks, unfixed tasks are second-level tasks, and fixed tasks are third-level tasks. `KHE_TASKING` is only needed for representing second-level nodes, since tasks at the other levels do not require assignment. By convention, then, taskings will contain only unfixed tasks.

## 11.3. Task tree construction

KHE offers a solver for building a task tree holding the tasks of a given solution:

```
bool KheTaskTreeMake(KHE_SOLN soln, KHE_OPTIONS options);
```

As usual, this solver returns `true` if it changes the solution. Like any good solver, this function has no special access to data behind the scenes. Instead, it works by calling basic operations and helper functions:

- It calls `KheTaskingMake` to make one tasking for each resource type of `soln`'s instance, and it calls `KheTaskingAddTask` to add the unfixed tasks of each type to the tasking it made for that type. These taskings may be accessed by calling `KheSolnTaskingCount` and `KheSolnTasking` as usual, and they are returned in an order suited to resource assignment, as follows. Taskings for which `KheResourceTypeDemandIsAllPreassigned(rt)` is `true` come first. Their tasks will be assigned already if `KheSolnAssignPreassignedResources` has been called, as it usually has been. The remaining taskings are sorted by decreasing order of `KheResourceTypeAvoidSplitAssignmentsCount(rt)`. These functions are described in Section 3.5.1. Of course, the user is not obliged to follow this ordering. It is a precondition of `KheTaskTreeMake` that `soln` must have no taskings when it is called.

- It calls `KheTaskAssign` to convert resource preassignments into resource assignments, and to satisfy avoid split assignments constraints, as far as possible. Existing assignments are preserved (no calls to `KheTaskUnAssign` are made).

- It calls `KheTaskAssignFix` to fix the assignments it makes to satisfy avoid split assignments constraints. These may be removed later. At present it does not call `KheTaskAssignFix` to fix assignments derived from preassignments, although it probably should.

- It calls `KheTaskSetDomain` to set the domains of tasks to satisfy preassigned resources, prefer resources constraints, and other influences on task domains, as far as possible. `KheTaskTreeMake` never adds a resource to any domain, however; it either leaves a domain unchanged, or reduces it to a subset of its initial value.

These elements interact in ways that make them impossible to separate. For example, a prefer resources constraint that applies to one task effectively applies to all the tasks that are linked to it, directly or indirectly, by avoid split assignments constraints.

`KheTaskTreeMake` does not refer directly to any options. However, it calls function `KheTaskingMakeTaskTree`, described below, and so it is indirectly influenced by its options.

The implementation of `KheTaskTreeMake` has two stages. The first creates one tasking for each resource type of `soln`'s instance, in the order described, and adds to each the unfixed tasks of its type. This stage can be carried out separately by repeated calls to

```
KHE_TASKING KheTaskingMakeFromResourceType(KHE_SOLN soln,
  KHE_RESOURCE_TYPE rt);
```

which makes a tasking containing the unfixed tasks of `soln` of type `rt`, or of all types if `rt` is `NULL`. It aborts if any of these unfixed tasks already lies in a tasking.

The second stage is more complex. It applies public function

```
bool KheTaskingMakeTaskTree(KHE_TASKING tasking,
  KHE_TASK_BOUND_GROUP tbg, KHE_OPTIONS options);
```

to each tasking made by the first stage. When `KheTaskingMakeTaskTree` is called from within `KheTaskTreeMake`, its `options` parameter is inherited from `KheTaskTreeMake`.

As described for `KheTaskTreeMake`, `KheTaskingMakeTaskTree` assigns tasks and tightens domains; it does not unassign tasks or loosen domains. Only tasks in `tasking` are affected. If `tbg` is non-`NULL`, any task bounds created while tightening domains are added to `tbg`. Tasks assigned to non-cycle tasks have their assignments fixed, so are deleted from `tasking`.

The implementation of `KheTaskingMakeTaskTree` imitates the layer tree construction algorithm: it applies *jobs* in decreasing priority order. There are fewer kinds of jobs, but the situation is more complex in another way: sometimes, some kinds of jobs are wanted but not others. The three kinds of jobs of highest priority install existing domains and task assignments, and assign resources to unassigned tasks derived from preassigned event resources. These jobs are always included; the first two always succeed, and so does the third unless the user has made peculiar task or domain assignments earlier. The other kinds of jobs are optional, and whether they are included or not depends on the options (other than `rs_invariant`) described next.

`KheTaskTreeMake` consults the following options.

`rs_invariant`

A Boolean option which, when `true`, causes `KheTaskTreeMake` to omit assignments and domain tightenings which violate the resource assignment invariant (Section 12.2).

`rs_task_tree_prefer_hard_off`
: A Boolean option which, when `false`, causes `KheTaskTreeMake` to make a job for each point of application of each hard prefer resources constraint of non-zero weight. The priority of the job is the combined weight of its constraint, and it attempts to reduce the domains of the tasks of `tasking` monitored by the constraint's monitors so that they are subsets of the constraint's domain.

`rs_task_tree_prefer_soft`
: Like `rs_task_tree_prefer_hard_off` except that it applies to soft prefer resources constraints instead of hard ones, and its sense is reversed so that the default value (`false` as usual) omits these jobs. The author has encountered cases where reducing domains to enforce soft prefer resources constraints is harmful.

`rs_task_tree_split_hard_off`
: A Boolean option which, when `false`, causes `KheTaskTreeMake` to make a job for each point of application of each hard avoid split assignments constraint of non-zero weight. Its priority is the combined weight of its constraint, and it attempts to assign the tasks of `tasking` to each other so that all the tasks of the job's point of application of the constraint are assigned, directly or indirectly, to the same root task.

`rs_task_tree_split_soft_off`
: Like `rs_task_tree_split_hard_off` except that it applies to soft avoid split assignments constraints rather than hard ones.

`rs_task_tree_limit_busy_hard_off`
: A Boolean option which, when `false`, causes `KheTaskTreeMake` to make a job for each point of application of each limit busy times constraint with non-zero weight and maximum limit 0. Its priority is the combined weight of its constraint, and it attempts to reduce the domains of those tasks of `tasking` which lie in events preassigned the times of the constraint, to eliminate its resources, since assigning them to these tasks must violate this constraint. However, the resulting domain must have at least two elements; if not, the reduction is undone, reasoning that it is too severe and it is better to allow the constraint to be violated.

  This flag also applies to cluster busy times constraints with maximum limit 0, or rather to their positive time groups. These are essentially the same as the time groups of limit busy times constraints when the maximum limit is 0.

`rs_task_tree_limit_busy_soft_off`
: Like `rs_task_tree_limit_busy_hard_off` except that it applies to soft limit busy times constraints rather than hard ones.

By default, all of these jobs except `rs_task_tree_prefer_soft` are run.

## 11.4. Resource supply and demand

This section covers several topics which are not closely related, except that, in a general way, they are all concerned with the supply of and demand for resources.

### 11.4.1. Accounting for supply and demand

This section aims to understand the supply and demand for resources in practice.

Let $S$, the *supply*, be the sum, over all resources `r` of type `rt`, of the number of times that `r` could be busy without violating any resource constraints, as calculated by `KheResourceMaxBusyTimes` (Section 4.7.1). Let $D$, the *demand*, be the total duration of tasks of type `rt` for which there are assign resource constraints of non-zero weight. $S$ and $D$ depend only on the instance; they are the same for every solution.

Let the *excess supply* of resource type `rt` be $S - D$, the amount by which the supply of resources of that type exceeds the demand for them. This could be negative, in which case unassigned tasks or overloaded resources are inevitable.

Other considerations arise when we try to understand how supply and demand play out in a solution. Some resources may be *overloaded*: their actual number of busy times is larger than the value calculated by `KheResourceMaxBusyTimes`. Let $O$ be the sum, over all overloaded resources, of the excess. Other resources may be *underloaded*: their actual number of busy times is smaller than the value calculated by `KheResourceMaxBusyTimes`. Let $U$ be the sum, over all underloaded resources, of the amount by which each underloaded resource falls short. HSEval prints $O$ (in fact $-O$) and $U$ below each planning timetable. It should be clear that in a given solution, the number of busy times that resources actually supply is $S + O - U$.

There are also adjustments needed on the demand side. Some tasks that require assignment may in fact not be assigned. Let $X$ be their total duration. HSEval prints these tasks in the Unassigned row at the bottom of the planning timetable. Also, some tasks that do not require assignment may in fact be assigned. Let $Y$ be their total duration. HSEval prints these tasks in italics in planning timetables, and prints their total duration at the bottom of the timetables. In a given solution, the total duration of the tasks that are actually assigned is $D - X + Y$.

But now, each task that is actually assigned consumes one unit of resource supply, and vice versa, so we must have

$$D - X + Y = S + O - U$$

and rearranging gives

$$S - D = U - O + Y - X$$

$S - D$, the excess supply, depends only on the instance. So the quantity on the right is constant over all solutions for a given instance.

Now each unit of $O + X$ incurs a cost, but each unit of $U + Y$ incurs no cost. Nevertheless, minimizing $O + X$ is the same as minimizing $U + Y$, because their difference is a constant.

### 11.4.2. Classifying resources by available workload

Resources with high workload limits, as indicated by functions `KheResourceMaxBusyTimes` and `KheResourceMaxWorkload` (Section 4.7), may be harder to exploit than resources with lower workload limits, so it may make sense to timetable them first. Function

```
bool KheClassifyResourcesByWorkload(KHE_SOLN soln,
  KHE_RESOURCE_GROUP rg, KHE_RESOURCE_GROUP *rg1,
  KHE_RESOURCE_GROUP *rg2);
```

helps with that. It partitions `rg` into two resource groups, `rg1` and `rg2`, such that the highest workload resources are in `rg1`, and the rest are in `rg2`. It returns `true` if it succeeds with this, and `false` if not, which will be because the resources of `rg` have equal maximum workloads.

If `KheClassifyResourcesByWorkload` returns `true`, every resource in `rg1` has a maximal value of `KheResourceMaxBusyTimes` and a maximal value of `KheResourceMaxWorkload`, and every element of `rg2` has a non-maximal value of `KheResourceMaxBusyTimes` or a non-maximal value of `KheResourceMaxWorkload`. If it returns `false`, then `rg1` and `rg2` are `NULL`.

### 11.4.3. Limits on consecutive days, and rigidity

Nurse rostering instances typically place minimum and maximum limits on the number of consecutive days that a resource can be free, busy, or busy working a particular shift. These limits are scattered through constraints and may be hard to find. This section makes that easy.

An object called a *consec solver* is used for this. To create one, call

```
KHE_CONSEC_SOLVER KheConsecSolverMake(KHE_SOLN soln, KHE_FRAME frame);
```

It uses memory from an arena taken from `soln`. Its attributes may be retrieved by calling

```
KHE_SOLN KheConsecSolverSoln(KHE_CONSEC_SOLVER cs);
KHE_FRAME KheConsecSolverFrame(KHE_CONSEC_SOLVER cs);
```

The frame must contain at least one time group, otherwise `KheConsecSolverMake` will abort.

To delete a solver when it is no longer needed, call

```
void KheConsecSolverDelete(KHE_CONSEC_SOLVER cs);
```

This works by returning the arena to the solution.

To find the limits for a particular resource, call

```
void KheConsecSolverFreeDaysLimits(KHE_CONSEC_SOLVER cs, KHE_RESOURCE r,
  int *history, int *min_limit, int *max_limit);
void KheConsecSolverBusyDaysLimits(KHE_CONSEC_SOLVER cs, KHE_RESOURCE r,
  int *history, int *min_limit, int *max_limit);
void KheConsecSolverBusyTimesLimits(KHE_CONSEC_SOLVER cs, KHE_RESOURCE r,
  int offset, int *history, int *min_limit, int *max_limit);
```

For any resource `r`, these return the history (see below), the minimum limit, and the maximum limit on the number of consecutive free days, the number of consecutive busy days, and the number of consecutive busy times which appear `offset` places into each time group of `frame`. Setting `offset` to 0 might return the history and limits on the number of consecutive early shifts, setting it to 1 might return the limits on the number of consecutive day shifts, and so on. The largest offset acceptable to `KheConsecSolverBusyTimesLimits` is returned by

```
int KheConsecSolverMaxOffset(KHE_CONSEC_SOLVER cs);
```

An `offset` larger than this, or negative, produces an abort.

The `*history` values return history: the number of consecutive free days, consecutive busy days, and consecutive busy times with the given `offset` in the timetable of `r` directly before the timetable proper begins. They are taken from the history values of the same constraints that determine the `*min_limit` and `*max_limit` values.

All these results are based on the frame passed to `KheConsecSolverFrame`, which would always be the common frame. They are calculated by finding all limit active intervals constraints with non-zero weight, comparing their time groups with the frame time groups, and checking their polarities. In effect this reverse engineers what programs like NRConv do when they convert specialized nurse rostering formats to XESTT.

If no constraint applies, `*history` is set to 0, `*min_limit` is set to 1 (a sequence of length 0 is not a sequence at all), and `*max_limit` is set to `KheFrameTimeGroupCount(frame)`. In the unlikely event that more than one constraint applies, `*history` and `*min_limit` are set to the largest of the values from the separate constraints, and `*max_limit` is set to the smallest of the values from the separate constraints.

The *rigidity* of a resource is how constrained it is to follow a particular pattern of busy and free days, assuming that it is utilized to the maximum extent that constraints allow, as reported by `KheResourceMaxBusyTimes` (Section 4.7.1). Rigidity takes account of constraints on the number of consecutive busy days and consecutive free days, plus history.

It is hard to see how a local repair method, for example ejection chains (Section 12.10), can just stumble on a good timetable for a rigid resource (although it often does). Something more targeted, like optimal assignment using dynamic programming (Section 12.6), seems indicated.

Suppose that resource *r* has 20 available busy times, that the cycle has 28 days, that *r*'s busy days are limited to at most 5 consecutive days, and that its free days are limited to at least 2 consecutive days. Then to reach the 20 busy days economically we need runs of 5 consecutive busy days, separated by runs of 2 consecutive free days. A typical pattern would be

(5 busy, 2 free, 5 busy, 2 free, 5 busy, 2 free, 5 busy, 2 free)

The only freedoms here are to move the last two free days to other points in the cycle, or else to move two or more busy times to the end.

Resources with few available times can also be rigid. Suppose that *r* has 6 available busy times, that the cycle has 28 days, that *r*'s busy days are limited to at least 2 consecutive days, and that its free days are limited to at most 7 consecutive days. (This is an actual example, from an INRC2 instance.) A typical pattern would be

(7 free, 2 busy, 7 free, 2 busy, 7 free, 2 busy, 1 free)

The only freedom here is to move up to 6 free days to the end, another rigid case. We've just shown, for example, that *r*'s first and last days must be free.

For an example of a resource which is *not* rigid, let *r* have 15 available busy times, subject to the same constraints as the two previous resources. A typical pattern would be

(5 busy, 2 free, 5 busy, 2 free, 5 busy, 9 free)

This is not quite legal because the last run of free days is too long, but it's close, and there are many choices for moving two or more of those 9 free days forward, and for regrouping the busy sequences, for example into three runs of 4 days and one run of 3 days.

The ideal measure of rigidity (actually non-rigidity) would be the number of distinct zero cost patterns of busy and free days. But that seems impracticable to calculate, and anyway we do not need a precise measure. The measure we choose is inspired by the examples given above. It is a weighted sum of two parts, $m_1$ and $m_2$:

- First, we ask what is the smallest number of runs of consecutive busy days that we can have and still reach our desired number of busy days without violating any minimum or maximum limits on consecutive busy or free days? And what is the largest number? The difference is $m_1$, our first measure of non-rigidity. (Other measures are correlated with this one. For example, if the number of runs can vary, their lengths can vary as well.)

- Second, we ask what choices there are for placing the first run of consecutive busy days, consistent with history. For example, if there are 2 busy days from history, and the minimum limit is 3, then there is no choice for the first run of busy days: it must start on the first day. Or if there are 5 free days in history, and the maximum number of consecutive free days is 7, then the first run of busy days must start on the first, second, or third day. The number of choices here is $m_2$, our second measure of non-rigidity.

We weight the first measure by 10 and the second by 1.

For the resource with 20 available times above, at least 4 runs are required, because each run can have at most 5 busy times. At most 5 runs can be used, because if 6 runs are used there are 5 gaps between runs, each containing at least 2 times, leaving at most 18 places for busy times. So $m_1 = 5 - 4 = 1$.

For the resource with 6 available times, at most 3 runs are possible, because each run has at least 2 busy times. And 2 runs doesn't work, because it leaves only three free runs, each with at most 7 free times, to hold the 22 free times. So $m_1 = 3 - 3 = 0$.

For the resource with 15 available times it is a little harder to see what the possibilities are. A somewhat rough and ready general method works like this. Suppose all busy runs have length $x$, except possibly one run that is shorter, and all free runs have length $y$. If the number of busy times we want is $a$, then the number of busy runs is $c = \lceil a/x \rceil$. We must place one free run of length $y$ between each adjacent pair of busy runs, and optionally we can place one free run of length $y$ before the first run and after the last run. This gives a total number of times (busy plus free) of between $a + y(c - 1)$ and $a + y(c + 1)$. If the total number of times in the cycle is between these limits, then $x$ and $y$ are workable choices and $c$ is a workable number of busy runs.

Now $x$ and $y$ are bounded by limits set by constraints. So we try each combination of one legal choice for $x$ and one for $y$ and see what workable values for $c$ we get. The first measure of non-rigidity, $m_1$, is the difference between the largest and smallest workable values for $c$.

A general method of calculating the second measure of non-rigidity goes like this. Suppose that the minimum length of a run of consecutive busy times is $b_{min}$, and the maximum length is $b_{max}$. Suppose that the minimum length of a run of consecutive free times is $f_{min}$, and the maximum length is $f_{max}$. And suppose that the number of consecutive busy days from history is $b$, and the number of consecutive free days from history is $f$. At most one of $b$ and $f$ can be non-zero, and we also have $1 \leq b_{min} \leq b_{max}$, and $1 \leq f_{min} \leq f_{max}$.

If $b = f = 0$, then the first day could be busy, contributing 1 to $m_2$, or else any number of initial days from $f_{min}$ to $f_{max}$ inclusive could be free, contributing a further $f_{max} - f_{min} + 1$ to $m_2$.

If $b > 0$, then $f = 0$. If $b < b_{min}$, then the first day must be busy, so $m_2 = 1$. If $b_{min} \leq b < b_{max}$, then the first day could be busy, contributing 1 to $m_2$, or free, contributing $f_{max} - f_{min} + 1$ to $m_2$. If $b \geq b_{max}$, then the first day must be free, and $m_2 = f_{max} - f_{min} + 1$.

If $f > 0$, then $b = 0$. If $f < f_{min}$, then the first day must be free, and the number of initial free days may be between $f_{min} - f$ and $f_{max} - f$ inclusive, making $m_2 = f_{max} - f_{min} + 1$ choices altogether. If $f_{min} \leq f < f_{max}$, then the first day could be busy, contributing 1 to $m_2$, or free, contributing a further $f_{max} - f$ to $m_2$. If $f_{max} \leq f$, then the first day must be busy, so $m_2 = 1$.

Function

```
int KheConsecSolverNonRigidity(KHE_CONSEC_SOLVER cs, KHE_RESOURCE r);
```

returns the non-rigidity as we have defined it here. There is no precise threshold separating non-rigidity from rigidity, but for the first measure a value of 0 is very rigid, 1 is somewhat rigid, and 2 is non-rigid, arguably. For the second measure a similar statement is reasonable. Rather than worrying about thresholds it may be better to sort the resources by increasing non-rigidity and treat, say, the first 20% or 30% of them as rigid.

Finally,

```
void KheConsecSolverDebug(KHE_CONSEC_SOLVER cs, int verbosity,
  int indent, FILE *fp);
```

produces the usual debug print of `cs` onto `fp` with the given verbosity and indent. When `verbosity >= 2`, this prints all results for all resources, using format `history|min-max`. For efficiency, these are calculated all at once by `KheConsecSolverMake`.

## 11.4.4. Tighten to partition

Suppose we are dealing with teachers, and that they have partitions (Section 3.5.1) which are their faculties (English, Mathematics, Science, and so on). Some partitions may be heavily loaded (that is, required to supply teachers for tasks whose total workload approaches the total available workload of their resources) while others are lightly loaded.

Some tasks may be taught by teachers from more than one partition. These *multi-partition tasks* should be assigned to teachers from lightly loaded partitions, and so should not overlap in time with other tasks from these partitions. *Tighten to partition* tightens the domain of each multi-partition task in a given tasking to one partition, returning `true` if it changes anything:

```
bool KheTaskingTightenToPartition(KHE_TASKING tasking,
  KHE_TASK_BOUND_GROUP tbg, KHE_OPTIONS options);
```

The choice of partition is explained below. All changes are additions of task bounds to tasks, and if `tbg` is non-`NULL`, all these task bounds are also added to `tbg`.

It is best to call `KheTaskingTightenToPartition` after preassigned meets are assigned, but before general time assignment. The tightened domains encourage time assignment to avoid the undesirable overlaps. After time assignment, the changes should be removed, since otherwise they constrain resource assignment unnecessarily. This is what the task bound group is for:

```
tighten_tbg = KheTaskBoundGroupMake(soln);
for( i = 0;  i < KheSolnTaskingCount(soln);  i++ )
  KheTaskingTightenToPartition(KheSolnTasking(soln, i),
    tighten_tbg, options);
... assign times ...
KheTaskBoundGroupDelete(tighten_tbg);
```

The rest of this section explains how `KheTaskingTightenToPartition` works in detail.

`KheTaskingTightenToPartition` does nothing when the tasking has no resource type, or `KheResourceTypeDemandIsAllPreassigned` (Section 3.5.1) says that the resource type's tasks are all preassigned, or the resource type has no partitions, or its number of partitions is less than four or more than one-third of its number of resources. No good can be done in these cases.

Tasks whose domains lie entirely within one partition are not touched. The remaining multi-partition tasks are sorted by decreasing combined weight then duration, except that tasks with a *dominant partition* come first. A task with an assigned resource has a dominant partition, namely the partition that its assigned resource lies in. An unassigned task has a dominant partition when at least three-quarters of the resources of its domain come from that partition.

For each task in turn, an attempt is made to tighten its domain so that it is a subset of one partition. If the task has a dominant partition, only that partition is tried. Otherwise, the partitions that the task's domain intersects with are tried one by one, stopping at the first success, after sorting them by decreasing average available workload (defined next).

Define the *workload supply* of a partition to be the sum, over the resources $r$ of the partition, of the number of times in the cycle minus the number of workload demand monitors for $r$ in the matching. Define the *workload demand* of a partition to be the sum, over all tasks $t$ whose domain is a subset of the partition, of the workload of $t$. Then the *average available workload* of a partition is its workload supply minus its workload demand, divided by its number of resources. Evidently, if this is large, the partition is lightly loaded.

Each successful tightening increases the workload demand of its partition. This ensures that equally lightly loaded partitions share multi-partition tasks equally.

In a task with an assigned resource, the dominant partition is the only one compatible with the assignment. In a task without an assigned resource, preference is given to a dominant partition, if there is one, for the following reason. Schools often have a few *generalist teachers* who are capable of teaching junior subjects from several faculties. These teachers are useful for fixing occasional problems, smoothing out workload imbalances, and so on. But the workload that they can give to faculties other than their own is limited and should not be relied on. For example, suppose there are five Science teachers plus one generalist teacher who can teach junior Science. That should not be taken by time assignment as a licence to routinely schedule six Science meets simultaneously. Domain tightening to a dominant partition avoids this trap.

Tightening by partition works best when the `rs_invariant` option of `options` is `true`. For example, in a case like Sport where there are many simultaneous multi-partition tasks, it will then not tighten more of them to a lightly loaded partition than there are teachers in that partition. Assigning preassigned meets beforehand improves the effectiveness of this check.

### 11.4.5. Balancing supply and demand

This section presents a solver for investigating the balance between supply of and demand for resources of a given type. Its main aim is to answer this question: if some resource is not used up to its full capacity, what cost will that have in terms of tasks not assigned?

To create a balance solver, call

```
KHE_BALANCE_SOLVER KheBalanceSolverMake(KHE_SOLN soln,
  KHE_RESOURCE_TYPE rt, HA_ARENA a);
```

It makes a solver for the supply of and demand for resources of type `rt` in `soln`, using memory from arena `a`. There is no deletion operation; the solver is deleted when `a` is freed.

To find the total supply of resources of type `rt`, call

```
int KheBalanceSolverTotalSupply(KHE_BALANCE_SOLVER bs);
```

This calls `KheResourceMaxBusyTimes(soln, r, &res)` for each resource `r` of type `rt`, and returns the sum of the `res` values. As documented in Section 4.7, `res` is an upper limit on `r`'s number of busy times (as imposed by constraints) minus its current number of busy times.

To find the total demand for resources of type `rt`, call

```
int KheBalanceSolverTotalDemand(KHE_BALANCE_SOLVER bs);
```

This is the sum, over all unassigned tasks `t` of type `rt`, of the total duration of `t`, as returned by `KheTaskTotalDuration(t)` (Section 4.6.1).

The balance solver analyses this demand by cost reduction. For each task `t` that contributes to `KheBalanceSolverTotalDemand(bs)`, it calls `KheTaskAssignmentCostReduction` (Section 4.6.1) on `t`, and groups tasks with equal cost reductions. To access these groups, call

```
int KheBalanceSolverDemandGroupCount(KHE_BALANCE_SOLVER bs);
void KheBalanceSolverDemandGroup(KHE_BALANCE_SOLVER bs, int i,
  KHE_COST *cost_reduction, int *total_durn);
```

`KheBalanceSolverDemandGroup` returns the information kept about the `i`th group: the cost reduction of each of its tasks, and their total duration. `KheBalanceSolverTotalDemand` returns the sum of these total durations. The groups are visited in order of decreasing cost reduction.

Using this information it is easy to work out the marginal cost of not utilising a resource `r` to its full capacity. Suppose that tasks are assigned in order of decreasing cost reduction, until all resources are used to capacity. The cost reduction of the last task assigned is the marginal cost of not fully utilizing `r`. This value is returned by

```
KHE_COST KheBalanceSolverMarginalCost(KHE_BALANCE_SOLVER bs);
```

If supply exceeds demand, there is no marginal cost, and so the value returned is 0. Finally,

```
void KheBalanceSolverDebug(KHE_BALANCE_SOLVER bs, int verbosity,
  int indent, FILE *fp);
```

produces the usual debug print of `bs` onto `fp` with the given verbosity and indent.

### 11.4.6. Resource flow

It is arguably too simple to just compare the total supply of resources with the total demand for them. The tasks which constitute the demand have prefer resources monitors (hard and soft) which restrict which resources can be used. There could be enough supply overall but not enough of a particular kind: enough nurses but not enough senior nurses, enough rooms but not enough Science laboratories, and so on.

We can detect such problems now using the global tixel matching. However, here we build a *flow graph* (a directed graph in which we will find a maximum flow) that is much smaller than the global tixel matching. This graph gives a clearer view of the overall situation than one can get from a bipartite matching. We call this general idea *resource flow*, or just *flow*.

A flow graph is for a given resource type `rt`. It is built from a set of *admissible resources* and a set of *admissible tasks*. The admissible resources are just the resources of type `rt`. A task is admissible when all of these conditions hold:

1. It has type `rt`.
2. It is a proper root task.
3. It is derived from an event resource (needed because we use the event resource's domain).
4. It is not preassigned.
5. Its assignment is not fixed.
6. `KheTaskNonAsstAndAsstCost` (Section 11.9.1) gives it a positive non-assignment cost.
7. It is not assigned a resource. This condition is optional; it is present when parameter `preserve_assts` of function `KheFlowMake` below has value `true`.

As usual, the *total duration* of a proper root task is the duration of the task plus the durations of all the tasks assigned to it, directly or indirectly.

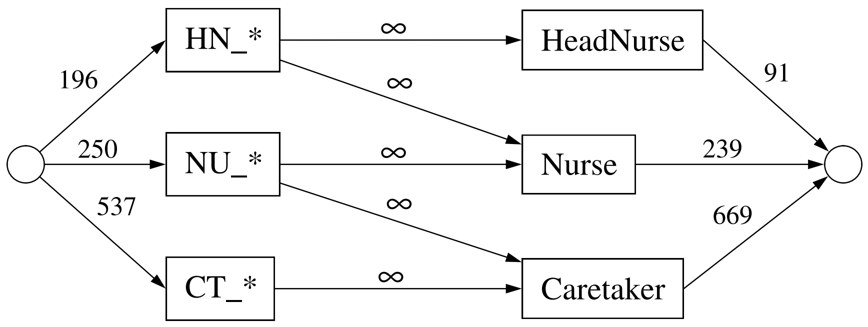
The flow graph contains a source node, some *resource nodes* (each containing a set of one or more admissible resources), some *task nodes* (each containing a set of one or more admissible tasks), and a sink node.

For each admissible resource, define a resource node $x$ containing just that resource. Add an edge from the source node to $x$, whose capacity $c(x)$ is the number of times that the resource is available, according to `KheResourceMaxBusyTimes` (Section 4.7.1). If the resource is currently assigned to any inadmissible proper root tasks, then reduce $c(x)$ by the total duration of those tasks (but not below 0) to compensate for their omission.

For each distinct set $R_y$ of resources preferred by at least one task, define a *task node* $y$ containing all tasks that prefer $R_y$, and add an edge from $y$ to the sink node, whose capacity $c(y)$ is the total duration of those tasks. Then for each resource node $x$ and each task node $y$, draw an edge from $x$ to $y$ of infinite capacity whenever $x$'s resource lies in $R_y$.

Before solving this graph, we compress it by merging resource nodes that are connected to the same task nodes. Each such merged node has incoming capacity equal to the total capacity of the nodes it replaces, and outgoing edges like the edges it replaces.

Here is an example of a flow graph from a real instance (INRC2-4-100-0-1108):



Node HN_* holds the head nurses, node HeadNurse holds the tasks that require a head nurse, and so on. This example substantiates our claim about the clarity of flow graphs: it shows that head nurses can do the work of ordinary nurses as well as their own, and ordinary nurses can do the work of caretaker nurses as well as their own. This is just as well, because, as the graph also shows, head nurses have a superfluity of available workload and caretakers have a shortage.

This flow graph can answer many questions. Each resource node is the answer to the question ‘What kind of resource is this?’, although that answer does not come with a simple name in general. (We will compare the sets of resources we get with existing resource groups, so that we can give the nodes familiar names whenever possible. But the algorithm deals with sets of resources that it defines itself, not with sets defined previously as resource groups.)

Call a maximum flow in this graph the *original flow*. By changing the graph and seeing whether the new maximum flow is less than the original, we can answer questions like these:

- Can at least one of the tasks of task node $y$ be assigned a resource from resource node $x$? Subtract 1 from $c(x)$ and $c(y)$ and find a maximum flow. If this flow is just 1 less than the original flow, then a maximum flow that uses this edge exists: take this flow and add one unit of flow from the source to $x$ to $y$ to the sink. If the answer is no, we might as well delete the edge from $x$ to $y$. This may interest callers since it simplifies the situation.

- Must the resources of $x$ be used exclusively by $y$? Yes, if for every other $y$ connected to $x$ the previous question has answer no.

- Can the tasks of $y$ be limited to resources from $x$? Remove all edges into $y$ other than the one from $x$ and find a maximum flow. The answer is yes if this equals the original flow.

There are many possible questions; our plan is to implement them as we need them.

The implementation defines three types. Type `KHE_FLOW` represents the entire flow graph; type `KHE_FLOW_RESOURCE_NODE` represents one resource node; and type `KHE_FLOW_TASK_NODE` represents one task node.

We start with type `KHE_FLOW_RESOURCE_NODE`. Its operations are

```
KHE_RESOURCE_SET KheFlowResourceNodeResources(
  KHE_FLOW_RESOURCE_NODE frn);
bool KheFlowResourceNodeResourceGroup(KHE_FLOW_RESOURCE_NODE frn,
  KHE_RESOURCE_GROUP *rg);
int KheFlowResourceNodeCapacity(KHE_FLOW_RESOURCE_NODE frn);
bool KheFlowResourceNodeFlow(KHE_FLOW_RESOURCE_NODE frn,
  KHE_FLOW_TASK_NODE *ftn, int *flow);
void KheFlowResourceNodeDebug(KHE_FLOW_RESOURCE_NODE frn,
  int verbosity, int indent, FILE *fp);
```

`KheFlowResourceNodeResources` returns the set of resources represented by flow resource node `frn`. If `KheFlowResourceNodeResourceGroup` returns `true`, then the pre-existing resource group `*rg` contains exactly these resources. `KheFlowResourceNodeCapacity` returns the total capacity of those resources (the sum of their individual capacities, defined above). `KheFlowResourceNodeFlow` reports the results of a max flow solve on the graph. It is to be called repeatedly, and each time it returns `true` it reports one edge with flow `*flow` from `frn` to `*ftn`. So it should be called like this:

```
while( KheFlowResourceNodeFlow(frn, &ftn, &flow) )
  ... there is a non-zero flow from frn to ftn ...
```

Finally, `KheFlowResourceNodeDebug` produces a debug print of `frn` in the usual way.

The operations on type `KHE_FLOW_TASK_NODE` are

```
KHE_TASK_SET KheFlowTaskNodeTasks(KHE_FLOW_TASK_NODE ftn);
KHE_RESOURCE_GROUP KheFlowTaskNodeDomain(KHE_FLOW_TASK_NODE ftn);
int KheFlowTaskNodeCapacity(KHE_FLOW_TASK_NODE ftn);
void KheFlowTaskNodeDebug(KHE_FLOW_TASK_NODE ftn, int verbosity,
  int indent, FILE *fp);
```

`KheFlowTaskNodeTasks` returns the set of tasks represented by `ftn`. `KheFlowTaskNodeDomain` returns the domain they share. `KheFlowTaskNodeCapacity` returns their capacity (their total duration); and `KheFlowTaskNodeDebug` produces a debug print of `ftn` in the usual way.

Now for the operations on type `KHE_FLOW`. A flow object is created and deleted by calling

```
KHE_FLOW KheFlowMake(KHE_SOLN soln, KHE_RESOURCE_TYPE rt,
  bool preserve_assts, bool include_soft);
void KheFlowDelete(KHE_FLOW f);
```

`KheFlowMake` builds the flow object in an arena taken from `soln`, including creating its resource and task nodes as defined above, and finding a maximum flow. `KheFlowDelete` returns the arena to `soln`, making `f`, its nodes, and the resource sets and task sets from its nodes undefined. (The task sets are not created within `f`'s arena, because the task set interface does not offer that option. But `KheFlowDelete` explicitly deletes them.)

The flow object returned by `KheFlowMake` accepts a variety of queries. Its resource nodes may be visited (sorted by increasing index of their resource sets' first resources) by

```
int KheFlowResourceNodeCount(KHE_FLOW f);
KHE_FLOW_RESOURCE_NODE KheFlowResourceNode(KHE_FLOW f, int i);
```

Its task nodes may be visited (in an unspecified order) by

```
int KheFlowTaskNodeCount(KHE_FLOW f);
KHE_FLOW_TASK_NODE KheFlowTaskNode(KHE_FLOW f, int i);
```

There is also

```
KHE_FLOW_RESOURCE_NODE KheFlowResourceToResourceNode(KHE_FLOW f,
  KHE_RESOURCE r);
KHE_FLOW_TASK_NODE KheFlowTaskToTaskNode(KHE_FLOW f, KHE_TASK task);
```

These return the resource node containing `r` and the task node containing `task`, or `NULL` if there is no such node (if `r` or `task` is not admissible). Finally,

```
void KheFlowDebug(KHE_FLOW f, int verbosity, int indent, FILE *fp);
```

produces a debug print of `f` onto `fp` with the given verbosity and indent.

### 11.4.7. Workload packing

The solver in this section is inspired by instance `COI-WHPP`, in which each resource has maximum workload 70, day shifts have workload 7, night shifts have workload 10, and the total supply of workload is just sufficient to meet the total demand. Given these conditions, and the absence of significant other conditions, it is not hard to see that in the best solutions, some resources will be assigned 10 day shifts only, and the rest will be assigned 7 night shifts only. It is not a real-world scenario, but it is the scenario in this instance.

Function

```
bool KheWorkloadPack(KHE_SOLN soln, KHE_OPTIONS options,
  KHE_RESOURCE_TYPE rt, KHE_TASK_BOUND_GROUP *tbg)
```

checks to see whether a scenario like the one above occurs in `soln` in the tasks and resources of type `rt`. If so, it installs task bounds into the tasks of type `rt` to enforce this kind of partitioned solution. This involves making heuristic decisions about which resources will get day shifts and which will get night shifts. If all goes well, it sets `*tbg` to a task bound group containing the task bounds it created and added to tasks, and returns `true`. Otherwise it adds no task bounds to tasks, sets `*tbg` to `NULL`, and returns `false`.

If task bounds were added, a call to `KheTaskBoundGroupDelete(*tbg)` can be used to remove them again. This deletes the task bound group, including deleting any task bounds in it, which in turn removes them from the tasks they were added to.

`KheWorkloadPack` does not assign resources to tasks. It leaves that to other solvers. They are forced by the task bounds to do it in the way that `KheWorkloadPack` has decided on.

The rest of this section presents the details of how `KheWorkloadPack` works. We begin with the conditions under which it acts.

Let *S* be the set of event resources of type `rt` with non-zero workload for which assign resource constraints with non-zero weight are present. (Event resources with zero workload can be assigned freely without affecting the workload packing calculation. Event resources without assign resource constraints of non-zero weight do not need to be assigned at all.) Over all elements of $S$ there must be exactly two distinct workloads, $w_1$ and $w_2$ say. Each is a workload, not a workload per time, and so is a positive integer. We require $w_1$ and $w_2$ to be relatively prime.

Now suppose that for some resource $r$ the workload limit is $W = w_1 w_2$. Then the only way to assign $r$ to event resources from $S$ that completely exhausts $r$'s workload is for all of the event resources assigned $r$ to have the same workload, say $w_i$, and for $r$ to be assigned $W/w_i$ such event resources. The proof of this is by contradiction, as follows.

Any other arrangement leads to a total workload for $r$ of the form

$$a_1 w_1 + a_2 w_2 = W = w_1 w_2$$

where $a_1$ and $a_2$ are positive integers. Dividing through by $w_1$ shows that $w_1$ divides $a_2$ (because $w_1$ and $w_2$ are relatively prime), and similarly $w_2$ divides $a_1$. So let $a_1 = b_1 w_2$ and $a_2 = b_2 w_1$ where $b_1$ and $b_2$ are positive integers. This gives

$$b_1 w_2 w_1 + b_2 w_1 w_2 = w_1 w_2$$

Dividing by $w_1 w_2$ gives $b_1 + b_2 = 1$, a contradiction, because $b_1$ and $b_2$ are positive integers.

Each resource of type `rt` must have limit workload constraints of non-zero weight which give it maximum workload $W = w_1 w_2$, according to `KheResourceMaxWorkload` (Section 4.7.1). If there are $n$ resources of type `rt`, then the total workload supply is $nW$. The total workload of the elements of $S$ must be at least $nW$, so that workload demand equals or exceeds supply.

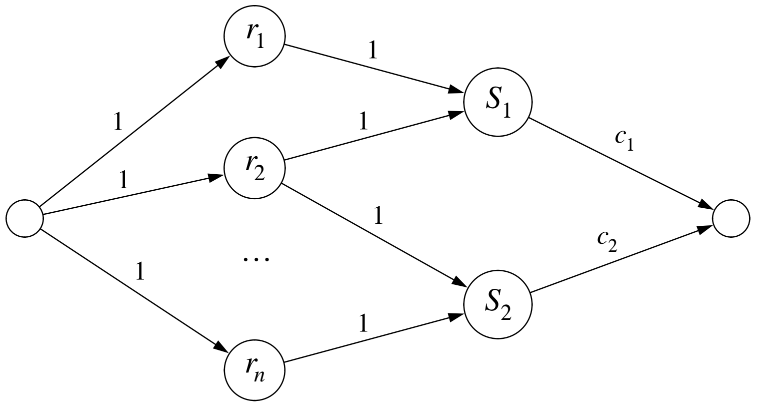
Finally, we need to decide which resources to assign to the event resources with workload $w_1$, and which to assign to the event resources with workload $s_2$. We do this as follows.

Let $S_1$ be the set of event resources from $S$ whose workload is $w_1$, and let $S_2$ be the set of event resources from $S$ whose workload is $w_2$. Let $R = \{r_1, \ldots, r_n\}$ be the resources of type `rt`. We need to partition $R$ into $R_1$, the resource group of the task bound applied to the event resources of $S_1$, and $R_2$, the resource group of the task bound applied to the event resources of $S_2$.

Each event resource of $S_1$ has workload $w_1$, making a total workload of $|S_1|w_1$. From the work done above, each resource has maximum workload $W = w_1 w_2$, so the number of resources needed to cover the event resources of $S_1$ is

$$c_1 = |S_1|w_1 / w_1 w_2 = |S_1| / w_2$$

Similarly, $c_2 = |S_2| / w_1$ resources are needed to cover the event resources of $S_2$. Suitable resources can be selected using a maximum flow in this graph:



The flow along each edge is an integral number of resources. Each resource $r_i$ is represented by a node at the end of an edge of weight 1 from the source, ensuring that each resource is utilized at most once. Each set of event resources $S_j$ is represented by a node at the start of an edge of weight $c_j$ to the sink, ensuring that at most $c_j$ resources are utilized by the event resources of $S_j$. An edge of weight 1 joins each $r_i$ to each $S_j$ such that $r_i$ is qualified for $S_j$, in the sense that $r_i$ lies in the domain of sufficiently many elements of $S_j$ to consume its entire maximum workload.

We don't actually build this flow graph, although we could. Instead, we find all the $r_i$ which are qualified for $S_1$ only and place them into $R_1$, taking care not to add more than $c_1$ resources to $R_1$. Then we find all the $r_i$ which are qualified for $S_2$ only and place them into $R_2$, taking care not to add more than $c_2$ resources to $R_2$. Finally we make arbitrary assignments of the remaining resources to $R_1$ or $R_2$, again taking care not to exceed the $c_1$ and $c_2$ limits.

At various points in this algorithm we may find that we are unable to utilize some resource. In that case the maximum flow is less than $n$, so we abandon workload packing.

## 11.5. Solution adjustments

This section presents solution adjustments (Section 8.6) for resource-structural applications.

### 11.5.1. Changing the multipliers of cluster busy times monitors

Cluster busy times monitors formerly had a *multiplier*, which was an integer that their true costs were multiplied by. Multipliers have been made redundant by `KheMonitorSetCombinedWeight` (Section 6.2), but the solver they supported is still available, with a change of interface:

```
void KheSetClusterMonitorMultipliers(KHE_SOLN soln,
  KHE_SOLN_ADJUSTER sa, char *str, int val);
```

This finds each cluster busy times constraint `c` whose name or Id contains `str`, and uses calls to `KheSolnAdjusterMonitorChangeWeight` to multiply the combined weight of each monitor derived from `c` by `val`. If `sa != NULL`, then the monitors can easily be returned to their previous state later:

```
    sa = KheSolnAdjusterMake(soln);
    KheSetClusterMonitorMultipliers(sa, str, val);
    do_something;
    KheSolnAdjusterDelete(sa);
```

The multipliers are in place while `do_something` is running, and removed afterwards.

### 11.5.2. Tilting the plateau

This section documents a rather left-field idea, which we call *tilting the plateau*. The idea is to consider a defect near the start of the timetable to be worse than an equally bad defect near the end of the timetable. A local search method like ejection chains will then believe that it has succeeded when it moves a defect towards the end of the timetable. The hope is that over the course of several repairs, defects will move all the way to the end and disappear.

The function for this is

```
    void KheTiltPlateau(KHE_SOLN soln, KHE_SOLN_ADJUSTER sa);
```

For each monitor $m$ of `soln` whose combined weight $w$ satisfies $w > 0$, `KheMonitorSetCombinedWeight` (Section 6.2) is called to change the combined weight of $m$ from $w$ to $wT - t$, where $T$ is the number of times in the instance, and $t$ is the index of the first time monitored by $m$, as returned by `KheMonitorTimeRange` (Section 6.4), or 0 if `KheMonitorTimeRange` returns `false`. Multiplying every monitor's weight by $T$ does not really change the instance, but subtracting $t$ makes monitors near the end of the timetable less costly than monitors near the start.

When $m$ is a limit active intervals monitor whose combined weight $w$ satisfies $w > 0$, the procedure is somewhat different. The new combined weight is $wT$, not $wT - t$; but then $m$ itself is informed that tilting is in force, by a call to `KheLimitActiveIntervalsMonitorSetTilt` (Section 6.7.7). This causes $m$ to perform its own subtraction of $t$ from each cost it reports, but using a different value of $t$ for each defective interval, namely the index of the first time in that interval. In this way, defective intervals near the end cost less than defective intervals near the start.

`KheTiltPlateau` may be used in conjunction with a solution adjuster:

```
    sa = KheSolnAdjusterMake(soln);
    KheTiltPlateau(soln, sa);
    do_something;
    KheSolnAdjusterDelete(sa);
```

The tilt applies during `do_something`; `KheSolnAdjusterDelete` removes it, including making the appropriate calls to `KheLimitActiveIntervalsMonitorClearTilt`. Alternatively, the `sa` parameter of `KheTiltPlateau` may be `NULL`, but then there will be no simple way to remove the tilt.

### 11.5.3. Propagating unavailable times to resource monitors

A resource $r$'s *unavailable times*, $U_r$, is a set of times taken from certain monitors of non-zero weight that apply to $r$: all times in avoid unavailable times monitors, all times in limit busy times monitors with maximum limit 0, and all times in positive time groups of cluster busy times constraints with maximum limit 0. In this section we do not care about the weight of these monitors, provided it is non-zero. We simply combine all these times into $U_r$.

Suppose that $r$ has a cluster busy times or limit active intervals monitor $m$ with a time group $T$ such that $T \subseteq U_r$. Then, although $T$ could be busy, it is not likely to be busy, and it is reasonable to let $m$ know this, by calling `KheClusterBusyTimesMonitorSetNotBusyState` (Section 6.7.4) or `KheLimitActiveIntervalsMonitorSetNotBusyState` (Section 6.7.7).

KHE offers a solver that implements this idea:

```
bool KhePropagateUnavailableTimes(KHE_SOLN soln, KHE_RESOURCE_TYPE rt);
```

For each resource $r$ of type `rt` in `soln`'s instance (or for each resource of the instance if `rt` is `NULL`), it calculates $U_r$, and, if $U_r$ is non-empty, it checks every time group $T$ in every cluster busy times and limit active intervals monitor for $r$. For each $T \subseteq U_r$, it calls the function appropriate to the monitor, with `active` set to `false` if $T$ is positive, and to `true` if $T$ is negative. It returns `true` if it changed anything.

There is no corresponding function to undo these settings. As cutoff indexes increase they become irrelevant anyway.

### 11.5.4. Changing the minimum limits of cluster busy times monitors

Cluster busy times monitors have a `KheClusterBusyTimesMonitorSetMinimum` operation (Section 6.7.4) which changes their minimum limits. This section presents a method of making these changes which might be useful during solving.

This method calculates the demand for resources at particular times, which only really makes sense after all times are assigned. So it could reasonably be classified as a resource structural solver, but since it helps to adjust monitor limits it has been documented here.

Consider this example from nurse rostering. Suppose each resource has a maximum limit on the number of weekends it can be busy. Since each resource can work at most 2 shifts per weekend, summing up these maximum limits and multiplying by 2 gives the maximum number of shifts that resources can work on weekends. We call this the *supply* of weekend shifts.

Now suppose we find the number of weekend shifts that the instance requires nurses for. Call this the *demand* for weekend shifts.

If demand equals or exceeds supply, each resource needs to work its maximum number of weekends, or else some demands will not be covered. In that case, the resources' maximum limits are also minimum limits. The solver described here calculates supply and demand. It leaves it to the user to call `KheClusterBusyTimesMonitorSetMinimum`, or whatever.

To create a solver for doing this work, call

```
KHE_CLUSTER_MINIMUM_SOLVER KheClusterMinimumSolverMake(HA_ARENA a);
```

It uses memory taken from arena `a`. There is no operation to delete the solver; it is deleted when `a` is freed. To carry out one solve, call

```
void KheClusterMinimumSolverSolve(KHE_CLUSTER_MINIMUM_SOLVER cms,
  KHE_SOLN soln, KHE_OPTIONS options, KHE_RESOURCE_TYPE rt);
```

It uses `options` to find the common frame and event timetable monitor. It considers tasks and resources of type `rt` only. It can be called any number of times to solve problems with unrelated values of `soln`, `options`, and `rt`.

The attributes of the most recent solve may be found by calling

```
KHE_SOLN KheClusterMinimumSolverSoln(KHE_CLUSTER_MINIMUM_SOLVER cms);
KHE_OPTIONS KheClusterMinimumSolverOptions(
  KHE_CLUSTER_MINIMUM_SOLVER cms);
KHE_RESOURCE_TYPE KheClusterMinimumSolverResourceType(
  KHE_CLUSTER_MINIMUM_SOLVER cms);
```

These will all be `NULL` before the first solve. If a new solve is begun with the same attributes as the previous solve, it will produce the same outcome if the solution has not changed.

The solve first finds the constraints suited to what it does: all cluster busy times constraints with non-zero cost and a non-zero number of time groups which are pairwise disjoint (always true in practice) and either all positive, in which case a non-trivial maximum limit must be present, or all negative, in which case a non-trivial minimum limit must be present.

For each maximal non-empty subset of these constraints with the same time groups (ignoring polarity) and the same 'applies to' time group, the solve makes one *group*, with its own supply and demand, for each offset of the 'applies to' time group. To visit these groups, call

```
int KheClusterMinimumSolverGroupCount(KHE_CLUSTER_MINIMUM_SOLVER cms);
KHE_CLUSTER_MINIMUM_GROUP KheClusterMinimumSolverGroup(
  KHE_CLUSTER_MINIMUM_SOLVER cms, int i);
```

There are several operations for querying a group. To visit its constraints, call

```
int KheClusterMinimumGroupConstraintCount(KHE_CLUSTER_MINIMUM_GROUP cmg);
KHE_CLUSTER_BUSY_TIMES_CONSTRAINT KheClusterMinimumGroupConstraint(
  KHE_CLUSTER_MINIMUM_GROUP cmg, int i);
```

To retrieve its constraint offset, call

```
int KheClusterMinimumGroupConstraintOffset(KHE_CLUSTER_MINIMUM_GROUP cmg);
```

The time groups may be retrieved from its first constraint. To find its supply, call

```
int KheClusterMinimumGroupSupply(KHE_CLUSTER_MINIMUM_GROUP cmg);
```

This is calculated as described above for weekends; here is a fully general description.

For each constraint `c` of `cmg` we calculate a supply, as follows. Suppose first that the constraint has non-trivial maximum limit `max` and that all its time groups are positive. Find, for each time group `tg` of `c`, the number of frame time groups that `tg` intersects with (taking the offset into account). This is the maximum number of times from `tg` that a resource can be busy for. Take the `max` largest of these numbers and add them to get the supply of `c`.

If `c` has a non-trivial minimum limit `min` and all its time groups are negative, set `max` to the number of time groups minus `min` and proceed as in the positive case. (For more on this transformation, see the theorem at the end of Section 3.7.14.)

For each resource `r` of type `rt` we find a supply, as follows. If `r` is a point of application of at least one constraint, its supply is the minimum of the supplies of its constraints. Otherwise, its supply is the sum, over all time groups `tg`, of the number of frame time groups `tg` intersects with. `KheClusterMinimumGroupSupply` is the sum, over all resources `r`, of the supply of `r`.

To find a group's demand, call

```
int KheClusterMinimumGroupDemand(KHE_CLUSTER_MINIMUM_GROUP cmg);
```

This is the sum, over all times in the time groups of the group's constraints (taking the offset into account), of the number of tasks of type `rt` running at each time.

Finally,

```
void KheClusterMinimumGroupDebug(KHE_CLUSTER_MINIMUM_GROUP cmg,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of `cmg` onto `fp` with the given verbosity and indent.

There is also an operation for finding the group of a given monitor:

```
bool KheClusterMinimumSolverMonitorGroup(KHE_CLUSTER_MINIMUM_SOLVER cms,
  KHE_CLUSTER_BUSY_TIMES_MONITOR cbtm, KHE_CLUSTER_MINIMUM_GROUP *cmg);
```

If `cms` has a group containing `cbtm`'s constraint and offset (there can be at most one), this function returns `true` and sets `*cmg` to that group. Otherwise it returns `false` and sets `*cmg` to `NULL`.

It is up to the caller to take it from here. For example, after carrying out a solve, for each cluster monitor `m` one could call `KheClusterMinimumSolverMonitorGroup` to see whether it is subject to a group. Then if that group's demand equals or exceeds its supply, a call to `KheClusterBusyTimesMonitorSetMinimum` increases `m`'s minimum limit. And so on. However, the solver does offer some convenience functions to help with this:

```
void KheClusterMinimumSolverSetBegin(KHE_CLUSTER_MINIMUM_SOLVER cms);
void KheClusterMinimumSolverSet(KHE_CLUSTER_MINIMUM_SOLVER cms,
  KHE_CLUSTER_BUSY_TIMES_MONITOR m, int val);
void KheClusterMinimumSolverSetEnd(KHE_CLUSTER_MINIMUM_SOLVER cms,
  bool undo);
```

`KheClusterMinimumSolverSetBegin` begins a run of changes to monitors' minimum limits. `KheClusterMinimumSolverSet` makes a call to `KheClusterBusyTimesMonitorSetMinimum`, and remembers that the call was made. `KheClusterMinimumSolverSetEnd` ends the run of changes, and if `undo` is `true` it also undoes them (in reverse order), returning the monitor limits to their values when the run began. Use of these functions is optional.

For convenience there is also

```
void KheClusterMinimumSolverSetMulti(KHE_CLUSTER_MINIMUM_SOLVER cms,
  KHE_RESOURCE_GROUP rg);
```

where `rg`'s resource type must equal `cms`'s. It calls `KheClusterMinimumSolverMonitorGroup` for each cluster busy times monitor `m` for each resource of `rg`. If that returns `true` and the group's demand equals or exceeds its supply, then `m`'s minimum limit is changed to its maximum limit. Neither `KheClusterMinimumSolverSetBegin` nor `KheClusterMinimumSolverSetEnd` are called. The user must call `KheClusterMinimumSolverSetBegin` first, as usual, and is free to call `KheClusterMinimumSolverSetEnd` immediately with `undo` set to `false`, or later with `undo` set to `true`. It is probably not a good idea to not call `KheClusterMinimumSolverSetEnd` at all, since that will leave `cms` unable to accept calls to `KheClusterMinimumSolverSetBegin`.

Finally, function

```
void KheClusterMinimumSolverDebug(KHE_CLUSTER_MINIMUM_SOLVER cms,
  int verbosity, int indent, FILE *fp);
```

produces the usual debug print of `cms` onto `fp` with the given verbosity and indent.

Cluster minimum solvers deal only with cluster busy times constraints. Other constraints might help to reduce supply further. For example, if a resource is unavailable for an entire day, that will reduce supply by 1. At present these kinds of ideas are not taken into account.

### 11.5.5. Unbalanced complete weekends

*Complete weekends* constraints, saying that each resource should either be busy on both days of each weekend, or free on both days, are common in nurse rostering. They help to minimize the number of weekends that nurses work. But they can cause a rather obscure problem, which we uncover and deal with in this section.

A task is *required* (meaning that assignment of the task is required) when it has non-zero non-assignment cost, according to `KheTaskNonAsstAndAsstCost` (Section 11.9.1). This has nothing to do with required constraints; the cost does not have to be a hard cost. A task is *optional* (meaning that assignment of the task is optional) when it has zero non-assignment cost, according to `KheTaskNonAsstAndAsstCost`. Every task is either required or optional.

If one day of a weekend subject to a complete weekends constraint is *busier* than (has more required tasks than) the other, then clearly something has to give. There are three possibilities:

1. Some nurses have complete weekends defects.

2. On the busier day, some required tasks are unassigned.

3. On the other (less busy) day, some optional tasks are assigned.

These could occur together. The first two give rise directly to defects—they are highly visible. The third does not give rise to any defects directly, but it will have a cost if there is a general shortage of nurses, because assigning nurses to optional tasks adds to the general overload.

Despite the absence of direct defects, the third possibility might not be the best, in which case we want to steer solvers away from it. There are several ways to do this, which we'll come to shortly. Whichever method is used, the aim is to rule out the third possibility without biasing the solve towards either of the others.

Function

```
void KheBalanceWeekends(KHE_SOLN soln, KHE_OPTIONS options,
  KHE_RESOURCE_TYPE rt, KHE_SOLN_ADJUSTER sa)
```

carries out this programme for `soln`'s resources and tasks of type `rt`. If `sa != NULL`, it uses solution adjuster `sa` to record the changes it made, so that someone else can undo them later. Parameter `options` is needed for accessing the common frame and the event timetable monitor. There is also this option for controlling `KheBalanceWeekends`:

`rs_balance_weekends_method`
: A string option that determines what is done when an unbalanced weekend is discovered. Value `"none"` turns weekend balancing off. Value `"fix_optional"` fixes all optional tasks on the less busy day. The third and default value, `"fix_required"`, fixes one or a few required tasks on the busier day, so as to equalize the number of unfixed required tasks on the two days. More details are given below.

In detail, `KheBalanceWeekends` works as follows.

Cluster busy times constraints have offsets; each legal offset defines a separate constraint. `KheBalanceWeekends` handles this, but for simplicity we will say 'constraint' here when we really mean 'constraint plus offset'.

`KheBalanceWeekends` begins by using a balance solver (Section 11.4.5) to compare the overall supply and demand for resources of type `rt`. If supply exceeds demand, case (3) above will not necessarily generate any cost, so the solve returns early, having changed nothing.

Next, the solve finds all cluster busy times constraints which apply to resources of type `rt`, have non-zero weight, and contain exactly two time groups (both positive), minimum limit 2, maximum limit 2, and allow zero flag `true`. These are the constraints that request complete weekends, although no-one checks (or needs to check) that the two time groups represent a Saturday and a Sunday. (A check is made that the two time groups are disjoint.) It groups these constraints into equivalence classes, placing two constraints into the same class when they have the same time groups. It then handles each class $C$ separately.

The first step in handling $C$ is to check that its constraints, taken together, apply to every resource of type `rt`. If not, $C$ is skipped, because not all resources require complete weekends.

Let $R(d)$ be the number of required tasks running on day $d$. Let $d_1$ and $d_2$ be the two days that $C$ monitors. If $R(d_1) = R(d_2)$ there is nothing to do and we skip $C$. If $R(d_1) < R(d_2)$ we swap the names of the two days. So we can assume from now on that $R(d_1) > R(d_2)$. This implies $R(d_1) > 0$. We've previously called $d_1$ the busier day, and $d_2$ the less busy day.

Let $O(d)$ be the number of optional tasks running on day $d$. If $O(d_2) = 0$, then our plan of fixing the assignments of the optional tasks of $d_2$ changes nothing, because there are no such tasks. So there is nothing to do and we skip $C$. So we can assume $O(d_2) > 0$.

We also check at this point whether any of the $O(d_2)$ optional tasks are currently assigned. If any are, then what we are trying to prevent has already occurred, so again we skip $C$.

The next step is to work out whether the costs involved are such that case (3) above is not the best choice. We'll return to that in a moment. If (3) is not the best choice, then the solution is changed as determined by the `rs_balance_weekends_method` option defined above. If `sa != NULL`, these changes are recorded in `sa` so that someone else can undo them later.

The cost calculation that decides whether to proceed is as follows. Suppose that all required tasks are assigned except for task $t$ on $d_1$, whose non-assignment cost, $n(t)$, is minimal. We find the cost of carrying on for each of the three cases above. Let $c$ be the cost incurred by resource constraints of assigning a task, given that demand exceeds supply, as returned by `KheBalanceSolverMarginalCost` (Section 11.4.5). The three cases and their costs are:

1. Assign a nurse to $t$ only. The cost of this (call it $c_1$) is the initial cost, minus $n(t)$, plus $c$, plus the minimum of the weights of the constraints of $C$.

2. Leave $t$ unassigned. Then the cost $c_2$ is the initial cost, since the solution does not change.

3. Assign a nurse to $t$ and to a task $t'$ on $d_2$ that does not need assignment. Then the cost $c_3$ is the initial cost, minus $n(t)$, plus $2c$. If all of the optional tasks on $d_2$ have non-zero assignment cost, add the minimum of those costs to $c_3$.

If $c_3 > c_1$ or $c_3 > c_2$, then we want to avoid the third case, so we fix the optional tasks on $d_2$.

When `rs_balance_weekends_method` has value `"fix_optional"`, what to do is clear: fix all optional tasks on the less busy day. When the value is `"fix_required"`, we need to fix one or more required tasks on the busier day so that the number of unfixed required tasks is equal on the two days. The number to fix is clear, but which ones? This is decided as follows.

Build a bipartite graph whose demand nodes are the required tasks on the busier day, and whose supply nodes are the required tasks on the less busy day. Join two nodes by an edge when their tasks' domains have a non-empty intersection, weighted by the cost of the initial solution minus the non-assignment costs of the two endpoints (this will favour tasks with large non-assignment costs), breaking ties by the difference in offset in the days frame of the starting times of the two tasks (this will favour pairs of tasks for the same shift). Find a maximum matching in this graph and fix every demand node that fails to match.

### 11.5.6. Allowing split assignments

A good way to minimize split assignments is to prohibit them at first but allow them later. To change a tasking from the first state to the second, call

```
bool KheTaskingAllowSplitAssignments(KHE_TASKING tasking,
  bool unassigned_only);
```

It unfixes and unassigns all tasks assigned to the tasks of `tasking` and adds them to `tasking`, returning `true` if it changed anything. If one of the original unfixed tasks is assigned (to a cycle task), the tasks assigned to it are assigned to that task, so that existing resource assignments are not forgotten. If `unassigned_only` is `true`, only the unassigned tasks of `tasking` are affected. (This option is included for completeness, but it is not recommended, since it leaves few choices open.) `KheTaskingAllowSplitAssignments` preserves the resource assignment invariant.

### 11.5.7. Enlarging task domains

If any room or any teacher is better than none, then it will be worth assigning any resource to tasks that remain unassigned at the end of resource assignment. Function

```
void KheTaskingEnlargeDomains(KHE_TASKING tasking, bool unassigned_only);
```

permits this by enlarging the domains of the tasks of `tasking` and any tasks assigned to them (and so on recursively) to the full set of resources of their resource types. If `unassigned_only` is true, only the unassigned tasks of `tasking` are affected. The tasks are visited in postorder—that is, a task's domain is enlarged only after the domains of the tasks assigned to it have been enlarged—ensuring that the operation cannot fail. Preassigned tasks are not enlarged.

This operation works, naturally, by deleting all task bounds from the tasks it changes. Any task bounds that become applicable to no tasks as a result of this are deleted.

## 11.6. Grouping by resource

*Grouping by resource* is a kind of task grouping, obtained by calling

```
bool KheTaskingGroupByResource(KHE_TASKING tasking,
  KHE_OPTIONS options, KHE_TASK_SET ts);
```

Similarly to grouping by resource constraints, to be described in Section 11.10, it groups tasks whose resource types are covered by `tasking` which lie in adjacent time groups of the common frame, and adds each task which it makes an assignment to to `ts` (if `ts` is non-`NULL`). However, the tasks are chosen in quite a different way: each group consists of a maximal sequence of tasks which lie in adjacent time groups of the frame and are currently assigned to the same resource. The thinking is that if the solution is already of good quality, it may be advantageous to keep these runs of tasks together while trying to assign them to different resources using an arbitrary repair algorithm.

When a grouping made by `KheTaskingGroupByResource` and recorded in a task set is no longer needed, function `KheTaskSetUnGroup` (Section 5.6) may be used to remove it.

## 11.7. The task grouper

A *task grouper* supports a form of task grouping which allows the grouping to be done, undone, and redone at will.

The first step is to create a task grouper object, by calling

```
KHE_TASK_GROUPER KheTaskGrouperMake(KHE_RESOURCE_TYPE rt, HA_ARENA a);
```

This makes a task grouper object for tasks of type `rt`. It is deleted when `a` is deleted. Also,

```
void KheTaskGrouperClear(KHE_TASK_GROUPER tg);
```

clears `tg` back to its state immediately after `KheTaskGrouperMake`, without changing `rt` or `a`.

To add tasks to a task grouper, make any number of calls to

```
bool KheTaskGrouperAddTask(KHE_TASK_GROUPER tg, KHE_TASK t);
```

Each task passed to `tg` in this way must be assigned directly to the cycle task for some resource `r` of type `rt`. The tasks passed to `tg` by `KheTaskGrouperAddTask` which are assigned `r` at the time they are passed are placed in one group. No assignments are made.

If `true` is returned by `KheTaskGrouperAddTask`, `t` is the *leader task* for its group: it is the first task assigned `r` which has been passed to `tg`. If `false` is returned, `t` is not the leader task.

Adding the same task twice is legal but is the same as adding it once. If the task is the leader task, it is reported to be so only the first time it is passed.

Importantly, although the grouping is determined by which resources the tasks are assigned to, it is only the grouping that the grouper cares about, not the resources. Once the groups are made, the resources that determined the grouping become irrelevant to the grouper.

At any time one may call

```
void KheTaskGrouperGroup(KHE_TASK_GROUPER tg);
void KheTaskGrouperUnGroup(KHE_TASK_GROUPER tg);
```

`KheTaskGrouperGroup` ensures that, in each group, the tasks other than the leader task are assigned directly to the leader task. It does not change the assignment of the leader task. `KheTaskGrouperUnGroup` ensures that, for each group, the tasks other than the leader task are assigned directly to whatever the leader task is assigned to (possibly nothing). As mentioned above, the resources which defined the groups originally are irrelevant to these operations.

If `KheTaskGrouperGroup` cannot assign some task to its leader task, it adds the task's task bounds to the leader task and tries again. If it cannot add these bounds, or the assignment still does not succeed, it aborts. In addition to ungrouping, `KheTaskGrouperUnGroup` removes any task bounds added by `KheTaskGrouperGroup`. In detail, `KheTaskGrouperGroup` records the number of task bounds present when it is first called, and `KheTaskGrouperUnGroup` removes task bounds from the end of the leader task until this number is reached.

A task grouper's tasks may be grouped and ungrouped at will. This is more general than using `KheTaskSetUnGroup`, since after ungrouping that way there is no way to regroup. The extra power comes from the fact that a task grouper contains, in effect, a task set for each group.

The author has encountered one case where `KheTaskGrouperUnGroup` fails to remove the task bounds added by `KheTaskGrouperGroup`. The immediate problem has probably been fixed, although it is hard to be sure that it will not recur. So instead of aborting in that case, `KheTaskGrouperUnGroup` prints a debug message and stops removing bounds for that task.

## 11.8. Task finding

*Task finding* is KHE's name for some operations, based on *task finder* objects, that find sets of tasks which are to be moved all together from one resource to another. Task finding is used by only a few solvers, because it has been replaced by *mtask finding*, the subject of Section 11.9. Only old code uses task finding now; it may eventually be removed altogether.

Task finding is concerned with which days tasks are running. A *day* is a time group of the common frame. The days that a task `t` is running are the days containing the times that `t` itself is running, plus the days containing the times that the tasks assigned to `t`, directly or indirectly, are running. The days that a task set is running are the days that its tasks are running.

Task finding represents the days that a task or task set is running by a *bounding interval*, a pair of integers: `first_index`, the index in the common frame of the first day that the task or task set is running, and `last_index`, the index of the last day that the task or task set is running. So task finding is unaware of cases where a task runs twice on the same day, or has a *gap* (a day within the bounding interval when it is not running). Neither is likely in practice. Task finding considers the duration of a task or task set to be the length of its bounding interval.

Task finding operations typically find a set of tasks, often stored in a task set object (Section 5.6). In some cases these tasks form a *task run*, that is, they satisfy these conditions:

1. The set is non-empty. An empty run would be useless.

2. Every task is a proper root task. The tasks are being found in order to be moved from one resource to another, and this ensures that the move will not break up any groups.

3. No two tasks run on the same day. This is more or less automatic when the tasks are all assigned the same resource initially, but it holds whether the tasks are assigned or not. If it didn't, then when the tasks are moved to a common resource there would be clashes.

4. The days that the tasks are running are consecutive. In other words, between the first day and the last there are no *gaps*: days when none of the tasks is running.

The task finder does not reject tasks which run twice on the same day or which have gaps. As explained above, it is unaware of these cases. So the last two conditions should really say that the task finder does not introduce any *new* clashes or gaps when it groups tasks into runs.

Some runs are *unpreassigned runs*, meaning that all of their tasks are unpreassigned. Only unpreassigned runs can be moved from one resource to another. And some runs are *maximal runs*: they cannot be extended, either to left or right. We mainly deal with maximal runs, but just what we mean by 'maximal' depends on circumstances. For example, we may want to exclude preassigned tasks from our runs. So our definition does *not* take the arguably reasonable extra step of requiring all runs to be maximal.

Some task finding operations find all tasks assigned a particular resource in a particular interval. In these cases, only conditions 2 and 3 must hold; the result need not be a task run.

Task finding treats non-assignment like the assignment of a special resource (represented by `NULL`). This makes it equally at home finding assigned and unassigned tasks.

A task `t` *needs assignment* if `KheTaskNeedsAssignment(t)` (Section 4.6.1) returns `true`, meaning that non-assignment of a resource to `t` would incur a cost, because of an assign resource constraint, or a limit resources constraint which is currently at or below its minimum limit, that applies to `t`. Task finding never includes tasks that do not need assignment when it searches for unassigned tasks, because assigning resources to such tasks is not a high priority. It does include them when searching for assigned tasks.

A resource is *effectively free* during some set of days if it is `NULL`, or it is not `NULL` and the tasks it is assigned to on those days do not need assignment. The point is that it is always safe to move some tasks to a resource on days when it is effectively free: if the resource is `NULL`, they are simply unassigned, and if it is non-`NULL`, any tasks running on those days do not need assignment, and can be unassigned, at no cost, before the move is made. Task finding utilizes the effectively free concept and offers move operations that work in this way.

### 11.8.1. Task finder objects

To create a task finder object, call

```
KHE_TASK_FINDER KheTaskFinderMake(KHE_SOLN soln, KHE_OPTIONS options,
  HA_ARENA a);
```

This returns a pointer to a private struct in arena `a`. Options `gs_common_frame` (Section 5.10) and `gs_event_timetable_monitor` (Section 8.4) are taken from `options`. If either is `NULL`, `KheTaskFinderMake` returns `NULL`, since it cannot do its work without them.

Ejection chain repair code can obtain a task finder from the ejector object, by calling

```
KHE_TASK_FINDER KheEjectorTaskFinder(KHE_EJECTOR ej);
```

This saves time and memory compared with creating new task finders over and over. Once again the return value is `NULL` if the two options are not both present.

The days tasks are running (the time groups of the common frame) are represented in task finding by their indexes, as explained above. The first legal index is 0; the last is returned by

```
int KheTaskFinderLastIndex(KHE_TASK_FINDER tf);
```

This is just `KheTimeGroupTimeCount(frame) - 1`, where `frame` is the common frame. Also,

```
KHE_FRAME KheTaskFinderFrame(KHE_TASK_FINDER tf);
```

may be called to retrieve the frame itself.

As defined earlier, the bounding interval of a task or task set is the smallest interval containing all the days that the task or task set is running. It is returned by these functions:

```
KHE_INTERVAL KheTaskFinderTaskInterval(KHE_TASK_FINDER tf,
  KHE_TASK task);
KHE_INTERVAL KheTaskFinderTaskSetInterval(KHE_TASK_FINDER tf,
  KHE_TASK_SET ts);
```

These return an interval (Section 8.12) holding the indexes in the common frame of the first and last days that `task` or `ts` is running. If `ts` is empty, the interval is empty. There is also

```
KHE_INTERVAL KheTaskFinderTimeGroupInterval(KHE_TASK_FINDER tf,
  KHE_TIME_GROUP tg);
```

which returns an interval holding the first and last days that `tg` overlaps with. If `tg` is empty, the interval is empty.

These three operations find task sets and runs:

```
void KheFindTasksInInterval(KHE_TASK_FINDER tf,
  KHE_INTERVAL in, KHE_RESOURCE_TYPE rt, KHE_RESOURCE from_r,
  bool allow_preassigned, bool allow_partial,
  KHE_TASK_SET res_ts, KHE_INTERVAL *res_in);
bool KheFindFirstRunInInterval(KHE_TASK_FINDER tf,
  KHE_INTERVAL in, KHE_RESOURCE_TYPE rt, KHE_RESOURCE from_r,
  bool allow_preassigned, bool allow_partial, bool sep_need_asst,
  KHE_TASK_SET res_ts, KHE_INTERVAL *res_in);
bool KheFindLastRunInInterval(KHE_TASK_FINDER tf,
  KHE_INTERVAL in, KHE_RESOURCE_TYPE rt, KHE_RESOURCE from_r,
  bool allow_preassigned, bool allow_partial, bool sep_need_asst,
  KHE_TASK_SET res_ts, KHE_INTERVAL *res_in);
```

All three functions clear `res_ts`, which must have been created previously, then add to it some tasks which are assigned `from_r` (or are unassigned if `from_r` is `NULL`). They set `*res_in` to the bounding interval of the tasks of `res_ts`.

Call `in` the *target interval*. A task `t` *overlaps* the target interval when at least one of the days on which `t` is running lies in it. Subject to the following conditions, `KheFindTasksInInterval` finds all tasks that overlap the target interval; `KheFindFirstRunInInterval` finds the first (leftmost) run containing a task that overlaps the target interval, or returns `false` if there is no such run; and `KheFindLastRunInInterval` finds the last (rightmost) run containing a task that overlaps the target interval, or returns `false` if there is no such run.

When `from_r` is `NULL`, only unassigned tasks that need assignment (as discussed above) are added. The first could be any unassigned task of type `rt` (it is this that `rt` is needed for), but the others must be compatible with the first, in that we expect these tasks to be assigned some single resource, and it would not do for them to have widely different domains.

Some tasks are *ignored*, which means that the operation behaves as though they are simply not there. Subject to this ignoring feature, the runs found are maximal. A task is ignored in this way when it is running on any of the days that the tasks that have already been added to `res_ts` are running. Preassigned tasks are allowed when `allow_preassigned` is `true`. Tasks that are running partly or wholly outside the target interval are allowed when `allow_partial` is `true`. When `allow_partial` is `true`, a run can extend an arbitrary distance beyond the target interval, and contain some tasks that do not overlap the target interval at all.

If `sep_need_asst` is `true`, all tasks `t` in the run found by `KheFindFirstRunInInterval` or `KheFindLastRunInInterval` have the same value of `KheTaskNeedsAssignment(t)`. This value could be `true` or `false`, but it is the same for all tasks in the run. If `sep_need_asst` is `false`, there is no requirement of this kind.

### 11.8.2. Daily schedules

Sometimes more detailed information is needed about when a task is running than just the bounding interval. In those cases, task finding offers *daily schedules*, which calculate both the bounding interval and what is going on on each day:

```
KHE_DAILY_SCHEDULE KheTaskFinderTaskDailySchedule(
  KHE_TASK_FINDER tf, KHE_TASK task);
KHE_DAILY_SCHEDULE KheTaskFinderTaskSetDailySchedule(
  KHE_TASK_FINDER tf, KHE_TASK_SET ts);
KHE_DAILY_SCHEDULE KheTaskFinderTimeGroupDailySchedule(
  KHE_TASK_FINDER tf, KHE_TIME_GROUP tg);
```

These return a *daily schedule*: a representation of what `task`, `ts`, or `tg` is doing on each day, including tasks assigned directly or indirectly to `task` or `ts`. Also,

```
KHE_DAILY_SCHEDULE KheTaskFinderNullDailySchedule(
  KHE_TASK_FINDER tf, KHE_INTERVAL in);
```

returns a daily schedule representing doing nothing during the given interval.

A `KHE_DAILY_SCHEDULE` is an object which uses memory taken from its task finder's arena. It can be deleted (which actually means being added to a free list in its task finder) by calling

```
void KheDailyScheduleDelete(KHE_DAILY_SCHEDULE ds);
```

It has these attributes:

```
KHE_TASK_FINDER KheDailyScheduleTaskFinder(KHE_DAILY_SCHEDULE ds);
bool KheDailyScheduleNoOverlap(KHE_DAILY_SCHEDULE ds);
KHE_INTERVAL KheDailyScheduleInterval(KHE_DAILY_SCHEDULE ds);
```

`KheDailyScheduleTaskFinder` returns `ds`'s task finder; `KheDailyScheduleNoOverlap` returns `true` when no two of the schedule's times occur on the same day, and `false` otherwise; and `KheDailyScheduleInterval` returns the interval of day indexs of the schedule's days. For each day between the interval's first and last inclusive,

```
KHE_TASK KheDailyScheduleTask(KHE_DAILY_SCHEDULE ds, int day_index);
```

returns the task running in `ds` on day `day_index`. It may be a task assigned directly or indirectly to `task` or `ts`, not necessarily `task` or a task from `ts`. `NULL` is returned if no task is running on that day. This is certain for schedules created by `KheTaskFinderTimeGroupDailySchedule` and `KheTaskFinderNullDailySchedule`, but it is also possible for schedules created by `KheTaskFinderTaskDailySchedule` and `KheTaskFinderTaskSetDailySchedule`. If there are two or more tasks running on that day, an arbitrary one of them is returned; this cannot happen when `KheDailyScheduleNoOverlap` returns `true`. Similarly,

```
KHE_TIME KheDailyScheduleTime(KHE_DAILY_SCHEDULE ds, int day_index);
```

returns the time in `ds` that is busy on day `day_index`. This will be `NULL` if there is no time in the schedule on that day, which is always the case when the schedule was created by a call to `KheTaskFinderNullDailySchedule`.

## 11.9. Multi-task finding

The author has made several attempts over the years to define an equivalence relation on tasks and use it to group equivalent tasks together into classes. The purpose is to avoid symmetrical assignments, in which a resource is assigned to several tasks in turn which are in fact equivalent, wasting time. This section describes what he hopes and believes will be his final attempt.

It could be argued that equivalence classes of tasks are only needed because XHSTT, and following it the KHE platform, allow at most one resource to be assigned to each task at any given moment during solving. If several could be assigned, equivalence would be guaranteed because the 'tasks' thus grouped would be indistinguishable. This would probably work for nurse rostering, but in high school timetabling it would not handle tasks that become equivalent when their meets are assigned the same time—requests for ordinary classrooms, for example.

Still, 'a task to which several resources can be assigned' is a valuable abstraction, better for the user than a set of equivalent tasks. So instead of defining a task group or task class (as in the author's previous attempts), we define a *multi-task* or *mtask* to be a task to which several resources can be assigned simultaneously. Behind the scenes, an mtask is a set of equivalent proper root tasks, but the user does not know or care which tasks those are, or which are assigned which resources: the mtask handles that, in a provably best possible way, as we'll see.

The idea, then, is to group tasks into mtasks and to write resource assignment algorithms that assign resources to mtasks rather than to tasks. Assigning resources to mtasks is somewhat harder to do than assigning them to tasks, because mtasks accept multiple assignments, but it should run faster because assignment symmetries are avoided.

Three types are defined here. Type `KHE_MTASK` represents one mtask. `KHE_MTASK_SET` represents a simple set of mtasks. And `KHE_MTASK_FINDER` creates mtasks and holds them. All older attemps at task equivalencing have been removed from the KHE platform and solvers.

### 11.9.1. Multi-tasks

A *multi-task* or *mtask* is a task to which several resources can be assigned simultaneously. Behind the scenes, it is a non-empty set of proper root tasks which are equivalent to one another in a sense to be defined in Section 11.9.4. This section presents the operations on mtasks.

There is no operation to create one mtask, because mtasks need to be made together all at once, which is what `KheMTaskFinderMake` (Section 11.9.3) does. After that, any changes to individual tasks which affect their equivalence will render these mtasks out of date. This includes assignments of one task to another task, changes to task domains, changes to whether a task assignment is fixed or not, meet splits and merges, and attaching and detaching event resource monitors. Because of this, it is best to create mtasks at the beginning of a call on some resource solver, after any such changes have been made, and delete them (by deleting the mtask finder's arena) at the end of that call, before later calls on other solvers can change things.

However, several of these 'forbidden' operations have mtask versions. These do what the forbidden operations do (indeed, each calls one forbidden operation), but they also update the mtasks to take account of the change. For example, the mtask version of assigning one task to another will cause the two tasks to be removed from their mtasks, and then the combined entity will be added to another mtask. This could make one or two mtasks disappear (since there are no empty mtasks), and it could bring a new mtask into existence. Operations of this type are too slow to call from the inner loops of solvers, but they can be called from less time-critical code.

Here now are the operations on mtasks. One advantage of the mtask abstraction is that we can model these operations on the corresponding task operations—although there are some differences, such as that we cannot assign one mtask to another.

First come some general operations:

```
char *KheMTaskId(KHE_MTASK mt);
```

This returns an Id for mtask `mt`, just the task Id of its first task.

```
KHE_RESOURCE_TYPE KheMTaskResourceType(KHE_MTASK mt);
bool KheMTaskIsPreassigned(KHE_MTASK mt, KHE_RESOURCE *r);
bool KheMTaskAssignIsFixed(KHE_MTASK mt);
KHE_RESOURCE_GROUP KheMTaskDomain(KHE_MTASK mt);
int KheMTaskTotalDuration(KHE_MTASK mt);
float KheMTaskTotalWorkload(KHE_MTASK mt);
```

Again, these come from `mt`'s first task; they must be the same for all `mt`'s tasks, otherwise those tasks would not have been placed into the same mtask. A preassigned task is the only member of its mtask, except in the unlikely case of equivalent tasks preassigned the same resource. A task with a fixed assignment is the only member of its mtask.

The proper root tasks of an mtask can come from the same meet, or from different meets. When they come from the same meet, function

```
bool KheMTaskHasSoleMeet(KHE_MTASK mt, KHE_MEET *meet);
```

sets `*meet` to that meet and returns `true`. Otherwise it sets `*meet` to `NULL` and returns `false`.

KHE allows the user to create tasks which are not derived from any event resource or meet. These are intended for use as proper root tasks to which ordinary tasks are assigned. However, if no ordinary tasks are assigned to them, the result is a task with duration 0. This is awkward, but careful examination (which we'll do later) shows that it is not really a special case.

An mtask *has fixed times* when none of its tasks (including tasks assigned, directly or indirectly, to those tasks) lie in meets with unassigned times, and the call to `KheMTaskSolverMake` that created the mtask had `fixed_times` set to `true`, meaning that there is an assumption that assigned times will not change. To check this condition, call

```
bool KheMTaskHasFixedTimes(KHE_MTASK mt);
```

When it returns `true`, these functions provide access to the times:

```
KHE_INTERVAL KheMTaskInterval(KHE_MTASK mt);
KHE_TIME KheMTaskDayTime(KHE_MTASK mt, int day_index,
  float *workload_per_time);
KHE_TIME_SET KheMTaskTimeSet(KHE_MTASK mt);
```

`KheMTaskInterval` returns the smallest interval of days in the days frame of `mt`'s task finder that contains `mt`'s times. In mtask finding generally, a value of type `KHE_INTERVAL` (defined in Section 8.12) always denotes an interval of days. For each index `day_index` in this interval, KheMTaskDayTime returns the time that mt is busy on the day of days_frame with index day_index, or NULL if mt does not run that day, as well as mt's workload per time on that day. Finally, KheMTaskTimeSet returns the set of times that the tasks of mt are running.

Many mtask operations utilize KheMTaskInterval(mt) as their representation of when mt is running. This representation is convenient but it does not recognize days within the interval where an mtask runs twice, or not at all. Two functions help to identify such cases:

```
bool KheMTaskNoOverlap(KHE_MTASK mt);
bool KheMTaskNoGaps(KHE_MTASK mt);
```

KheMTaskNoOverlap returns true when no two of mt's busy times lie on the same day, and KheMTaskNoGaps returns true when none of the calls to KheMTaskDayTime return NULL.

Returning to functions that do not need fixed times, to visit the tasks of an mtask we have

```
int KheMTaskTaskCount(KHE_MTASK mt);
KHE_TASK KheMTaskTask(KHE_MTASK mt, int i,
  KHE_COST *non_asst_cost, KHE_COST *asst_cost);
```

KheMTaskTask returns the ith task t, plus a cost *non_asst_cost which will be included in the solution cost whenever t is unassigned (as reported by assign resource monitors) and a cost *asst_cost which will be included in the solution cost whenever t is assigned (as reported by prefer resources monitors with empty sets of preferred resources). Actually, these costs can vary depending on other task assignments; the costs returned here are lower bounds that do not depend on other assignments. The tasks are returned so that those most in need of assignment come first, that is, in order of decreasing *non_asst_cost - *asst_cost. Tasks for which this order is not certain lie in different mtasks. All this is explained in detail in Section 11.9.4.

For the convenience of solvers that need these costs but not mtasks, there is also

```
void KheTaskNonAsstAndAsstCost(KHE_TASK task, KHE_COST *non_asst_cost,
  KHE_COST *asst_cost);
```

It returns these costs, as defined above, for task, quite independently of mtask finding. Here task would usually be a proper root task, but it does not need to be; the costs depend on task itself and on all tasks assigned, directly or indirectly, to task.

Next come operations concerned with resource assignment. Each mtask has a set of resources currently assigned to it (that is, assigned to some of its tasks). This set is in fact a multi-set: a resource may be currently assigned to a given mtask more than once. Assigning a resource more than once to a given mtask inevitably causes clashes, but it is better to let it happen than to waste time preventing it. The resource assignment operations are

```
bool KheMTaskMoveResourceCheck(KHE_MTASK mt, KHE_RESOURCE from_r,
  KHE_RESOURCE to_r, bool disallow_preassigned);
bool KheMTaskMoveResource(KHE_MTASK mt, KHE_RESOURCE from_r,
  KHE_RESOURCE to_r, bool disallow_preassigned);
```

KheMTaskMoveResourceCheck returns true when changing one of mt's assignments from from_r to to_r would succeed, and false when it would not succeed. Here from_r could be NULL, in which case the request is to add to_r to the set of resources assigned to mt, that is, to increase the multiplicity of its assignments to `mt` by one. We call this an *assignment*, although we have not provided a `KheMTaskAssignResourceCheck` operation for it. And `to_r` could be `NULL`, in which case the request is to remove `from_r` from the set of resources assigned `mt`, that is, to reduce the multiplicity of its assignments to `mt` by one. We call this an *unassignment*. although again there is no `KheMTaskUnAssignResourceCheck` operation. `KheMTaskMoveResource` actually makes the change, returning `true` if it was successful, and `false` if it wasn't (in that case, nothing is changed).

Parameter `disallow_preassigned` is concerned with the awkward question of what to do with preassigned mtasks. The corresponding functions for tasks allow a preassigned task to be assigned, unassigned, and moved to another task which is preassigned the same resource. If `disallow_preassigned` is `false`, the equivalent behaviour is permitted here, allowing a preassigned mtask to be assigned and unassigned. However, in practice callers of these functions are more likely to want all changes to preassigned tasks to be disallowed: such tasks will already be assigned their preassigned resources, and changes to those assignments are not wanted. This is what happens when `disallow_preassigned` is `true`.

Here is the full list of reasons why an mtask move might not succeed:

- `from_r == to_r`, so the move would change nothing.
- `mt` contains only fixed tasks; their assignments cannot change.
- `mt` contains only preassigned tasks, and either the `disallow_preassigned` parameter is `true`, so that their assignments cannot change, or else it is `false`, and `to_r` is neither of the two permitted values (the preassigned resource and `NULL`).
- `to_r != NULL` and the domain of `mt` (the same for all its tasks) does not contain `to_r`.
- `from_r != NULL` and `from_r` is not one of the resources assigned to `mt`.
- `from_r == NULL` (and therefore `to_r != NULL`) and `mt` does not contain at least one unassigned task to assign `to_r` to.

As usual, returning `false` when the reassignment changes nothing reflects the practical reality that no solver wants to waste time on such changes.

This next function may be useful for suggesting a suitable resource for assignment:

```
bool KheMTaskResourceAssignSuggestion(KHE_MTASK mt, KHE_RESOURCE *to_r);
```

It returns `true` with `*to_r` set to a suggestion for an assignment to `mt`, if one can be found, and `false` if no suggestion can be made. The suggestion comes by looking for tasks which share an event resource with the next unassigned task of `mt` and are already assigned a resource: if that resource can be assigned to `mt`, then it becomes the suggestion. The idea here is to promote resource constancy (assigning the same resource to all the tasks of a given event resource) even when it is not required by an avoid split assignments constraint.

For visiting the assignments to `mt` there is

```
int KheMTaskAsstResourceCount(KHE_MTASK mt);
KHE_RESOURCE KheMTaskAsstResource(KHE_MTASK mt, int i);
```

which return the number of non-`NULL` resources in the multi-set of resources assigned to `mt`, and the `i`th resource, in the usual way. There are also

```
int KheMTaskAssignedTaskCount(KHE_MTASK mt);
int KheMTaskUnassignedTaskCount(KHE_MTASK mt);
```

which returns the number of assigned tasks in `mt`, and the number of unassigned tasks in `mt`. Naturally, they sum to `KheMTaskTaskCount(mt)`. `KheMTaskAssignedTaskCount` is a synonym for `KheMTaskAsstResourceCount`. The assigned tasks always come first in an mtask, so the first unassigned task (if there is one) is

```
KheMTaskTask(mt, KheMTaskAssignedTaskCount(mt), &non_asst_cost, &asst_cost);
```

There is also

```
bool KheMTaskNeedsAssignment(KHE_MTASK mt);
```

which returns `true` when `mt` contains at least one unassigned task such that the costs returned by `KheMTaskTask` satisfy `*non_asst_cost - *asst_cost > 0`. In other words, the cost of the solution would be reduced if this task was assigned, as far as the event resource monitors that determine `*non_asst_cost` and `*asst_cost` are concerned. Also,

```
int KheMTaskNeedsAssignmentCount(KHE_MTASK mt);
```

returns the number of tasks in `mt` that need assignment, as just defined. One could write

```
KheMTaskNeedsAssignmentCount(mt) > 0
```

instead of `KheMTaskNeedsAssignment(mt)`. And

```
bool KheMTaskContainsNeedlessAssignment(KHE_MTASK mt);
```

returns `true` if `mt` contains a task which is assigned but does not need to be. This means that a call to `KheMTaskNeedsAssignment(mt)` would return `false`, but furthermore, after any one resource is unassigned from `mt`, `KheMTaskNeedsAssignment(mt)` would still return `false`.

Any given set of resources is always assigned to the tasks of an mtask in a best possible (least cost) way. When a resource is unassigned from an mtask, the remaining assignments may no longer have this property. In that case, they are adjusted to make them best possible again.

A similar issue arises when an mtask is constructed: if the initial resource assignments are not best possible, they will be moved from one task to another within the mtask until they are. So there may be calls on task assignment operations while `KheMTaskSolverMake` is running. These are guaranteed to not increase the cost of the solution. They might decrease it.

An mtask's tasks all have the same domain, making the following operations well-defined:

```
bool KheMTaskAddTaskBoundCheck(KHE_MTASK mt, KHE_TASK_BOUND tb);
bool KheMTaskAddTaskBound(KHE_MTASK mt, KHE_TASK_BOUND tb);
bool KheMTaskDeleteTaskBoundCheck(KHE_MTASK mt, KHE_TASK_BOUND tb);
bool KheMTaskDeleteTaskBound(KHE_MTASK mt, KHE_TASK_BOUND tb);

int KheMTaskTaskBoundCount(KHE_MTASK mt);
KHE_TASK_BOUND KheMTaskTaskBound(KHE_MTASK mt, int i);
```

`KheMTaskAddTaskBound` adds its bound to each task. It returns `false` and changes nothing if any of the underlying `KheTaskAddTaskBound` operations would return `false`.

If the domain of an mtask is changed in this way, its tasks could become equivalent to the tasks of some other mtask that already have the new domain. However, no attempt is made to find and merge such mtasks. It does no harm, apart from wasting solve time, to have two mtasks on hand which could be merged into one.

Mtasks work correctly with marks and paths. Operations on mtasks are not stored in paths, but the underlying operations on tasks are, and that is enough to make everything work.

Finally,

```
void KheMTaskDebug(KHE_MTASK mt, int verbosity, int indent, FILE *fp);
```

produces a debug print of `mt` onto `fp`. It calls `KheTaskDebug` for each task of `mt`.

### 11.9.2. Multi-task sets

Just as type `KHE_TASK_SET` represents a simple set of tasks, so `KHE_MTASK_SET` represents a simple set of mtasks. The only wrinkle is that an mtask set remembers the interval that it covers (the union of the values of `KheMTaskInterval(mt)` for each of its mtasks `mt`). This is done to make function `KheMTaskSetInterval`, presented below, very efficient.

The operations on mtask sets follow those on task sets, with a few adjustments. To create and delete an mtask set, call

```
KHE_MTASK_SET KheMTaskSetMake(KHE_MTASK_FINDER mtf);
void KheMTaskSetDelete(KHE_MTASK_SET mts, KHE_MTASK_FINDER mtf);
```

Deleted mtask sets are held in a free list in `mtf`, and freed when `mtf`'s arena is freed.

Three operations are offered for reducing the size of an mtask set:

```
void KheMTaskSetClear(KHE_MTASK_SET mts);
void KheMTaskSetClearFromEnd(KHE_MTASK_SET mts, int count);
void KheMTaskSetDropFromEnd(KHE_MTASK_SET mts, int n);
```

`KheMTaskSetClear` clears `mts` back to the empty set. `KheMTaskSetClearFromEnd` removes mtasks from the end until `count` mtasks remain. If `count` is larger than the number of mtasks in `mts`, none are removed. `KheMTaskSetDropFromEnd` removes the last `n` mtasks from `mts`. If `n` is larger than the number of mtasks in `mts`, all are removed.

Two operations are offered for adding mtasks to an mtask set:

```
void KheMTaskSetAddMTask(KHE_MTASK_SET mts, KHE_MTASK mt);
void KheMTaskSetAddMTaskSet(KHE_MTASK_SET mts, KHE_MTASK_SET mts2);
```

`KheMTaskSetAddMTask` adds `mt` to the end of `mts`; `KheMTaskSetAddMTaskSet` appends the elements of `mts2` to the end of `mts` without disturbing `mts2`.

Here are two operations for deleting one mtask:

```
void KheMTaskSetDeleteMTask(KHE_MTASK_SET mts, KHE_MTASK mt);
KHE_MTASK KheMTaskSetLastAndDelete(KHE_MTASK_SET mts);
```

`KheMTaskSetDeleteMTask` deletes `mt` from `mts` (it must be present). Assuming that `mts` is not empty, `KheMTaskSetLastAndDelete` deletes and returns the last mtask of `mts`.

To find out whether an mtask set contains a given mtask, call

```
bool KheMTaskSetContainsMTask(KHE_MTASK_SET mts, KHE_MTASK mt, int *pos);
```

If found, this sets `*pos` to `mt`'s index in `mts`. To visit the mtasks of an mtask set, call

```
int KheMTaskSetMTaskCount(KHE_MTASK_SET mts);
KHE_MTASK KheMTaskSetMTask(KHE_MTASK_SET mts, int i);
```

in the usual way. There is also

```
KHE_MTASK KheMTaskSetFirst(KHE_MTASK_SET mts);
KHE_MTASK KheMTaskSetLast(KHE_MTASK_SET mts);
```

which return the first and last elements when `mts` is non-empty.

For sorting an mtask set there is

```
void KheMTaskSetSort(KHE_MTASK_SET mts,
  int(*compar)(const void *, const void *));
```

where `compar` compares mtasks. There is also

```
void KheMTaskSetUniqueify(KHE_MTASK_SET mts);
```

which uses a call to `HaArraySortUnique` with a suitable comparison function to uniqueify `mts`, that is, to ensure that each mtask in `mts` appears there at most once. The mtasks are sorted by increasing starting time, with ties broken by increasing order of `KheTaskSolnIndex` applied to each mtask's first task. This does what is wanted, given than every mtask contains at least one task, and no task appears in two mtasks.

When `mts`'s mtasks all have fixed times, function

```
KHE_INTERVAL KheMTaskSetInterval(KHE_MTASK_SET mts);
```

returns the smallest interval containing the indexes in the days frame of the days of all of their times. As mentioned earlier, this interval is kept up to date as mtasks are added and removed, ensuring that `KheMTaskSetInterval` just has to return a field of `mts`, making it very fast.

Next come operations for changing the assignments of resources to an mtask set:

```
bool KheMTaskSetMoveResourceCheck(KHE_MTASK_SET mts,
  KHE_RESOURCE from_r, KHE_RESOURCE to_r, bool disallow_preassigned,
  bool unassign_extreme_unneeded);
bool KheMTaskSetMoveResource(KHE_MTASK_SET mts,
  KHE_RESOURCE from_r, KHE_RESOURCE to_r, bool disallow_preassigned,
  bool unassign_extreme_unneeded);
```

`KheMTaskSetMoveResource` calls `KheMTaskMoveResource` for each mtask `mt` of `mts`, and `KheMTaskSetMoveResourceCheck` checks whether this would succeed, without doing it.

When `to_r != NULL` and `unassign_extreme_unneeded` is `true`, the first and last mtasks in `mts` are treated differently. For each, if there is a needless assignment in the mtask, according to `KheMTaskContainsNeedlessAssignment` (Section 11.9.1), the mtask is unassigned instead of moved. Over the course of a solve this reduces the number of needless assignments, reducing resource workloads and generally improving solutions, as the author's tests have shown.

Two similar functions are

```
bool KheMTaskSetMoveResourcePartialCheck(KHE_MTASK_SET mts,
  int first_index, int last_index, KHE_RESOURCE from_r, KHE_RESOURCE to_r,
  bool disallow_preassigned, bool unassign_extreme_unneeded);
bool KheMTaskSetMoveResourcePartial(KHE_MTASK_SET mts,
  int first_index, int last_index, KHE_RESOURCE from_r, KHE_RESOURCE to_r,
  bool disallow_preassigned, bool unassign_extreme_unneeded);
```

These are like `KheMTaskSetMoveResourceCheck` and `KheMTaskSetMoveResource` except that they only apply to some of the mtasks of `mts`, those whose index in `mts` lies between `first_index` and `last_index` inclusive—just as though these were the only mtasks in `mts`.

Finally we have

```
void KheMTaskSetDebug(KHE_MTASK_SET mts, int verbosity, int indent,
  FILE *fp);
```

which produces a debug print of `mts` onto `fp` with the given verbosity and indent.

### 11.9.3. Multi-task finders

The operation for creating mtasks is

```
KHE_MTASK_FINDER KheMTaskFinderMake(KHE_SOLN soln, KHE_RESOURCE_TYPE rt,
  KHE_FRAME days_frame, bool fixed_times, HA_ARENA a);
```

Using memory from arena `a`, this makes a `KHE_MTASK_FINDER` object containing mtasks such that every proper root task of `soln` whose type is `rt` lies in exactly one mtask. Or `rt` may be `NULL`, and then mtasks are created for every resource type. Parameter `days_frame` holds the common frame and influences the operations below that depend on days. An mtask finder is deleted when its arena is deleted, along with its mtasks and mtask sets.

If `fixed_times` is `true`, the finder assumes that any times currently assigned to meets will remain as they are for its entire lifetime. (This is not checked, so care is needed here.) This allows it to treat tasks from different meets as equivalent, if they run at the same times and satisfy all other requirements. If `fixed_times` is `false`, the finder does not make this assumption. Instead, equivalent tasks must come from the same meet, so that they always run at the same times, even if those times change or are unassigned. For full details, consult Section 11.9.4.

These simple queries return the attributes passed in:

```
KHE_SOLN KheMTaskFinderSoln(KHE_MTASK_FINDER mtf);
KHE_FRAME KheMTaskFinderDaysFrame(KHE_MTASK_FINDER mtf);
bool KheMTaskFinderFixedTimes(KHE_MTASK_FINDER mtf);
KHE_ARENA MTaskFinderArena(KHE_MTASK_FINDER mtf);
```

To find out which resource types the mtask finder is handling, there are functions

```
int KheMTaskFinderResourceTypeCount(KHE_MTASK_FINDER mtf);
KHE_RESOURCE_TYPE KheMTaskFinderResourceType(KHE_MTASK_FINDER mtf, int i);
bool KheMTaskFinderHandlesResourceType(KHE_MTASK_FINDER mtf,
  KHE_RESOURCE_TYPE rt);
```

The first two allow you to visit the resource types handled by `mtf`; the third tells you whether `mtf` handles a given resource type. These functions are arguably overkill, since `mtf` either handles one resource type or all resource types; but in principle it could handle any subset of the resource types, so this approach has seemed best.

When dealing with mtasks, the days of the common frame that they are running on loom large. These days are often represented by their indexes in the common frame (parameter `days_frame` of `KheMTaskFinderMake`). The index of the first day is 0, and of the last day is

```
int KheMTaskFinderLastIndex(KHE_MTASK_FINDER mtf);
```

This is one less than the number of time groups in `days_frame`.

To visit the mtasks of a `KHE_MTASK_FINDER` object, the calls are

```
int KheMTaskFinderMTaskCount(KHE_MTASK_FINDER mtf);
KHE_MTASK KheMTaskFinderMTask(KHE_MTASK_FINDER mtf, int i);
```

as usual. The order that the mtasks appear here is arbitrary, unless one chooses to first call

```
void KheMTaskFinderMTaskSort(KHE_MTASK_FINDER mtf,
  int (*compar)(const void *, const void *));
```

to sort the mtasks using function `compar`. One comparison function is provided:

```
int KheMTaskDecreasingDurationCmp(const void *, const void *);
```

`KheMTaskFinderMTaskSort(mtf, &KheMTaskDecreasingDurationCmp)` sorts the mtasks by decreasing duration, which might be a good heuristic for ordering them for assignment.

When mtasks are in use, it is best to deal only with them and not access tasks directly. When a task is returned by some function and has to be dealt with, the right course is to call

```
KHE_MTASK KheMTaskFinderTaskToMTask(KHE_MTASK_FINDER mtf, KHE_TASK t);
```

to move from task `t` to its proper root task and from there to the mtask containing that proper root task. This function will abort if there is no such mtask. That should never happen, provided the resource type of `t` is the resource type, or one of the resource types, handled by `mtf`.

When the `fixed_times` parameter of `KheMTaskFinderMake` is `true`, one can call

```
KHE_MTASK_SET KheMTaskFinderMTasksInTimeGroup(KHE_MTASK_FINDER mtf,
  KHE_RESOURCE_TYPE rt, KHE_TIME_GROUP tg);
KHE_MTASK_SET KheMTaskFinderMTasksInInterval(KHE_MTASK_FINDER mtf,
  KHE_RESOURCE_TYPE rt, KHE_INTERVAL in);
```

These return the set of mtasks of resource type `rt` that are running at any time of `tg` (which must be non-empty), or at any time of any time group of interval `in` of `mtf`'s days frame (again, `in` must be non-empty). Each set is built on demand (except that for singleton time groups `tg` the sets are built when `mtf` itself is built), sorted by increasing start time, uniqueified by `KheMTaskSetUniqueify` when necessary, and cached within `mtf` so that subsequent requests for it run quickly. The caller must not modify these mtask sets.

A similar function is

```
void KheMTaskFinderAddResourceMTasksInInterval(KHE_MTASK_FINDER mtf,
  KHE_RESOURCE r, KHE_INTERVAL in, KHE_MTASK_SET mts);
```

This adds to `mts` the mtasks that `r` is assigned to that lie wholly within interval `in` in the current frame, in chronological order. These mtasks can change as resource assignments change, so there is no caching of the results. One can also do a similar job avoiding mtasks by calling

```
void KheAddResourceProperRootTasksInInterval(KHE_RESOURCE r,
  KHE_INTERVAL in, KHE_SOLN soln, KHE_FRAME days_frame,
  KHE_TASK_SET ts);
```

to add to `ts` the proper root tasks assigned `r` in `soln` that lie wholly within `in` of `days_frame`.

When `fixed_times` is `false`, or tasks lie in unassigned meets, the functions just given aren't really useful. But there are other ways to visit mtasks. `KheMTaskFinderMTaskCount` and `KheMTaskFinderMTask` will visit them all, for example. Another option is to visit the tasks of a given meet and use `KheMTaskFinderTaskToMTask` to find the mtasks containing those tasks.

We return now to functions that are available irrespective of the value of `fixed_times`. It was mentioned at the start of this section that several operations on tasks which are forbidden (because they would change the mtask structure) have mtask versions which both carry out the forbidden operation and change the mtask structure, possibly creating or destroying some mtasks as they do so. These operations are

```
bool KheMTaskFinderTaskMove(KHE_MTASK_FINDER mtf, KHE_TASK task,
  KHE_TASK target_task);
bool KheMTaskFinderTaskAssign(KHE_MTASK_FINDER mtf, KHE_TASK task,
  KHE_TASK target_task);
bool KheMTaskFinderTaskUnAssign(KHE_MTASK_FINDER mtf, KHE_TASK task);
bool KheMTaskFinderTaskSwap(KHE_MTASK_FINDER mtf, KHE_TASK task1,
  KHE_TASK task2);
void KheMTaskFinderTaskAssignFix(KHE_MTASK_FINDER mtf, KHE_TASK task);
void KheMTaskFinderTaskAssignUnFix(KHE_MTASK_FINDER mtf, KHE_TASK task);
```

`KheMTaskFinderTaskMove` (for example) calls `KheTaskMove`, and it also updates `mtf`'s data structures so that the right results continue to be returned by `KheMTaskFinderMTaskCount`, `KheMTaskFinderMTask`, `KheMTaskFinderMTaskFromMeetCount`, `KheMTaskFinderMTaskFromMeet`, and also by functions `KheMTaskFinderTaskToMTask`, `KheMTaskFinderMTasksInTimeGroup`, and `KheMTaskFinderMTasksInInterval`. Mtasks held by the user, either directly or in user-defined mtask sets, may become undefined when mtasks are created and destroyed.

Because of these updates, `KheMTaskFinderTaskMove` and the other functions above are too slow to be called from within time-critical code; but they are fine for other applications. Structural solvers, for example, are usually not time-critical. The related checking and query functions (`KheTaskMoveCheck` and so on) are safe to call directly, since they change nothing.

To *group* some tasks means to move them to a common *leader task*, forcing solvers to assign the same resource to each task in the group (by assigning a resource to the leader task). All tasks involved must be proper root tasks. If any of them are assigned a resource before grouping, then it must be the same resource, and the leader task will be assigned that resource after grouping.

The mtask finder offers operations for task grouping:

```
void KheMTaskFinderGroupBegin(KHE_MTASK_FINDER mtf, KHE_TASK leader_task);
bool KheMTaskFinderGroupAddTask(KHE_MTASK_FINDER mtf, KHE_TASK task);
void KheMTaskFinderGroupEnd(KHE_MTASK_FINDER mtf, KHE_SOLN_ADJUSTER sa);
```

`KheMTaskFinderGroupBegin` clears out any previous task grouping information and sets the leader task (a proper root task). Then any number of calls to `KheMTaskFinderGroupAddTask` set the tasks (also proper root tasks) to be assigned to the leader task, without actually carrying out those assignments. The return value is `true` if `task` can be included; if it is `false`, `task` is omitted from the grouped tasks, either because it cannot be moved to `leader_task`, or because it is assigned a resource and some other task in the group is assigned a different resource. Finally, `KheMTaskFinderGroupEnd` actually carries out the moves. If `sa != NULL` these are recorded in solution adjuster `sa`, allowing them to be undone later if desired.

A sequence of calls to `KheMTaskFinderTaskAssign` would do what these calls do. But these calls are faster because they build only the final mtask which reflects all the assignments.

Finally,

```
void KheMTaskFinderDebug(KHE_MTASK_FINDER mtf, int verbosity,
  int indent, FILE *fp);
```

produces a debug print of `mtf` onto `fp` with the given verbosity and indent.

### 11.9.4. Behind the scenes 1: defining task similarity

It is now time to look behind the scenes, and see how mtasks guarantee that symmetrical assignments will be avoided, and at the same time that nothing useful will be missed.

Behind the scenes, then, an mtask is a sequence (not a set) of proper root tasks, each optionally assigned a resource. When *m* resources are assigned to an mtask, they are assigned to the first *m* proper root tasks in the sequence. Each mtask contains the proper root tasks of one equivalence class of an equivalence relation between proper root tasks that we call *task similarity*. To turn this set into a sequence we sort the elements into non-decreasing order of an attribute of each task called its *task cost*.

It is easy to see how mtasks avoid many assignments. Suppose we have $n$ unassigned tasks, and that we decide to assign $m$ resources to these tasks, where $m \leq n$. For the first resource there are $n$ unassigned tasks to choose from, for the second there are $n - 1$ to choose from, and so on, giving $n(n-1)\dots(n-m+1)$ choices altogether. This could be a very large number. But now suppose that these $n$ tasks are grouped into an mtask. Then the mtask tries just one of these choices, the one which assigns the first resource that comes along to the first task, the second to the second, and so on. So there is a large reduction in the number of choices. The question is whether anything useful has been missed.

'Missing something useful' is really an appeal to a dominance relation between solutions (Appendix C). We claim that any solution containing assignments of any $m$ resources to the $n$ tasks is dominated by the solution containing the assignments chosen by the mtask. The proof will go like this. Limit all consideration to the $m$ resources and $n$ tasks of interest. If a resource is assigned to a task that appears later in the mtask's sequence than some other task which is unassigned, then we can move the resource to that earlier unassigned task, and the move will not increase the cost of the solution, in fact it might decrease it. And then, exchanging the assignments of any two resources can be done and will not change solution cost. These two facts, if we can prove them, will together show that we can transform our solution into the mtask's solution with no increase in cost.

We call a task, considered independently of any tasks that may be assigned to it, an *atomic task*. We view one proper root task as the set of all the atomic tasks assigned to it, directly or indirectly, including itself. Apart from domains, preassignments, and fixed assignments, which relate specifically to the root task, only this set matters, not which tasks are assigned to which.

As mentioned earlier, KHE allows tasks to be created that are not derived from any meet. These would typically serve as proper root tasks to which tasks derived from meets could be assigned. Such tasks are consulted to find domains, preassignments, and fixed assignments when they are proper root tasks, but since they do not run at any times and have no effect on any monitors they are ignored otherwise: they are not included among the atomic tasks. This means that the set of atomic tasks could be empty. However we do not treat this case as special. Conditions of the form 'for each atomic task, ...' are vacuously true.

A proper root task is said to have fixed times if each of its atomic tasks lies in a meet with an assigned time, and the `fixed_times` parameter of `KheMTaskFinderMake` is `true`, allowing us to assume that these assigned times will not change. In that case, similarity is based on the assigned times of the tasks' meets. Otherwise, things are handled as though none of the tasks have assigned times, and similarity is based on their meets.

Two proper root tasks are similar when they satisfy these conditions:

(1) They have equal domains.

(2) They are either both unpreassigned, or both preassigned the same resource. This second possibility inevitably causes clashes, which means that in practice a preassigned task will usually not be similar to any other task, making it the only member of its mtask.

(3) The assignments of both tasks are not fixed. In other words, a task whose assignment is fixed is always the only member of its mtask.

(4) The number of atomic tasks must be the same for both tasks, and taking them in a canonical order based on their assigned times and meets, corresponding atomic tasks must be similar, according to a definition to be given below. This condition is vacuously true when both tasks have no atomic tasks.

Assuming that the similarity relation for atomic tasks is an equivalence relation, this evidently defines an equivalence relation on proper root tasks, as required.

Two atomic tasks are similar when they satisfy these conditions:

(1) They have equal durations and workloads.

(2) Either they both have an assigned time, in which case those times are equal, or they both don't, in which case their meet indexes are equal. This second case is always followed when `fixed_times` is `false`, consistent with what was said about this above. It is also followed when at least one of the atomic tasks in question has no assigned time.

(3) They are similar in their effects on monitors. There are many details to cover here; these are tackled below.

Once again, this is clearly an equivalence relation, provided that (3) is an equivalence relation.

These rules could be improved on. For example, if there are no limit workload monitors, then task workloads do not matter. Still, what we have is simple and works well in practice.

The rest of this section is concerned with similarity of two atomic tasks in their effect on monitors. The general idea is that this similarity holds when, for all resources *r*, assigning *r* to one of the tasks has the same effect on monitors as assigning it to the other task. But there are complications in making this general idea concrete, as we are about to see.

We can safely ignore unattached monitors and monitors with weight 0. A monitor can be an *event monitor*, monitoring the times assigned to a specified set of events, or an *event resource monitor*, monitoring the resources assigned to a specified set of tasks, or a *resource monitor*, monitoring the busy times or workload of a specified resource. We'll take each kind in turn.

*Event monitors* are unaffected by the assignments of resources to tasks. They depend only on the times assigned to meets. So we can ignore them here.

*Resource monitors* are not directly concerned with which tasks a resource is assigned to, but rather with those tasks' busy times and workloads. We have already required similar tasks to be equal in those respects, so that moving a resource from one similar task to another leaves its resource monitors unaffected. This is true whether or not times are assigned.

*Event resource monitors* (assign resource, prefer resources, avoid split assignments, and limit resources monitors) are where things get harder. The tests we have so far included in the similarity condition do not guarantee that event resource monitors will be unaffected when a resource is moved from one task to another—far from it.

Before we delve into event resource monitors, there is a special case we need to dispose of. Consider an avoid split assignments monitor *m* whose monitored tasks are all assigned to each other (have the same proper root). At most one distinct resource can be assigned to these tasks, so *m* must have cost 0. It can be and is ignored. This case is quite likely to arise in practice, although *m* might be detached when it does. It includes the case where *m* monitors a single task.

The author spent some time considering what happens with other kinds of event resource monitors when their tasks have the same proper root. These monitors monitor a single task, in effect, which is helpful for similarity. However these cases seem unlikely to arise in practice, and some of their details are not obvious, so nothing special has been done about them.

Event resource monitors explicitly name the tasks (always atomic) that they *monitor* (are affected by). We divide them into two groups. A *separable monitor* is one whose cost may be apportioned to the tasks it monitors, each portion depending only on the assignment of that one task. A *inseparable monitor* is one whose cost cannot be apportioned in this way.

A monitor that monitors just one task is separable, because all its cost can be apportioned to that task. But there are less trivial examples. Consider an assign resource constraint with a linear cost function. Its cost is its weight times the total duration of its unassigned tasks, and this may be apportioned to the individual unassigned tasks, making the monitor a separable one. But if the cost function is not linear, one cannot apportion the cost in this way.

We analyse inseparable monitors first. If task $t$ is monitored by inseparable monitor $m$, the cost of assigning a resource to $t$ cannot be apportioned to $t$. This indeterminacy in cost prevents us from saying definitely what the effect on $m$ of a resource assignment is. So in this case, $t$ cannot be considered similar to any other task.

There is however an exception to this rule. Consider two tasks both monitored by $m$. An examination of the event resource constraints will show that, provided the two tasks have equal durations, the effect on $m$ of assigning a given resource $r$ to either task must be the same. So $m$ does not prevent the two tasks from being declared similar. Altogether, then, for two atomic tasks to be similar they must have the same inseparable monitors—not monitors with the same attributes, but the exact same monitors.

We turn now to separable monitors. Each task has its own individually apportionable cost, dependent only on its own assignment. Again we divide these monitors into two groups: *resource-dependent separable monitors*, for which the cost depends on the choice of resource, and *resource-independent separable monitors*, for which the cost depends only on whether the task is assigned or not, not on the choice of resource.

For example, a separable prefer resources monitor will usually be resource-dependent, because the cost depends on whether the assigned resource is a preferred one or not. But if the set of preferred resources is empty, assigning any resource produces the same cost, and the monitor is resource-independent.

To analyse the resource-dependent separable monitors, consider the usual kind of separable prefer resources monitor. The cost depends on which resource is assigned, so the permutations of resource assignments that mtasks rely on could produce virtually any cost changes. So we require, for similarity, that the resource-dependent separable monitors of the two tasks can be put into one-to-one correspondence such that corresponding monitors have the same attributes (type, hardness, cost function, weight, preferred resources, and limits where present).

We are left with just the resource-independent separable monitors, whose cost depends only on whether each task is assigned or not, not on which resource is assigned. We could repeat the previous work and require a one-to-one correspondence between these monitors such that corresponding monitors have the same attributes. But we can do better.

Consider three tasks, $t_1$, $t_2$, and $t_3$, that are similar according to the rules so far. Suppose $t_1$

is monitored by a separable assign resource monitor with weight 20, $t_2$ is not monitored, and $t_3$ is monitored by a separable prefer resources monitor with an empty set of resources and weight 10. Assuming duration 1, assigning any resource to $t_1$ reduces the cost of the solution by 20; assigning any resource to $t_2$ does not change the cost; and assigning any resource to $t_3$ increases the cost by 10. Examples like this are common in nurse rostering, to place limits on the number of nurses assigned to a shift. Here, at least one nurse is wanted, but three is too many.

Let the *task cost* of a task $t$ be the sum, over all resource-independent separable monitors $m$ that monitor $t$, of the change in cost reported by $m$ when $t$ goes from being unassigned to being assigned. In the example above, assuming duration 1, the task costs are $-20$ for $t_1$, 0 for $t_2$, and 10 for $t_3$. These values are independent of all other assignments, and also of which resource is being assigned, and so they can be calculated in advance of any solving, while mtasks are being constructed. When adding an assignment to an mtask, it will always be better to choose a remaining unassigned task with minimum task cost. So the mtask sorts its tasks by non-decreasing task cost at the start, and assigns them in sorted order.

It remains to state, for each monitor type, the conditions under which it is separable, and if separable, resource-independent. The examples given earlier cover most of these cases.

An assign resource monitor is separable when it monitors a single task, or its cost function is linear, or both. It is then always resource-independent. Otherwise it is inseparable.

A prefer resources monitor is ignored when its set of preferred resources includes every resource of its resource type, since its cost is always 0 then. Otherwise, it is separable when it monitors a single task, or its cost function is linear, or both. It is then resource-independent when its set of preferred resources is empty. Otherwise it is inseparable.

An avoid split assignments monitor is ignored when its tasks all have the same proper root (including when it monitors a single task). Otherwise it is always considered to be inseparable.

It would not be unreasonable to declare all limit resources monitors to be inseparable, since in practice they apply to multiple tasks and have non-trivial limits. However, they can also be used to do what assign resource monitors do, by selecting all resources and setting the minimum limit to the total duration of the tasks monitored. They can also be used to do what prefer resources monitors do, by selecting those resources that are not selected by the prefer resources monitor, and setting the maximum limit to 0. In these cases, we want a limit resources monitor to be classified in the same way that the assign resource or prefer resources monitor would be.

If a limit resources monitor is equivalent to an assign resource or prefer resources monitor as just described, it is classified as that other monitor would be. Otherwise, it is separable when its cost function is linear and its maximum limit is 0. It is then resource-independent when its set of preferred resources contains every resource of its type. It is also separable when it monitors a single task. In that case it is resource-independent when its set of preferred resources contains every resource of its type, in which case the assignment cost depends on how the duration of the task compares with the monitor's limits. (Curiously, if the task's duration is less than the minimum limit, there will be both a non-assignment cost and an assignment cost, because the minimum limit is not reached whether the task is assigned or not.) Otherwise it is inseparable.

To recapitulate, then, two proper root tasks are similar when they have equal domains and preassignments, they are both not fixed, and they have similar atomic tasks. Two atomic tasks are similar when they have equal durations, workloads, and start times (or meets), their inseparable monitors are the same, and their resource-dependent separable monitors have equal attributes.

Their resource-independent separable monitors (usually assign resource monitors, and prefer resources monitors with no preferred resources) may differ: instead of influencing similarity, they determine the task's position in the sequence of tasks of its mtask.

### 11.9.5. Behind the scenes 2: accessing mtasks and mtask sets

This section describes the mtask finder's moderately efficient data structure for accessing mtasks by signature, and for finding the mtask sets returned by `KheMTaskFinderMTasksInTimeGroup` and `KheMTaskFinderMTasksInInterval`. It has been written to clarify the ideas of its somewhat confused author, and is not likely to be of any value to the user.

Quite a few objects are created and deleted in the operations that follow. Deleted objects are added to free lists in the mtask finder, where they are available for future creations.

The data structure allows proper root tasks to be inserted and deleted at any moment, not just during initialization. This flexibility is needed to support the 'forbidden' operations, which work by deleting from the data structure the proper root tasks they affect, carrying out the operation requested, and then inserting the result tasks back into the data structure.

Actually there are three data structures. First, there is an array of all mtasks, included to support `KheMTaskFinderMTaskCount` and `KheMTaskFinderMTask`. Each mtask contains its index in this array. To add a new mtask we add it to the end and set its index; to delete it we use its index to find its position, and move the last element to that position, changing its index.

Second, there is an array of mtasks indexed by task index (function `KheTaskSolnIndex` from the KHE platform). For each task handled by the mtask finder (each proper root task of a suitable resource type), the value at its index is its mtask. Other indexes have `NULL` values and are never accessed. This supports a trivial implementation of `KheMTaskFinderTaskToMTask`. When a task is added to an mtask or removed from it, the value at its index is changed.

We won't mention these two arrays again, although they are kept up to date as the structure changes. All subsequent data structure descriptions relate to the third data structure.

Every task has a resource type, and every mtask has one too, because its tasks all have the same domain. `KheMTaskFinderMTasksInTimeGroup` and `KheMTaskFinderMTasksInInterval` have a resource type parameter and return sets of mtasks which all have that resource type.

So all operations that we are concerned with here have a parameter which is a non-`NULL` resource type; call it `rt`. Each operation traverses a short list of tables (this list is the entry point for the third data structure), one table for each resource type supported by the mtask finder, to find the table for `rt`. The rest takes place in that table; everything in it has resource type `rt`.

***Task insertion.*** To add a proper root task to the structure, first we build its *signature*. This is an object containing everything needed to decide whether two proper root tasks are similar, as defined in Section 11.9.4, including one *atomic signature* for each atomic task assigned, directly or indirectly, to the proper root task. Atomic signatures are sorted into a canonical order for ease of comparison. The non-assignment and assignment costs, as returned by `KheMTaskTask`, are calculated at the same time as the signature but are not part of it and are stored separately.

The tasks of an mtask have equal signatures. This shared signature is stored in the mtask. A task belongs in an mtask if its signature is equal to the mtask's stored signature.

So after calculating the signature of the new task, the second step is to search the appropriate table to see if it contains an mtask with the same signature as the signature of the new task. There are three different ways to do this, depending on the *type* of the signature:

`KHE_SIG_FIXED_TIMES`
: A task's signature has this type when the `fixed_times` parameter of `KheMTaskFinderMake` is `true`, each of its atomic tasks derived from a meet has an assigned time, and there is at least one such atomic task. So the task has a chronologically first assigned time, and we use that as an index into the table. We'll explain how this is done later on.

`KHE_SIG_MEETS`
: A task's signature has this type when the `fixed_times` parameter of `KheMTaskFinderMake` is `false`, or not every atomic task derived from a meet has an assigned time, and there is at least one atomic task derived from a meet. We use any one of these meets to find other tasks with the same signature: we traverse the set of all tasks of the meet, and for each of those of the right resource type that has an mtask, we compare the mtask's signature with the new task's signature. So there is no third data structure for this case; the meet itself provides a suitable structure. This would not work for `KHE_SIG_FIXED_TIMES`, because fixed-time tasks with the same signatures can come from different meets.

`KHE_SIG_OTHER`
: A task's signature has this type when neither of the other two cases applies. This means that the task has no atomic tasks derived from meets; its duration is therefore zero and it is basically useless. Still, for uniformity it must lie in an mtask. These mtasks are likely to be very few, so they are stored in a separate list in the table, and this list is searched to find the mtask (if any) with this signature.

Whichever way the search is done, if it finds an existing mtask whose signature is equal to the new task's signature, all we have to do is add the new task to that mtask and throw away the new task's signature. If it does not find an existing mtask with that signature, we have to create a new mtask with that signature, add the new task to it as its first task, and insert the new mtask into the data structure. This insertion does nothing if the signature type is `KHE_SIG_MEETS`, and it is a simple addition to the end of the table's separate list if the signature type is `KHE_SIG_OTHER`. How an mtask is inserted when its type is `KHE_SIG_FIXED_TIMES` is a subject for later.

***Task deletion.*** To delete a task, we first delete it from its mtask, obtained by the usual call to `KheMTaskFinderTaskToMTask`. If the mtask becomes empty, we then have to delete the mtask (we don't allow empty mtasks). We do this in one of three ways depending on the signature type. If the type is `KHE_SIG_MEETS` there is nothing to do; if it is `KHE_SIG_OTHER` we search the appropriate table's separate list of mtasks of this type and delete the mtask from there. If the type is `KHE_SIG_FIXED_TIMES` we use the first assigned time to index the table, as for insertion, and carry on as described below.

***The third data structure.*** The third data structure supports five operations: mtask retrieval by signature, mtask insertion, mtask deletion, `KheMTaskFinderMTasksInTimeGroup`, and `KheMTaskFinderMTasksInInterval`. The last two operations are supposed to cache their results so that multiple calls with the same parameters run quickly. These cached values must be kept up to date as mtasks are inserted and deleted.

We've already shown how the first three operations are done when the signature type is `KHE_SIG_MEETS` or `KHE_SIG_OTHER`. The last two, `KheMTaskFinderMTasksInTimeGroup` and

KheMTaskFinderMTasksInInterval, do not deal in these two types of mtasks anyway. So we need to consider here only mtasks whose signatures have type KHE_SIG_FIXED_TIMES.

One entry in the third data structure has type

```
typedef struct khe_entry_rec {
  KHE_TIME_GROUP              tg;
  KHE_INTERVAL                in;
  KHE_MTASK_SET              mts;
} *KHE_ENTRY;
```

Entry e means: 'the value of KheMTaskFinderMTasksInTimeGroup(mtf, rt, tg) is e->mts when tg == e->tg, and the value of KheMTaskFinderMTasksInInterval(mtf, rt, in) is e->mts when in == e->in.' The rt parameter is not mentioned because e lies within one table of the third data structure, as defined above, and rt is taken care of when this table is selected.

One table of the third data structure, then, consists of an array indexed by time, where each element contains a list of these entries. An entry appears once in each list indexed by a time that is one of the times of its time group or interval (considered as a set of time groups). This means that an entry appears in the table as many times as its time group or interval has times.

As we will see, from time to time it will be necessary to add an entry to a table. However we never delete an entry. Once we begin keeping track of the mtasks of a particular time group or interval, we continue doing that until the mtask finder is deleted. This is arguably wasteful, but the perennial caching question (is this cache entry still needed?) has no easy answer here, and we expect to receive queries for only a moderate number of distinct time groups and intervals.

Let us see now how to implement the five operations.

To retrieve an mtask by signature, we take the chronologically first time of the signature (call it t), and we take the first entry of the list indexed by t. As we'll see later, this entry is always present and its mtask set contains every mtask whose signature includes t. So we search that mtask set for an mtask containing the signature we are looking for.

To insert a new mtask mt, we have to find every mtask set that mt belongs in and add it. So for each of mt's fixed times we traverse the list of entries indexed by that time and add mt to the mtask set in each entry. It is easy to see that these are exactly the mtask sets that mt needs to be added to. An entry can appear in several lists, so we only add mt to an mtask set when it is not already present. If it is present it will be at the end, so that condition can be checked quickly.

To delete an mtask mt we have to find every mtask set that mt is currently in and remove it. So for each of mt's fixed times we traverse the list of entries indexed by that time and delete mt from the mtask set in each entry. Because an entry can appear in several lists, we only attempt to delete mt from an entry's mtask set when it is present.

To implement KheMTaskFinderMTasksInTimeGroup(mtf, rt, tg), we first need to check whether the rt table contains an entry for tg. We do this by searching the list of entries indexed by the first time of tg (it is a precondition that tg cannot be empty) for an entry containing tg. If we find one, we return its mtask set and we are done.

If there is no entry containing tg, we have to make one and add it to each list indexed by a time of tg, which is straightforward. The hard part is that we also have to build the mtask set of all mtasks whose fixed times have a non-empty intersection with tg, so that we can add it to the new entry and also return it to the caller. We could do this from scratch, by finding all tasks running at the relevant times, then building and uniqueifying the set of all these tasks' mtasks. But we do it in a faster way, as follows.

As we saw when inserting and deleting mtasks, once an entry is present it is kept up to date as mtasks come and go. So during the initialization of the mtask finder, before any mtasks have been created, we add one entry to the start of each list. If the list is for time `t`, the entry contains a time group containing just `t` (as returned by platform function `KheTimeSingletonTimeGroup`) and an empty mtask set. As mtasks are inserted and deleted, this mtask set will always hold the set of all fixed-time mtasks whose times include `t`. This entry will always be first in its list.

So to build the new mtask set, we take the union of the mtask sets in the first entries of the lists indexed by the times of the new time group. We call `KheMTaskSetAddMTaskSet` repeatedly to build the union, then we call `KheMTaskSetUniqueify` to uniqueify it.

`KheMTaskFinderMTasksInInterval` is similar to `KheMTaskFinderMTasksInTimeGroup`. Its `in` parameter is just a shorthand for the union of the time groups of `in`'s days.

Where then is the confusion? The author was not sure whether each entry had to be added to multiple lists. Suppose each entry was added to just one list, the one for its time group's first time. `KheMTaskFinderMTasksInTimeGroup` and `KheMTaskFinderMTasksInInterval` at least would be fine: they use only that first time to access the table. Would anything go wrong?

Just one thing would go wrong, as it turns out. When a new mtask is added, it would be added to the mtask set of each entry whose time group's first time is one of the mtask's fixed times. But that is not enough. For example, an mtask holding tasks of the Wednesday night shift would not be added to the mtask set holding all mtasks running on Wednesday, because that mtask set's entry would lie only in the list indexed by the first time on Wednesday.

The mtask finder's similarity rule must be complicated, but are the complications just described necessary? The author believes that they are. `KheMTaskFinderMTasksInTimeGroup` and `KheMTaskFinderMTasksInInterval` are used frequently by the ejection chain solver, so they must run quickly. The symmetry elimination provided by mtasks is essential for grouping by resource constraints (Section 11.10), and that solver also needs the 'forbidden' operations. We don't want multiple multi-task software modules, so one module has to do it all.

## 11.10. Task grouping by resource constraints

*Task grouping by resource constraints*, or *TGRC*, is KHE's term for grouping tasks together, forcing the tasks in each group to be assigned the same resource, based on analyses of resource constraints which suggest that solutions in which the tasks in each group are not assigned the same resource are likely to be inferior. That does not mean that those tasks will always be assigned the same resource in good solutions, any more than, say, a constraint requiring nurses to work complete weekends is always satisfied in good solutions. However, in practice those tasks usually do end up being assigned the same resource, so it makes sense to require it, at least to begin with. Later we can remove the groups and see what happens.

`KheTaskTreeMake` also groups tasks, but its groups are based on avoid split assignments constraints, whereas here we make groups based on resource constraints.

The function is

```
bool KheGroupByResourceConstraints(KHE_SOLN soln, KHE_RESOURCE_TYPE rt,
  KHE_OPTIONS options, KHE_SOLN_ADJUSTER sa);
```

There is no `tasking` parameter because this kind of grouping cannot be applied to an arbitrary set of tasks, as it turns out. Instead, it applies to all tasks of `soln` whose resource type is `rt`, which lie in a meet which is assigned a time, with some exceptions, discussed below. If `rt` is `NULL`, `KheGroupByResourceConstraints` applies itself to each of the resource types of `soln`'s instance in turn. It tries to group these tasks, returning `true` if it groups any. If `sa != NULL`, it saves any changes in solution adjuster `sa` (Section 8.6.1), so that they can be undone later.

`KheGroupByResourceConstraints` finds whatever groups it can among these tasks. It makes each such *task group* by choosing one of its tasks as the *leader task* and assigning the others to it. It makes assignments only to proper root tasks (non-cycle tasks not already assigned to other non-cycle tasks), so it does not disturb existing groups. But it does take existing groups into account: it will use tasks to which other tasks are asssigned in its own groups.

Tasks which are initially assigned a resource could, in principle, participate in TGRC. The rule then would be that two tasks put into the same group may not be assigned different resources initially; and if any of the grouped tasks are assigned a resource initially, then the whole group is assigned that resource finally. `KheMTaskFinderGroupBegin`, `KheMTaskFinderGroupAddTask`, and `KheMTaskFinderGroupEnd` from Section 11.9.3 implement this rule.

However, in practice, when `KheGroupByResourceConstraints` is called the only tasks assigned a resource have been assigned by `KheAssignByHistory` (Section 12.4.2). In effect, those tasks are already grouped. Given that `KheGroupByResourceConstraints` does not take account of history (ideally it would, but it does not at present), the practical way forward is for it to ignore tasks which are assigned a resource, just as though they were not there.

Tasks whose assignments are fixed (even to `NULL`) are also ignored. It is true that they could become leader tasks, since the assignments of leader tasks are not changed, but there are other considerations when choosing leader tasks, and to add fixing to the mix has seemed to the author to be a bridge too far. In any case there are not likely to be any fixed unassigned proper root tasks when `KheGroupByResourceConstraints` is called.

Tasks for which non-assignment has no cost also do not participate in TGRC, because they cause problems in some places (notably profile grouping). In practice only a few such tasks are assigned resources in good solutions. It would probably help to allow a few of them into groups, when there is no other way to build the needed groups. But that is not being done at present.

To summarize, then, `KheGroupByResourceConstraints` applies to each proper root task of `soln` whose resource type is `rt` (or any type if `rt` is `NULL`), which lies in a meet which is assigned a time, is not assigned a resource, does not have a fixed assignment, and for which non-assignment has a cost.

`KheGroupByResourceConstraints` uses two kinds of grouping. The first, *combinatorial grouping*, tries all combinations of assignments over a few consecutive days, building a group when just one of those combinations has zero cost, according to the cluster busy times and limit busy times constraints that monitor those days. The second, *profile grouping*, uses limit active intervals constraints to find different kinds of groups. All this is explained below.

`KheGroupByResourceConstraints` consults option `rs_invariant`, and also

`rs_group_by_rc_off`
A Boolean option which, when `true`, turns grouping by resource constraints off.

`rs_group_by_rc_max_days`
An integer option which determines the maximum number of consecutive days (in fact, time groups of the common frame) examined by combinatorial grouping (Section 11.10.2). Values 0 or 1 turn combinatorial grouping off. The default value is 3.

`rs_group_by_rc_combinatorial_off`
A Boolean option which, when `true`, turns combinatorial grouping off.

`rs_group_by_rc_profile_off`
A Boolean option which, when `true`, turns profile grouping off.

It also calls `KheFrameOption` (Section 5.10) to obtain the common frame.

The following subsections describe the algorithms used behind the scenes for TGRC. There are many details; some have been omitted. The last subsections document the interface used by the TGRC modules to communicate with each other, as found in header file `khe_sr_tgrc.h`.

### 11.10.1. Combinatorial grouping

Suppose that there are two kinds of shifts, day and night; that each nurse must be busy on both days of the weekend or neither; and that nurses cannot work a day shift on the day after a night shift. Then nurses assigned to the Saturday night shift must work on Sunday, and so must work the Sunday night shift. So it makes sense to group one Saturday night shift with one Sunday night shift, and to do so repeatedly until night shifts run out on one of those days.

Suppose that the groups just made consume all the Sunday night shifts. Then nurses working the Saturday day shifts cannot work the Sunday night shifts, because the Sunday night shifts are grouped with Saturday night shifts now, which clash with the Saturday day shifts. So now it is safe to group one Saturday day shift with one Sunday day shift, and to do so repeatedly until day shifts run out on one of those days.

Groups made in this way can be a big help to solvers. In instance `COI-GPost.xml`, for example, each Friday night task can be grouped with tasks for the next two nights. Good solutions always assign these three tasks to the same resource, owing to constraints specifying that the weekend following a Friday night shift must be busy, that each weekend must be either free on both days or busy on both, and that a night shift must not be followed by a day shift.

*Combinatorial grouping* realizes these ideas. It enumerates a space whose elements are sets of mtasks (Section 11.9.1). The space is defined by *requirements* supplied by the caller. As explained in Section 11.10.6, the requirements could state that the sets must cover a given time group or mtask, or must not cover a given time group or mtask, and so on. For each set of mtasks $S$ in the search space, it calculates a cost $c(S)$, by evaluating the resource constraints that apply to one resource in the part of the cycle covered by $S$, and selects a set $S'$ such that $c(S')$ is minimal, or zero. It then makes one group by selecting one task from each mtask of $S'$ and grouping those tasks, and then repeating that until as many tasks as possible or desired have been grouped.

As formulated here, combinatorial grouping is a low-level algorithm which finds and groups one set of mtasks $S'$. It is called on by higher-level algorithms to do their actual grouping. For example, a higher-level algorithm might try combinatorial grouping at various points through the cycle, or even try it repeatedly at the same points, as in the example above, where grouping the Saturday and Sunday night shifts would be one application of combinatorial grouping, then grouping the Saturday and Sunday day shifts would be another.

The number of sets of mtasks tried by combinatorial grouping will usually be exponential in the number of days involved in the search. So the number of days has to be small, unless the choices on each day are very limited.

### 11.10.2. Using combinatorial grouping with combination reduction

This section describes one way in which the general idea of combinatorial grouping, as just presented, is applied by TGRC.

Let $m$ be the value of the `rs_group_by_rc_max_days` option described earlier. Iterate over all pairs $(f, t)$, where $f$ is a subset of the common frame containing $k$ adjacent time groups, for all $k$ such that $2 \leq k \leq m$, and $t$ is an mtask that covers $f$'s first or last time group.

For each $(f, t)$ pair, run combinatorial grouping, set up to require that $t$ be covered and that each of the $k$ time groups of $f$ be free to be either covered or not, and only doing grouping when there is a unique zero-cost grouping satisfying these requirements.

If $f$ has $k$ time groups, each with $n$ mtasks, say, there are up to $(n + 1)^{k-1}$ combinations for each run, so `rs_group_by_rc_max_days` must be small, say 3, or 4 at most. In any case, unique zero-cost groupings typically concern weekends, so larger values are unlikely to yield anything.

If one $(f, t)$ pair produces some grouping, then return to the first pair containing $f$. This handles cases like the one described earlier, where a grouping of Saturday and Sunday night shifts opens the way to a grouping of Saturday and Sunday day shifts.

The remainder of this section describes *combination reduction.* This is a refinement that TGRC uses to make unique zero-cost combinations more likely in some cases.

Some combinations examined by combinatorial grouping may have zero cost as far as the monitors used to evaluate it are concerned, but have non-zero cost when evaluated in a different way, involving the overall supply of and demand for resources. Such combinations can be ruled out, leaving fewer zero-cost combinations, and potentially more task grouping.

For example, suppose there is a maximum limit on the number of weekends each resource can work. If this limit is tight enough, it will force every resource to work complete weekends, even without an explicit constraint, if that is the only way that the available supply of resources can cover the demand for weekend shifts. This example fits the pattern to be given now, setting $C$ to the constraint that limits the number of busy weekends, $T$ to the times of all weekends, $T_i$ to the times of the $i$th weekend, and $f_i$ to the number of days in the $i$th weekend.

Take any any set of times $T$. Let $S(T)$, the *supply during* $T$, be the sum over all resources $r$ of the maximum number of times that $r$ can be busy during $T$ without incurring a cost. Let $D(T)$, the *demand during* $T$, be the sum over all tasks $x$ for which non-assignment would incur a cost, of the number of times $x$ is running during $T$. Then $S(T) \geq D(T)$ or else a cost is unavoidable.

In particular, take any cluster busy times constraint $C$ which applies to all resources, has time groups which are all positive, and has a non-trivial maximum limit $M$. (The analysis also applies when the time groups are all negative and there is a non-trivial minimum limit, setting $M$

to the number of time groups minus the minimum limit.) Suppose there are $n$ time groups $T_i$, for $1 \leq i \leq n$, and let their union be $T$.

Let $f_i$ be the number of time groups from the common frame with a non-empty intersection with $T_i$. This is the maximum number of times from $T_i$ during which any one resource can be busy without incurring a cost, since a resource can be busy for at most one time in each time group of the common frame.

Let $F$ be the sum of the largest $M$ $f_i$ values. This is the maximum number of times from $T$ that any one resource can be busy without incurring a cost: if it is busy for more times than this, it must either be busy for more than $f_i$ times in some $T_i$, or else it must be busy for more than $M$ time groups, violating the constraint's maximum limit.

If there are $R$ resources altogether, then the supply during $T$ is bounded by

$$S(T) \leq RF$$

since $C$ is assumed to apply to every resource.

As explained above, to avoid cost the demand must not exceed the supply, so

$$D(T) \leq S(T) \leq RF$$

Furthermore, if $D(T) \geq RF$, then any failure to maximize the use of workload will incur a cost. That is, every resource which is busy during $T_i$ must be busy for the full $f_i$ times in $T_i$.

So the effect on grouping is this: if $D(T) \geq RF$, a resource that is busy in one time group of the common frame that overlaps $T_i$ should be busy in every time group of the common frame that overlaps $T_i$. TGRC searches for constraints $C$ that have this effect, and informs its combinatorial grouping solver about what it found by changing the requirements for some time groups from 'a group is free to cover this time group, or not' to 'a group must cover this time group if and only if it covers the previous time group'. When searching for groups, the option of covering some of these time groups but not others is removed. With fewer options, there is more chance that some combination might be the only one with zero cost, allowing more task grouping.

Instance `CQ14-05` has two constraints that limit busy weekends. One applies to 10 resources and has maximum limit 2; the other applies to the remaining 6 resources and has maximum limit 3. So combination reduction actually takes sets of constraints with the same time groups that together cover every resource once. Instead of $RF$ (above), it uses the sum over the set's constraints $c_j$ of $R_jF_j$, where $R_j$ is the number of resources that $c_j$ applies to, and $F_j$ is the sum of the largest $M_j$ of the $f_i$ values, where $M_j$ is the maximum limit of $c_j$. The $f_i$ are the same for all $c_j$.

## 11.10.3. Profile grouping

Suppose 6 nurses are required on the Monday, Tuesday, Wednesday, Thursday, and Friday night shifts, but only 4 are required on the Saturday and Sunday night shifts. Consider any division of the night shifts into sequences of one or more shifts on consecutive days. However these sequences are made, at least two must begin on Monday, and at least two must end on Friday.

Now suppose that the intention is to assign the same resource to each shift of any one sequence, and that a limit active intervals constraint, applicable to all resources, specifies that night shifts on consecutive days must occur in sequences of at least 2 and at most 3. Then the two sequences of night shifts that must begin on Monday must contain a Monday night and a Tuesday night shift at least, and the two that end on Friday must contain a Thursday night and a Friday night shift at least. So here are two groupings, of Monday and Tuesday nights and of Thursday and Friday nights, for each of which we can build two task groups.

Suppose that we already have a task group which contains a sequence of 3 night shifts on consecutive days. This group cannot be grouped with any night shifts on days adjacent to the days it currently covers. So for present purposes the tasks of this group can be ignored. This can change the number of night shifts running on each day, and so change the amount of grouping. For example, in instance `COI-GPost.xml`, all the Friday, Saturday, and Sunday night shifts get grouped into sequences of 3, and 3 is the maximum, so those night shifts can be ignored here, and so every Monday night shift begins a sequence, and every Thursday night shift ends one.

We now generalize this example, ignoring for the moment a few issues of detail. Let $C$ be any limit active intervals constraint which applies to all resources, and whose time groups $T_1, \ldots, T_k$ are all positive. Let $C$'s limits be $C_{max}$ and $C_{min}$, and suppose $C_{min}$ is at least 2 (if not, there can be no grouping based on $C$). What follows is relative to $C$, and is repeated for each such constraint. Constraints with the same time groups are notionally merged, allowing the minimum limit to come from one constraint and the maximum limit from another.

Let $n_i$ be the number of tasks of interest that cover $T_i$. The $n_i$ make up the *profile* of $C$.

A *long task* is a task which covers at least $C_{max}$ adjacent time groups from $C$. Long tasks can have no influence on grouping to satisfy $C$'s minimum limit, so they may be ignored, that is, profile grouping may run as though they are not there. This applies both to tasks which are present at the start, and tasks which are constructed along the way. As profile grouping proceeds, some tasks become grouped into larger tasks which are no longer relevant because they are long. This causes some of the $n_i$ values to decrease. We always base our decisions on the current profile, not the original profile.

For each $i$ such that $n_{i-1} < n_i$, $n_i - n_{i-1}$ groups of length at least $C_{min}$ must start at $T_i$ (more precisely, they must cover $T_i$ but not $T_{i-1}$). They may be constructed by combinatorial grouping, passing in time groups $T_i, \ldots, T_{i+C_{min}-1}$ with cover type 'yes', and $T_{i-1}$ and $T_{i+C_{max}}$ with cover type 'no', asking for $m = n_i - n_{i-1} - c_i$ tasks, where $c_i$ is the number of existing tasks (not including long ones) that satisfy these conditions already. The new groups must group at least 2 tasks each. Some of the time groups may not exist; in that case, omit them, but still do the grouping if there are at least 2 'yes' time groups. The case for sequences ending at $j$ is symmetrical.

If $C$ has no history, we set $n_0$ and $n_{k+1}$ to 0, encouraging groups to begin at $T_1$ and end at $T_k$. If $C$ has history, we still set $n_0$ to 0, reasoning that assign by history (Section 12.4.2) has taken care of history at that end; but we set $n_{k+1}$ to $\infty$, preventing groups from being formed to end at $T_k$.

Groups made by one round of profile grouping may participate in later rounds. Suppose $C_{min} = 2$, $C_{max} = 3$, $n_1 = n_5 = 0$, and $n_2 = n_3 = n_4 = 4$. Profile grouping builds 4 groups of length 2 begining at $T_2$, then 4 groups of length 3 ending at $T_4$, incorporating the length 2 groups.

We turn now to some issues of detail.

**Singles.** A *single* is a set of mtasks that satisfies the requirements of combinatorial grouping but contains only one mtask. We need to consider how singles affect profile grouping. Singles of length $C_{max}$ or more are ignored, but there may be singles of smaller length.

The $n_i - n_{i-1}$ groups that must start at $T_i$ include singles. Singles are already present, just as though they were made first. The combinatorial grouping solver has a variant that applies the given requirements, but instead of doing any grouping, returns $c_i$, the number of tasks of interest that lie in the mtasks of singles. Then we ask combinatorial grouping to make up to $n_i - n_{i-1} - c_i$ groups, not $n_i - n_{i-1}$, with an extra requirement that singles are to be exluded. If $n_i - n_{i-1} - c_i \leq 0$ we skip the call; the sequences that need to start at $T_i$ are already present.

**Varying task domains.** Suppose that one senior nurse is wanted each night, four ordinary nurses are wanted each week night, and two ordinary nurses are wanted each weekend night. Then two groups still need to start on Monday nights, but they should group demands for ordinary nurses, not senior nurses. Nevertheless, tasks with different domains are not totally unrelated. A senior nurse could very well act as an ordinary nurse on some shifts.

We still aim to build $M = n_i - n_{i-1} - c_i$ groups as before. However, we do this by making several calls on combinatorial grouping. For each resource group $g$ appearing as a domain in any mtask running at time $T_i$, find $n_{gi}$, the number of tasks (not including long ones) with domain $g$ running at $T_i$, and $n_{g(i-1)}$, the number at $T_{i-1}$. For each $g$ such that $n_{gi} > n_{g(i-1)}$, call combinatorial grouping, with a requirement expressing a preference for domain $g$, and asking for $\min(M, n_{gi} - n_{g(i-1)})$ groups. Then subtract from $M$ the number of groups actually made. Stop when $M = 0$ or the list of domains is exhausted.

**Varying task costs.** The tasks participating in profile grouping might well differ in their non-assignment cost. It feels wrong to group tasks with very different costs. Although this is not currently prevented, it is likely to be fairly harmless, for two reasons.

First, in grouping generally we only consider tasks which need assignment—tasks whose cost of non-assignment exceeds their cost of assignment. So we won't be grouping a task that needs assignment with a task that doesn't.

Second, the most cost-reducing tasks in each mtask are assigned first. That should encourage task groups to contain tasks of similar cost.

**Non-uniqueness of zero-cost groupings.** The main problem with profile grouping is that there may be several zero-cost groupings in a given situation. For example, a profile might show that a group covering Monday, Tuesday, and Wednesday may be made, but give no guidance on which shifts on those days to group.

There are various ways to deal with this problem. At present we are limiting profile grouping to constraints $C$ whose time groups all contain a single time. Thus profile grouping will group sequences of day shifts, sequences of night shifts, and so on, but it will not group sequences of days, even when there is a constraint limiting the number of consecutive busy days whose profile shows that sequences must begin on a certain day. An exception to this is the case $C_{min} = C_{max}$, discussed below.

**An overall algorithm.** We are now in a position to present an overall algorithm for profile grouping. Find all limit active intervals constraints $C$ which apply to all resources and whose time groups are all singletons and all positive. Notionally merge constraints that share the same time groups; for example, we could take $C_{min}$ from one and $C_{max}$ from another. For each of these merged constraints $C$ such that $C_{min} \geq 2$, proceed as follows.

Traverse the profile repeatedly, looking for cases where $n_i > n_{i-1}$ and $n_j < n_{j+1}$, and use combinatorial grouping (aiming to find zero-cost groups, not unique zero-cost groups) to build groups which cover between $C_{min}$ and $C_{max}$ time groups starting at $T_i$ (or ending at $T_j$). Continue traversing the profile until until no points which allow grouping can be found.

As groups are made, the $n_i$ will often decrease. At some point they might all be zero, or the $n_i - n_{i-1} - c_i$ might all be zero. Alternatively, they might all be non-zero but all equal, and we need to think about what to do then. Further grouping is possible but would involve arbitrary choices, making whether to go further a matter of experience and experiment.

One case where going further is worthwhile is when $C_{min} = C_{max}$. It is very constraining to insist, as this does, that every sequence of consecutive busy days (say) away from the start and end of the cycle must have a particular length. Indeed, it changes the problem into a combinatorial one of packing these rigid sequences into the profile. Local repairs cannot do this well, because to increase or decrease the length of one sequence, we must decrease or increase the length of a neighbouring sequence, and so on all the way back to the start or forward to the end of the cycle (unless there are shifts nearby which can be assigned or not without cost). So we turn to profile grouping to find suitable groups before assigning any resources. Some of these groups may be less than ideal, but still the overall effect should be better than no grouping at all.

Another case for going further is when $C_{min} + 1 = C_{max}$ and the time groups are singletons. This case arises in instance `INRC2-4-100-0-1108`, where night shifts preferably come in sequences of length 4 or 5. The author's other solvers struggle with this requirement, making it very tempting to build these sequences before doing any assignment.

If we do decide to keep going, one way to do that is as follows. From among all time groups $T_i$ where $n_i > 0$, choose one which has been the starting point for a minimal number of groups (to spread out the starting points as much as possible) and make a group there if combinatorial grouping allows it. Then return to traversing the profile repeatedly. There should now be an $n_i > n_{i-1}$ case just before the latest group, and an $n_j < n_{j+1}$ case just after it. Repeat until there is no $T_i$ where $n_i > 0$ and combinatorial grouping can build a group.

Another way to keep going is to use the dynamic programming algorithm from the next section. Although it is not globally optimum, it is an efficient way to find high-quality groups.

### 11.10.4. A dynamic programing algorithm for profile grouping

This section presents a dynamic programming algorithm for profile grouping which can be applied to any subsequence $[a, b]$ of the profile such that $n_i > 0$ for all $i$ in the range $a \le i \le b$, and $n_{a-1} = n_{b+1} = 0$. The algorithm reduces each $n_i$ in the range by one, using groups of minimum total cost. Applied repeatedly, it can produce many very good groups, although there is no suggestion that they are globally optimum.

We have one hard constraint and one soft constraint. The hard constraint is that we require the algorithm to produce a set of groups, each of length between $C_{min}$ and $C_{max}$ inclusive, such that every position in the range is covered by exactly one group. The last group, however, may have length less than $C_{min}$ when it is the last time group of $C$ and history (i.e. a future) is present, since short sequences at the end do not violate $C$ in that case. The soft constraint is that the total cost of the groups (as reported by combinatorial grouping) should be minimized.

One could ask whether there will be any cost: a sequence of night shifts (say) whose length satisfies $C$ is not likely to violate any other constraints. In practice this is largely true. The main exception is that complete weekend constraints may combine with unwanted pattern constraints to cause sequences that end on a Saturday or begin on a Sunday to have non-zero cost.

Our dynamic programming algorithm finds a solution $S(i)$ which is optimal among all solutions which cover the first $i$ time groups of $[a, b]$, for each $i$ such that $0 \leq i \leq b - a + 1$.

The first of these optimal solutions, $S(0)$, is required to cover no time groups, so it is the empty set of sequences, with cost 0. Assume inductively that we have found $S(k)$ for each $k$ such that $0 \leq k < i$. We need to find $S(i)$.

To do this, for each $j$ such that $C_{min} \leq j \leq C_{max}$, find the solution which consists of $S(i - j)$ plus a single sequence covering time groups $T_{i-j+1} \dots T_i$. The cost of this solution is the cost of $S(i - j)$ plus the cost of the additional sequence, as reported by combinatorial grouping, tasked with finding a sequence of minimum cost covering $T_{i-j+1} \dots T_i$ but not $T_{i-j}$ and not $T_{i+1}$. Find the solution of minimum cost over all $j$ and declare that to be $S(i)$.

As explained above, the last group may have length less than $C_{min}$ when history is present. In that case, we allow the last sequence to have any length $j$ such that $1 \leq j \leq C_{max}$.

The main problem with this algorithm is that there may be no $S(i)$ at all. For example, $S(1)$ does not exist because there are no legal sequences of length 1; legal sequences start only with $S(C_{min})$. Even after that, there may be gaps. For example, if every sequence must have length 4 or 5, there is no $S(6)$ or $S(7)$. There is also the possibility that sequences of the right lengths might exist but combinatorial grouping finds no way to group their tasks, even though we ask it only for sequences of minimum, not necessarily zero, cost. We treat missing solutions of this kind as though they had cost $\infty$. We also do this when we need an $S(i - j)$ but $i - j < 0$.

Another problem is that if $C_{max}$ is relatively large, combinatorial grouping could be too slow. This has not been a problem in practice, but it is probably safest to limit dynamic programming to cases where either the time groups each contain a single time, or else $C_{max} \leq 4$.

Normally, we remove a sequence from the profile only when it has length $C_{max}$, because only then is it unable to participate in further grouping. However, after one round of dynamic programming we remove every sequence in the optimal solution from the profile, reasoning that collectively they are finished and should not participate further. We can repeat this, reducing each $n_i$ by one on each round, until some $n_i = 0$ or the round fails to find a solution.

Although the dynamic programming algorithm finds an optimal way to reduce each $n_i$ by one, the *general profile grouping problem*, which is to find an optimal way to fill an arbitrary profile with minimum-cost sequences of length between $C_{min}$ and $C_{max}$, remains unsolved. Even when the $n_i$ are equal there is no proof that a sequence of rounds, each of which finds an optimal way to reduce them all by one, is guaranteed to find an optimal solution overall. (It is true that an optimal solution in this case can be divided into a sequence of rounds, each of which reduces all the $n_i$ by one, but that does not prove that our sequence of rounds is optimal.) When arbitrary task domains are added, it is easy to see that the problem includes the NP-hard multi-dimensional matching problem. However, task domains do not seem to be a problem in practice.

The author has considered using dynamic programming for the general profile grouping problem, inspired by the dynamic programming algorithm for optimal resource assignment (Section 12.6). Such an algorithm seems to be possible, but it would be complicated, especially since it would need to take into account any task grouping that has already occurred. The optimal resource assignment algorithm treats grouped tasks heuristically; that would not suffice here.

### 11.10.5. Implementation notes 1: mtask groups

File `khe_sr_tgrc.h` contains the interfaces that the TGRC source files use to communicate with each other. It declares a type `KHE_MTASK_GROUP` representing one *mtask group*. This is an mtask set with additional features relevant to grouping: it contains an mtask set, plus it keeps track of which mtask will be the leader mtask, and the cost to a resource of assigning it to the group.

For creating and deleting an mtask group object there are

```
KHE_MTASK_GROUP KheMTaskGroupMake(KHE_COMB_GROUPER cg);
void KheMTaskGroupDelete(KHE_MTASK_GROUP mg);
```

Here `KHE_COMB_GROUPER` is another type defined in `khe_sr_tgrc.h`. It is mainly concerned with running combinatorial grouping, but it also holds a free list of mtask group objects.

There are operations for clearing an mtask group object and overwriting its contents with the contents of another mtask group object:

```
void KheMTaskGroupClear(KHE_MTASK_GROUP mg);
void KheMTaskGroupOverwrite(KHE_MTASK_GROUP dst_mg,
  KHE_MTASK_GROUP src_mg);
```

For visiting its mtasks there are

```
int KheMTaskGroupMTaskCount(KHE_MTASK_GROUP mg);
KHE_MTASK KheMTaskGroupMTask(KHE_MTASK_GROUP mg, int i);
```

as usual, along with

```
bool KheMTaskGroupIsEmpty(KHE_MTASK_GROUP mg);
```

which is the same as testing whether the count is 0. For adding and deleting mtasks there are

```
bool KheMTaskGroupAddMTask(KHE_MTASK_GROUP mg, KHE_MTASK mt);
void KheMTaskGroupDeleteMTask(KHE_MTASK_GROUP mg, KHE_MTASK mt);
```

`KheMTaskGroupAddMTask` adds `mt` to `mg` and returns `true`, or if the addition cannot be carried out (because `mt` runs on the same day as one of the mtasks that is already present, or because no leader mtask can be found that suits both the existing mtasks and `mt`), it changes nothing and returns `false`. `KheMTaskGroupDeleteMTask` deletes `mt` from `mg`. Owing to issues around calculating leader mtasks, `mt` must be the most recently added but not deleted mtask, otherwise `KheMTaskGroupDeleteMTask` will abort. Function

```
bool KheMTaskGroupContainsMTask(KHE_MTASK_GROUP mg, KHE_MTASK mt);
```

returns `true` when `mg` contains `mt`.

An mtask group `mg` has a cost, which is the cost of the resource monitors of some resource `r` when `r` is assigned to one task from each mtask of `mg`. Not all monitors are included, only cluster busy times and limit busy times monitors whose monitoring is limited to the days during which the mtasks of `mg` are running, plus one extra day on each side. (We do not want wider issues, such as global workload limits, to influence this cost.) The mtask group module is responsible for finding a suitable resource, making the assignments, measuring the cost, and taking the assignments away again, all of which is done by

```
bool KheMTaskGroupHasCost(KHE_MTASK_GROUP mg, KHE_COST *cost);
```

If a cost can be calculated, `KheMTaskGroupHasCost` sets `*cost` to its value and returns `true`. If a cost cannot be calculated, because `mg` is empty, or a suitable resource `r` cannot be found, or cannot be assigned to every mtask of `mg` (none of these conditions is likely to occur in practice), then `false` is returned. There is also

```
bool KheMTaskGroupIsBetter(KHE_MTASK_GROUP new_mg,
  KHE_MTASK_GROUP old_mg);
```

which returns `true` when `old_mg` is empty or else both `new_mg` and `old_mg` have a cost, and the cost of `new_mg` is smaller than the cost of `old_mg`.

Calculating the cost is slow, so mtask group objects cache the most recently calculated cost, and only recalculate it when the set of mtasks has changed since it was last calculated.

To actually carry out some grouping, the function is

```
int KheMTaskGroupExecute(KHE_MTASK_GROUP mg, int max_num,
  KHE_SOLN_ADJUSTER sa, KHE_SOLN_ADJUSTER fix_leaders_sa, char *debug_str);
```

By making calls to functions `KheMTaskFinderGroupBegin`, `KheMTaskFinderGroupAddTask`, and `KheMTaskFinderGroupEnd` (Section 11.9.3), it makes up to `max_num` groups from the mtasks of `mg`. It returns the number of groups actually made. If `sa != NULL` the task assignments made are recorded in `sa` so that they can be undone later. If `fix_leaders_sa != NULL`, the `NULL` assignments in the leader tasks of the groups are fixed and stored in `fix_leaders_sa` so that they can be undone later. The point of this is that fixing their assignments removes them from the profile, which is what is wanted when finding groups using dynamic programming. Parameter `debug_str` is used for debugging only, and should contain some indication of how the group came to be formed: `"combinatorial grouping"`, `"profile grouping"`, or whatever.

Finally,

```
void KheMTaskGroupDebug(KHE_MTASK_GROUP mg,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of `mg` onto `fp` with the given verbosity and indent. This includes the cost, if currently known, and it highlights the leader mtask.

### 11.10.6. Implementation notes 2: the combinatorial grouper

Combinatorial grouping is a low-level solve algorithm that provides services to higher-level grouping solvers. It allows those solvers to load a variety of different requirements, and it then will search for groups that satisfy those requirements.

This is done by a *combinatorial grouper* object, made like this:

```
KHE_COMB_GROUPER KheCombGrouperMake(KHE_MTASK_FINDER mtf,
  KHE_RESOURCE_TYPE rt, HA_ARENA a);
```

It finds groups of `mtf`'s mtasks of type `rt`, using memory from arena `a`. There is no `Delete` operation; the grouper is deleted when `a` is freed. It calls `KheMTaskGroupExecute` from Section 11.10.5 to actually make its groups, and this updates `mtf`'s mtasks, so that `mtf` does not go out of date as grouping proceeds. Functions

```
KHE_MTASK_FINDER KheCombGrouperMTaskFinder(KHE_COMB_GROUPER cg);
KHE_SOLN KheCombGrouperSoln(KHE_COMB_GROUPER cg);
KHE_RESOURCE_TYPE KheCombGrouperResourceType(KHE_COMB_GROUPER cg);
HA_ARENA KheCombGrouperArena(KHE_COMB_GROUPER cg);
```

return various attributes of `cg`; the solution comes from `mtf`.

The resource type passed to `KheCombGrouperMake` must be non-`NULL`, and it must be one of the resource types handled by `mtf`. An mtask finder is able to handle either one resource type or all resource types, but a comb grouper can only handle one resource type.

Incidentally to its other functions, a `KHE_COMB_GROUPER` object holds a free list of mtask group objects. Functions

```
KHE_MTASK_GROUP KheCombGrouperGetMTaskGroup(KHE_COMB_GROUPER cg);
void KheCombGrouperPutMTaskGroup(KHE_COMB_GROUPER cg,
  KHE_MTASK_GROUP mg);
```

get an object from this list (returning `NULL` if the list is empty) and put an object onto the list.

A `KHE_COMB_GROUPER` object can solve any number of combinatorial grouping problems for a given `mtf`, one after another. The user loads the grouper with one problem's requirements, then requests a solve, then loads another lot of requirements and solves, and so on.

We'll present the functions which load requirements informally now. Precise descriptions of what each requirement does are given at the end of this section. These requirements make a rather eclectic bunch. They are all needed, however, to support the various kinds of grouping.

It is usually best to start the process of loading requirements by calling

```
void KheCombGrouperClearRequirements(KHE_COMB_GROUPER cg);
```

This clears away any old requirements.

A key requirement for most solves is that the groups it makes should cover a given time group. Any number of such requirements can be added and removed by calling

```
void KheCombGrouperAddTimeGroupRequirement(KHE_COMB_GROUPER cg,
  KHE_TIME_GROUP tg, KHE_COMB_COVER_TYPE cover);
void KheCombGrouperDeleteTimeGroupRequirement(KHE_COMB_GROUPER cg,
  KHE_TIME_GROUP tg);
```

any number of times. `KheCombSolverAddTimeGroup` specifies that the groups must cover `tg` in a manner given by the `cover` parameter, whose type is

```
typedef enum {
  KHE_COMB_COVER_YES,
  KHE_COMB_COVER_NO,
  KHE_COMB_COVER_PREV,
  KHE_COMB_COVER_FREE
} KHE_COMB_COVER_TYPE;
```

We'll explain this fully later, but just briefly, `KHE_COMB_COVER_YES` means that we are only interested in sets of mtasks that cover the time group, `KHE_COMB_COVER_NO` means that we are not interested in sets of mtasks that cover the time group, and so on.

`KheCombGrouperDeleteTimeGroupRequirement` undoes a previous call to `KheCombGrouperAddTimeGroupRequirement` with the same time group. If there has been no such call, `KheCombGrouperDeleteTimeGroupRequirement` aborts.

Any number of requirements that the groups should cover a given mtask may be added:

```
void KheCombGrouperAddMTaskRequirement(KHE_COMB_GROUPER cg,
  KHE_MTASK mt, KHE_COMB_COVER_TYPE cover);
void KheCombGrouperDeleteMTaskRequirement(KHE_COMB_GROUPER cg,
  KHE_MTASK mt);
```

These work in the same way as for time groups. Care is needed because `mt` may be rendered undefined, if groups are made that leave `mt` empty afterwards. The safest option after a solve whose requirements include an mtask is to call `KheCombGrouperClearRequirements`.

Next we have

```
void KheCombSolverAddNoSinglesRequirement(KHE_COMB_SOLVER cs);
void KheCombSolverDeleteNoSinglesRequirement(KHE_COMB_SOLVER cs);
```

This is concerned with whether mtask sets that contain a single mtask are acceptable—an awkward question, as we'll see. And

```
void KheCombGrouperAddPreferredDomainRequirement(KHE_COMB_GROUPER cg,
  KHE_RESOURCE_GROUP rg);
void KheCombGrouperDeletePreferredDomainRequirement(KHE_COMB_GROUPER cg);
```

specifies that mtasks whose domains resemble `rg` are preferred. We'll return to all these requirements later.

There is no need to reload requirements between solves. Requirements stay in effect until they are either deleted individually or cleared out by `KheCombGrouperClearRequirements`.

After all the requirements are added, an actual solve is carried out by calling

```
int KheCombGrouperSolve(KHE_COMB_GROUPER cg, int max_num,
  KHE_COMB_VARIANT_TYPE cg_variant, KHE_SOLN_ADJUSTER sa,
  char *debug_str);
```

`KheCombGrouperSolve` searches the space of all sets of mtasks $S$ that satisfy the requirements passed in by the user, and selects one set $S'$ of minimal cost $c(S')$. Using $S'$, it makes as many groups as it can, up to `max_num` groups, and returns the number it actually made, between `0` and `max_num`. If $S'$ contains a single mtask, no groups are made and the value returned is 0.

`KheCombGrouperSolve` offers several variants of the algorithm just described, selected by parameter `cg_variant`, which we'll describe later. If parameter `sa` is non-`NULL`, any task assignments made by `KheCombGrouperSolve` are stored in `sa`, so that they can be undone later. Parameter `debug_str` is used only by debug code, to say how the grouping came about. It might be `"combinatorial grouping"` or `"profile grouping"`, for example.

One variant of `KheCombGrouperSolve` is different and has been given its own interface:

```
int KheCombGrouperSolveSingles(KHE_COMB_GROUPER cg);
```

It makes no groups. Instead, it counts the number of tasks needing assignment that lie in mtasks which satisfy the requirements by themselves (not grouped with any other mtasks). These are the tasks we called singles above.

Our tour of the interface of `KHE_COMB_GROUPER` ends with function

```
void KheCombGrouperDebug(KHE_COMB_GROUPER cg, int verbosity,
  int indent, FILE *fp);
```

This produces the usual debug print of `cg` onto `fp` with the given verbosity and indent.

The rest of this section is devoted to a precise description of `KheCombGrouperSolve`. There are three things to do here. First, we need to specify how the search space of mtask sets is determined. Second, for each mtask set $S$ in the search space we need to define a cost $c(S)$. And third, we need to explain the algorithm variants selected by `cg_variant`.

For the search space we need some definitions. A task *covers* a time if it, or a task assigned to it directly or indirectly, runs at that time (and possibly at other times). A task covers a time group if it covers one or more of the time group's times. An mtask covers a time or time group if its tasks do (they run at the same times). An mtask covers an mtask if it is that mtask. An mtask covers a time group or mtask requirement if it covers that requirement's time group or mtask.

A set of mtasks $S$ lies in the search space if it satisfies all of the following conditions. The letters in parentheses at the end of each condition will be explained afterwards.

1. Each mtask in $S$ covers at least one time group or mtask requirement whose `cover` is not `KHE_COMB_COVER_NO`. This condition allows for a generate-and-test approach to building the search space: find the set $X$ of all mtasks that satisfy this condition, then use the usual recursive algorithm to generate all subsets $S$ of $X$, then test each $S$ against each of the following conditions. (a)

2. For each `mt` in $S$, `mt` does not cover any time group or mtask requirement whose `cover` is `KHE_COMB_COVER_NO`. (a)

3. For each `mt` in $S$, `mt` contains at least one task which not fixed, not assigned, and for which non-assignment has a cost. That is, `KheMTaskAssignIsFixed(mf)` must be `false` and `KheMTaskNeedsAssignment(mt)` must be `true`. Only tasks with these properties participate in grouping, as discussed above. (a)

4. For each pair of distinct mtasks `mt1` and `mt2` in *S*, `KheMTaskInterval(mt1)` and `KheMTaskInterval(mt2)` are disjoint. We intend to assign some resource to one task from each mtask of *S*, so no two of those tasks can run on the same day. (b)

5. If *S* is non-empty then it contains a *leader mtask*, that is, an mtask containing tasks that can serve as leader tasks for the tasks in the other mtasks of *S*. This rules out sets *S* whose mtasks have incompatible domains. (b)

6. If `cg_variant == KHE_COMB_VARIANT_SINGLES`, then *S* contains at most one mtask. We say more about this below. (b)

7. If `KheCombGrouperAddNoSinglesRequirement` was called, then *S* contains at least two mtasks. Otherwise *S* contains at least one mtask. (c)

8. Each time group or mtask requirement *C* must be satisfied. What this means depends on the value of *C*'s `cover` parameter, as follows:

   `KHE_COMB_COVER_YES`
   At least one of the mtasks of *S* covers *C*'s time group or mtask.

   `KHE_COMB_COVER_NO`
   None of the mtasks of *S* cover *C*'s time group or mtask.

   `KHE_COMB_COVER_PREV`
   This is interpreted like `KHE_COMB_COVER_YES` if the preceding time group or mtask requirement is covered, and like `KHE_COMB_COVER_NO` if the preceding time group or mtask requirement is not covered.

   `KHE_COMB_COVER_FREE`
   *C* is free to be covered by *S*'s mtasks, or not.

   If the first time group or mtask has cover `KHE_COMB_COVER_PREV`, this is treated like `KHE_COMB_COVER_FREE`. (c)

Time groups and mtasks not mentioned in any requirement may be covered, or not. The difference between this and a time group or mtask with cover `KHE_COMB_COVER_FREE` is that the mtasks that cover a free time group or mtask may be included in the search space.

We have so far given the impression that `KheCombGrouperSolve` generates all subsets *S* of the set *X* defined in condition (1) above, and then tests each *S* against these conditions. In fact, it does better. The letter at the end of each condition says when that condition is evaluated:

(a) This condition is evaluated just once for each mtask `mt`, at the start of the solve. If it does not hold, then `mt` is omitted from the set *X* of mtasks that we find all subsets of.

(b) When some set *S* does not satisfy this condition, every superset of *S* also does not satisfy it. So it is evaluated each time we add an mtask to *S* when generating all subsets. If it fails, that path of the recursive generation of all subsets is truncated immediately.

(c) This condition is (and can only be) evaluated when a complete subset has been generated.

In addition, for each mtask `mt` a list is kept of all time group and mtask requirements *C* with cover `KHE_COMB_COVER_YES` for which `mt` is the last mtask that covers *C*. Before trying the branch of the recursion that omits `mt`, the list is traversed and if there are any requirements in it that are not yet covered, that branch is not taken.

There is no prohibition on passing in a Yes cover requirement for an mtask which cannot be part of any *S* because it fails to satisfy one of the (a) conditions. For example, we could require the solve to cover an mtask whose tasks were all assigned. This condition is impossible to satisfy, so the result will be that `KheCombGrouperSolve` finds no groups and returns 0.

We said above that the first step is to build the set *X* of all mtasks that satisfy the first condition. Before doing anything further, this set is sorted so that mtasks whose first busy day is earlier come before mtasks whose first busy day is later. If there is a preferred domain (if `KheCombGrouperAddPreferredDomainRequirement` was called), then as a second priority, mtasks whose domain is a superset of the preferred domain come before mtasks whose domain is not a superset of the preferred domain, and as a third priority, mtasks whose domain is smaller come before mtasks whose domain is larger. This ensures that mtasks with preferred domains are tried first, which means that sets of mtasks with preferred domains are tested first, making them more likely to be chosen, but without actually ruling out any set of mtasks.

The second thing we need to do is to explain how the cost $c(S)$ of each set of mtasks *S* is defined. By the conditions above, *S* is non-empty and contains a leader mtask.

Let *I* be the smallest interval of days such that all the mtasks in *X*, as defined by conditions (1) and (a) above, run entirely within those days, plus (for safety) one extra day on each side. This is the grouper's idea of the part of the cycle affected by the current solve. Take the leader mtask of *S* and search its domain (as returned by `KheMTaskDomain`) for a resource *r* which is free and available throughout *I*. Most resources are free during grouping, and most resources are available (not subject to avoid unavailable times constraints) most of the time, so *r* should be easy to find; but if there is no such *r*, ignore *S*.

Assign *r* to each mtask of *S*. The cost $c(S)$ of *S* is determined while the assignments are in place. It is the total cost of all cluster busy times and limit busy times monitors which monitor *r* and have times lying entirely within the times of the days *I*. We limit ourselves to monitors within *I* because we don't want *r*'s global workload, for example, to influence the outcome. We add one day on each side so as not to miss monitors that prohibit certain local patterns, such as incomplete weekends. This is admittedly ad-hoc but it seems to work. After the cost is worked out, the assignments of *r* added to the mtasks of *S* are removed.

The third and last thing we need to do is to explain the `cg_variant` parameter. It has type

```
typedef enum {
  KHE_COMB_VARIANT_MIN,
  KHE_COMB_VARIANT_ZERO,
  KHE_COMB_VARIANT_SOLE_ZERO,
  KHE_COMB_VARIANT_SINGLES
} KHE_COMB_VARIANT_TYPE;
```

and allows the user to select one of four variants of the basic algorithm, as follows.

If `cg_variant` is `KHE_COMB_VARIANT_MIN`, then a subset $S'$ is chosen such that $c(S')$ is minimal among all $c(S)$, as described above. This will be possible as long as the search space contains at least one $S$ satisfying the conditions. If it doesn't, no groups are made.

If `cg_variant` is `KHE_COMB_VARIANT_ZERO` or `KHE_COMB_VARIANT_SOLE_ZERO`, then $c(S')$ must also be 0, and in the second case there must be no other $S$ satisfying the conditions such that $c(S)$ is 0. If these conditions are not met, no groups are made.

If `cg_variant` is `KHE_COMB_VARIANT_SINGLES`, the behaviour is different. No groups are made. Instead, `KheCombGrouperSolve` returns the number of individual, ungrouped tasks which satisfy the given requirements. (If the requirements include 'no singles', this will be 0.) This variant is accessed by calling `KheCombGrouperSolveSingles`, not `KheCombGrouperSolve`.

Let us call an mtask that satisfies the requirements without any grouping a *single*. Singles raise some awkward questions for combinatorial grouping. What to do about them seems to vary depending on why combinatorial grouping is being called, so instead of dealing with them in a fixed way, the grouper offers three features that help with them.

First, if the set of mtasks $S'$ with minimum or zero cost contains only one mtask, `KheCombSolverSolve` accepts it as best but makes no groups from it, returning 0 for the number of groups made. It is natural not to make any task assignments, because each of them is from a task from one mtask of $S'$ to a task from another mtask of $S'$, which is not possible when $S'$ contains only one mtask. But it is arguable that each unassigned task from that one mtask is a satisfactory group which should be reported. However, the value returned here is 0, as we said.

Second, by calling `KheCombSolverAddNoSinglesRequirement`, the user may declare that a set $S$ containing just one mtask should be excluded from the search space. But this is not a magical solution to the problem of singles. For example, when we need a unique zero-cost set of mtasks, we may want to include singles in the search space, to show that grouping is better than doing nothing. We need to think about the significance of singles in the current context.

Third, after setting up a problem, one can call `KheCombGrouperSolveSingles`. This searches the requested space, but, as we have seen, it does no grouping, instead returning the total number of tasks lying in singles. If our aim is to produce a certain number of groups, we can treat these singles as pre-existing groups, subtract their number from our target, and run again with 'no singles' on.

# Chapter 12. Resource Solvers

A *resource solver* assigns resources to tasks, or changes existing resource assignments. This chapter presents the resource solvers packaged with KHE.

## 12.1. Specification

The recommended interface for resource solvers, defined in `khe_solvers.h`, is

```
typedef bool (*KHE_TASKING_SOLVER)(KHE_TASKING tasking,
  KHE_OPTIONS options);
```

It assigns resources to some of the tasks of `tasking`, influenced by `options`, returning `true` if it changed, or at least usually changes, the solution. Taskings were defined in Section 5.5. The `options` parameter is as in Section 8.2; by convention, options consulted by resource solvers have names beginning with `rs_`.

A resource solver could focus on the initial *construction* of a resource assignment, or on the *repair* of an existing resource assignment. It is not wise, however, to try to classify solvers rigidly in this way, because some can be used for both. A construction solver can be converted into a repair solver by prefixing it with some unassignments, and a repair solver can be converted into a construction solver by including missing assignments among the defects that it is able to repair.

Except for preassignments, there is no reason to assign resources, at least in large numbers, before times are assigned. Accordingly, a resource solver may choose to assume that all meets have been assigned times. It may alter time assignments in its quest for resource assignments.

The usual way to convert preassignments in the instance into assignments in the solution is to call `KheTaskTreeMake` (Section 11.3); this is one of several routine jobs that it carries out. `KheTaskTreeMake` does not fix these assignments, although it does reduce the domains of the affected tasks to singletons. So other solvers should not be able to move preassigned tasks to other resources, but they can unassign them, which will produce errors if any preassigned tasks are unassigned when the solution is written.

A *split assignment* is an assignment of two or more distinct resources to the tasks monitored by an avoid split assignments monitor. A *partial assignment* is an assignment of resources to some of these same tasks, but not all. An assignment can be both split and partial.

## 12.2. The resource assignment invariant

If all tasks have duration 1, then the matching defines an assignment of resources to tasks which maximizes the number of assignments. Although larger durations are common, and maximizing the number of assignments is not the only objective, still it is clear from this fact that the matching deserves a central place in resource assignment.

Accordingly, the author's work in resource assignment [9] emphasizes algorithms that preserve the following condition, called the *resource assignment invariant*:

*The number of unmatchable demand tixels equals its initial value.*

Resource assignments are allowed only when the number of unmatchable demand tixels does not increase. This keeps the algorithms on a path that cannot lead to new violations of required avoid clashes constraints, avoid unavailable times constraints, limit busy times constraints, and limit workload constraints. In practice, most tasks can be assigned while preserving this invariant.

The Boolean option `rs_invariant` is used to tell resource solvers whether they should preserve the resource assignment invariant or not. In principle, every resource solver should consult and obey this option; in practice, many do but not all. A reasonable strategy is to preserve the invariant for most of the solve, but to relax it near the end, to allow as many assignments as possible to be made. Section 12.15 explains how to do this, or something else, as you prefer.

The invariant is not usually checked after each individual operation. Rather, a sequence of related operations is carried out, and then the number of unmatchable demand tixels at the end of the sequence is compared with the number at the start. If it has increased, the sequence of operations needs to be undone. Such sequences were called *atomic sequences* in Section 4.8, where the following code (using a mark object) was recommended for obtaining them:

```
mark = KheMarkBegin(soln);
success = SomeSequenceOfOperations(...);
KheMarkEnd(mark, !success);
```

When preserving the resource invariant, this needs to be changed to

```
mark = KheMarkBegin(soln);
init_count = KheSolnMatchingDefectCount(soln);
success = SomeSequenceOfOperations(...);
if( KheSolnMatchingDefectCount(soln) > init_count )
  success = false;
KheMarkEnd(mark, !success);
```

This works without the matching too, since then `KheSolnMatchingDefectCount` returns 0.

As a simple but effective aid to getting this right, this code is encapsulated in functions

```
void KheAtomicOperationBegin(KHE_SOLN soln, KHE_MARK *mark,
  int *init_count, bool resource_invariant);
bool KheAtomicOperationEnd(KHE_SOLN soln, KHE_MARK *mark,
  int *init_count, bool resource_invariant, bool success);
```

which may be placed before and after a sequence of operations, like this:

```
KheAtomicOperationBegin(soln, &mark, &init_count, resource_invariant);
success = SomeSequenceOfOperations(...);
KheAtomicOperationEnd(soln, &mark, &init_count, resource_invariant,
  success);
```

Here `mark` and `init_count` are variables of type `KHE_MARK` and `int`, not used for anything else, `resource_invariant` is `true` if the operations must preserve the resource invariant to be considered successful, and `success` is their diagnosis of their own success, not including checking the resource invariant. `KheAtomicOperationEnd` returns `true` if `success` is `true` and (if `resource_invariant` is `true`) the number of unmatchable demand tixels did not increase:

```
void KheAtomicOperationBegin(KHE_SOLN soln, KHE_MARK *mark,
  int *init_count, bool resource_invariant)
{
  *mark = KheMarkBegin(soln);
  *init_count = KheSolnMatchingDefectCount(soln);
}

bool KheAtomicOperationEnd(KHE_SOLN soln, KHE_MARK *mark,
  int *init_count, bool resource_invariant, bool success)
{
  if( resource_invariant &&
      KheSolnMatchingDefectCount(soln) > *init_count )
    success = false;
  KheMarkEnd(*mark, !success);
  return success;
}
```

The code is trivial, but useful because it encapsulates a common but slightly confusing pattern.

If the resource invariant is being enforced, there is no need to include the cost of demand monitors in the solution cost, since their cost cannot increase. They must continue to monitor the solution, however, so detaching is not appropriate. Function

```
void KheDisconnectAllDemandMonitors(KHE_SOLN soln, KHE_RESOURCE_TYPE rt);
```

disconnects all demand monitors (or all demand monitors which monitor entities of type `rt`, if `rt` is non-`NULL`) from all their parents, including the solution object if it is a parent. Thus, as required, they continue to monitor the solution, but the costs they compute are not added to the cost of any group monitor. `KheSolnMatchingDefectCount` still works, however, and there is nothing to prevent them from being made children of other group monitors later.

In general, time solvers cannot be expected to maintain the resource assignment invariant. There are cases where, whatever time is assigned, the number of unmatchable demand tixels must increase. Accordingly, the usual way to take the matching into account during time assignment is to add a hard cost of 1 to the solution cost for each unmatchable demand tixel.

Do-it-yourself solving offers both ways to use the global tixel matching. As documented in Section 8.4, item `gta` installs the global tixel matching in such a way that every unassigned demand node contributes hard cost 1 to the solution cost, as is usual during time assignment; while item `gtb` installs it so that it is available but without contributing to solution cost, as is usual during resource assignment.

## 12.3. Unchecked, checked, ejecting, and Kempe task and task set moves

The operation of assigning a resource to a task is fundamental to resource solving. This section defines four variants of this operation (unchecked, checked, ejecting, and Kempe), and presents functions for applying them to individual tasks and to task sets (Section 5.6).

In all cases, the task or tasks to be moved can be assigned or unassigned initially; either way, they are reassigned to the given resource. If the given resource is `NULL`, that's fine too; it means unassignment, even for the Kempe functions, where it would be more natural, arguably, for the operation to be undefined. The functions all return `false` when they either cannot carry out the requested changes, or they can but that changes nothing. Failed operations leave the solution in its state at the point of failure, so calls on these functions (except `KheTaskMoveResource`) should be enclosed in `KheMarkBegin` and `KheMarkEnd`, to undo failed attempts properly.

An *unchecked task move* is just a call on platform function

```
bool KheTaskMoveResource(KHE_TASK task, KHE_RESOURCE r);
```

Although it makes the checks described in Section 4.6, it is called 'unchecked' here, because it does not check whether the move introduces any incompatible tasks (defined below).

An *unchecked task set move* is a set of unchecked task moves to the same resource, as implemented by function

```
bool KheTaskSetMove(KHE_TASK_SET ts, KHE_RESOURCE r);
```

defined here. It moves the tasks of `ts` to `r` using calls to `KheTaskMoveResource`. It returns `true` when `ts` is non-empty and the individual moves all succeed.

An *ejecting task move* is a task move which both moves a resource to a task and *ejects* (that is, unassigns) the resource from all incompatible tasks. This is done by function

```
bool KheEjectingTaskMove(KHE_TASK task, KHE_RESOURCE r, bool allow_eject);
```

when `allow_eject` is `true`. It moves `task` to `r`, unassigning `r` from all incompatible tasks (defined below), and returning `true` if it succeeds. Failure can be due to `task` being fixed, or `r` not lying in the domain of `task`, or `r` being already assigned to `task`, or because some incompatible task cannot be unassigned, or it can be but `allow_eject` is `false`, meaning that ejection is not allowed (this is called an checked task move above).

`KheEjectingTaskMove` considers two tasks to be incompatible when they overlap in time. However, in nurse rostering, two tasks are often considered incompatible when they occur on the same day, so another function is offered which handles such cases using frames (Section 5.10):

```
bool KheEjectingTaskMoveFrame(KHE_TASK task, KHE_RESOURCE r,
  bool allow_eject, KHE_FRAME frame);
```

This is the same as `KheEjectingTaskMove` except that two tasks are considered incompatible if any time that one task is running lies in the same time group of `frame` as some time that the other task is running. Here `frame` may not be a null frame.

Unlike the corresponding function for ejecting meet moves, `KheEjectingTaskMove` and `KheEjectingTaskMoveFrame` do not consult the matching or use a group monitor. Instead, when r is non-NULL, they use r's timetable monitor to find the tasks assigned r that are incompatible with task and unassign them, returning false if any cannot be unassigned, because they are fixed or preassigned. Then they call KheTaskMoveResource and return what it returns.

It is not likely that some incompatible task task2 cannot be unassigned because it is fixed. This is because KheTaskFirstUnFixed(task2) (Section 4.6.1) is unassigned, not task2.

An *ejecting task set move* is a set of ejecting task moves to the same resource. This operation is carried out by functions

```
bool KheEjectingTaskSetMove(KHE_TASK_SET ts, KHE_RESOURCE r,
  bool allow_eject);
bool KheEjectingTaskSetMoveFrame(KHE_TASK_SET ts, KHE_RESOURCE r,
  bool allow_eject, KHE_FRAME frame);
```

which perform ejecting task moves on the elements of ts, without or with a frame, returning true when ts is non-empty and all of the individual ejecting task moves succeed.

A *Kempe task move* is carried out by functions

```
bool KheKempeTaskMove(KHE_TASK task, KHE_RESOURCE r);
bool KheKempeTaskMoveFrame(KHE_TASK task, KHE_RESOURCE r, KHE_FRAME frame);
```

If r is NULL, this is just an unassignment as usual. Otherwise, if task is initially unassigned, or assigned r, false is returned. Otherwise, let r2 be the resource initially assigned to task. KheKempeTaskMove performs a sequence of ejecting task moves, first of task to r, then of the tasks ejected by this move to r2, then of the tasks ejected by those moves to r, and so on until there are no ejected tasks. It fails if any of these ejecting task moves fails, or if tries to move some task twice. There is no allow_eject parameter because it is inherent in the Kempe idea to keep going until all tasks are assigned.

A *Kempe task set move* is approximately a set of Kempe task moves, carried out by

```
bool KheKempeTaskSetMove(KHE_TASK_SET ts, KHE_RESOURCE r);
bool KheKempeTaskSetMoveFrame(KHE_TASK_SET ts, KHE_RESOURCE r,
  KHE_FRAME frame);
```

The tasks must initially be assigned the same resource. This is not exactly like moving the tasks one by one, because the rule about not moving a task twice applies to the operation as a whole.

Finally, there is a way to select the kind of move to make on the fly, defined by type

```
typedef enum {
  KHE_MOVE_UNCHECKED,
  KHE_MOVE_CHECKED,
  KHE_MOVE_EJECTING,
  KHE_MOVE_KEMPE,
} KHE_MOVE_TYPE;
```

The usual four functions are offered:

```
bool KheTypedTaskMove(KHE_TASK task, KHE_RESOURCE r, KHE_MOVE_TYPE mt);
bool KheTypedTaskMoveFrame(KHE_TASK task, KHE_RESOURCE r,
  KHE_MOVE_TYPE mt, KHE_FRAME frame);
bool KheTypedTaskSetMove(KHE_TASK_SET ts, KHE_RESOURCE r,
  KHE_MOVE_TYPE mt);
bool KheTypedTaskSetMoveFrame(KHE_TASK_SET ts, KHE_RESOURCE r,
  KHE_MOVE_TYPE mt, KHE_FRAME frame);
```

These switch on `mt`, then call one of the functions above. There is also

```
char *KheMoveTypeShow(KHE_MOVE_TYPE mt);
```

which returns the obvious one-word description of `mt`: `"unchecked"` and so on.

## 12.4. Resource assignment algorithms

This section presents four algorithms for constructing initial assignments of resources to tasks. The next section documents another.

As explained at the start of this chapter, it is not wise to emphasise the distinction between construction and repair. Although the author has not found any uses for these algorithms in repair, there may be some; and later in this chapter there is another algorithm (resource matching) which is useful for both. Indeed, the time sweep algorithm built on resource matching is the author's method of choice for constructing an initial assignment in nurse rostering.

### 12.4.1. Satisfying requested task assignments

When an event resource must be assigned a particular resource, that should appear in the instance as a preassignment. Such preassignments in the instance are converted to assignments in the solution by function `KheSolnAssignPreassignedResources` (Section 4.3).

When the assignment is merely a preference, it will be included as a request, in the form of a constraint, not as a preassignment. Function

```
bool KheSolnAssignRequestedResources(KHE_SOLN soln,
  KHE_RESOURCE_TYPE rt, KHE_OPTIONS options);
```

may be used to make these requested assignments. It returns `true` if it changes the solution.

It is quite likely that some of the requested assignments will be incompatible with finding a good solution. That's fine: the assignments made by `KheSolnAssignRequestedResources` are not fixed in any sense; they are open to change by repair algorithms later.

`KheSolnAssignRequestedResources` works as follows. First, it finds all limit busy times and cluster busy times monitors which monitor resources of type `rt`, have non-zero cost, and have non-zero minimum limit without allowing zero. For the cluster busy times constraint, a non-trivial maximum limit can also be used if there are negative time groups, using the transformation given at the end of Section 3.7.14. We'll assume a minimum limit and positive time groups here, but the equivalent case of a maximum limit and negative time groups is also handled.

These monitors all require a resource to be assigned one or more tasks. In some cases, which we call *forcing* cases, they force the resource to be assigned a task at a particular time. For limit busy times constraints, this is true for each time in each time group whose cardinality is not larger than the minimum limit. For cluster busy times constraints, it is true for each time in time groups of cardinality one, when the number of time groups is not larger than the minimum limit. In all other cases, which we call *non-forcing* cases, they force the resource to be assigned a task, but not at a particular time.

Sort the monitors into decreasing combined weight order. Make two passes over the monitors, handling forcing cases on the first pass, and non-forcing cases on the second.

To handle forcing cases, find each particular time that the resource has to be busy. Try assigning the resource to each task of its type running at that time, and keep the assignment which produces the smallest solution cost.

To handle non-forcing cases, determine a set of times such that one of those times has to be busy (for cluster busy times monitors this will be the set of all times in all time groups that are not already busy), try assigning the resource to each task of its type running at any of those times, and keep the assignment which produces the smallest solution cost.

A monitor may need several repeats of this treatment to reduce its cost to 0. It is important to, in effect, start again on the monitor after keeping an assignment, since it is possible for one assignment to affect several times or time groups, especially when tasks have been grouped.

`KheSolnAssignRequestedResources` consults two options:

`rs_requested_off`
: A Boolean option which, when `true`, causes `KheSolnAssignRequestedResources` to do nothing.

`rs_requested_nonforced_on`
: A Boolean option which, when `true`, causes `KheSolnAssignRequestedResources` to carry out its second pass over the monitors (the pass that handles non-forced requests). By default this pass is omitted, because it is harder to justify and less obviously useful than the first (forced) pass.

It also uses the `gs_event_timetable_monitor` option (Section 8.4), to find the events running at each time. It aborts if this option is not in `options`.

There is another function, closely related to `KheSolnAssignRequestedResources`:

```
bool KheMonitorRequestsSpecificBusyTimes(KHE_MONITOR m);
```

It returns `true` if `m` requests that a resource be busy at one or more specific times, triggering a forcing case for `KheSolnAssignRequestedResources`. Precisely, it returns `true` when given:

1. A limit busy times monitor with a non-zero minimum limit, with `false` for `allow_zero`, and with one or more time groups whose cardinality is at least the minimum limit.

2. A cluster busy times monitor with a minimum limit equal to or greater than its number of time groups, with `false` for `allow_zero`, whose time groups are all positive, with one or more of them containing just one time.

3. A cluster busy times monitor such that the transformation documented by the theorem at the end of Section 3.7.14 produces the previous case.

The correspondence with `KheSolnAssignRequestedResources` is not quite exact, as it turns out, but the differences are insignificant, practically speaking.

### 12.4.2. Resource assignment by history

Section 12.4.1 presented an algorithm for assigning resources based on requests expressed by limit busy times and cluster busy times constraints. Here we carry out a similar programme, but this time based on the history values of limit active intervals constraints. The function is

```
int KheAssignByHistory(KHE_SOLN soln, KHE_RESOURCE_TYPE rt,
  KHE_OPTIONS options, KHE_SOLN_ADJUSTER sa);
```

It makes some assignments to resources of type `rt` which are forced in the sense that not making them would incur a cost in a limit active intervals constraint involving history. As usual, that does not always mean that these assignments are the best ones to make. If `sa != NULL`, the assignments made here are fixed and added to `sa`, so that someone else can unfix them later. If `sa == NULL`, the assignments are still made, but they are not fixed. The return value is the number of distinct resources assigned to tasks.

`KheAssignByHistory` is affected by one option:

`rs_assign_by_history_off`
A Boolean option which, when `true`, causes `KheAssignByHistory` to do nothing except return immediately with result value 0.

Why can't KHE's other solvers handle history themselves? The two solvers in question are `KheTimeSweepAssignResources` (Section 12.7.3) and `KheDynamicResourceSequentialSolve` (Section 12.6.6). The answer is partly that they both run after grouping, and grouping does not take history into account; and partly that `KheDynamicResourceSequentialSolve` may make arbitrary choices for the assignments it makes which cause problems for resources that are not assigned until later, including problems with history.

***Constraints.*** The algorithm handles each constraint $C$ that satisfies these conditions:

1. $C$ is a limit active intervals constraint with at least one time group.

2. Each time group of $C$ is positive.

3. Each time group $g$ of $C$ contains a single time. The day (that is, the time group of the common frame) containing this time is called $g$'s *associated day*.

4. As we proceed from one time group of $C$ to the next, the associated days are consecutive.

5. The associated day of the first time group of $C$ is the first day of the cycle.

These constraints are called *admissible constraints*. The conditions are checked, and if any fail to hold, $C$ is ignored. They seem restrictive, but constraints that satisfy them are common: limits on consecutive day shifts, on consecutive night shifts, and so on. Some limit active intervals constraints do not satisfy them, mainly limits on the number of consecutive busy or free days. However, those are less constraining than these ones, and it is not clear that trying to satisfy them at the start of the solve would be helpful.

One limit active intervals constraint may have several offsets, each representing a different instantiation of the constraint. We treat each offset as a distinct constraint, but for simplicity of presentation we say 'constraint' here when we should, strictly, say 'constraint plus offset'.

Let $C$'s minimum limit be $C_{min}$, and let its maximum limit be $C_{max}$. Both are compulsory in XESTT. If they were missing we could easily substitute the obvious values: 1 for $C_{min}$, and a large number, larger than the number of days plus the largest history value, for $C_{max}$.

We assume here that $C$'s cost function is not a step function. In the rare cases where it is a step function, our analysis does not always hold—but we apply our algorithm anyway.

***Resources***. For a given admissible constaint $C$, we are only interested in resources that must be busy during $C$'s first time group in order to avoid a cost for $C$ caused by history. Let $h(r)$ be $C$'s history value for resource $r$.

- If $h(r) = 0$, or equivalently if $C$ contains no value for $h(r)$, then there is no constraint on $r$'s timetable at the start of the cycle, so we are not interested in $r$.

- If $C_{min} \le h(r) \le C_{max}$, there is no need to extend the existing sequence of $h(r)$ tasks, since as it stands it generates zero cost. If $C_{max} < h(r)$, then it would be a bad idea to extend it, because it is already generating a cost which will increase if we extend it further. So we are not interested in $r$ in these cases.

So we are interested in resources $r$ such that $0 < h(r) < C_{min}$, called *admissible resources.*

We don't need to worry about $r$ having history in two or more constraints with different time groups. If one constraint monitors night shifts and another monitors day shifts, say, then we cannot have $h(r) > 0$ in both.

***Tasks***. We want to assign resources with non-zero history to tasks running at the start of the cycle. Each task $t$ used for this must satisfy these conditions:

(a) Task $t$ has the given resource type `rt`.

(b) Task $t$ is a proper root task.

(c) The times that $t$ is running (including the times of any tasks assigned, directly or indirectly, to $t$) include at least one time.

(d) The times that $t$ is running (including the times of any tasks assigned, directly or indirectly, to $t$) do not include two or more times from the same day.

(e) Every time that $t$ is running is a time monitored by $C$.

(f) The days of the times that $t$ is running are consecutive.

Tasks satisfying these conditions are *admissible*. The *domain* $d(t)$ of admissible task $t$ is the set of resources that may be assigned to $t$: its domain, according to function `KheTaskDomain`.

The first four conditions are not really restrictions. Condition (e) is needed because if $t$ is running at a time not monitored by $C$, then assigning a resource to $t$ will make that resource busy on the day of that time, preventing it from being busy at a time needed to satisfy $C$.

Although we expect to run our algorithm before grouping, we allow an admissible task to run for several consecutive days, for generality. To agree with the C language convention, we assign index 0 to the first day, rather than 1. Condition (f) allows us to represent the days that $t$ is running as an interval: a pair of integer indexes $(a, b)$ satisfying $0 \le a \le b$ which we call $i(t)$. This is both an interval in the sequence of days of the cycle and an interval in the sequence of time groups of $C$, given the restrictions above on how these two sequences of time groups are related. We write $l(t)$ for the length of $i(t)$.

The algorithm relies on sets $T_i$, each of which contains all admissible tasks $t$ such that $i(t) = (i, k)$ for some $k \ge i$; that is, all admissible tasks whose first day has index $i$. Building $T_i$ is a straightforward matter of retrieving from the event timetable monitor all meets running at $C$'s time on day $i$, finding all the tasks of type `rt` lying within those meets, finding their proper root tasks, then building their intervals and omitting those tasks that do not satisfy all the conditions for admissibility. Each $T_i$ is built only when it is needed.

***The algorithm***. Our algorithm handles each constraint $C$ satisfying the conditions above in turn. It is a time sweep which builds one weighted bipartite matching for each day in turn, starting from the first day of the cycle, until there is nothing more to do. The demand nodes on the first day are all resources satisfying the conditions above, and on subsequent days, all of those resources which are neither finished with, nor assigned a task on that day. The supply nodes on day $i$ are the tasks of the set $T_i$ defined above. An edge joins an $r$ to a $t$ when

1. If $t$ is assigned a resource $r'$, then $r = r'$; if $t$ is not assigned a resource, $r$ is assignable to $t$;

2. $h(r) + l(t) \le C_{max}$.

A maximum matching in this graph is found and used to decide which assignments to make. If we assign $r$ to $t$ on day $i$ and we have $C_{min} \le h(r) + i + l(t)$, that is, if $r$ has everything it needs, then $r$ is finished with. When all resources have been finished with, the algorithm terminates.

It remains to assign weights to the edges to encourage good decisions. These weights will naturally be related to the costs of the constraints affected by the decisions. We will focus here on just two kinds of constraints: the limit active intervals constraint $C$ that started all this, and the event resource constraints that are affected by the assignment or non-assignment of task $t$.

Taking only $C$ into account, let $a(r)$ be the cost of assigning $r$, and let $n(r)$ be the cost of not assigning $r$. Similarly, taking only event resource constraints relevant to $t$ into account, let $a(t)$ be the cost of assigning $t$, and let $n(t)$ be the cost of not assigning $t$. Then

$$w(r, t) = a(r) - n(r) + a(t) - n(t)$$

is a suitable weight. The more the cost of non-assignment exceeds the cost of assignment, the smaller this will be (very likely it will be negative, but that does not matter), and the greater the chance will be of choosing this edge and thus avoiding the expensive non-assignment.

Concretely, $a(r)$ is 0 and $n(r)$ is the cost due to $h(r) + i$ being smaller than $C_{min}$, while $n(t)$ and $a(t)$ are the values returned by `KheTaskNonAsstAndAsstCost` (Section 11.9.1).

***Miscellaneous points of detail***. Although this algorithm works off limit active intervals constraints, it is quite different from profile grouping. It can handle grouped tasks, but it is best to run it before any kind of grouping is run. There is one point of potential overlap with grouping. As described here, for the most part we assign the minimum number of tasks to each resource—tasks that cover $C_{min} - h(r)$ days. We could choose to assign more tasks than this, as long as we respect the $C_{max}$ upper limit. This might be useful if profile grouping determines that a set of grouped tasks should end on a later day. At present we are not doing this; we are relying on other parts of the overall solve to extend the sequence if needed.

A review of this section will show that the algorithm still works if different resources have different values for $C_{min}$ and $C_{max}$, as long as the time groups of $C$ are the same for all resources. So we partition the admissible constraints into equivalence classes. Two constraints lie in the same class when they have the same time groups in the same order. We then treat each class like a single constraint. The admissible resources are all resources with non-zero history in any of the class's constraints, and $C_{min}$ and $C_{max}$, as well as the constraint weight and cost function, can differ between resources.

We said earlier that we do not have to worry about a resource $r$ having non-zero history in two or more constraints with different time groups. However, $r$ could have non-zero history in two or more constraints with the *same* time groups (constraints in the same equivalence class). This will happen, for example, if the user wants to give weight 10 to sequences that are too short, and weight 20 to sequences that are too long. In such cases it would be a contradiction if the history values were different, but the limits, weights, and cost functions could be different. The solver handles such cases; it aims for a sequence for $r$ that satisfies all of its constraints by having length at least equal to the largest minimum, and no more than the smallest maximum. In the highly unlikely case that no such length exists, it ignores the resource. It also ignores resources which have different history values in different constraints. As mentioned, such cases, although legal, are basically self-contradictory.

***Incompatible consecutive busy days constraints***. A *consecutive busy days constraint* is an admissible constraint as defined above, with one change: instead of containing a single time, each time group $g$ contains a set of two or more times lying within a single day. In practice this will be the entire set of times of the day.

Consecutive busy days constraints can have history values that are incompatible with the history values in admissible constraints. For example, suppose that a consecutive night shifts constraint specifies that night shifts must come in sequences of 4 or 5, and a consecutive busy days constraint specifies that busy days must come in sequences of 3, 4, or 5. Suppose that just before the current instance begins, some resource has 2 consecutive morning shifts followed by one night shift. Then according to the consecutive night shifts constraint, in the current solution the resource must begin with 3 or 4 night shifts, but according to the consecutive busy days constraint it must begin with 0, 1, or 2 busy days. There is no way to satisfy both requirements.

If the consecutive busy days constraint has weight no larger than the weight of the consecutive night shifts constraint, it can be ignored. There will be a cost but it is unavoidable. If its weight is larger, however, it must take priority, which means that (in the example) we need to assign 2 night shifts, not 3 or 4. So when the weights dictate it, the number of night shifts to be assigned by assign by history must be reduced to the largest value that does not cause a violation of the incompatible consecutive busy days constraint.

### 12.4.3. Most-constrained-first assignment

When each unfixed task has no followers, so that each demands a resource for a single interval of time, as is usual with room assignment, a simple 'most constrained first' heuristic assignment algorithm that maintains the resource assignment invariant is usually sufficient to obtain a virtually optimal assignment (in high school timetabling, not nurse rostering). Function

```
bool KheMostConstrainedFirstAssignResources(KHE_TASKING tasking,
  KHE_OPTIONS options);
```

does this. It tries to assign each unassigned unfixed task of `tasking`, leaving assigned ones untouched. For each such task, it maintains the set of resources that can currently be assigned to the task without increasing the number of unmatchable demand tixels. It selects a task with the fewest such resources, assigns it if possible, and repeats until all tasks have been handled.

Each assignment preserves the resource assignment invariant. If no assignment can do that, the task remains unassigned. Among all resources that preserve it, as a first priority an assignment that minimizes `KheSolnCost` is chosen, and as a second priority, resources that have already been assigned to other tasks of the event resources of the task and the tasks assigned to it are preferred. So even when an avoid split assignments constraint is not present, the algorithm favours assigning the same resource to all the tasks of a given event resource, for regularity.

In fact, `KheMostConstrainedFirstAssignResources` assigns multi-tasks (Section 11.9), not individual tasks. Each task of a multi-task is assignable by the same resources, so one list of suitable resources is kept per multi-task. At each step, a multi-task is selected for assignment for which the number of suitable resources minus the number of unassigned tasks is minimal.

When a resource is assigned to a task, it becomes less available, so its suitability for assignment to its other multi-tasks is rechecked. If it proves to be no longer assignable to some of them, their priorities are changed. The multi-tasks are held in a priority queue (Section A.4), which allows their queue positions to be updated efficiently when their priorities change.

### 12.4.4. Resource packing

To *pack* a resource means to find assignments of tasks to the resource that make the solution cost as small as possible, while preserving the resource assignment invariant, in effect utilizing the resource as much as possible [9]. Following the recommended interface for resource assignment functions (Section 12.1), function

```
bool KheResourcePackAssignResources(KHE_TASKING tasking,
  KHE_OPTIONS options);
```

assigns resources to the unassigned tasks of `tasking` using resource packing, as follows.

The tasks are grouped into multi-tasks (Section 11.9). Two numbers help to estimate the difficulty of utilizing a resource effectively: the *demand duration* and the *supply duration*. A resource's demand duration is the total duration of the multi-tasks it is assignable to. Its supply duration is the number of times it is available for assignment: the cycle length, minus the number of its workload demand monitors, minus the total duration of any tasks it is already assigned to.

The resources are placed in a priority queue, ordered by increasing demand duration minus supply duration. That is, the less demand there is for the resource, or the more supply, the more important it is to pack it sooner rather than later. In practice, part-time teachers come first in this order, which is good, because they are difficult to utilize effectively.

The main loop of the algorithm removes a resource of minimum priority from the priority queue and packs it. If this causes any multi-tasks to become completely assigned, they are unlinked from the resources assignable to them, reducing those resources' demand durations and thus altering their position in the priority queue. This is repeated until the queue is empty.

Each resource `r` is packed using a binary tree search: at each tree node, one available task group is either assigned to `r`, or not. The multi-tasks are taken in decreasing order of the maximum, over all tasks `t` of the multi-task, of `KheMeetDemand(m)`, where `m` is the first unfixed meet on the chain of assignments out of the meet containing `t`. This gives preference to tasks whose meets are hard to move, reasoning that the leftovers will be given split assignments, and repairing them may require moving their meets. The search tree has a moderate depth limit. At the limit, the algorithm switches to a simple heuristic which assigns as many tasks as it can.

### 12.4.5. Split assignments

After solver functions such as `KheMostConstrainedFirstAssignResources` (Section 12.4.3) and `KheEjectionChainRepairResources` (Section 12.10) have assigned resources to most tasks, some tasks may remain unassigned. These will have to receive split assignments. Function

```
bool KheFindSplitResourceAssignments(KHE_TASKING tasking,
  KHE_OPTIONS options);
```

reduces the cost of the solution as much as it can, by making split assignments to the unassigned tasks of `tasking` while maintaining the resource assignment invariant. Any tasks which were unassigned to begin with are replaced in `tasking` by their child tasks.

At the core of `KheFindSplitResourceAssignments` is a procedure which takes every pair of resources capable of constituting a split assignment to some task and tries to assign them greedily to the task, keeping the assignment that produces the lowest solution cost. However, before entering on that, `KheFindSplitResourceAssignments` eliminates resources that cannot be assigned even to one child task, makes assignments that are forced because there is only one available resource (not forgetting that one forced assignment might lead to another, or that once a resource has been assigned to one child task it makes sense to assign it to as many others as possible), and divides each task into independent components (in the sense that no resource is assignable to two components). In practice, much of what it does is more or less forced.

## 12.5. Single resource assignment using dynamic programming

This section presents a polynomial-time dynamic programming algorithm that finds an optimal timetable for a single resource, assuming that time assignments are fixed, and that the resource's timetable can be built up step by step in chronological order, as in nurse rostering.

Let the single resource be *r*. The algorithm finds one timetable for *r* for each distinct total number of assigned times. Each timetable minimizes the total cost of the resource constraints that monitor the timetable of $r$, among all timetables with that number of assigned times. The caller is then free to adopt any one of these timetables. The algorithm does not minimize other costs, such as the cost of assigning or not assigning tasks, or costs that depend on the timetables of two resources. It chooses unassigned tasks whose assignment minimizes these costs at the moment they are chosen, but that is not the same as minimizing them overall.

To run the algorithm, the first step is to create a *single resource solver*, by calling

```
KHE_SINGLE_RESOURCE_SOLVER KheSingleResourceSolverMake(KHE_SOLN soln,
  KHE_OPTIONS options);
```

Among other things, parameter `options` is used to access the common frame, defining the days of the cycle. The solver can be deleted when it is no longer wanted, by calling

```
void KheSingleResourceSolverDelete(KHE_SINGLE_RESOURCE_SOLVER srs);
```

To solve for a particular resource `r`, call

```
void KheSingleResourceSolverSolve(KHE_SINGLE_RESOURCE_SOLVER srs,
  KHE_RESOURCE r, KHE_SRS_DOM_KIND dom_kind, int min_assts,
  int max_assts, KHE_COST cost_limit);
```

This does not change the solution. Instead, it carries out the solve and finds a number of distinct timetables. The timetables vary in the number of assignments they contain, as explained above.

The type of parameter `dom_kind` is defined in `khe_solvers.h` as

```
typedef enum {
  KHE_SRS_DOM_WEAK,
  KHE_SRS_DOM_MEDIUM,
  KHE_SRS_DOM_SEPARATE,
  KHE_SRS_DOM_TRIE
} KHE_SRS_DOM_KIND;
```

This determines whether the solve uses weak dominance, medium dominance, separate dominance, or trie dominance. These terms are explained below. Solutions of minimum cost are found in any case; there may be some difference in running time.

The solve only finds timetables whose number of assignments is at least `min_assts` and at most `max_assts`; if these restrictions are not wanted, simply pass `0` and `INT_MAX`. The result of `KheResourceMaxBusyTimes` (Section 4.7) would be a good starting point for constructing more interesting values for `min_assts` and `max_assts`.

The solve only finds timetables whose cost is no larger than `cost_limit`. A reasonable value for this in nurse rostering would be `KheCost(0, INT_MAX)`, since hard constraint violations are unacceptable. To have no cost limit at all, use `KheCost(INT_MAX, INT_MAX)`.

To find out about the timetables produced by `KheSingleResourceSolverSolve`, call

```
int KheSingleResourceSolverTimetableCount(KHE_SINGLE_RESOURCE_SOLVER srs);
void KheSingleResourceSolverTimetable(KHE_SINGLE_RESOURCE_SOLVER srs,
  int i, int *asst_count, KHE_COST *r_cost);
```

afterwards. `KheSingleResourceSolverTimetableCount` returns the number of timetables that were found, and `KheSingleResourceSolverTimetable` reports on the `i`th timetable, for `i` in the range 0 to `KheSingleResourceSolverTimetableCount(srs) - 1`. It reports the number of assignments, and the total cost of the resource monitors of `r` (the quantity that is optimized). The timetables are returned in increasing order of `*asst_count`.

It is up to the caller to decide which of these timetables, if any, to take into the solution. To actually change the original solution, call

```
void KheSingleResourceSolverAdopt(KHE_SINGLE_RESOURCE_SOLVER srs, int i);
```

This will change the solution to include the `i`th timetable.

One way (not the only way) to decide among the different solutions is to assume that not assigning one time has cost `c`, and choose a solution that minimizes

```
*r_cost - *asst_count * c
```

breaking ties in favour of solutions whose `*asst_count` is larger. Function

```
int KheSingleResourceSolverBest(KHE_SINGLE_RESOURCE_SOLVER srs,
  KHE_COST cost_reduction);
```

does this (the `cost_reduction` parameter is `c`), and returns the index of the best timetable by this measure. It may only be called when there is at least one timetable. The result may be passed to `KheSingleResourceSolverTimetable` or `KheSingleResourceSolverAdopt`.

The caller must choose a suitable value of `c`. The best way to do this, probably, is to create a balance solver (Section 11.4.5) and use the result of `KheBalanceSolverMarginalCost` for `c`.

To move on to another resource, call `KheSingleResourceSolverSolve` again. It saves some time (not a huge amount) to use one solver on many resources. All memory is reclaimed by `KheSingleResourceSolverDelete`. Finally,

```
void KheSingleResourceSolverDebug(KHE_SINGLE_RESOURCE_SOLVER srs,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of `srs` onto `fp` with the given verbosity and indent; and

```
void KheSingleResourceSolverTest(KHE_SOLN soln, KHE_OPTIONS options,
  KHE_RESOURCE r);
```

creates a single resource solver and tests it by finding optimal timetables for `r`. It produces some debug output, including graphs (Section 8.9.2) in subdirectory `stats` of the current directory. The user must create this subdirectory before `KheSingleResourceSolverTest` is called.

This algorithm is a precursor of the dynamic resource solver from the next section, and many of that algorithm's ideas were first tried out here. The main differences here are that we assign a single resource for the full set of days, that multi-day tasks do not spoil the optimality, and that we find a set of alternative solutions with varying numbers of assignments.

## 12.6. Optimal resource reassignment using dynamic programming

This section presents a function for finding an optimal reassignment of an arbitrary subset of the resources of an instance over an arbitrary subset of the days of the instance, assuming time assignments are fixed. The function uses a straightforward generalization of a dynamic programming algorithm that has long been used within column generation solvers for nurse rostering to produce an optimal timetable for a single nurse.

The algorithm is designed to support a very large-scale neighbourhood (VLSN) search algorithm which repeatedly chooses some resources and some days, unassigns those resources on those days, and reassigns them optimally. The initial setup creates data structures whose size is proportional to the size of the entire solution, but each call on the solver after that has running time which depends only on the number of selected resources and days. This is important for efficiency: if there are 50 resources and 28 days, but one call on the solver reassigns, say, 5 resources over 7 days, we want the running time to depend on 5 and 7, not on 50 and 28.

As Appendix C.2 shows, the running time of one call on the solver is $O(n(a+1)^m mn^{cm})$, where $n$ is the number of selected days, $m$ is the number of selected resources, $a$ is a constant, the number of shift types, and $c$ is another constant, the number of constraints (usually 1 or 2) per resource whose maximum limits increase with $n$ (such as limits on the total number of shifts). As $n$ and $m$ increase, this increases rapidly. Precisely which values lead to feasible running times is a matter for empirical experiment. The fact that the running time is polynomial in $n$ and exponential in $m$ comes through clearly in experiments: it can be quite feasible to reassign 14 days or more, but not 14 resources.

Optimality is only guaranteed when each task lies within a single day. Multi-day tasks, whether derived from events of duration greater than 1 or from task grouping, are handled, but the solver is only a good heuristic, not optimal, when they are present. Another limitation is that the solver ignores avoid split assignments and limit idle times constraints. These are never present in the nurse rostering instances that it is mainly intended for.

### 12.6.1. Invoking optimal resource reassignment

Functions

```
KHE_DYNAMIC_RESOURCE_SOLVER KheDynamicResourceSolverMake(KHE_SOLN soln,
  KHE_RESOURCE_TYPE rt, KHE_OPTIONS options);
void KheDynamicResourceSolverDelete(KHE_DYNAMIC_RESOURCE_SOLVER drs);
```

create and delete a dynamic resource solver for `soln` which reassigns resources of type `rt`.

`KheDynamicResourceSolverMake` places the proper root tasks of type `rt` into multi-tasks (Section 11.9) which last for the life of the solver. During that time, any changes to their assigned times, domains, or groupings will not change the state of the solver, and so may cause errors. Calls which assign and unassign resources to proper root tasks are also a problem, although assignments and unassignments made by the solver itself are not.

After creating, but before solving, make any number of calls to

```
void KheDynamicResourceSolverAddDayRange(KHE_DYNAMIC_RESOURCE_SOLVER drs,
  int first_day_index, int last_day_index);
void KheDynamicResourceSolverAddResource(KHE_DYNAMIC_RESOURCE_SOLVER drs,
  KHE_RESOURCE r);
```

The first adds the days from `first_day_index` to `last_day_index` inclusive to the solve. The values are indexes into the common frame. The union of the ranges is solved; this could be several disjoint intervals. The second adds resource `r` (which must have type `rt` and must not be already added) to the solve. These calls define the *selected days* and the *selected resources.*

After loading the day ranges and resources, to do the actual solve call

```
bool KheDynamicResourceSolverSolve(KHE_DYNAMIC_RESOURCE_SOLVER drs,
  bool priqueue, bool extra_selection, bool expand_by_shifts,
  bool shift_pairs, bool correlated_exprs, int daily_expand_limit,
  int daily_prune_trigger, int resource_expand_limit, int dom_approx,
  KHE_DRS_DOM_KIND main_dom_kind, bool cache,
  KHE_DRS_DOM_KIND cache_dom_kind);
```

This changes `drs`'s solution for the selected resources on the selected days, to an optimal solution as explained above. It returns `true` when the new solution has lower cost than the old one. After it returns, `drs` is ready to set up for another solve: it is expecting another set of calls to `KheDynamicResourceSolverAddResource` and `KheDynamicResourceSolverAddDayRange`, or a call to `KheDynamicResourceSolverDelete`.

`KheDynamicResourceSolverSolve` may overrun its time limit, but not by much. It checks the time limit frequently (by calls to `KheOptionsTimeLimitReached`), and terminates early, without finding a new best solution, if it has been reached.

There is also

```
void KheDynamicResourceSolverTest(KHE_DYNAMIC_RESOURCE_SOLVER drs,
  bool priqueue, bool extra_selection, bool expand_by_shifts,
  bool shift_pairs, bool correlated_exprs, int daily_expand_limit,
  int daily_prune_trigger, int resource_expand_limit, int dom_approx,
  KHE_DRS_DOM_KIND main_dom_kind, bool cache,
  KHE_DRS_DOM_KIND cache_dom_kind, KHE_COST *cost);
```

which is the same as `KheDynamicResourceSolverSolve` except that it never changes the solution, not even when it finds a new best. This helps when testing. To give some idea of how things went, it sets `*cost` to the cost of the solution it finds, either a new best or the original.

For both functions, `priqueue` says whether to use a priority queue of solutions. If it is `false`, the solve proceeds day by day. If it is `true`, the solve proceeds solution by solution, always choosing for expansion the solution with the lowest cost, irrespective of the day. This will expand fewer solutions, since it stops as soon as it chooses a solution lying on the last day. But whether this will be faster is hard to say, because of the extra overhead of the priority queue.

Parameter `extra_selection` says whether to use one- and two-extra selection when expanding each solution into the solutions for the next day. This should reduce the number of new solutions generated without sacrificing optimality.

Parameter `expand_by_shifts` says whether to expand by shifts; if not, expansion by resources is used. Expansion by shifts should be best.

Parameter `shift_pairs` says whether to test pairs of shifts for dominance during expansion by shifts.

Parameter `correlated_exprs` says to adjust dominance testing to take account of correlated expressions, as described in Appendix D.6.5. It should be best to do this.

Parameter `daily_expand_limit` is an upper limit on the number of undominated solutions that the call may expand on each day. When the priority queue is not in use, at the end of each day the undominated solutions for that day are sorted by increasing cost, the most costly are deleted until at most `daily_expand_limit` undominated solutions remain, and those are expanded. When the priority queue is in use, the algorithm keeps track of the number of solutions it has expanded on each day, and when it reaches `daily_expand_limit`, any further solutions for that day are deleted and freed rather than expanded. Setting the value to 0 turns off this feature.

Parameter `daily_prune_trigger` is also a number of undominated solutions used to limit the number of solutions on each day, but its method is quite different. When an undominated solution is added on some day which causes the number of undominated solutions on that day to be exactly `daily_prune_trigger`, the cost of the costliest solution on that day at that moment becomes the value used to prune the search for further solutions on that day, if that causes a reduction in that value. Any generation path leading to solutions for that day for which it can be shown that all solutions generated will have at least that cost are abandoned. This applies whether the priority queue is in use or not. Setting the value to 0 turns off this feature.

Potentially, `daily_prune_trigger` is better than `daily_expand_limit` at reducing run time, because it avoids ever generating the solutions that `daily_expand_limit` merely deletes. However, `daily_expand_limit` deletes solutions that are definitely uncompetitive on their day, whereas `daily_prune_trigger` avoids generating solutions that are probably uncompetitive. Also, `daily_prune_trigger` only works with the indexed set data structure, because only that data structure gives efficient access to the costliest undominated solution on a given day.

Parameter `resource_expand_limit` is the maximum number of alternative assignments tried for each resource on each day. The lowest cost `resource_expand_limit` alternatives are tried, or more if there are ties in cost. Setting the value to 0 turns off this feature.

Parameter `dom_approx` exploits the fact that dominance testing is conservative: it deletes only provably uncompetitive solutions. Hand analysis reveals many cases where dominance is clear, but not provable. When `dom_approx > 0`, the initial available cost is increased by a factor of `dom_approx/10`: by 10% when `dom_approx` is 1, by 20% when `dom_approx` is 2, and so on. This makes the dominance test more likely to succeed, so it reduces the number of undominated solutions and speeds up the algorithm, although provable optimality is lost. Setting `dom_approx` to 0 turns off this feature. Only dominance tests between $d_k$-complete solutions are affected.

Parameter `main_dom_kind` has type

```
typedef enum {
  KHE_DRS_DOM_LIST_NONE,                    /* UNUSED     */
  KHE_DRS_DOM_LIST_SEPARATE,                /* SEPARATE   */
  KHE_DRS_DOM_LIST_TRADEOFF,                /* TRADEOFF   */
  KHE_DRS_DOM_LIST_TABULATED,               /* TABULATED  */
  KHE_DRS_DOM_HASH_EQUALITY,                /* UNUSED     */
  KHE_DRS_DOM_HASH_MEDIUM,                  /* SEPARATE   */
  KHE_DRS_DOM_INDEXED_TRADEOFF,             /* TRADEOFF   */
  KHE_DRS_DOM_INDEXED_TABULATED             /* TABULATED  */
} KHE_DRS_DOM_KIND;
```

(For the comments, see below.) This determines two things at once: the data structure to hold sets of solutions in, which may be a list, hash table, or array indexed by cost, and the kind of dominance testing to use, which may be none (no solution dominates another), separate, tradeoff, tabulated, equality, or medium. These are mixed together because different data structures support different kinds of testing. `KHE_DRS_DOM_INDEXED_TABULATED` is usually best.

Tradeoff and tabulated dominance only work in places along the signature where the value relates to a root expression. Furthermore, tradeoff dominance only works in places where the cost of that expression is calculated using a cost function that is not quadratic. At places where these conditions do not hold, separate dominance is used, regardless of what the user requests. If correlated expressions are used, they provide a better way to handle most non-root expressions.

The two functions also offer the option of using caching. If `cache` is `true`, caching is used and `cache_dom_kind` says what kind of data structure and dominance testing to use within the caches. If `cache` is `false`, caching is not used and `cache_dom_kind` is not used either.

A cache is a set of solutions for a given day which have the same parent solution, making it likely that there will be dominance relations between them. When caching is used, new solutions are inserted into the cache for the new day rather than the main table for the day. The usual dominance testing goes on within the cache, which therefore contains only undominated solutions. Then after all solutions with the same parent solution are tried, the surviving members of the cache are inserted into the main table, once again with the usual dominance testing, and the cache is emptied out. The hope is that dominance testing within the small cache will run much faster than dominance testing within the large main table, saving time overall.

It has been thought best to not worry the dominance testing code about whether it is going on within the main table or within the cache. So although the main table and the cache may use quite different data structures, when both use dominance testing that is more than simple equality they must use the same kind of dominance testing. This is expressed by the comments above: when the cache is in use, if both of `main_dom_kind` and `cache_dom_kind` have values whose comments above are not `UNUSED`, then those comments must be the same.

Behind the scenes, the solver also caches the results of some partial dominance tests, to speed up the solve when those tests are made repeatedly. This behind-the-scenes caching is always on, but it is likely to be invoked more often when caching is used. The author thought that the two together would provide a significant speedup, but that has not eventuated in practice.

### 12.6.2. Statistics and debug

After one call to `KheDynamicResourceSolverSolve` and before the next, these two functions may be called to retrieve some statistics about the solve:

```
int KheDynamicResourceSolverSolveStatsCount(
  KHE_DYNAMIC_RESOURCE_SOLVER drs);
void KheDynamicResourceSolverSolveStats(KHE_DYNAMIC_RESOURCE_SOLVER drs,
  int i, int *solns_made, int *table_size, float *running_time);
```

`KheDynamicResourceSolverSolveStatsCount` is the number of statistics, usually the number of open days plus one. `KheDynamicResourceSolverSolveStats(drs, i)` sets `*solns_made` to the number of solutions created on open day `i`, `*table_size` to the size of the table on open day `i`, and `*running_time` to the running time, to be described next. When `i` is 0 these numbers will be 1, 1, and the time to open for solving. For these functions to work correctly, the `TESTING` compiler flag (defined near the top of file `khe_sr_dynamic_resource.c`) must be set to 1.

When the priority queue is not in use, the running time for a given day is the time in seconds from the start of the solve to the end of the day. When the priority queue is in use, the running time for a given day is the time from the start of the solve until the first moment that a solution for the given day is deleted from the priority queue. Either way, on the last day this is the time that the algorithm stops, not counting closing, which is omitted from the timings.

Function

```
void KheDynamicResourceSolverDebug(KHE_DYNAMIC_RESOURCE_SOLVER drs,
  int verbosity, int indent, FILE *fp);
```

produces the usual debug print of `drs` onto `fp` with the given verbosity and indent.

### 12.6.3. Precise specification

Here is an exact statement of what the solver does. The following applies only to resources and tasks of the type passed to `KheDynamicResourceSolverMake`. Resources and tasks of other types are irrelevant. The solver uses multi-tasks (Section 11.9), and everything required for a successful grouping of tasks into multi-tasks is required here too.

For each proper root task `t`, let the *busy times* of `t` be the times that `t` is running, including the times that the tasks assigned (directly or indirectly) to `t` are running. If `t` has at least one busy time, then let the *first busy day* of `t` be the day holding the chronologically first busy time of `t`, let the *last busy day* of `t` be the day holding the chronologically last busy time of `t`, and let the *busy day range* of `t` be the range of days from the first busy day to the last busy day inclusive.

First, `KheDynamicResourceSolverMake` may return `NULL`, meaning that the solver could not be made. This happens when any of the following conditions occurs:

- `KheResourceTypeDemandIsAllPreassigned(rt)` (Section 3.5.1) is `true`; there is no point in trying to solve, because the assignments of the resources of `rt` cannot be changed.

- There is no common frame (option `gs_common_frame`, needed to define the days), or there is no event timetable monitor (option `gs_event_timetable_monitor`).

- Some task (not necessarily a proper root task) lies in a meet whose time is not assigned. The solver assumes that all tasks run at known times, so it cannot handle such tasks. They are discovered when the solver calls `KheMTaskSolverMake` (Section 11.9.3) and finds that `incomplete_times` is `false`. Tasks that do not lie in any meet are allowed.

- Some proper root task with at least one busy time has a busy day range with a gap: a day in the busy range when the task is not running. Tasks with gaps can occur, so should really be allowed, but they cause implementation problems that would be hard to solve, so it has seemed better to avoid them altogether by means of this rule, given that tasks with gaps are very unlikely to occur in practice. This case is uncovered when the solver calls `KheMTaskNoGaps` (Section 11.9.1) and finds that the result is `false`.

- Some proper root task runs twice on one day, taking into account the tasks assigned to it (directly or indirectly); or some resource is initially assigned to two tasks on one day. The solver uses a representation of solutions which assumes that each resource can be assigned to at most one task on each day, which is why it cannot handle these cases. The first case is uncovered when the solver calls `KheMTaskNoOverlap` (Section 11.9.1) and finds that the result is `false`; the second is uncovered by the solver as it initializes itself.

Although not required, it is better if, in the initial solution, each preassigned task is assigned. The solver limits its search to solutions in which each preassigned task is assigned. This can place it at an unfair disadvantage to the initial solution if that contains an unassigned preassigned task.

Second, if `KheDynamicResourceSolverMake` returns a non-`NULL` value, the solver selects those tasks `t` that satisfy all of the following conditions:

1. Task `t` is a proper root task of the given resource type.

2. Task `t` has at least one busy time. Given the conditions above, this can fail only if `t`, and all tasks assigned to `t` (directly or indirectly), do not lie in any meet. That is not a problem, but such tasks are not selected.

These first two conditions are consequences of using multi-tasks, and are applied when the solver is created. The following three conditions are applied when starting one solve:

3. Every day in the busy day range of `t` is a selected day.

4. `KheTaskDomain(t)` has a non-empty intersection with the set of selected resources.

5. Initially, `t` is either unassigned, or else it is assigned one of the selected resources. In the second case, it must be possible to unassign `t` (unless `t` is preassigned).

Finally, the result of a solve is a solution which is identical to the initial solution except that changes to the assignments of the selected tasks, to either a selected resource or to `NULL`, are allowed. This result solution will be optimal when there are no multi-day tasks, no avoid split assignments constraints, and no limit idle times constraints.

Appendices C and D contain a detailed account of the algorithm and its implementation.

### 12.6.4. Using optimal resource reassignment in practice

`KheDynamicResourceSolverSolve` supports very large-scale neighbourhood (VLSN) search:

```
bool KheDynamicResourceVLSNSolve(KHE_SOLN soln,
  KHE_RESOURCE_TYPE rt, KHE_OPTIONS options);
```

This calls `KheDynamicResourceSolverMake`, returning immediately if its result is `NULL`, then calls `KheDynamicResourceSolverSolve` repeatedly with various choices of resources and days, ending by deleting the solver. It returns `true` and changes `soln` if it was able to improve the solution, and returns `false` with `soln` unchanged if not. It is influenced by these options:

`rs_drs_off`
: A Boolean option which, when `true`, makes `KheDynamicResourceVLSNSolve` do nothing.

`rs_drs_time_limit`
: A soft time limit for the entire call to `KheDynamicResourceVLSNSolve`, in the format of `KheTimeFromString` (Section 8.1): `secs`, or `mins:secs`, or `hrs:mins:secs`, or `-`, meaning 'set no limit' (the default). Time limits set by the caller also work. If it were not for this option and the `rs_drs_solve_limit` and `rs_drs_fail_limit` options (just below), `KheDynamicResourceVLSNSolve` could well run forever, so care is needed. The limit is checked after each call to `KheDynamicResourceSolverSolve`, so the end could come some time after the limit is reached.

`rs_drs_solve_limit`
: An integer upper limit on the number of calls to `KheDynamicResourceSolverSolve`. The special value `-` means 'set no limit'. The default value is `1000`.

`rs_drs_fail_limit`
: An upper limit on the number of consecutive calls to `KheDynamicResourceSolverSolve` which produce no improvement; if this limit is reached, `KheDynamicResourceVLSNSolve` terminates. The special value `-` means 'set no limit'. The default value is `100`.

The options so far affect the test as a whole. The following options affect individual solves. Each has its usual name plus a much briefer name, just a single capital letter, which appears in parentheses below and will be used later.

`rs_drs` (R)
: A string option defining how the resources and days are to be selected on each solve. The default value is `"targeted(4:8, 5:6, 6:3)"`. A detailed explanation appears below.

`rs_drs_priqueue` (Q)
: An Boolean option which becomes the `priqueue` argument of each call made to `KheDynamicResourceSolverSolve`. The default value is `false`.

`rs_drs_extra_selection` (E)
: A Boolean option which becomes the `extra_selection` argument of each call to `KheDynamicResourceSolverSolve`. The default value is `true`.

`rs_drs_expand_by_shifts` (S)
: A Boolean option which becomes the `expand_by_shifts` argument of each call to `KheDynamicResourceSolverSolve`. The default value is `true`.

`rs_drs_shift_pairs` (P)
: A Boolean option which becomes the `shift_pairs` argument of each call to `KheDynamicResourceSolverSolve`. The default value is `false`.

`rs_drs_correlated_exprs` (C)
: A Boolean option which becomes the `correlated_exprs` argument of each call to `KheDynamicResourceSolverSolve`. The default value is `false`.

`rs_drs_daily_expand_limit` (L)
: An integer option which becomes the `daily_expand_limit` argument of each call to `KheDynamicResourceSolverSolve`. The default value is `10000`. This removes the guarantee of optimality but brings some sanity to running time. To remove the limit and so restore the guarantee of optimality, use value `0`.

`rs_drs_daily_prune_trigger` (T)
: An integer option which becomes the `daily_prune_trigger` argument of each call to `KheDynamicResourceSolverSolve`. The default value is `0`, meaning no trigger. Any other value takes away the guarantee of optimality.

`rs_drs_resource_expand_limit` (M)
: An integer option which becomes the `resource_expand_limit` argument of each call to `KheDynamicResourceSolverSolve`. The default value is `0`, meaning no limit. Any other value takes away the guarantee of optimality.

`rs_drs_dom_approx` (A)
: An integer option which becomes the `dom_approx` argument of each call to `KheDynamicResourceSolverSolve`. The default value is `0`, meaning exact dominance, not approximate. Any other value takes away the guarantee of optimality.

`rs_drs_dom_kind` (K)
: A string option which becomes the `main_dom_kind` argument of each of the calls to `KheDynamicResourceSolverSolve`. The value consists of two parts separated by an underscore. The first part says which data structure to use when storing a solution set; it may be `"list"` (an unsorted list), `"hash"` (a hash table), or `"indexed"` (an array, indexed by cost, of lists). The second part says the kind of dominance testing to use, and may be `"none"` (don't test for dominance), `"separate"` (separate dominance), `"equality"` (equality dominance), `"medium"` (medium dominance), `"tradeoff"` (for tradeoff dominance), or `"tabulated"` (tabulated dominance). The two parts are not independent, because each data structure only supports some kinds of dominance testing. The legal values of `rs_drs_dom_kind` are `"list_none"`, `"list_separate"`, `"list_tradeoff"`, `"list_tabulated"`, `"hash_equality"`, `"hash_medium"`, `"indexed_tradeoff"`, and `"indexed_tabulated"`. The default is `"indexed_tabulated"`.

`rs_drs_cache_dom_kind` (J)
A string option defining the dominance testing to use when caching. It takes the same values as `rs_drs_dom_kind` just above, plus `"nocache"` (omit caching) and `"same"` (use the value of `rs_drs_dom_kind`). It sets the `cache` and `cache_dom_kind` arguments of each call to `KheDynamicResourceSolverSolve`. The default value is `"nocache"`.

Here now are the details of the `rs_drs` option, which determines how the resources and days are selected. Unlike the other options, which define simple Boolean, integer, or enumerated values, this one defines sequences of sets of resources and days.

We routinely place quotes around the value of the `rs_drs` option, and also the `rs_drs_test` option below. We do this because these values often contain characters that have a special meaning for the operating system's command interpreter.

Values for the `rs_drs` option come in two types, which we call *separate* and *combined*. The author implemented separate values initially, and only later discovered the arguably superior virtues of combined values.

Separate values contain a bar character:

```
rs_drs="...|..."
```

The part before the bar defines the resources `R`, and the part after the bar defines the days `D`. The two parts are quite separate, hence the name. When `rs_drs` has a separate value, `KheDynamicResourceVLSNSolve` implements this algorithm:

```
for each set of resources R defined by the first part of rs_drs do
  for each set of days D defined by the second part of rs_drs do
    solve for R and D;
```

There are options for choosing `R` and `D` randomly, giving a traditional VLSN search which ends when time runs out. Alternatively the user can give specific values for either or both.

***Choosing resources separately***. The first part of a separate `rs_drs` value, the part before the bar character, specifies the resources that are to be included in the solve. There are several ways to do it. The first is by giving resource indexes:

```
rs_drs="0,2,4;1,3,5|..."
```

This defines a sequence of two sets of resources. The first contains those resources whose indexes in the current resource type (call it `rt`) are 0, 2, and 4; the second contains those resources whose indexes in `rt` are 1, 3, and 5. Any out of range indexes are ignored.

The second way to choose resources separately is

```
rs_drs="all(3)|..."
```

This works sequentially through the resources, taking them in groups of three (or whatever number is specified between the parentheses). The last group may have fewer resources.

The third way to choose resources separately is

```
rs_drs="choose(3)|..."
```

This defines a potentially infinite sequence of sets of resources, each set containing three (or whatever is specified) elements chosen at random from `rt`. If the request is for more resources than `rt` contains, the number is silently reduced to the number of resources that `rt` contains.

The fourth way to choose resources separately is

```
rs_drs="cluster(str)|..."
```

where `str` is a string of characters not including the right parenthesis character. This selects resources in the following way.

For each resource `r` of type `rt`, scan the monitors `m` that monitor the timetable of `r`. For each cluster busy times monitor whose constraint's Id or Name contains `str`, and whose time groups are all positive (these are `r`'s *selected monitors*), decide whether `r` is *overloaded* (more active time groups than the maximum limit), *maximal* (the same number of active time groups as the maximum limit), or *underloaded* (otherwise). Then classify `r` itself as overloaded if any of these classifications are overloaded, or else maximal if any are maximal, or else underloaded (including the case where `r` has no selected monitors).

The sequence of sets of resources iterated over is then defined as follows. If there are no overloaded resources or no underloaded resources, the sequence of sets is empty. If there is at least one maximal resource, then up to three sets of resources is defined for each pair of resources $(r_1, r_2)$, where $r_1$ is an overloaded resource and $r_2$ is an underloaded resource. Each set has the form $\{r_1, r_2, r_3\}$, where $r_3$ is a maximal resource. Or if there are no maximal resources, sets consisting of just all pairs $\{r_1, r_2\}$ are defined.

Each set offers a distinct opportunity to move load from an overloaded resource to an underloaded one, which, all else being equal, will reduce cost. If there is a cost reduction, the algorithm is restarted from the beginning, including a fresh classification of each resource as overloaded, maximal, or underloaded.

The fifth way to choose resources separately is similar to the third:

```
rs_drs="avail(int)|..."
```

where `int` is a positive integer, usually 3. Calling on `KheResourceAvailableBusyTimes` and `KheResourceAvailableWorkload` (Section 4.7.1) to determine the availability of each resource, it chooses sets of resources $\{r_1, r_2, \ldots, r_n\}$ at random subject to four conditions: $n$ is the number given in the `rs_drs_resources` option; $r_1$ has negative availability; the other $r_i$ have positive availability; and the resources are assignable to most of each others' tasks.

The sixth way to choose resources separately is

```
rs_drs="similar(int)|..."
```

It is like `avail(int)`, except that availability is not taken into account: any set of the given number of resources, such that each resource is assignable to most of the others' tasks, may be chosen (at random) for reassignment.

***Choosing days separately***. The second part of a separate `rs_drs` option specifies the days that are to be included in the solve. For each call on the solver, a set of days is needed. Each day is represented by its index in the common frame: 0 for the first day, 1 for the second, and so on.

Instead of giving each day individually, as is done for resources, sequences of consecutive days are given, each by means of its *interval*: the index of its first day plus the index of its last day. One call on the solver requires one *interval set*, that is, one set of intervals.

One way to define the days is to give an explicit sequence of one or more interval sets separated by semicolons. Each interval set is a sequence of one or more intervals separated by commas. Each interval contains two integers separated by a hyphen. For example,

```
rs_drs="...|0-13;14-27"
```

requests two interval sets, `0-13` and `14-27`, while

```
rs_drs="...|5-6,12-13,19-20,26-27"
```

requests one interval set with four intervals. Assuming that the cycle begins on a Monday, this example requests that the first four weekends be reassigned.

The value `all` is equivalent to a single interval set holding a single interval which spans the entire cycle. The value `all(x)` is equivalent to a sequence of interval sets, following each other through the cycle, each containing one interval of length `x`. The last interval will have length less than `x` if `x` does not divide evenly into the cycle length. For example, `all(14)` is equivalent to `0-13;14-27;...` and so on to the end of the cycle. The value `all(x,y)` is like `all(x)` except that each interval begins `y` days further on: `all(14,7)` is equivalent to `0-13;7-20;14-27;...` and so on.

The value `choose(x)` is equivalent to a single interval set with a single interval whose length is `x` and whose first day is chosen randomly. In the algorithm above, when `choose(x)` is used, a fresh random choice is made each time the algorithm reaches its second line. If `x` exceeds the number of days in the cycle, it is reduced to that number.

Here are three examples of how separate `rs_drs` values work together to define VLSN searches of various kinds. First,

```
rs_drs="similar(3)|choose(28)"
```

(the default value of this option) generates a potentially infinite sequence of solves, each for three similar resources and an interval of 28 days chosen at random. Second,

```
rs_drs="similar(3)|all(14,7)"
```

generates a potentially infinite sequence of solves, but this time, for each set of similar resources the solves work through the cycle. Either way, the solves end when time runs out. And finally:

```
rs_drs="all(3)|all(28)"
```

This works systematically through the resources in groups of 3. For each group of 3 resources it works systematically through the cycle in groups of 28 consecutive days.

***Choosing resources and days together***. We turn now to combined values for `rs_drs`, those that do not contain a bar character. These choose sets of resources and days together, allowing each to influence the other. `KheDynamicResourceVLSNSolve` implements this algorithm when `rs_drs` has a combined value:

```
for each choice (R, D) of resources and days defined by rs_drs do
  solve for R and D;
```

By carefully crafting a combined choice, we hope to increase the chance of improvement.

The first combined value is

```
rs_drs="same_shift(max_resources, days)"
```

where `max_resources` and `days` are positive integers. First, it chooses a sequence of `days` consecutive days at random. Then it chooses an offset within the days at random (this represents a particular shift type, for example the night shift). Finally, it chooses all resources whose timetable during the selected days contains only shifts of the chosen shift type, including at least one shift of the chosen shift type, as well as zero or more free days. If the number of chosen resources exceeds `max_resources`, it is reduced to `max_resources` by deleting resources at random. The solve is then carried out for the chosen resources and days, and then the method repeats using new random choices until time runs out.

The second combined value is

```
rs_drs="defective_shift(max_resources, days)"
```

It is the same as `same_shift`, except that its starting point is a limit active intervals monitor *m* with non-zero cost, chosen at random from all limit active intervals monitors that limit the number of consecutive shifts of a specific shift type. The resource monitored by *m* is included in the chosen `max_resources` resources, one of *m*'s defective intervals is chosen at random and its days are included in the chosen `days` consecutive days, and the chosen shift type is the shift type monitored by *m*. Other choices are random, as for `same_shift`.

The third combined value is the one that selects targeted VLSN search. There is a lot to say about it, so it has been given its own section of this Guide (Section 12.6.5).

### 12.6.5. Targeted VLSN search

A natural generalization of our previous ideas, *targeted VLSN search*, is obtained by

```
khe_drs="targeted(r1:d1, r2:d2, ... , rn:dn)"
```

where `r1, d1, r2, d2, ... , rn, dn` are positive integers defining feasible neighbourhood sizes. We write them in mathematical notation as a set of pairs:

$$\{(r_1, d_1), (r_2, d_2), \ldots, (r_n, d_n)\}$$

Each $(r_i, d_i)$ means 'it is feasible to reassign up to $r_i$ resources over up to $d_i$ days.' We call it a *neighbourhood shape.* It represents practical experience with the solver. There must be at least one $(r_i, d_i)$. A reasonable value is

```
khe_drs="targeted(3:28, 4:12, 5:6, 6:3)"
```

This says that you can reassign 3 resources over 28 days, or 4 resources over 12 days, and so on. It would be reasonable to require $r_i$ to be strictly monotone increasing and the $d_i$ to be strictly monotone decreasing, but the solver does not do that.

Targeted VLSN search is a normal VLSN search for the most part: at each step, it unassigns a certain set of resources $R$ on a certain set of days $D$, and reassigns them optimally by calling the dynamic programming solver. But instead of choosing $R$ and $D$ randomly, targeted VLSN search cycles through the defects of the solution, or rather through those event resource and resource defects whose resource type is the given resource type `rt`. For each defect it builds an $R$ and $D$ around that defect whose optimal reassignment seems likely to succeed. Its choice of $R$ and $D$ is partly determined by the defect, and partly random. It then optimally reassigns those resources on those days, and then moves on to the next defect. It continues to cycle around the defects until time runs out or the list becomes empty, relying on the randomness to vary the reassignments it tries. If an optimal reassignment is successful, it rebuilds its list of defects and carries on.

The defects are sorted initially by decreasing cost. This is done so that solves that run out of time spend the time that they do have on defects whose removal would give a larger payoff.

Targeted VLSN search builds different neighbourhoods for different types of defects. Our main task, then, is to define a neighbourhood for each type of defect. But before we do that, there are several general points to make.

*Choice.* Neighbourhoods are built around specific defects, but there is still scope for randomness in their construction. In the following, the word 'choose' means 'choose randomly from a uniform distribution'. Sometimes there is no choice that satisfies the conditions placed on the choice. If that happens, the conditions are relaxed by the smallest amount that makes a choice possible. In those few cases where that still does not work, the defect is dropped from the defect list and not tried again until the list is rebuilt after a successful reassignment.

*Coherence.* Although the days and resources chosen for reassignment can be arbitrary, they need to be coherent: we don't want unrelated reassignment problems bracketed together. So we always choose a sequence of consecutive days, and we always choose a first resource $r$ and then choose other resources that are similar to $r$. (As defined above, two resources are similar when each is assignable to most of the other's tasks.)

*Focus.* An important point for any repair method, but especially for optimal reassignment, which is conceived of as a method of last resort, is that the focus needs to be on those defects that are present at the end of the solve. They will be very difficult to remove. One consequence of this is that it is probably best not to try to remove a defect completely (that is, reduce its deviation to zero) in a single step. Instead, repairs that try to reduce the deviation by one seem best.

*Shape.* It is not easy to say which neighbourhood shape $(r_i, d_i)$ would be best for a given defect. For localized defects, such as unavailable times or defective active intervals, the author's intuition is that reassigning just a few days around the defect is best, to allow as many resources as possible to be involved. For global defects, such as a workload overload, it is harder to say what is best. One can localize such a defect by choosing one busy day and building a neighbourhood around it, hoping to free up that day; or one can try a complete reassignment. Both approaches seem plausible. So rather than committing ourselves to either, we choose a neighbourhood shape at random from the list given in the `rs_drs` option. As the solver returns to a given defect repeatedly, different shapes are tried randomly.

Another point to consider when choosing $(r_i, d_i)$ is that for small $r_i$ and large $d_i$ there are relatively few distinct neighbourhoods. For example, if there are 20 resources and 28 days, then there are only $20 \cdot 19 \cdot 18 / 3 \cdot 2 \cdot 1$ neighbourhoods with shape $(3, 28)$. When $r_i$ is larger and $d_i$ is smaller there are many more neighbourhoods.

*Consecutive days.* We often need to convert a non-empty set of days into a sequence $D$ of $d_i \geq 1$ consecutive days. We do this in the following way, which we call the *usual conversion.*

The first step is to convert the initial set of days into a sequence of consecutive days, by adding to it every day after its first day and before its last day that is not already present. For example, if the initial set contains the weekend days, this step would add in the week days between the weekend days. Often the initial set is a sequence of consecutive days to begin with, and in that case this step does nothing.

Now suppose the sequence we have needs to grow by $\delta$ days, and that $a$ days are available to the left of it, and $b$ days are available to the right. If $\delta > a + b$ then it can't grow by $\delta$, so just grow it by $a$ days to the left and $b$ to the right. Otherwise, choose an amount $x$ to grow at the left and an amount $y$ to grow at the right, such that (1) $x + y = \delta$, (2) $0 \leq x \leq \min(\delta, a)$, and (3) $0 \leq y \leq \min(\delta, b)$.

There is really only one choice here, so we need to eliminate $y$ from these formulas. From (1) and (2) we find $0 \leq y \leq \delta$, so (3) can be replaced by $y \leq b$. Combining this with (1) gives $\delta - x \leq b$, that is, $x \geq \delta - b$. So altogether we need to choose an $x$ such that

$$\max(0, \delta - b) \leq x \leq \min(\delta, a)$$

and then we just take $y = \delta - x$.

If the sequence we have needs to shrink by $\delta$ days, we simply choose an $x$ such that $0 \leq x \leq \delta$, and shrink the sequence at the left end by $x$ and at the right end by $\delta - x$.

*Tasks with gaps.* The optimal reassignment solver cannot handle tasks whose busy days have gaps in them, and will refuse to come into existence when such tasks are present. So in the following, where it says 'the set of days when task $t$ is running', it is sufficient to implement this set by an interval, representing a sequence of consecutive days.

*Tasks that need assignment.* Some tasks need assignment, in the sense that there is a cost incurred if they are not assigned. `KheTaskNeedsAssignment` (Section 4.6.1) reports this condition. Neighbourhoods often take this into account. The reasoning is that if it would be better if task $t$ is not assigned to resource $r$, then if $t$ needs assignment it will probably have to be assigned to some other resource, and so the neighbourhood needs to include resources suited to $t$; whereas if $t$ does not need assignment, resources suited to $t$ are not needed.

*Task grouping.* A standard feature of KHE is that tasks may be grouped, in which case they must all be assigned the same resource (possibly `NULL`). In what follows, wherever we say 'Choose a task $t$ such that ...', we really mean 'Choose a task $t$ such that ..., then replace $t$ by its proper root task.' This ensures that task groups are handled correctly and never broken apart.

*Assignable, free, and available resources.* A resource $r$ is *assignable, free, and available for task $t$* when $r$ is assignable to $t$, free (not assigned to any tasks) during the *days* that $t$ is running, as reported by function `KheResourceTimetableMonitorFreeForTask` (Section 6.8.2), and available (not subject to any avoid unavailable times constraints) during the *times* that $t$ is running, as reported by function `KheResourceTimetableMonitorAvailableForTask` (Section 6.8.2). This reflects the fact that $r$ can be assigned at most one task per day, but it can be assigned a task at one time of a day and be unavailable at another.

*Similar resources.* We stated earlier that two resources are similar if they are assignable to many of the other's tasks. Here we tighten this up for targeted VLSN search. Let $r$ be a resource and let $r_1, \ldots, r_{m-1}$ be the other resources of its type. Let $s(r, r_i)$ be the *similarity* of $r$ and $r_i$, a value that increases as the resources become more similar. We'll make this concrete shortly.

Sort the $r_i$ by non-increasing $s(r, r_i)$, then relabel them so that $r_1$ is the most similar, and so on. Suppose that we need to choose $r$ plus $k$ other similar resources. Then choose those $k$ resources from the first $\min(2k, m-1)$ elements of $r_1, \ldots, r_{m-1}$. This returns the wanted number of resources whenever possible, chosen randomly, but favouring similar resources over less similar ones.

Often we choose a certain number of similar resources, preferring those with a certain property $P$. We do this by breaking the sorted list $r_1, \ldots, r_{\min(2k,m-1)}$ into two lists, one holding the resources with property $P$, the other holding the rest. We then choose resources at random from the first list (without replacement) until either we have all we need or we run out of resources. In the second case we choose resources at random (again without replacement) from the other list until either we have all we need or we run out of resources. Either way, we then stop.

It remains to define the similarity function $s(r, r_i)$. We want larger values to indicate that the resources are assignable to each others' tasks. We take this to mean the tasks that they are actually assigned to when we make the calculation (when the call on `KheDynamicResourceVLSNSolve` is made) rather than the tasks that they could be assigned to, because we want to be able to actually make these reassignments in the current state. So let $T(r)$ be the number of proper root tasks that $r$ is actually assigned to, and let $R(r, r_i)$ be the number of those tasks that $r_i$ could be assigned to. Then define the floating-point value

$$s(r, r_i) \ = \ \frac{R(r, r_i) + R(r_i, r)}{T(r) + T(r_i)}$$

or 0.0 when $T(r) + T(r_i) = 0$. This will be 0.0 when neither resource can be assigned to any of the other's tasks, and 1.0 when each resource can be assigned to all of the other's tasks.

We are now ready to choose a neighbourhood for each type of defect. We choose the shape, then the part of the defect to build the neighbourhood around, then the days, then the resources. Let $m$ be a defect (a monitor with non-zero cost). We'll start with event resource defects.

*Assign resource defects.* Choose $(r_i, d_i)$. Choose an unassigned task $t$ monitored by $m$. Let $D$ be the usual conversion of the set of days that $t$ is running. There is no natural first resource $r$, so proceed as follows. Let $X$ be the set of resources that are assignable, free, and available for $t$. If $X$ is empty, drop $m$. Otherwise choose one resource $r$ from $X$, then choose $r_i - 1$ similar resources, preferring resources that are assignable, free, and available for $t$.

*Prefer resources defects.* Choose $(r_i, d_i)$. Choose a task $t$ monitored by $m$ that is assigned an unpreferred resource $r$. Let $D$ be the usual conversion of the days $t$ is running. The choice of resources depends on whether $t$ needs assignment (discussed above), as follows.

If $t$ needs assignment, then in practice $m$ will have a non-empty set of preferred resources. Choose $r$ plus $r_i - 1$ similar resources, preferring resources that are assignable, free, and available for $t$ and preferred for $m$. If $t$ does not need assignment, the problem is to find a good timetable for $r$, not to find a good assignment for $t$. So choose $r$ plus $r_i - 1$ similar resources, ignoring $t$.

*Avoid split assignments defects.* These are not handled by the optimal reassignment solver, so the VLSN module will return immediately when they are present.

*Limit resources defects.* Choose $(r_i, d_i)$. Let $T_m$ be the tasks monitored by $m$, and let $R_m$ be the resources from $m$'s constraint. What we do depends on whether $m$ is overloaded (too many assignments of resources from $R_m$ to $T_m$) or underloaded (too few of those assignments).

If $m$ is overloaded, choose a task $t$ from those members of $T_m$ that are assigned a resource from $R_m$. There must be such a task. Let $D$ be the usual conversion of the days when $t$ is running. Let $t$'s assigned resource be $r$. Choose $r$ plus $r_i - 1$ similar resources, preferring resources that are assignable, free, and available for $t$ but not in $R_m$.

If $m$ is underloaded, choose a task $t$ from those members of $T_m$ that are not assigned a resource from $R_m$. If there are no such tasks, drop $m$ (it cannot be repaired). Let $D$ be the usual conversion of the days when $t$ is running. If $t$ is unassigned, choose a resource $r$ from those elements of $R_m$ that are assignable, free, and available for $t$. If there are no such resources, drop $m$. Otherwise, $t$ is assigned a resource; let $r$ be that resource. Either way, choose $r$ plus $r_i - 1$ similar resources, preferring resources that are assignable, free, and available for $t$ and also in $R_m$.

We turn now to resource defects. Let $m$ be the defective resource monitor, and let $r$ be the $m$'s resource. Our first resource will always be $r$, since we are trying to improve its timetable.

*Avoid clashes defects.* The optimal reassignment module will not build a solver when any resource has a clash (or even two assignments on the same day). When it finds a new best solution, that solution will also have no clashes. So avoid clashes defects cannot arise here.

*Avoid unavailable times defects.* Choose an $(r_i, d_i)$. Choose one task $t$ assigned $r$ when $r$ is unavailable. Let $D$ be the usual conversion of the days when $t$ is running. If $t$ needs assignment, choose $r$ plus $r_i - 1$ similar resources, preferring resources that are assignable, free, and available for $t$. If $t$ does not need assignment, choose $r$ plus $r_i - 1$ similar resources, ignoring $t$.

*Limit idle times defects.* These are not handled by the optimal reassignment solver, so the VLSN module will return immediately when they are present.

*Cluster busy times defects.* Choose an $(r_i, d_i)$. What we do depends on whether the defect is an overload (too many active time groups) or underload (too few active time groups).

If the defect is an overload, choose an active time group $g$. If $g$ is a positive time group, $r$ is busy during $g$ and it would help if it was free. Let $T$ be the set of one or more tasks assigned $r$ that are making $r$ busy during $g$. Let $D$ be the usual conversion of the days containing the times of the tasks of $T$. Choose $r$ plus $r_i - 1$ similar resources, preferring resources that are assignable, free, and available for at least one task from $T$.

If $g$ is a negative time group, $r$ is free during $g$ and it would help if it was busy. Let $D$ be the usual conversion of the days containing the times of $g$. (There are no tasks here to guide $D$'s selection.) Choose $r$ plus $r_i - 1$ similar resources, preferring resources that are busy during $g$.

If the defect is an underload, choose a non-active time group $g$. If $g$ is a positive time group, then $r$ is free during $g$ and it would help if it was busy. If $g$ is negative, $r$ is busy during $g$ and it would help if it was free. These are just the overload cases again, so proceed as before.

There is another possibility when the defect is an underload and the `allow_zero` flag is set: make $r$ completely free. So when the defect is an underload, the `allow_zero` flag is set, and $r$ has exactly one active time group, we make a Boolean choice and either proceed as described or else find the single active time group and proceed as described for an overload.

*Limit busy times defects.* Choose an $(r_i, d_i)$ such that $r_i \geq 4$. Each time group of $m$ can be defective individually, so choose one defective time group $g$. The rest depends on whether $g$ is overloaded (too many busy times for $r$ in $g$) or underloaded (too few busy times for $r$ in $g$).

If $g$ is overloaded, then we need $r$ to be less busy during $g$. Let $T$ be the set of one or more tasks assigned $r$ that are making $r$ busy during $g$. Choose one task $t$ from $T$. Let $D$ be the usual conversion of the days containing the times that $t$ is running. Choose $r$ plus $r_i - 1$ similar resources, preferring resources that are assignable, free, and available for $t$. This is like repairing a cluster busy times defect when the chosen resource group is overloaded and positive, except that here it is enough to unassign $r$ from one of the tasks of $T$, whereas with a cluster busy times defect it only helps if $r$ is unassigned from all of $T$.

An exception to this is that when the cost function is a step function, it only helps if $r$ is unassigned from all of $T$. So in that case we follow the cluster busy times defect plan exactly.

If $g$ is underloaded, then we need $r$ to be busier during $g$. Let $D$ be the usual conversion of the days containing the times of $g$. Choose $r$ plus $r_i - 1$ similar resources, preferring resources that are busy during $g$. This is like repairing a cluster busy times defect when the chosen time group is underloaded and positive.

There is another possibility when the defect is an underload and the `allow_zero` flag is set: we could make $g$ completely free for $r$. So when the defect is an underload, the `allow_zero` flag is set, and the number of $r$'s busy times during $g$ is at most 2, we make a Boolean choice and either proceed as described or else try to make $r$ completely free during `tg`, in which case we proceed as for a cluster busy times overload when we have chosen a positive time group $g$.

*Limit workload defects.* These are like limit busy times defects, only applied to workload rather than to the number of busy times, so they are handled in the same way. There is one difference. The KHE platform offers no way to tell whether a workload is close to 0.0, so when the defect is an underload and the `allow_zero` flag is set we make a Boolean choice to either increase the workload or decrease it to 0.0, without knowing whether it is close to 0.0 already.

*Limit active intervals defects.* Choose $(r_i, d_i)$. Choose one defective interval $I$. If $I$ is too long, choose one of these possibilities:

1. Let $g$ be the time group at the left end of $I$. Treat $g$ as it would be treated when handling a cluster busy times defect which is an overload.

2. Let $g$ be the time group at the right end of $I$. Treat $g$ as it would be treated when handling a cluster busy times defect which is an overload.

These two cases must exist and be distinct, otherwise $I$ has length 0 or 1 and cannot be too long. If $I$ is too short, choose one of these possibilities:

3. Let $g$ be the time group just to the left of $I$ (this choice is only open if $g$ exists). Treat $g$ as it would be treated when handling a cluster busy times defect which is an underload.

4. Treat the entire set of times of the days covered by $I$ as though it was a single time group $g$ and the defect was a cluster busy times overload. Hopefully, this will remove $I$ from $r$ altogether and redistribute it among similar resources.

5. Let $g$ be the time group just to the right of $I$ (again, this choice is only open if $g$ exists). Treat $g$ as it would be treated when handling a cluster busy times defect which is an underload.

There is an additional wrinkle here beyond what is done for cluster busy times defects. Suppose that *m* monitors a particular shift *s* on each day (the day shift, the night shift, etc.). Concretely this means that each time group of *m* contains one time, and for each time group that time has the same offset in its day. Then, when choosing the other $r_i - 1$ resources, we prefer resources whose only assignments during *D* are to *s*. The reasoning is that it will be easier to redistribute (say) a set of night shifts among these resources than a set of miscellaneous shifts.

### 12.6.6. Finding an initial solution with the dynamic solver

The ability to find an optimal assignment for a few resources opens up an interesting possibility for finding an initial solution. Instead of proceeding from left to right through the planning timetable, as done by time sweep (Section 12.7.3), we can proceed from top to bottom, one or a few resources at a time. This approach has its own solver function:

```
bool KheDynamicResourceSequentialSolve(KHE_SOLN soln,
  KHE_RESOURCE_TYPE rt, KHE_OPTIONS options);
```

It returns `true` when the cost of `soln` on exit is less than its cost on entry.

`KheDynamicResourceSequentialSolve` reads three options from `options`:

`rs_drs_seq_resources`
An integer with default value 1 holding the number of resources to solve for at each step.

`rs_drs_seq_days`
An integer with default value 28 holding the number of days to solve for at each step.

`rs_drs_seq_frac`
A floating-point number with default value 0.6 holding the fraction of the resources to assign, as explained below.

Roughly speaking, this is like calling `KheDynamicResourceVLSNSolve` with option settings `rs_drs_resources=all(rs_drs_seq_resources)` and `rs_drs_days=all(rs_drs_seq_days)`. But there is quite a lot more than this going on.

The order in which the resources are assigned is important. Resources for which good timetables are hard to find should be assigned first. `KheDynamicResourceSequentialSolve` sorts the resources by increasing non-rigidity (Section 11.4.3). Resources with equal non-rigidity are sorted by decreasing `KheResourceAvailableBusyTimes` (Section 4.7.1).

If `rs_drs_seq_frac` is less than 1.0, `KheDynamicResourceSequentialSolve` only assigns the given fraction of the resources in this way (after sorting them all). It then builds a resource group `rg` containing the remaining unassigned resources and calls

```
KheTimeSweepAssignResources(soln, rg, options)
```

to assign them. This reflects the intuition that an initial optimal assignment will utilize difficult resources fully, but that to assign every resource in this way would lead to very poor outcomes towards the end of the solve.

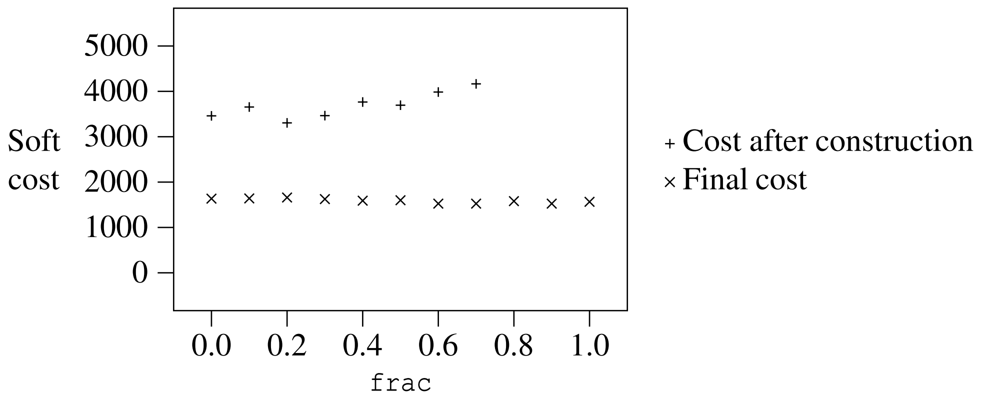
There are cases where a dynamic resource solver cannot be created owing to issues with the tasks. Rather than failing in those cases, `KheDynamicResourceSequentialSolve` passes the entire job on to time sweep.

Event resource monitors can mislead this algorithm. For example, suppose that on every day there is a task subject to a hard assign resource constraint, and that all the resource constraints are soft. Then the first resource assigned will be assigned a task on every day, which is mad.

`KheDynamicResourceSequentialSolve` compensates for this by adjusting the weights of all event resource monitors at the start. Hard monitors are given soft cost 2, and soft monitors are given soft cost 1. This way of compensating for resources that have not been assigned yet is analogous to, but more simple-minded than, time sweep's way of compensating for times that have not been assigned yet. These changes are undone after the optimal resource assigning is finished, before any call to time sweep.

With event resource constraints drastically weakened in this way, the assignment of one resource has very little influence over the assignment of another. Accordingly it is probably best to leave option `rs_drs_seq_resources` at its default value, 1.

Here is a graph of solution cost (after construction and final) when solving instance INRC2-4-100-0-1108 for various values of `rs_drs_seq_frac`, called `frac` here:



Cost after construction is quite variable, tending to a minimum when `frac` is 0.2. The values beyond 0.7 have been omitted because they include a hard cost. The final costs are the best of twelve 5-minute full runs. They are more uniform, but the best of them (1525 when `frac` is 0.6, 0.7, or 0.9) is better than either method alone (1635 when `frac` is 0.0, 1565 when `frac` is 1.0).

The construction phase running time on this instance is 1.2 seconds when `frac` is 0.0 (so no dynamic solver is created). It then grows from 2.9 seconds to 3.4 seconds as `frac` increases from 0.1 to 1.0. These times do not matter if the total time limit is, say, 5 minutes.

There is some evidence here that using dynamic programming on rigid resources and time sweep on non-rigid ones is better than either method alone. The author carried out a larger test, of twelve 5-minute full runs on each of the 20 INRC2-4 instances. The average cost without dynamic programming was 2383. Using 0.6 for `frac` (the best value tried), the average cost with dynamic programming was 2345. The difference is 38, which is presumably significant.

### 12.6.7. Testing

The module that implements `KheDynamicResourceVLSNSolve` also offers

```
void KheDynamicResourceVLSNTest(KHE_SOLN soln, KHE_RESOURCE_TYPE rt,
  KHE_OPTIONS options);
```

This creates a solver, calls it multiple times, accumulates some statistics, and prints them to one or more output files in the form of Lout source files for graphs, which the user can convert later to EPS files using Lout. It only works when the `TESTING` compiler flag, defined near the top of source file `khe_sr_dynamic_resource.c`, is set to 1. It also requires the user to have previously created a sub-directory of the current directory called `stats`. This is where the output files go.

To ensure that `KheDynamicResourceVLSNTest`'s calls on the solver all start from the same solution, `KheDynamicResourceVLSNTest` calls `KheDynamicResourceSolverTest` rather than `KheDynamicResourceSolverSolve`. As explained above, this is the same except that it never changes the solution; so `KheDynamicResourceVLSNTest` never does either.

The options of `KheDynamicResourceVLSNSolve` that have capital letters are used by `KheDynamicResourceVLSNTest` as default values. However, the double-loop algorithm given above for `KheDynamicResourceVLSNSolve` is no part of `KheDynamicResourceVLSNTest`, so only the first element of the sequence of sets of resources defined by `rs_drs_resources` is used by `KheDynamicResourceVLSNTest`, and similarly only the first element of the sequence of sets of days defined by `rs_drs_days` is used by `KheDynamicResourceVLSNTest`.

The data generated by one call to `KheDynamicResourceVLSNTest` has a multi-dimensional structure, specified by the `rs_drs_test` option that we will come to shortly. If there are $k$ dimensions of lengths $d_1, d_2, \ldots, d_k$, then $d_1 \times d_2 \times \cdots \times d_k$ tests will be carried out, one for each point in this $k$-dimensional structure. We'll see that different choices for the parameters may be specified at each point along each dimension, allowing these choices to be compared.

Imagine for the moment that each test produces a single number for its result. The value produced by the test at position $(c_1, \ldots, c_k)$ in the multi-dimensional structure, where $1 \le c_i \le d_i$ for $1 \le i \le k$, appears as the $c_k$th point in the $c_{k-1}$st data set of the graph named $(c_1, \ldots, c_{k-2})$.

In reality, one test does not produce a single number for its result, for two reasons. First, function `KheDynamicResourceSolverSolveStats` (documented above) returns three numbers: an integer number of solutions made, an integer table size, and a real-valued running time in seconds. To handle this, `KheDynamicResourceVLSNTest` produces three entire families of graphs, one for solutions made, one for table size, and one for running time.

Second, each test produces a sequence of numbers, one for each day. The user has two options here. By default, what is recorded is the total number of solutions made over all days, the total table size over all days, and the final running time. The other option is to use the `Z` notation, explained below, which adds an extra dimension at the end, along which the numbers for each day are placed. Running time is cumulative; solutions made and table size are not.

To specify the multi-dimensional structure, the `rs_drs_test` option is used. Its default value is `none`, meaning 'perform no testing'. Otherwise its general format is

```
dim "|" dim { "|" dim }
```

Each `dim` specifies one dimension. There must be at least two dimensions, as shown. Each `dim` consists of a sequence of one or more positions, separated by semicolons:

```
dim ::= pos { ";" pos }
```

Each position specifies one position along its dimension, and consists of zero or more options, separated by colons:

```
pos ::= [ option { ":" option } ]
```

Each option consists of a `capital_letter` and a `value`:

```
option ::= capital_letter value
```

The capital letter must be one of those given in parentheses earlier, and specifies which option is being set: `Q` means `priqueue`, and so on. The `value` must be a legal value for that option as defined above. For example, `Qtrue` and `Qfalse` are the only legal values for `Q`. A test of the priority queue might look like this:

```
rs_drs_test="Qfalse;Qtrue|Z"
```

This has two dimensions, so it produces one graph (actually three, as explained above), showing one data set for when the queue is not used, and one for when it is used.

The `Z` dimension is special and indicates that the last dimension is to contain the results for each day. When `Z` is used it must be given alone as the last dimension. As explained above, when `Z` is not used the graphs contain only a single number for each test.

In general, a `value` may contain any legal value for its option, as defined earlier for each option. The exceptions are `R` and `D`, where the option's value may define a sequence of sets of resources or days, as in

```
rs_drs_resources="0,1,2;3,4,5"
```

but `R` and `D` may define only a single set of resources or days, as in `R0,1,2`. We still allow values of the form `choose(x)` for `R` and `D`, but within `KheDynamicResourceVLSNTest` the first value produced by any given `choose(x)` is used every time a choice is made.

There is not much use for more than two dimensions. One is to include default option values in `rs_drs_test` rather than inheriting them from the other options. For example, although

```
rs_drs_test="Rchoose(3):D0-13|Qfalse;Qtrue|Z"
```

is three-dimensional, the first dimension has only one position so it still produces only one graph. The test at point $(c_1,\dots,c_k)$ takes the default values of all options, applies the modifications given in $c_1$, then the modifications given in $c_2$, and so on, so options given in a singleton first dimension apply to all tests.

A good reason for placing options into `rs_drs_test` rather than inheriting them from other options is that only options that appear explicitly within `rs_drs_test` are printed as labels on the graphs. So if you want the graph to say how many resources are being reassigned (as you probably do), then an `R` option is needed within `rs_drs_test`.

Some options, currently `rs_drs_daily_expand_limit`, `rs_drs_daily_prune_trigger`, `rs_drs_resource_expand_limit`, and `rs_drs_dom_approx`, take away the guarantee of optimality when they have non-default values. When they all have their default values, all tests with the same resources and days but differences in other parameters should return solutions with the same cost. `KheDynamicResourceVLSNTest` checks that they do, using the `*cost` parameter of each call to `KheDynamicResourceSolverTest` to find out what its result's cost was.

## 12.7. Resource matching

Consider the tasks running at some time *t*. Each task can be assigned at most one resource. Assuming the resources have hard avoid clashes constraints, each resource can be assigned to at most one of the tasks. So the assignments to these tasks form a matching in the bipartite graph with tasks for demand nodes, resources for supply nodes, and feasible assignments for edges.

Consider an initial state in which none of the tasks running at time *t* is assigned. For each edge in the bipartite graph, carry out the indicated assignment, label the edge with the cost of the solution after the assignment is made, and then remove the assignment. The result is a bipartite graph with edge weights representing the badness of each individual assignment.

Assuming that all tasks have hard assign resource constraints, a maximum matching of minimum cost in this graph will be a very desirable assignment. Indeed, it will often be optimal. This can be seen by examining all constraint types: each is either unaffected by the assignment, or else its effect is independent for each edge, so that the edge weights are valid in combination as well as individually. *Resource matching* is KHE's name for this general idea.

There is one constraint whose effect is not independent for each edge: the limit resources constraint from employee scheduling. Resource matching handles this constraint specially, as described in Section 12.7.2. This special arrangement is exact (preserves optimality) in many common cases, but in general it is merely heuristic. The resource assignment invariant is another problem: it may hold for each element of a set of assignments individually, but fail on the whole set. However, this does not seem to be a problem in practice.

Not all tasks have hard assign resource constraints. In nurse rostering, for example, a shift requiring between 3 and 5 nurses is modelled by an event with 5 tasks, only 3 of which have assign resource constraints. Fortunately, missing assign resource constraints are easily handled. For each task, add a supply node, linked only to that task, representing non-assignment of the task. The edge weight is just the initial solution cost, because choosing that edge changes nothing.

As described, resource matching constructs assignments; it does not repair them. However, a repair algorithm is easily made from it: choose a time *t*, unassign all the tasks running at that time, reassign them using resource matching, and then either keep the new assignments if they improve the solution, or revert to the original assignments if they do not.

During initial construction it may be that some tasks are already assigned, and what is wanted is to assign the unassigned ones without disturbing the assigned ones. In that case, simply omit the demand nodes for assigned tasks.

Instead of selecting all tasks running at a single time *t*, KHE's implementation selects all tasks whose times overlap with an arbitrary set of times $T$. For $|T| \geq 2$, this does not make sense in general, because one resource could be assigned to two or more of the tasks, and the rationale for using matching is lost. However, there are at least two cases where it does make sense.

First, when $T$ is a time group from the common frame (Section 5.10), hard limit busy times constraints prohibit resources from being assigned to two or more tasks that overlap $T$.

Second, when resource matching is used for repair, KHE's version of it specifies that tasks which are assigned the same resource at the start must be assigned the same resource at the end. Of course, this does not produce an optimal reassignment of the tasks, because it requires some tasks to be assigned to the same resources. However, minimum cost weighted matchings can be found in polynomial time, whereas true optimal reassignment is NP-complete.

### 12.7.1. A solver for resource matching

This section presents a solver for resource matching. It can be used directly via the interface given in this section, or indirectly via the applications given in the following two sections.

One solver may be used for many solves. To create and delete a solver, call

```
KHE_RESOURCE_MATCHING_SOLVER KheResourceMatchingSolverMake(
  KHE_SOLN soln, KHE_RESOURCE_GROUP rg, HA_ARENA a);
void KheResourceMatchingSolverDelete(KHE_RESOURCE_MATCHING_SOLVER rms);
```

The deletion really only happens when arena `a` is deleted or recycled; but before then a call to `KheResourceMatchingSolverDelete` is needed to carry out some tidying up (there are group monitors to remove).

The solves have one supply node for each resource of `rg`, plus supply nodes representing non-assignment. Typically, `rg` would be `KheResourceTypeFullResourceGroup(rt)` for some resource type `rt`, but it can be any resource group. It is fixed for the lifetime of the solver.

To carry out one solve, call

```
bool KheResourceMatchingSolverSolve(KHE_RESOURCE_MATCHING_SOLVER rms,
  KHE_RESOURCE_MATCHING_DEMAND_SET rmds, bool edge_adjust1_off,
  bool edge_adjust2_off, bool edge_adjust3_off, bool edge_adjust4_off,
  bool ejection_off, KHE_OPTIONS options);
```

If this can find a way to improve the solution, it does so and returns `true`. Otherwise it leaves the solution unchanged and returns `false`. Parameter `rdms` is the set of demand nodes to match against the supply nodes already present in `rms`; how to construct it is explained below. The other parameters affect the detailed behaviour of the solver, as follows.

When `true`, parameters `edge_adjust1_off`, `edge_adjust2_off`, `edge_adjust3_off`, and `edge_adjust4_off` turn off the four edge adjustments. These adjust edge costs so that, in cases which would otherwise be tied, resources with certain properties are preferred, as follows.

Edge adjustment 1 gives preference to resources with a larger number of available times (Section 4.7) over resources with a smaller number. This seems likely to be the most effective form of edge adjustment, so it is given twice the weight of the others.

Edge adjustment 2 gives preference to resources whose assignment brings a smaller number of constraints from below their maximum values to their maximum values. Hopefully this will keep more resources available for assignment for longer.

Edge adjustment 3 tracks the number of consecutive assignments to the resource in recent solves, and favours resources for which this is smaller. This encourages smaller sequences of consecutive assignments, which hopefully will give more flexibility when repairing later.

Edge adjustment 4 tracks the time of day of the most recent assignment to the resource, and favours assignments that repeat that time of day. This encourages sequences of shifts of the same type. These always seem to be acceptable in nurse rostering, and they are often preferable.

At the end of the call, if limit resources monitors are involved and any of them have non-zero cost, function `KheEjectionChainRepairInitialResourceAssignment` is called to repair them. This call is omitted if parameter `ejection_off` is `true`.

Three options from `options` are consulted. Option `rs_invariant` determines whether the resource assignment invariant is in effect, as usual. If it is, only individual edges that preserve the invariant are included in the graph, and if, when the solution is changed to reflect the minimum matching, any of the individual assignments fail the invariant, those assignments are omitted. Option `gs_common_frame` supplies the common frame, needed even when edge adjustment is not in effect, for ejecting task moves. Finally, the first time that `rmds` is solved, option `gs_event_timetable_monitor` (Section 8.4), which must be present, is used to obtain efficient access to the tasks which overlap its times.

A demand set is constructed by a sequence of calls beginning with

```
KHE_RESOURCE_MATCHING_DEMAND_SET KheResourceMatchingDemandSetMake(
  KHE_RESOURCE_MATCHING_SOLVER rms);
```

and continuing with any number of calls to

```
void KheResourceMatchingDemandSetAddTime(
  KHE_RESOURCE_MATCHING_DEMAND_SET rmds, KHE_TIME t);
void KheResourceMatchingDemandSetAddTimeGroup(
  KHE_RESOURCE_MATCHING_DEMAND_SET rmds, KHE_TIME_GROUP tg);
void KheResourceMatchingDemandSetAddFrame(
  KHE_RESOURCE_MATCHING_DEMAND_SET rmds, KHE_FRAME frame);
```

in any order. These define a set of times $T$: the union of the times `t`, the time groups `tg`, and the time groups of `frame`. $T$ may not be empty.

`KheResourceMatchingDemandSetMake` formerly had a `preserve_existing` flag which, when `true`, caused existing assignments to be preserved, by omitting assigned tasks. This has been withdrawn, partly because it was not being enforced when the resource matching called the ejection chain solver to repair its work, and partly because it now seems better to the author for the solver that added those assignments to be the one to decide whether they need to be preserved (including for how long), and to enforce its decisions using `KheTaskAssignFix`.

A demand set may be saved, and solved multiple times. When it is no longer needed it may be deleted explicitly, by calling

```
void KheResourceMatchingDemandSetDelete(KHE_RESOURCE_MATCHING_DEMAND_SET rmds);
```

Alternatively, deleting its solver's arena will also delete it, because it is stored in that arena. A less drastic alternative to deletion is

```
void KheResourceMatchingDemandSetClear(KHE_RESOURCE_MATCHING_DEMAND_SET rmds);
```

which clears out `rmds` ready for a fresh lot of times.

The demand nodes of one demand node set are specified in two steps: first the tasks to include, called the *selected tasks*, are specified, then the grouping of those tasks into demand nodes. A task `t` is selected (its assignment may be changed) when it satisfies these conditions:

1. It has the same resource type as the solver's `rg` attribute;

2. It is either assigned directly to the cycle task of a resource of `rg`, or else it is unassigned;

3. It, or some task assigned directly or indirectly to it, lies in a meet which is assigned a time, directly or indirectly, so as to cause the task to share at least one time with *T*;

4. It is not derived from a preassigned event resource;

5. Its assignment is not fixed (by `KheTaskAssignFix`, or because it is a cycle task);

6. Not assigning it might attract a cost. All tasks subject to assign resource constraints of non-zero cost are included. Some tasks subject to limit resources constraints with minimum limits are also included, chosen heuristically, as explained in the depths of Section 12.7.2.

The last item is a compromise. If too few tasks are included, the assignment will be too far from final to be useful; but if too many are included (including tasks for which assignment is not needed), then if the resources have minimum workload limits these will favour assigning all these tasks, over-using the resources early in the cycle and causing workload shortages later.

For each resource `r` of `rg` there is one demand node containing all selected tasks which are initially assigned `r`. If there are no such tasks, there is no such node. There is also one demand node for each of the remaining selected tasks. (We are speaking of logical demand nodes here; as the next section explains, equivalent logical demand nodes are grouped into single nodes by the implementation, for efficiency.)

The supply nodes of one solve consist of one for each resource `r` of `rg`, representing assignment of `r`, and one for each demand node, representing non-assignment of its tasks. (Again, these are logical supply nodes; in the implementation, all supply nodes representing non-assignment are grouped into a single supply node.)

When determining which edges are present and their weights, the first step is to unassign every initially assigned selected task using `KheTaskUnAssign`. This must succeed, because the selected tasks are not fixed. Then, for each demand node `d`, for each supply node `s` representing assignment of a resource `r`, draw an edge between `d` and `s` when the tasks of `d` can be assigned `r`. This is tested by calling `KheEjectingTaskMoveFrame` for each task of `d`; an edge is added when all these calls succeed. The edge cost is the solution cost after they are done, optionally with edge adjustment as described above. There is also an edge from `d` to the supply node `s` representing non-assignment of the tasks of `d`, whose cost is the (unchanged) solution cost.

### 12.7.2. Implementing resource matching

This section describes the implementation of resource matching in detail.

As mentioned earlier, limit resources constraints (or rather monitors) are a problem for resource matching, because they take away the independence of the edge weights. Suppose that on the current day there is a requirement for at least one senior nurse. If no special arrangements are made, every edge to a non-senior nurse will carry a cost. That is not right, because only one task needs a senior nurse. This problem strongly influences the implementation.

Resource matching detaches all limit resources monitors that affect the current match and replaces them by adjustments to the edge weights. This restores the lost independence. These adjustments are often *exact*: they have the same effect on cost as the monitors. When they are not exact, resource matching loses its local optimality, although it is still a good heuristic.

The algorithm has two parts. The first part, *preparation*, builds the demand nodes and does a few other things explained below. It has three phases. The second part, *solving*, adds the edges, finds the matching, and makes the assignments. A demand set may be solved repeatedly, but it is prepared only once, just before it is solved for the first time.

*Preparation (first phase): find and group selected tasks.* A *selected task* is a task that may be assigned by the current match. An *affected task* is a task whose assignment is affected by the current match: it is either selected, or it is assigned, directly or indirectly, to a selected task (its *selected task*). For example, if a Saturday night task is grouped with a Sunday night task, then when solving either Saturday or Sunday, one of the tasks is affected and selected and the other is affected but not selected.

Given the demand set's set of times $T$, the selected tasks are easily found. For each time in $T$, use the event timetable monitor from option `gs_event_timetable_monitor` to find the meets running at that time. For each task of the wanted type in each meet, follow its chain of assignments to its proper root. By the way it was found, the proper root must satisfy conditions 1, 2, and 4; if it also satisfies conditions 3, 5, and 6, then make it a selected task. Condition 7 (concerning the cost of non-assignment) is not checked here; that will be done later.

A selected task might be encountered more than once while doing this. So to finish this step, the array of selected tasks it builds is sorted and uniqueified.

The next step is to traverse the uniqueified array of selected tasks, doing two things. First, if several selected tasks are assigned the same resource when resource matching is called, the specification states that they should be assigned the same resource by resource matching. So a task grouper (Section 11.7) is used to group these tasks. In each group, the leader task remains selected but its followers are assigned to it, demoting them to affected but not selected. From now on, 'selected' means 'selected after grouping'. The grouping is removed at the end of the solve: the follower tasks are then assigned directly to whatever the leader task is assigned to. Second, each selected task is placed into its own *demand node*, a node of the bipartite graph.

*Preparation (second phase): find task profiles and merge equivalent nodes.* One selected task per node would work. But many tasks are *equivalent*: for each resource $r$, assigning $r$ to one of these tasks has the same effect as assigning $r$ to another. Given that the following calculations are not cheap and that underlying the weighted bipartite matching algorithm is a flow algorithm, able to handle multiple equivalent nodes as single nodes with multiplicities represented by edge capacities, it makes sense to merge nodes containing equivalent tasks into a single node whose incoming edge has its number of tasks as a capacity limit. This phase does this.

Determining whether selected tasks are equivalent is done by building a *task profile* for each, such that two tasks are equivalent if their profiles are equal. A selected task's profile depends on the task itself and on the tasks assigned to it, directly or indirectly. It consists of the set of times occupied by those tasks, their total workload, and a set of *preferences*. A preference is a pair $(g_i, c_i)$, where $g_i$ is a set of resources, and $c_i$ is a cost. Its meaning is that the resources of $g_i$ are preferred for this task, and assigning something not in $g_i$ attracts cost $c_i$.

For convenience of presentation, an artificial resource $r_0$ is defined, such that assigning $r_0$ to some task means non-assignment of that task. A preference's $g_i$ may include $r_0$.

At the start of this phase, each node contains a single task. It also has a task profile attribute, which is now initialized to the task profile for the node's sole task, $t$ say. This is done by traversing $t$, the tasks assigned to $t$, the tasks assigned to those tasks, and so on recursively.

While doing this, the set of times occupied by those tasks, and their total workload, are added to the profile. Also, for each point where an assign resource or prefer resources monitor $m$ monitors a task $t'$ which is either $t$ or assigned directly or indirectly to it, one preference $(g_i, c_i)$ is added:

- If $m$ is an assign resource monitor, $g_i$ is the full set of resources of the task's resource type, and $c_i$ is the duration of $t'$ multiplied by the weight of $m$'s constraint.
- If $m$ is a prefer resources monitor, $g_i$ is $m$'s resource group plus $r_0$, and $c_i$ is the duration of $t'$ multiplied by the weight of $m$'s constraint.

These preferences express the effect of these monitors exactly. A prefer resources constraint does not penalize non-assignment, which is why $r_0$ is included.

Two preferences with the same set of resources may be merged into one, whose cost is the sum of the two original costs. These merges are done as preferences are added to profiles.

After the traversal of the affected tasks of selected task $t$ ends, the preferences in $t$'s profile are sorted, to expedite comparing profiles. After the profiles are done, the nodes are sorted to bring nodes with equal profiles together, then adjacent nodes with equal profiles are merged.

*Preparation (third phase): add preferences representing limit resources monitors.* This phase adds preferences representing limit resources monitors. The representation is often exact, and when it isn't, it is usually close.

While preferences representing assign resource and prefer resources monitors were being added to task profiles in the previous step, a list of all limit resources monitors that monitor affected tasks was built. This list is now sorted and uniqueified. Each monitor on it is then visited and preferences are added to represent it.

Before visiting the first limit resources monitor, all affected tasks are visited, and the back pointer in each is set to its selected task. (All that is actually needed is a boolean mark to indicate that the task is affected.) After the last limit resources monitor is visited, the affected tasks are visited again and their back pointers are cleared.

Handling one limit resources monitor $m_i$ proceeds in two steps. In the first step, several quantities are calculated: the total duration $N$ of the affected tasks monitored by $m_i$; lower and upper limits $L$ and $U$ on the total duration of these tasks which may be assigned resources from its resource group $g_i$ without incurring a penalty; and for each selected task $t$, its *monitored duration* $t_d$: the total duration of its affected tasks that are monitored by $m_i$. In the second step, preferences are added based on these quantities.

The first step proceeds as follows. $N$ is initialized to 0, and $L$ and $U$ to $m_i$'s minimum and maximum limits. If $m_i$ has no minimum limit, $L$ is set to 0. If $m_i$ has no maximum limit, $U$ is set to a very large number. For each selected task $t$, $t_d$ is set to 0.

Now $m_i$ may monitor non-affected tasks as well as affected ones. In practice, limit resources monitors always limit what is happening at a particular moment in time, so non-affected tasks might seem to be not a live issue. But consider the grouped Saturday and Sunday tasks above. While matching Saturday, there may well be a limit resources monitor which monitors the Sunday task and also other, non-affected Sunday tasks.

A complete traversal of the tasks monitored by $m_i$ is carried out. For each task, the back pointer set earlier tells whether the task is affected by the current match or not. If it is affected, its duration is added to $N$ and also to the monitored duration of its selected task. If it is not affected, there are two cases. If it is assigned, directly or indirectly, to a resource from $g_i$, then its duration is subtracted from both $L$ and $U$. If it is not assigned to any resource, directly or indirectly, then its duration is subtracted from $L$ only. This is analogous to how history is handled by cluster busy times and limit active intervals monitors.

After this, if $L$ is negative, set it to 0, and if it exceeds $N$, set it to $N$. Do the same for $U$. After that we must have $0 \leq L \leq U \leq N$. Here $L \leq U$ is an invariant of this whole step, established by a requirement of the limit resources constraint and preserved as the step proceeds.

That concludes the first step in the handling of limit resources monitor $m_i$, the calculation of $N$, $L$, $U$, and the $d_t$. The second step adds preferences to demand nodes, as follows.

Selected tasks with total monitored duration at least $L$ should be assigned resources from $g_i$, so find selected tasks $t$ whose total monitored duration is as large as possible not exceeding $L$, and add preference $(g_i, w_i d_t)$ to their nodes, to encourage these assignments. Similarly, selected tasks of monitored duration at least $N - U$ should not be assigned resources from $g_i$, so find selected tasks $t$ whose total monitored duration is as large as possible not exceeding $N - U$, and add preference $(G \cup \{r_0\} - g_i, w_i d_t)$ to each of them, to discourage these assignments. Each demand node receives at most one preference derived from $m_i$, since $L + (N - U) \leq N$.

Given that all the tasks in one node share the same preferences, it may be necessary to split nodes while doing this. Only one of the two resulting nodes receives the new preference.

Which demand nodes should these preferences be added to? It is easy to add preferences derived from assign resource and prefer resources monitors, because the tasks and hence the nodes are determined; but here the selected tasks must be chosen, from the selected tasks with positive monitored duration.

Nodes need to be chosen whose preferences are as similar as possible to the new preference that will come in. It would be wrong to encourage some resources with one preference and a completely different set of resources with another. This will be investigated further below. For now, it is assumed that there is an integer *compatibility* for each node with respect to $m_i$, such that when compatibility is high, adding preference $(g_i, w_i d_t)$ works well, and when it is low, adding preference $(G \cup \{r_0\} - g_i, w_i d_t)$ works well.

The algorithm for adding limit resources preferences, then, is as follows. Calculate the compatibility of each node, and store it in the node. Sort the nodes into decreasing order of compatibility. Consider the tasks as forming a single sequence, beginning with the tasks in the first node, then the second, and so on. Ignoring tasks of zero monitored duration, find the largest prefix of this sequence whose tasks have total monitored duration at most $L$, and ensure that preference $(g_i, w_i d_t)$ applies to each task $t$ of them and to no other tasks. This may involve some node splitting, as mentioned earlier. Then find the largest suffix of this sequence whose tasks have monitored duration at most $N - U$, and ensure that preference $(G \cup \{r_0\} - g_i, w_i d_t)$ applies to each task $t$ of them and to no other tasks. Again, this may require some node splitting.

The implementation is slightly different. At each node $n$, it builds a set $A$ of tasks from $n$. First it adds to $A$ the first task it can find whose monitored duration is non-zero and would not cause the target (initially either $L$ or $N - U$) to be exceeded. Then it adds to $A$ as many more tasks from $n$ as it can, subject to them all having the same monitored duration as the first, and collectively not exceeding the target. After this is done, if $A$ is empty it proceeds to the next node. If $A$ contains every task of $n$ it adds the new preference to $n$, updates the target, and proceeds to the next node. Otherwise it makes a new node holding copies of $n$'s preferences plus the new preference, moves the tasks of $A$ to it, updates the target, then restarts on $n$. It does the $N - U$ preferences before the $L$ ones, because this is slightly simpler to implement, given that new nodes go on the end of the sorted sequence of nodes.

A formula is needed for the compatibility of a preference $(g_i, w_i d_t)$ with a node $n_j$. Let the intersection of the resource groups of the preferences already present in $n_j$ be $G_j$. If there are no preferences, $G_j = G \cup \{r_0\}$, where $G$ is the full set of resources. Finding a suitable formula is a rather puzzling problem; the author's current choice is

$$\frac{|G_j \cap g_i|}{|G_j|}$$

or 0 if $|G_j| = 0$ (unlikely). This reaches its maximum value, 1, when $G_j \subseteq g_i$, which is reasonable since adding $(g_i, w_i d_t)$ to $n_j$ does not reduce $G_j$, and its minimum value, 0, when $G_j \cap g_i$ is empty.

Although the result will in general be heuristic, not exact, the difficulty should not be overstated. A typical example might be (a) 5 or 6 nurses, with (b) at least one senior nurse and (c) at most two trainee nurses. For cases like this, a simple heuristic should do very well.

Let $S$ be the senior nurses and $T$ be the trainee nurses. Constraint (a) adds preferences of the form $(G, w_i)$, where $G$ is the full set of resources, to 5 tasks, leaves one task untouched, and adds preferences of the form $(\{r_0\}, w_i)$ to the remaining tasks; (b) adds one preference of the form $(S, w_i)$; and (c) adds preferences of the form $(G \cup \{r_0\} - T, w_i)$ to all but two of the tasks. It is easy to verify that (b) will prefer nodes containing $(G, w_i)$, while (c) will prefer nodes containing $(\{r_0\}, w_i)$, and also nodes containing $(S, w_i)$, since $S \cap (G \cup \{r_0\} - T) = S$, because $S$ and $T$ are disjoint.

Set operations are slow, so four optimizations are used. First, in preferences derived from assign resource constraints, $g_i$ is in fact `NULL`. This is because it has no effect on intersections (except by omitting $r_0$, but $r_0$ is handled separately). Only non-`NULL` sets of resources need to be intersected. Second, an intersection is only performed when it is actually needed: when a task profile already contains at least two preferences with non-`NULL` sets of resources, and a third is being considered for adding to it. That makes three non-`NULL` sets—quite unlikely in practice. Third, only the size of the intersection in the formula is calculated, not the actual set. And fourth, intersections are stored as resource sets (Section 5.9), which are cheaper than resource groups.

Consider any demand node $d$. Suppose that every preference in its profile contains $r_0$. This means that, after careful consideration, preparation has concluded that not assigning $d$'s tasks would not incur a cost. As explained earlier, it is better not to include such tasks at all, because they could over-use resources early in the cycle. Accordingly, such nodes $d$ are now deleted.

Finally, preparation ends with the deletion of preferences derived from assign resource and prefer resources monitors (they are not needed by solving, as explained below). The results of preparation are stored in the demand set object: the demand nodes with their tasks, profiles, and preferences; the uniqueified list of relevant limit resources monitors; and the task grouper recording which tasks have to be assigned the same resource.

*Solving.* Solving is much easier to describe than preparation. Group tasks as indicated by the task grouper. Detach the limit resources monitors. From each demand node, add an edge of capacity 1 to each supply node representing a resource, and an edge of unlimited capacity to the supply node representing non-assignment. Find a maximum matching of minimum cost and make the assignments indicated by it. Reattach the limit resources monitors. Ungroup the task grouper. If parameter `ejection_off` is `false` and limit resources monitors were involved and any of them have non-zero cost, call `KheEjectionChainRepairInitialResourceAssignment` to repair them. After that, using a mark, if the solution is not improved, undo the assignments.

The cost of the edge from demand node $d$ to the supply node for resource $r$ (possibly $r_0$) is the cost of the solution after that one assignment is made. In addition, to compensate for the detached limit resources monitors, for each preference $(g_i, c_i)$ in $d$ derived from a limit resources constraint such that $g_i$ does not contain $r$, $c_i$ is added to the edge cost. Edge costs are not affected by preferences derived from assign resource and prefer resources monitors, because those monitors are not detached. Their preferences are needed for task equivalence and to guide the placement of preferences derived from limit resources monitors, but they are not used when solving, so they can be and are deleted at the end of preparation. There are also the separate adjustments described earlier, the ones controlled by parameters `edge_adjust1_off`, `edge_adjust2_off`, `edge_adjust3_off`, and `edge_adjust4_off`.

When there are no limit resources monitors, the algorithm does not waste time on work inspired by them: grouping equivalent tasks using task profiles is a valuable optimization in any case, and the third phase of preparation does nothing. Whether the preparation time spent on limit resources monitors is significant is a question that can only be answered definitely by testing, but the running time is probably dominated by solving, in which case the answer is no.

This section sheds light on how event resource constraints should be modelled. It is best in principle to use assign resource and prefer resources constraints, because they affect each task independently. But if they are replaced by equivalent limit resources constraints, this algorithm will produce the same matching graph. This opens a path to a useful generalization—the expression of all event resource constraints by limit resources constraints—by showing that the efficiency advantage of assign resource and prefer resources constraints need not be lost.

### 12.7.3. Time sweep resource assignment

In a planning timetable whose columns represent times and whose rows represent resources, resource packing proceeds vertically: it assigns one row after another. *Time sweep* proceeds horizontally, assigning one time (that is, the tasks running at that time) after another. This is likely to be useful in nurse rostering, where many constraints link nearby times.

KHE offers this function for time sweep resource assignment:

```
bool KheTimeSweepAssignResources(KHE_SOLN soln, KHE_RESOURCE_GROUP rg,
  KHE_OPTIONS options);
```

Using resource matching, it assigns resources to those tasks of `soln` whose resource type is that of `rg`, and which are initially unassigned. It does not disturb any existing assignments. For how it handles fixed and preassigned tasks, and other such details, see Section 12.7.1.

`KheTimeSweepAssignResources` obtains a frame from `KheFrameOption` (Section 5.10). It visits each time group of the frame in chronological order, and uses one resource matching to assign or reassign the tasks which overlap this time group. It is influenced indirectly by the resource matching options, and directly by these options:

`rs_time_sweep_daily_time_limit`
A string option defining a soft time limit for each day. The format is the one accepted by `KheTimeFromString` (Section 8.1): `secs`, or `mins:secs`, or `hrs:mins:secs`. There is also the special value `-`, meaning 'set no limit'. The default value is `3`, that is, three seconds per day.

`rs_time_sweep_edge_adjust1_off`
A Boolean option which, when `true`, causes edge adjustment 1 to be turned off, by passing `true` to `KheResourceMatchingSolverSolve` for `edge_adjust1_off`.

`rs_time_sweep_edge_adjust2_off`
A Boolean option which, when `true`, causes edge adjustment 2 to be turned off, by passing `true` to `KheResourceMatchingSolverSolve` for `edge_adjust2_off`.

`rs_time_sweep_edge_adjust3_off`
A Boolean option which, when `true`, causes edge adjustment 3 to be turned off, by passing `true` to `KheResourceMatchingSolverSolve` for `edge_adjust3_off`.

`rs_time_sweep_edge_adjust4_off`
A Boolean option which, when `true`, causes edge adjustment 4 to be turned off, by passing `true` to `KheResourceMatchingSolverSolve` for `edge_adjust4_off`.

`rs_time_sweep_ejection_off`
A Boolean option which, when `true`, causes ejection chain repair to be turned off, by passing `true` to `KheResourceMatchingSolverSolve` for `ejection_off`.

`rs_time_sweep_lookahead`
An integer option which, when it has a positive value *k*, causes time sweep to look ahead *k* time groups when calculating edge costs. A full description appears below (Section 12.7.4). The default value, 0, produces no lookahead.

`rs_time_sweep_cutoff_off`
A Boolean option which, when `true`, causes cutoff times to be omitted. When `false`, cutoff times are installed in all cluster busy times and limit active intervals monitors for the resources of `rg`, making them ignore all time groups after the largest time of the current time group. Cluster busy times monitors that request their resources to be busy at specific times, as reported by `KheMonitorRequestsSpecificBusyTimes` (Section 12.4.1), are excepted: they are not cut off. Cutoff times are removed after the last time group.

`rs_time_sweep_redo_off`
A Boolean option which, when `true`, causes redoing to be omitted. When `false`, after the last time group is assigned, the algorithm returns to the first time group and reassigns it using resource matching with the same options. The result may be different, because the following time groups are assigned now, and there are no cutoffs. It sweeps through all the time groups in this way. At the end, it checks whether the cost improved, and if so it does another redo sweep, continuing until a complete redo sweep has no effect on cost.

`rs_time_sweep_rematch_off`
A Boolean option which, when `true`, causes rematching to be omitted. When `false`, after

each time group is assigned during the initial sweep, the most recently assigned 2, 3, and so on up to `rs_time_sweep_rematch_max_groups` time groups are reassigned, using resource matching with the same options. This rematching is omitted during redoing.

`rs_time_sweep_rematch_max_groups`
The maximum number of time groups rematched (see just above). The default value is 7.

`rs_time_sweep_two_phase`
A Boolean option which, when `true`, causes time sweep to run twice. The first run assigns the resources of `rg` with the largest workload limits according to `KheClassifyResourcesByWorkload` (Section 11.4.2). The second run assigns the rest.

On one instance, cutoff times and redoing had a very significant effect. Without redoing, cutoff times reduced final cost from 185 to 149. With redoing, they reduced final cost from 95 to 72. Edge adjustment produced mixed results. Rematching during time sweep also produced mixed results, reducing one solution cost by 40 (from 107 to 67), but increasing another by 20.

### 12.7.4. Time sweep with lookahead

If option `rs_time_sweep_lookahead` has value $k > 0$, `KheTimeSweepAssignResources` looks ahead $k$ time groups when calculating edge costs, as follows.

Lookahead is similar to combinatorial grouping (Section 11.10.1). Suppose that while we are matching time group $i$ we need to determine the cost of the edge that connects task $t$ to resource $r$. Set the cutoffs of resource monitors so that they monitor everything up to and including time group $i + k$. Detach all assign resource and limit resources monitors that monitor tasks running during time groups after time group $i$. Then try all combinations of assignments which include assigning $t$ to resource $r$ during time group $i$, and assigning any task (in fact, the first task in each demand node, since the others are equivalent) or nothing during time groups $i + 1 , \dots , i + k$. Take the minimum of these values and use it as the cost of $e$, plus edge adjustments as usual.

The point of including resource monitors up to time group $i + k$ is to include the cost of the minimum-cost combination for the resource in the edge cost. The point of excluding assign resource and limit resources monitors after time group $i$ is that if some combination leaves some event resource unassigned, that does not matter because some other resource might eventually be assigned to it. For limit resources monitors it would be ideal to 'detach' any minimum limit but leave any maximum limit 'attached'; but we don't do that. In any case a maximum limit will be at least 1 in practice, and we are only assigning one resource.

To support lookahead, a variant of `KheResourceMatchingSolverSolve` is offered:

```
bool KheResourceMatchingSolverSolveWithLookahead(
  KHE_RESOURCE_MATCHING_SOLVER rms,
  ARRAY_KHE_RESOURCE_MATCHING_DEMAND_SET *rmds_array,
  int first_index, int last_index, bool edge_adjust1_off,
  bool edge_adjust2_off, bool edge_adjust3_off,
  bool edge_adjust4_off, bool ejection_off, KHE_OPTIONS options);
```

The matched demand set is in `*rmds_array` at `first_index`. The lookahead demand sets follow, ending at `last_index`. So `last_index == first_index` means no lookahead, `last_index == first_index + 1` means one day's worth, and so on. The other parameters are as for `KheResourceMatchingSolverSolve`. `ARRAY_KHE_RESOURCE_MATCHING_DEMAND_SET` is defined alongside `KHE_RESOURCE_MATCHING_DEMAND_SET` in `khe_solvers.h`.

### 12.7.5. Resource rematching repair

*Resource rematching* repairs a solution using resource matching. KHE's function for this is

```
bool KheResourceRematch(KHE_SOLN soln, KHE_RESOURCE_GROUP rg,
  KHE_OPTIONS options, int variant);
```

It creates a resource matching solver for `soln` and `rg` and calls it on many sets of times.

Parameter `variant` may be any integer and causes some change in behaviour when it changes. At present, depending on whether it is odd or even, the time sets rematched are traversed in forward or reverse order. This can be significant, especially when a time limit prevents all of them from being visited.

`KheResourceRematch` is influenced indirectly by the resource matching solver options, and directly by these options:

`rs_rematch_off`
: A Boolean option which, when `true`, causes `KheResourceRematch` to do nothing.

`rs_rematch_select`
: This determines how `KheResourceRematch` selects sets of times for solving. Its values are `"none"`, `"defective_tasks"`, `"frame"`, `"intervals"`, and `"auto"`, for which see below.

`rs_rematch_max_groups`
: An integer option which instructs `KheResourceRematch` to try sequences of adjacent time groups of length 1, 2, and so on up to its value. Its default value is 7. It is only consulted when `rs_rematch_select` is `"frame"` or `"intervals"`.

`rs_rematch_edge_adjust1_off`
: A Boolean option which, when `true`, causes edge adjustment 1 to be turned off, by passing `true` to `KheResourceMatchingSolverSolve` for `edge_adjust1_off`.

`rs_rematch_edge_adjust2_off`
: A Boolean option which, when `true`, causes edge adjustment 2 to be turned off, by passing `true` to `KheResourceMatchingSolverSolve` for `edge_adjust2_off`.

`rs_rematch_edge_adjust3_off`
: A Boolean option which, when `true`, causes edge adjustment 3 to be turned off, by passing `true` to `KheResourceMatchingSolverSolve` for `edge_adjust3_off`.

`rs_rematch_edge_adjust4_off`
: A Boolean option which, when `true`, causes edge adjustment 4 to be turned off, by passing `true` to `KheResourceMatchingSolverSolve` for `edge_adjust4_off`.

`rs_rematch_ejection_off`
: A Boolean option which, when `true`, causes ejection chain repair to be turned off, by

passing `true` to `KheResourceMatchingSolverSolve` for `ejection_off`.

The choices for `rs_rematch_select` are as follows. In each case, a set of times may be selected several times over, but each distinct set is solved only once. As explained above at the end of the introduction to resource matching, when the selected tasks are initially assigned (as is assumed here), tasks which share a resource initially will share one finally.

If `rs_rematch_select` is `"none"`, rematching is turned off, like `rs_rematch_off`.

If `rs_rematch_select` is `"defective_tasks"`, sets of times suited to repairing high school timetables are selected. Find the first tasking of `soln` whose resource type is the resource type of `rg`. For each task *t* of that tasking which is unassigned or assigned a resource from `rg`, and which is defective (unassigned, assigned an unpreferred resource, part of a split assignment, or involved in a clash), make one set of times equal to the set of times that *t* is running, including the times of all tasks connected with *t* by assignments not involving a cycle task.

If `rs_rematch_select` is `"frame"`, sets of times suitable for repairing nurse rostering timetables are selected. For each index in the common frame (Section 5.10), the time group at that index, plus $m - 1$ immediately following time groups, are united to form one of the sets of times. There is one set for each value of *m* between 1 and `rs_rematch_max_groups` inclusive.

If `rs_rematch_select` is `"intervals"`, then for each limit active intervals constraint in the instance, for each index into the sequence of time groups of that constraint, the time group at that index, plus $m - 1$ immediately following time groups, are united to form one of the sets of times. There is one set for each value of *m* between 1 and `rs_rematch_max_groups` inclusive. To these are added the sets of times solved when `rs_rematch_select` is `"frame"`.

Finally, if `rs_rematch_select` is `"auto"` (the default value), then `"defective_tasks"` is chosen when the model is high school timetabling, otherwise `"frame"` is chosen. The author had high hopes for `"intervals"`, but his tests showed an improvement in only one instance, from 107 to 105, which did not justify the increased running time, averaging one or two seconds.

## 12.8. Run homogenization

A *run* is a maximal sequence of assignments to tasks for the same resource on consecutive days. A *homogeneous run* is a maximal sequence of assignments to tasks for the same resource on consecutive days such that each assignment is to the same shift (to the morning shift, or the day shift, or whatever). Each run consists of a sequence of one or more homogeneous runs. The *length* of a run (homogeneous or otherwise) is its number of assignments.

There may be no incentive in the cost function to make runs homogeneous, but doing so may increase the chances of finding repairs for other defects. Accordingly, KHE offers

```
void KheRunHomogenize(KHE_SOLN soln, KHE_RESOURCE_TYPE rt,
  KHE_OPTIONS options);
```

which swaps homogeneous runs between pairs of resources with the aim of reducing the number of homogeneous runs (and thus increasing their length), wherever this can be done without increasing the cost of `soln`.

## 12.9. Moving unnecessary assignments

Unnecessary assignments are assignments which can be removed without incurring any event resource defects. That is, they are assignments made for the sake of giving some resource a good timetable, not because the event resource itself needs to be assigned.

When total workload is tight, every unnecessary assignment leads to one deviation in total workload, although there is usually no close connection between the two things. So an unnecessary assignment is, in effect, a defect for which there is no monitor with non-zero cost.

To make unnecessary assignments more visibly defective, KHE offers resource solver

```
void KheMoveUnnecessaryAssignments(KHE_SOLN soln, KHE_RESOURCE_TYPE rt,
  KHE_OPTIONS options);
```

It attempts to move unecessary assignments from resources of type `rt` that are not overloaded to resources that are overloaded. It does this by swapping assignments between pairs of resources, wherever it can find a way to do it without increasing the total cost of the solution. It may swap several assignments on adjacent days, only one of which is actually unnecessary.

The value of this is that when a repair method like ejection chains comes to repair the overloaded resource defect, these unnecessary assignments will be available for unassignment.

## 12.10. Ejection chain repair

Function

```
bool KheEjectionChainRepairResources(KHE_TASKING tasking,
  KHE_OPTIONS options);
```

uses ejection chains (Chapter 13) to improve the solution by changing the assignments of the tasks of `tasking`. It is influenced by many options, including

`rs_eject_off`
A Boolean option which, when `true`, causes this function to do nothing.

For full details, consult Section 13.6.

## 12.11. Global load balancing

*Global load balancing* moves tasks between resources to even out the resources' loads. (By *load* we mean either number of busy times or workload.) The method is global in two senses: its focus is on total load, and it deals with all resources of a given resource type at once.

The function is

```
bool KheGlobalLoadBalance(KHE_SOLN soln, KHE_RESOURCE_TYPE rt,
  KHE_OPTIONS options);
```

This performs global load balancing on the resources of type `rt` in `soln`. If it succeeds in finding a balancing that reduces the cost of `soln`, it carries out that balancing and returns `true`.

Otherwise it changes nothing and returns `false`.

Global load balancing is similar to ejection chain repair in that it tries to find a sequence (possibly a very long sequence) of linked changes to the solution that together reduce its cost. Compared with ejection chains, it is limited in the defects it tackles and the changes it allows; but if an improvement of the kind it seeks exists at all, it will find it.

The rest of this section explains how global load balancing works.

We will be moving task sets (that is, sets of tasks) from one resource to another. We only use *admissible task sets*: task sets $T$ whose tasks are proper root tasks which are all assigned to the same resource in the initial solution (the solution we are trying to improve) and cover at most one time on each day. Furthermore, the days covered are consecutive, so that they can be represented by an interval $i(T)$ of length $l(T)$.

There are two other conditions on what makes a task set admissible, both controlled by user options. The first is the maximum number of consecutive days it can cover, determined by option `rs_glb_max_len`, whose default value is 3. The second is that an admissible task set must lie at the start or end of a maximal sequence of consecutive busy days for the resource it is assigned in the initial solution. Setting option `rs_glb_in_seq` to `true` turns off this condition.

In fact, our algorithm could handle any task sets at all, provided the tasks of each task set are assigned the same resource in the initial solution. Requiring task sets to cover consecutive days limits the graph $G$ defined below to polynomial size while retaining everything that could realistically lead to success.

The algorithm is based on a directed graph $G$, containing three kinds of nodes, as follows. *Source nodes* are labelled by a resource, so are shown like this:

$$\boxed{r}$$

They have no incoming edges. $G$ contains one for each resource $r$ satisfying this condition:

1. Resource $r$ has type `rt` and is *overloaded* in the initial solution: that is, one or both of `KheResourceAvailableBusyTimes` and `KheResourceAvailableWorkload` (Section 4.7.1) returns `true` for $r$ with a negative availability.

*Intermediate nodes* are labelled by a resource and a task set, and are shown like this:

$$\boxed{r + T}$$

These nodes may have both incoming and outgoing edges, and must satisfy these conditions:

1. Resource $r$ has type `rt` and is *fully loaded* in the initial solution: not overloaded, and one or both of `KheResourceAvailableBusyTimes` and `KheResourceAvailableWorkload` (Section 4.7.1) returns `true` for $r$ with zero availability.

2. $T$ is an admissible task set which is not assigned $r$ in the initial solution.

Every $(r, T)$ pair that satisfies these conditions and is reachable from a source node (more on this below) is included. Each intermediate node represents a state in which the tasks assigned $r$ are those assigned *r* in the initial solution plus the tasks of task set *T*.

Finally, *sink nodes* are also labelled by a resource and a task set. They look like intermediate nodes in our diagrams:

$$\boxed{r+T}$$

These nodes have no outgoing edges. A sink node must satisfy these conditions:

1. Resource *r* has type `rt` and is neither overloaded nor fully loaded in the initial solution.

2. *T* is an admissible task set which is not assigned *r* in the initial solution.

Every $(r, T)$ pair that satisfies these conditions and is reachable from a source node (more on this below) is included. Like intermediate nodes, each sink node represents a state in which the tasks assigned *r* are those assigned *r* in the initial solution plus the tasks of task set *T*.

Each edge of *G* has the form

$$\overset{v}{\boxed{r_1}} \xrightarrow{-T_2 \qquad +T_2} \overset{w}{\boxed{r_2+T_2}} \qquad \text{or} \qquad \overset{v}{\boxed{r_1+T_1}} \xrightarrow{-T_2 \qquad +T_2} \overset{w}{\boxed{r_2+T_2}}$$

and means 'unassign $r_1$ from $T_2$ and assign $r_2$ to $T_2$, leading to a node in which $r_2$ has its original assignments, plus $T_2$.' *G* has one edge for each pair of nodes $(v, w)$ satisfying these conditions:

1. $r_1 \neq r_2$.

2. *v* is a source or intermediate node, and *w* is an intermediate or sink node.

3. $T_2$ is assigned to $r_1$ in the initial solution.

4. A single compound operation consisting of assigning $r_1$ to $T_1$ (but only if $T_1$ is present), unassigning $r_1$ from $T_2$, and assigning $r_2$ to $T_2$ can be carried out and does not increase the total cost of the resource monitors that monitor $r_1$, plus the event resource monitors that monitor the tasks of $T_2$, plus (but only if *w* is a sink node) the resource monitors that monitor $r_2$. If *v* is a source node, it must decrease this cost.

We build *G* while performing a breadth-first search of it, starting from the source nodes. For each node, we try all possible edges out of it, and for those which satisfy the four conditions, we either retrieve an existing endpoint node or else create one and enqueue it for later expansion.

If we can find a path in *G* from a source node to a sink node, then carrying out the unassignments and assignments specified by the edges along that path will usually produce a new solution whose cost is smaller. This is because cost decreases at the start and does not increase thereafter. So after constructing the graph, we sort its sink nodes by the total cost reduction of the path to the sink node. Then each sink node is tried in turn, largest total cost reduction first, until all are tried or applying the task set moves along the path does actually reduce solution cost, at which point the solve ends—except that, if we do succeed in reducing solution cost, we run the whole algorithm again from the beginning, and continue in this way until there is no improvement.

The cost reduction predicted by the path is usually realized, but there are two reasons why it might not be. First, limit resources constraints might cause interference between edges which is not taken account of by this algorithm. Second, if a path passes through two nodes containing the same resource, then the changes along that path will change a resource's assignments twice, which again is not taken account of. (Using breath-first search should keep the chance of this small.) The algorithm simply accepts the first cost reduction, without concern for whether it is the expected amount or why it isn't, if not.

## 12.12. Resource pair repair

One idea for repairing resource assignments is to unassign all tasks assigned to two resources, then try to reassign those tasks to the same two resources in a better way—an example of very large-scale neighbourhood (VLSN) search [1, 12]. The search space, although formally exponential in size, is often small enough to search completely, giving an optimal result.

This section is devoted to function `KheResourcePairReassign`, which carries out this idea while trying to save time by detecting symmetries. Section 12.13 offers another way of reassigning resources. It does not detect symmetries, but it is more general in several respects.

### 12.12.1. The basic function

The basic function for carrying out this kind of repair is

```
bool KheResourcePairReassign(KHE_SOLN soln, KHE_RESOURCE r1,
  KHE_RESOURCE r2, bool resource_invariant, bool fix_splits);
```

It knows that when one task is assigned to another, the two tasks must be assigned the same resource; and it believes that tasks that overlap in time must be assigned different resources. It does not change task domains, fixed assignments, or assignments of tasks to non-cycle tasks. If it can find a reassignment to `r1` and `r2` of the tasks currently assigned to `r1` and `r2` which satisfies these conditions and gives `soln` a lower cost, it makes it and returns `true`; otherwise it changes nothing and returns `false`. If `resource_invariant` is `true`, only changes that preserve the resource assignment invariant are allowed. `KheResourcePairReassign` accepts any resources, but it is most likely to succeed on resources with similar capabilities that are involved in defects.

If `fix_splits` is `true`, the algorithm focuses on repairing split assignments, by forcing tasks unassigned by the algorithm which are linked by avoid split assignments constraints of non-zero cost to be assigned the same resource in the reassignment. This runs faster, because it has fewer choices to try, but it may overlook other kinds of improvements.

Within the set of tasks assigned to `r1` and `r2` originally, there may be subsets which are not assignable to two resources without introducing clashes. Clashes in the original assignments can cause this, as can split assignments when `fix_splits` is set. Such subsets are ignored by `KheResourcePairReassign`; their original assignments are left unchanged.

### 12.12.2. A resource pair solver

Resource solver

```
bool KheResourcePairRepair(KHE_TASKING tasking, KHE_OPTIONS options);
```

calls KheResourcePairReassign for many pairs of resources. The resource_invariant arguments of all these calls are set to the rs_invariant option of options. Two other options control the behaviour of KheResourcePairRepair:

rs_pair_off
: A Boolean option which, when true, turns resource pair repair off.

rs_pair_select
: This option determines which pairs of resources are tried. If it is "none", no pairs are tried, giving another way to turn this repair off. If it is "splits" (the default), then for all pairs of resources involved in all split assignments of tasking, KheResourcePairRepair calls KheResourcePairReassign for those two resources, with the fix_splits parameter set to true. This focuses the solver on repairing split assignments. If it is "partitions", then KheResourcePairReassign calls KheResourcePairRepair for each pair of resources in each partition of the resource type of tasking, or in all resource types if tasking has no resource type, with fix_splits set to false. Each resource type with no partitions is treated as though all resources lie in a single shared partitition. This focuses the solver on improving resources' assignments generally. However the search space is often larger, increasing the chance that the search will be cut short, losing optimality. Value "all" is the same as "partitions" except that partitions are ignored, so that there is a call on KheResourcePairReassign for every pair of distinct resources of the types involved.

KheResourcePairRepair collects statistics about its calls to KheResourcePairReassign, held in the rs_pair_calls, rs_pair_successes, and rs_pair_truncs options. Each time KheResourcePairReassign is called, rs_pair_calls is incremented. Each time it returns true, rs_pair_successes is incremented. And each time it truncates an overlong search (at most once per call), rs_pair_truncs is incremented. The caller must initialize and retrieve these options at the right moments, using the usual options functions (Section 8.2).

### 12.12.3. Partition graphs

Resource pair repair is essentially about two-colouring a clash graph whose nodes are tasks and whose edges join pairs of tasks that overlap in time. Although the basic idea is simple enough, the details become quite complicated, especially when optimizing by removing symmetries in the search. It has proved convenient to build on a separate *partition graph* module, which is the subject of this section. It finds the connected components of a graph (called *components* here), and, if requested, partitions components into two *parts* by two-colouring them.

The module stores a graph whose nodes are represented by values of type void *. There are operations for creating a graph in a given arena, adding nodes to it, and visiting those nodes:

```
KHE_PART_GRAPH KhePartGraphMake(KHE_PART_GRAPH_REL_FN rel_fn,
  HA_ARENA a);
void KhePartGraphAddNode(KHE_PART_GRAPH graph, void *node);
int KhePartGraphNodeCount(KHE_PART_GRAPH graph);
void *KhePartGraphNode(KHE_PART_GRAPH graph, int i);
```

Deleting the arena deletes the graph, including its components and parts, but not its nodes. These functions and the others in this section are declared in include file khe_part_graph.h.

To define the edges, the user passes in a *relation function* of type KHE_PART_GRAPH_REL_FN which the module calls back whenever it needs to know whether two nodes are connected by an edge. As the user would define it, this function looks like this:

```
KHE_PART_GRAPH_REL RelationFn(void *node1, void *node2)
{
  ...
}
```

where type KHE_PART_GRAPH_REL is

```
typedef enum {
  KHE_PART_GRAPH_UNRELATED,
  KHE_PART_GRAPH_DIFFERENT,
  KHE_PART_GRAPH_SAME
} KHE_PART_GRAPH_REL;
```

Values KHE_PART_GRAPH_UNRELATED and KHE_PART_GRAPH_DIFFERENT are the usual options for clash graphs, the first saying that there is no edge between the two nodes, the second that there is an edge which requires the two nodes to be coloured with different colours. The third value, KHE_PART_GRAPH_SAME, says that the two nodes must be coloured the same colour. It is used, for example, when the two nodes represent tasks which are linked by an avoid split assignments constraint, and the fix_splits option is in force.

After all nodes have been added, the user may call

```
void KhePartGraphFindConnectedComponents(KHE_PART_GRAPH graph);
```

to find the connected components, which may then be visited by

```
int KhePartGraphComponentCount(KHE_PART_GRAPH graph);
KHE_PART_GRAPH_COMPONENT KhePartGraphComponent(KHE_PART_GRAPH graph, int i);
```

The graph that a component is a component of may be found by

```
KHE_PART_GRAPH KhePartGraphComponentGraph(KHE_PART_GRAPH_COMPONENT comp);
```

and the nodes of a component may be visited by

```
int KhePartGraphComponentNodeCount(KHE_PART_GRAPH_COMPONENT comp);
void *KhePartGraphComponentNode(KHE_PART_GRAPH_COMPONENT comp, int i);
```

KhePartGraphFindConnectedComponents considers two nodes to be connected when rel_fn returns KHE_PART_GRAPH_SAME or KHE_PART_GRAPH_DIFFERENT when passed those nodes.

If requested, the module will partition the nodes of a component into two sets, such that two-colouring the component will give the nodes in one set one colour, and the nodes in the other set the other colour. This gives exactly two ways to two-colour the component, which is all there are, since once a colour is assigned to one node, its neighbours must be assigned the other colour, their neighbours must be assigned the first colour, and so on. To carry out this partitioning, call

```
void KhePartGraphComponentFindParts(KHE_PART_GRAPH_COMPONENT comp);
```

After that, to retrieve the two parts, call

```
bool KhePartGraphComponentParts(KHE_PART_GRAPH_COMPONENT comp,
  KHE_PART_GRAPH_PART *part1, KHE_PART_GRAPH_PART *part2);
```

If `KhePartGraphComponentFindParts` was able to partition the component into two parts, `KhePartGraphComponentParts` returns `true` and sets `*part1` and `*part2` to non-`NULL` values; otherwise it returns `false` and sets them to `NULL`. To find a part's enclosing component, call

```
KHE_PART_GRAPH_COMPONENT KhePartGraphPartComponent(
  KHE_PART_GRAPH_PART part);
```

The nodes of a part may be visited by

```
int KhePartGraphPartNodeCount(KHE_PART_GRAPH_PART part);
void *KhePartGraphPartNode(KHE_PART_GRAPH_PART part, int i);
```

as usual.

### 12.12.4. The implementation of resource pair reassignment

This section describes the implementation of `KheResourcePairReassign`. It builds two partition graphs altogether, a *first graph* which does the basic analysis, and a *second graph* which is used to find and remove symmetries in the first graph.

The same node type is used in both graphs. A node holds a set of tasks. A resource is *assignable to a node* when it is assignable to each task of the node. A resource is assignable to a fixed task when it is assigned to that task (fixed tasks are never unassigned). A resource is assignable to an unfixed task when it lies in the domain of that task. It is possible for neither, one, or both resources to be assignable to a node. If neither is assignable, the node is *unassignable*, otherwise it is *assignable.*

When a resource is assignable to a node, there are operations for assigning and unassigning it. To assign it, assign it to each unfixed task of the node. To unassign it, unassign it from each unfixed task of the node.

The first graph contains one node for each task initially assigned `r1` or `r2`, containing just that task. Thus, in the first graph there are no unassignable nodes. Given two nodes, the first graph's relation function first checks which resources are assignable to each. If there is no way to assign the same resource to both nodes, it returns `KHE_PART_GRAPH_DIFFERENT`. Otherwise, if there is no way to assign different resources to the nodes, it returns `KHE_PART_GRAPH_SAME`. Otherwise, if `fix_splits` is `true` and the two nodes share an avoid split assignments monitor of non-zero cost, it returns `KHE_PART_GRAPH_SAME`. Otherwise, if the two nodes overlap in time, it returns `KHE_PART_GRAPH_DIFFERENT`. Otherwise it returns `KHE_PART_GRAPH_UNRELATED`.

Next, the graph's connected components are found and partitioned. It is easy to see, referring to the relation function, that if a component was successfully partitioned there must be at least one way (and possibly two ways) to assign `r1` to the nodes of one part and `r2` to the nodes of the other part. So a component of the first graph is called *assignable* if it was successfully partitioned, and *unassignable* otherwise.

For each assignable component, the nodes of one part are merged into one node, and the nodes of the other are merged into a second node. These two nodes are assignable to different resources in one or two ways. For each unassignable component, all the nodes are merged into a single node. It does not matter whether this node is assignable or not; it is never assigned.

Next, the assignable components are sorted into increasing order of number of possible assignments. Each of the $C$ assignable components has 1 or 2 possible assignments. A tree search is carried out which tries each of these on each component in turn. The total search space size is at most $2^C$. This is often small enough to search completely. For safety, the search only explores both assignments until 512 tree nodes have been visited; after that it tries only one assignment for each component. In the usual way, each time the tree search reaches a leaf it compares its solution cost with the best so far, and if it is better (and if the resource assignment invariant is preserved, if required) it takes a copy of its decisions. At the end, the cost of the best solution found is compared with the initial solution cost, and if the best solution is better it is installed; otherwise the initial solution is restored.

The search space often has symmetries which would waste time and cause the node limit to be reached often enough to compromise optimality in practice if they were not removed. The rest of this section describes them and how `KheResourcePairReassign` removes them.

Suppose `r1` and `r2` are Mathematics teachers assigned to two Mathematics courses from the same form, each split into 4 meets of the same durations, running simultaneously. This gives 4 components and a search space of size $2^4$, yet clearly this could be reduced safely to 1. If two of the simultaneous meets are made not simultaneous, the search space size can still be reduced safely, to 2. If `fix_splits` is `true`, each set of 4 meets is related, making 1 component and a search space of size 2—still unnecessarily large when the meets are simultaneous.

A component is *symmetrical* if it makes no difference which of its two assignments is chosen. In that case, its assignment choices can be reduced from 2 to 1 by arbitrarily removing one, halving the search space size. But note the complicating factor in the Mathematics example: one cannot arbitrarily remove one choice from each component, because some combinations of choices lead to split assignments and others do not. Instead, a way must be found to first merge the four components into one, which can then be assigned arbitrarily.

Symmetry arises when the two assignment choices of a component affect monitors in the same way. They need to have the same effect on the state of monitors, so that no difference arises when the monitors change state again later in response to changes outside the component.

The two choices always have the same effect on the state of event monitors (no effect at all), and on the state of assign resources monitors, which care only whether tasks are assigned resources, not which resources. As far as these kinds of monitors are concerned, all components are symmetrical. Classify the remaining monitors into three groups: resource monitors, prefer resources monitors, and avoid split assignments monitors. (This description was written before the advent of limit resources monitors, and does not take them into account.)

A component is *r-symmetrical*, *p-symmetrical*, or *s-symmetrical* when it is assignable both ways and they affect in the same way all resource, prefer resources, or avoid split assignments monitors that monitor tasks of the component. (In particular, if there are no monitors of some type, the component is vacuously symmetrical in that type.) Combinations of prefixes denote conjunctions of these conditions. For example, *symmetrical* is shorthand for *rps-symmetrical.*

Although these definitions are clear in principle, they are rather abstract. An algorithm needs concrete, easily computable conditions that imply the abstract ones and are likely to hold in practice. Here are the concrete conditions used by `KheResourcePairReassign`, assuming that the component is assignable both ways.

Suppose that some component's two parts run at the same times and have the same total workload. Then the component is r-symmetrical, because only these things affect resource monitors, except clashes—but component assignments have no clashes in themselves, and since the two parts run at the same times, they have the same clashes with tasks outside the component.

Suppose that, for every prefer resources monitor of non-zero cost which monitors any task of some component, either `r1` and `r2` are both preferred by the monitor's constraint, or they are both not preferred. Then the component is p-symmetrical.

Suppose that, for each task in some component $c$ which is monitored by an avoid split assignments monitor of non-zero cost, every task monitored by that monitor either was not assigned `r1` or `r2` originally, or else it lies in $c$. Then the component is s-symmetrical.

To prove this, take one avoid split assignments monitor, and partition the set of tasks monitored by it into those that were not assigned `r1` or `r2` originally, and so are beyond the scope of the reassignment (call them $S_1$), and those that were (call them $S_2$). If the tasks of $S_2$ lie within two or more components, then which way those components are assigned does matter. But if they lie within one component, then the cost of the monitor will be the same whichever assignment is chosen. This is because `r1` and `r2` do not appear among the resources assigned to the tasks of $S_1$ (if they did, those tasks would be in $S_2$), so the assignments to $S_2$ introduce fresh resources to the monitor. If all the tasks of $S_2$ lie in one part of the component, one fresh resource is introduced by both assignments; if some lie in one part and the others in the other, two fresh resources are introduced by both assignments. Either way, the effect on the monitor is the same.

When `fix_splits` is `true`, all tasks which share an avoid split assignments monitor lie in the same part, so in the same component. So every component is s-symmetrical in that case.

It is easy to check whether a component is rp-symmetrical. This is done as each component is partitioned. Merely checking for s-symmetry is not enough: as illustrated by the Mathematics example, several components may need to be merged (by merging their parts) to produce one s-symmetrical component. This is done using the second partitioning graph, as follows.

The second-graph nodes are the merged nodes from the first-graph components. When two nodes come from the same first-graph component, `KHE_PART_GRAPH_DIFFERENT` is returned by the relation function. Otherwise, if they share an avoid split assignments monitor of non-zero cost, it returns `KHE_PART_GRAPH_SAME`. Otherwise it returns `KHE_PART_GRAPH_UNRELATED`.

Two nodes representing the two parts of a first-graph component must lie in the same second-graph component, because there is an edge between them. So each second-graph component is a set of first-graph components linked by avoid split assignments constraints.

For each second-graph component, its first-graph components may be merged if it does not contain an unassignable first-graph component, at most one of its first-graph components is not rp-symmetrical, and it is partitionable. The two nodes of the merged component are built by merging the nodes of each part of the second-graph component. If all the first-graph components being merged are rp-symmetrical, the resulting component is rps-symmetrical, so either one of its assignments may be removed. But component merges are valuable even without rps-symmetry.

## 12.13. Resource reassignment

This section describes an operation called *resource reassignment* which in principle can optimally reassign the tasks assigned to an arbitrary number of resources. We say 'in principle' because as the number of resources increases the running time increases dramatically, so that in practice it can only handle 3 resources, or 4 at most; or alternatively it can handle all resources, but only when the tasks are running at a very limited range of times.

The first step in resource reassignment is to create a *reassign solver*, by calling

```
KHE_REASSIGN_SOLVER KheReassignSolverMake(KHE_SOLN soln,
  KHE_RESOURCE_TYPE rt, KHE_OPTIONS options);
```

The solver uses `gs_common_frame` and `rs_invariant` from `options`. These values, along with `soln` and `rt`, are fixed for the lifetime of the solver.

A solver object may be deleted by calling

```
void KheReassignSolverDelete(KHE_REASSIGN_SOLVER rs);
```

This should be done after solving is completed, and also if any changes to the solution are made other than those carried out by `rs`. This is because `rs` keeps information between solves, so it will go wrong if the solution changes in ways that it does not know about.

To say which resources are to be involved in the next solve, call

```
bool KheReassignSolverAddResource(KHE_REASSIGN_SOLVER rs, KHE_RESOURCE r);
bool KheReassignSolverDeleteResource(KHE_REASSIGN_SOLVER rs, KHE_RESOURCE r);
void KheReassignSolverClearResources(KHE_REASSIGN_SOLVER rs);
```

`KheReassignSolverAddResource` adds `r` to the solver, returning `false` and changing nothing when `r` is already present. `KheReassignSolverDeleteResource` deletes `r`, returning `false` and changing nothing when `r` is not present. Both abort if `r` does not have the resource type `rt` passed to `KheReassignSolverMake`. `KheReassignSolverClearResources` deletes all resources.

One of the resources may be `NULL`. This causes the solver to select as many non-overlapping unassigned tasks of type `rt` as it can easily find, and try assigning them, and also unassigning other tasks, as though unassigned tasks were assigned a resource called `NULL`. Only unassigned tasks in need of assignment according to `KheTaskNeedsAssignment` (Section 4.6.1) are included.

To carry out an actual solve, call

```
bool KheReassignSolverSolve(KHE_REASSIGN_SOLVER rs, int first_index,
  int last_index, KHE_REASSIGN_GROUPING grouping, bool ignore_partial,
  KHE_REASSIGN_METHOD method, int max_assignments);
```

This optimally reassigns the solver's resources to the tasks assigned those resources in the *target interval* (`first_index` to `last_index` inclusive), returning `true` if it improves the solution.

When deciding whether a task `t` lies in the target interval, `t`'s own interval, as returned by `KheTaskFinderTaskInterval` (Section 11.8.1), is used to determine which days it is running. These include days when tasks assigned directly or indirectly to `t` are running.

Tasks are organized into groups during the solve, and parameter `grouping` determines how these groups are made. Two tasks are only eligible to be in the same group if they are assigned the same resource (possibly `NULL`) initially. The tasks of each group are assigned the same resource throughout the solve. The type of `grouping` is

```
typedef enum {
  KHE_REASSIGN_MINIMAL,
  KHE_REASSIGN_RUNS,
  KHE_REASSIGN_MAXIMAL
} KHE_REASSIGN_GROUPING;
```

`KHE_REASSIGN_MINIMAL` produces no grouping beyond the initial grouping of the tasks (which is not disturbed); `KHE_REASSIGN_RUNS` groups sequences of tasks assigned the same resource on adjacent days (*runs*); and `KHE_REASSIGN_MAXIMAL` groups all tasks initially assigned the same resource which participate in the solve. The meaning of ‘optimal reassignment’ is relative to these groupings; only `KHE_REASSIGN_MINIMAL` produces true optimal reassignment.

When `ignore_partial` is `true`, tasks that lie partly inside and partly outside the target interval are ignored, just as though they were not there. When `grouping` is `KHE_REASSIGN_RUNS`, this causes some runs to be shorter than they otherwise would be.

When `ignore_partial` is `false`, tasks that lie partly inside and partly outside the target interval are included in the solve. Furthermore, when `grouping` is `KHE_REASSIGN_RUNS`, tasks that lie entirely outside the target interval are included when they are part of a run that lies partly within the target interval.

We say that a group *needs assignment* when at least one of its tasks needs assignment, according to `KheTaskNeedsAssignment` (Section 4.6.1). If a group does not need assignment, then, as well as trying to assign it to the resources of the solve, `KheReassignSolverSolve` will try unassigning it. When `grouping` is `KHE_REASSIGN_RUNS`, runs are built such that all tasks in any given run have the same value for `KheTaskNeedsAssignment`. As previously stated, for the `NULL` resource this value must be `true`; but for non-`NULL` resources it may be `true` or `false`.

The type of `method` is

```
typedef enum {
  KHE_REASSIGN_EXHAUSTIVE,
  KHE_REASSIGN_MATCHING
} KHE_REASSIGN_METHOD;
```

It determines the algorithm used for solving: exhaustive search or weighted bipartite matching. The latter is reasonable only when there is one group of tasks per resource: when `grouping` is `KHE_REASSIGN_MAXIMAL`, or the target interval is narrow. Parameter `max_assignments` limits the number of alternatives tried on each call to `KheReassignSolverSolve` when `method` is `KHE_REASSIGN_EXHAUSTIVE`. It is not consulted when `method` is `KHE_REASSIGN_MATCHING`.

If there are $k$ resources and initially those resources are assigned $R$ groups of tasks, then each group could be assigned any one of the $k$ resources, making a search space of size $k^R$ when `method` is `KHE_REASSIGN_EXHAUSTIVE`. However there is some pruning. Before starting the search, the solver checks the $(k - 1)R$ possible new assignments to see which succeed, and does not try failed ones again. Any group that cannot move at all is omitted from the search, and groups that overlap with that group have its resource removed from their list of resources to try.

And any assignment which would give a resource two tasks on the same day is not tried. The test for this is carried out efficiently using intervals.

Unavailable times are taken into account when solution costs are reported, but they are not taken as a reason to exclude a resource from being assigned to a group. It is not unusual for an optimal solution to contain a few assignments of resources to tasks at unavailable times.

Function

```
void KheReassignSolverDebug(KHE_REASSIGN_SOLVER rs,
  int verbosity, int indent, FILE *fp);
```

produces a debug print of `rs` onto `fp` with the given verbosity and indent. Between solves there is not much to display, mainly the resources.

A convenient way to call `KheReassignSolverSolve` repeatedly is

```
bool KheReassignRepair(KHE_SOLN soln, KHE_RESOURCE_TYPE rt,
  KHE_OPTIONS options);
```

This creates a reassign solver, tries a variety of sets of resources of type `rt` and target intervals, and ends by deleting the solver and returning `true` if any of the solves improved the solution. `KheResourceReassignRepair` consults these options:

`rs_reassign_resources`
: This integer option says how many non-`NULL` resources from `rt` to select for each solve. The default value is 2. The special value `all` selects all resources of type `rt`. This is only feasible when `method` is `KHE_REASSIGN_MATCHING`. When `method` is `KHE_REASSIGN_EXHAUSTIVE`, larger values (sometimes even 3) can produce long run times.

`rs_reassign_select`
: All sets of resources tried contain `rs_reassign_resources` non-`NULL` resources. All have type `rt`. This option determines which of these sets are tried. Its value may be `"none"` (the default), meaning that no sets are tried, turning `KheReassignRepair` off; `"all"`, meaning that all sets are tried; `"adjacent"`, meaning that each set of resources which are adjacent to each other in `rt` are tried; or `"defective"`, meaning that all sets in which at least one of the resources has a defective resource monitor are tried.

: For experimental use there is also `constraint:xxx` where `xxx` stands for any non-empty string. The cluster busy times constraints of the instance whose names contain `xxx` are found, and then all sets of resources are selected such that one resource violates one of these constraints, and the rest are slack (strictly below the maximum) for all of them. The hope is that optimal reassignment might move tasks from the violating resource to the slack ones.

`rs_reassign_null`
: When `true`, this Boolean option says to include `NULL` in the set of resources passed to the solver on each call. This is in addition to the `rs_reassign_resources` non-`NULL` resources selected by `rs_reassign_select`. The default value is `false` as usual.

`rs_reassign_parts`
: The value of `first_index - last_index + 1` on each call. For example, setting this value to 14 (the default) reassigns two weeks. If `rs_reassign_parts` is larger than the

total number of days, it is silently reduced to the total number of days.

For experimental use there is also `constraint:xxx` where `xxx` stands for any non-empty string. The cluster busy times constraints of the instance whose names contain `xxx` are found, and for each time group of each of them, the smallest target interval covering that time group is one of the target intervals tried. A target interval may be found several times over in this way, but it is only tried once. For this value of `rs_reassign_parts`, options `rs_reassign_start` and `rs_reassign_increment` (just below) are not consulted.

`rs_reassign_start`, `rs_reassign_increment`
: The value of `first_index` on the first call for a given set of resources, and how much it is incremented by on each subsequent call for that set of resources. The default values are 0 and `rs_reassign_parts`. Only intervals lying entirely within the legal range are tried.

`rs_reassign_grouping`
: Determines the `grouping` argument of each call (see above). It may be `"minimal"` (the default), `"runs"`, or `"maximal"`.

`rs_reassign_ignore_partial`
: A Boolean option which determines the `ignore_partial` argument of each call (see above). The default value is `false`.

`rs_reassign_method`
: Determines the `method` argument of each call (see above). Its value may be either `"exhaustive"` (the default) or `"matching"`.

`rs_reassign_max_assignments`
: An integer option which determines the `max_assignments` argument of each call (see above). Its default value is 1000000.

To allow for up to three calls to `KheReassignRepair` with separate options, there are also

```
bool KheReassign2Repair(KHE_SOLN soln, KHE_RESOURCE_TYPE rt,
  KHE_OPTIONS options);
bool KheReassign3Repair(KHE_SOLN soln, KHE_RESOURCE_TYPE rt,
  KHE_OPTIONS options);
```

`KheReassign2Repair` is the same except that it consults options `rs_reassign2_resources`, `rs_reassign2_select`, and so on; `KheReassign3Repair` is the same except that it consults options `rs_reassign3_resources`, `rs_reassign3_select`, and so on.

## 12.14. Trying unassignments

KHE's solvers assume that it is always good to assign resources to tasks. But sometimes cost can be reduced by unassigning a task, because the resulting assign resource defect costs less than the defects introduced by the assignment. In acknowledgement of this, KHE offers

```
bool KheSolnTryTaskUnAssignments(KHE_SOLN soln, bool strict,
  KHE_OPTIONS options);
```

for use at the end. It tries unassigning each proper root task of soln. If any unassignment reduces the cost of soln, it is not reassigned. The result is true if any unassignments were kept.

If strict is true, all unassignments that reduce the cost of soln are kept. If strict is false, all unassignments that do not increase the cost of soln are kept.

Restricting KheSolnTryTaskUnAssignments to proper root tasks ensures that it does no task ungrouping. By the end there will probably be no groups anyway, but it seems best to keep the ideas of ungrouping and unassigning distinct.

It might pay to unassign two or more adjacent tasks. KheSolnTryTaskUnAssignments consults an option for this:

rs_max_unassign
: This integer option determines the maximum number of adjacent tasks to try unassigning. The default value is 1.

For example, setting rs_max_unassign to 2 will try unassigning entire weekends (among other things), which might pay off if the resource is working on too many weekends.

## 12.15. Putting it all together

There is an rs option which allows you to define a do-it-yourself resource solver (Section 8.3) built from the pieces presented in this and the preceding chapter. The default value of this option defines a reasonable solver which we will come to later.

The value of rs must be a <solver> as defined in Section 8.3. Here is the current list of solver names and their meanings. We start with resource structural solvers (from Chapter 11):

| `<item>` | Meaning |
|---|---|
| `rt <solver>` | For each resource type, run `<solver>` for that type. Most of the following solvers must be inside `rt`, since they apply to a resource type (or tasking, holding all tasks of one resource type). |
| `rrd <solver>` | If the total demand for resources of the current resource type equals or exceeds the total supply, according to a balance solver (Section 11.4.5), call `KheSolnRedundancyBegin` (Section 7.6) for the current resource type, run `<solver>`, then call `KheSolnRedundancyEnd`. Otherwise just run `<solver>`. The cost function and combined weight parameters of `KheSolnRedundancyBegin` are fixed to linear and `KheCost(0, 1)`. |
| `rem <solver>` | Make an event monitor for all events, add it to `options` with name `gs_event_timetable_monitor`; run `<solver>`; delete it. Several resource solvers need it. It is slow when time assignments change. |
| `rwp <solver>` | Call `KheWorkloadPack` (Section 11.4.7), run `<solver>`, then restore by calling `KheTaskBoundGroupDelete`. |
| `rsm <solver>` | Call `KheSetClusterMonitorMultipliers` (Section 11.5.1), run `<solver>`, then call `KheResetClusterMonitorMultipliers`. The monitors to change and the multiplier are stored in the `rs_multiplier` option (e.g. `"100:BusyWeekends"`), which must be present. |
| `rcm <solver>` | Install cluster minimum limits (Section 11.5.4), run `<solver>`, then return the limits to their previous values. |
| `rbw <solver>` | Call `KheBalanceWeekends` (Section 11.5.5), run `<solver>`, then unfix any tasks that were fixed. |
| `rgc <solver>` | Group tasks using `KheGroupByResourceConstraints` (Section 11.10), run `<solver>`, then undo the groupings. |
| `rgr <solver>` | Group tasks using `KheTaskingGroupByResource` (Section 11.6), run `<solver>`, then undo the grouping. |
| `red` | Call `KheTaskingEnlargeDomains` (Section 11.5.7). |

And here is the list of resource solver items (from Chapter 12):

| `<item>` | Meaning |
|---|---|
| `rin <solver>` | Set the `rs_invariant` option (Section 12.2) to `true`, run `<solver>`, and then set the `rs_invariant` option to `false`. |
| `rrq` | Call `KheSolnAssignRequestedResources` (Section 12.4.1). |
| `rah <solver>` | Call `KheAssignByHistory` (Section 12.4.2), fix the assignments it makes, run `<solver>`, then remove the fixes. |
| `rmc` | Call `KheMostConstrainedFirstAssignResources` (Section 12.4.3). |
| `rpk` | Call `KheResourcePackAssignResources` (Section 12.4.4). |
| `rcx` | Call `KheResourcePackAssignResources` if any tasks of the current resource type are subject to avoid split assignments constraints, and `KheMostConstrainedFirstAssignResources` if not. |
| `rfs` | If `KheResourceTypeAvoidSplitAssignmentsCount` is 0, do nothing, otherwise call `KheFindSplitResourceAssignments` (Section 12.4.5) followed by `KheTaskingAllowSplitAssignments` (Section 11.5.6). |
| `rdv` | Call `KheDynamicResourceVLSNSolve` (Section 12.6). |
| `rdt` | Call `KheDynamicResourceVLSNTest` (Section 12.6). |
| `rds` | Call `KheDynamicResourceSequentialSolve` (Section 12.6.4). |
| `rts` | Call `KheTimeSweepAssignResources` (Section 12.7.3). |
| `rrm` | Call `KheResourceRematch` (Section 12.7.5). |
| `rrh` | Call `KheRunHomogenize` (Section 12.8). |
| `rmu` | Call `KheMoveUnnecessaryAssignments` (Section 12.9). |
| `rec` | Call `KheEjectionChainRepairResources` (Section 12.10). |
| `rgl` | Call `KheGlobalLoadBalance` (Section 12.11). |
| `rrs` | Call `KheResourcePairRepair` (Section 12.12.2). |
| `rrp` | Call `KheReassignRepair` (Section 12.13). |

`KheSolnTryTaskUnAssignments` has been omitted because the general solver calls it.

What each solver does in detail depends on its own options; `rs` only specifies when (if at all) it gets called, and how much time it has to run, not what it does when it is called. For example, a solver may have an option that turns it off. If that is set, the solver does nothing.

The remarks about time limits in Section 8.3 apply to resource solving. There is an `rs_time_limit` option for placing a time limit on resource assignment. For example,

```
rs_time_limit="10:0"
```

sets an overall time limit for resource assignment (including repair) of 10 minutes. Time weights may be used to apportion the available time among the various solvers as usual.

The `rt <solver>` item calls `<solver>` once for each resource type, raising the question of how the available time is apportioned among the resource types. It is apportioned using the number of resources of each type as the time weight. During setup, each `rt` item is converted into a sequence of solver items, with one item for each resource type, whose time weight is the number of resources in the resource type and whose `<solver>` is `rt`'s `<solver>`.

All of this is carried out by function

```
bool KheCombinedResourceAssign(KHE_SOLN soln, KHE_OPTIONS options);
```

This sets the `rs_time_limit` time limit if that option is present, then assigns resources using the value of option `rs` as its guide, and finally deletes the time limit if it sets it. The easiest way to call it is to include `rs` in the `gs` option (Section 8.4), causing the general solver to call it for you.

The `rs` option has default value

```
gts!do rem rt(
  1: rin!do rrd rwp rcm rbw rah rgc rcx!rts,
  2: rin!do rgc (rfs, rrm, rec, rdv),
  1: (red, rrm, rec, rdv)
)
```

It is spread over several lines here for readability.

The first line, enclosing everything, installs a separate global tixel matching (high school timetabling only), creates an event timetable monitor, then solves each resource type.

The second line is the construction phase. It installs several refinements: the resource assignment invariant (high school timetabling only), redundancy handling, workload packing, cluster minimum limits, weekend balancing, assign by history, and grouping by constraints. It then does the actual construction, using `rcx` (high school timetabling) or `rts` (nurse rostering).

The third line is the first repair phase. After installing fewer refinements, it proceeds to finding split assignments, resource rematching, ejection chains, and VLSN search using optimal reassignment by dynamic programming.

The third line is the second repair phase. It repeats the first repair phase, minus refinements and plus domain enlargement. It is given half the running time of the first repair phase.

All this represents the author's current idea of how best to use the various resource solvers. It will change as his ideas change.

# Chapter 13. Ejection Chains

Ejection chains are sequences of repairs that generalize the augmenting paths from bipartite matching. They are credited to Glover [3], who named them and applied them to the travelling salesman problem. Similar ideas, often not clearly expressed, appear in a few earlier papers.

In this chapter, the first of two devoted to ejection chains, we introduce ejection chains and present *ejectors*, which are KHE objects that supply a general framework for ejection chain solving. This material reflects the content of file `khe_se_ejector.c`, which is unlikely to change much in the future. In the next chapter, we introduce the more volatile code that KHE offers to make ejectors handle specific kinds of defects.

## 13.1. Introduction

This section introduces ejection chains. Some of its statements will need to be qualified later, but they are close enough to the truth to serve for purposes of introduction.

A *defect* is a specific point in a solution where something is not perfect and incurs a non-zero cost. Concretely, it is a monitor (possibly a group monitor) with non-zero cost.

An ejection chain algorithm iterates over the defects of a solution. For each defect it tries a set of alternative *repairs*. A repair could be a simple move or swap, or something arbitrarily complex. It is targeted specifically at the defect, and usually removes it, but it may introduce new defects. If no new defects of significant cost appear, that is success. If just one significant new defect appears, the algorithm calls itself recursively to try to remove it, building up a chain of coordinated repairs. If several significant new defects appear, or the recursive call fails to remove the new defect, it undoes the repair and continues with alternative repairs.

Corresponding to the well-known function for finding an augmenting path in a bipartite graph, starting from a given unassigned node, is this function, formulated by the author, for 'augmenting' (improving) a solution, starting from a given defect:

```
bool Augment(Solution s, Cost c, Defect d);
```

(KHE's interface is somewhat different to this.) `Augment` has precondition

```
cost(s) >= c  &&  cost(s) - cost(d) < c
```

If `Augment` can change `s` to reduce its cost to less than `c`, it does so and returns `true`; if not, it leaves `s` unchanged and returns `false`. Importantly, the precondition implies that removing `d` without adding new defects would succeed. Here is an abstract implementation of `Augment`:

```
bool Augment(Solution s, Cost c, Defect d)
{
  repair_set = RepairsOf(d);
  for( each repair r in repair_set )
  {
    new_defect_set = Apply(s, r);
    if( cost(s) < c )
      return true;
    for( each e in new_defect_set )
      if( cost(s) - cost(e) < c && Augment(s, c, e) )
        return true;
    UnApply(s, r);
  }
  return false;
}
```

It begins by finding a set of alternative ways that `d` could be repaired. For example, an unassigned task could be repaired by assigning any suitable resource to it; an overloaded resource could be repaired by reassigning or unassigning any task that contributes to the overload; and so on. In practice, these repairs don't have to be placed into a set—they just need to be iterated over. For each repair `r`, `Augment` applies `r` and receives the set of new defects introduced by `r`, looks for success in two ways, then if neither of those works out it unapplies the repair and continues by trying the next repair, returning `false` when all repairs have been tried without success.

Success could come in two ways. Either a repair reduces `cost(s)` to below `c`, or some new defect `e` has cost large enough to ensure that removing it alone would constitute success, and a recursive call targeted at `e` succeeds. Notice that `cost(s)` may grow without limit as the chain deepens, while there is a defect `e` whose removal would reduce the solution's cost to below `c`.

The key observation that justifies the whole approach is this: the new defects targeted by the recursive calls are not known to have resisted attack before. It might be possible to repair one of them without introducing any new defects of significant cost.

The algorithm stops at the first successful chain. An option for finding the best successful chain rather than the first has been withdrawn, because of problems in combining it with ejection beams (Section 13.4.5). It is no loss: it produced nothing remarkable, and ran slowly. Another option, for limiting the disruption caused by the repairs, has also been withdrawn. It too was not very useful. It can be approximated by limiting the maximum chain length, as described next.

The tree searched by `Augment` as presented may easily grow to exponential size, which is not the intention. KHE offers two methods of limiting its size, both of which seem useful. They may be used separately or together.

The first method is to limit the maximum chain length to a fixed constant, perhaps 3 or 4. The maximum length is passed as an extra parameter to `Augment`, and reduced on each recursive call, with value 0 preventing further recursion. Not only is this attractive in itself, it also supports *iterative deepening*, in which `Augment` is called several times on the same defect, with the length parameter increased each time. Another idea is to use a small length on the first iteration of the main loop (see below), and increase it on later iterations.

The second method is used by augmenting paths in bipartite matching. Just before each call on `Augment` from the main loop, the entire solution is marked unvisited (by incrementing a global visit number, not by traversing the entire solution). When a repair changes some part of the solution, that part is marked visited. Repairs that change parts of the solution that are already marked visited are tabu. This limits the size of the tree to the size of the solution.

Given a solution and a list of its defects, the main loop cycles through the list repeatedly, calling `Augment` on each defect in turn, with `c` set to `cost(s)`. When the main loop exits, every defect has been tried at least once without success since the most recent success, so no further successful augments are possible, unless there is a random element within `Augment`. Under reasonable assumptions, this very clear-cut stopping criterion ensures that the whole algorithm runs in polynomial time, for the same reason that hill-climbing does.

Defects come in a variety of types, basically one for each monitor type. For example, a missing task assignment (an assign resource monitor defect) is different from a resource overload (a limit busy times or cluster busy times monitor defect). Each type of defect needs its own specialized set of repairs. This idea seems to have been first clearly articulated by the present author, who named the result *polymorphic ejection chains*: the first step in each augment is a dynamic dispatch on the defect type (not shown above).

Careful work is needed to maximize the effectiveness of ejection chains. Grouping together monitors that measure the same thing is important, because it reduces the number of defects and increases their cost, increasing the chance that a chain will be continued. Individual repair operations should actually remove the defects that they are called to repair (the framework does not check this), and should do whatever seems most likely to avoid introducing new defects. We'll consider these points in detail in Section 13.6.

## 13.2. Ejector construction

KHE offers *ejector* objects which provide a framework for ejection chain algorithms, reducing the implementation burden to writing just the augment functions. An ejector object, stored in a given arena `a`, is constructed by a sequence of calls beginning with

```
KHE_EJECTOR KheEjectorMakeBegin(char *schedules, HA_ARENA a);
```

followed by calls which load augment functions, as explained below, and ending with

```
void KheEjectorMakeEnd(KHE_EJECTOR ej);
```

The ejector is then ready to do some solving (Section 13.3). The two attributes are returned by

```
char *KheEjectorSchedules(KHE_EJECTOR ej);
HA_ARENA KheEjectorArena(KHE_EJECTOR ej);
```

There is no function to delete an ejector; it is deleted when its arena is deleted.

The `schedules` string consists of one or more *major schedules* separated by semicolons:

```
<major_schedule>;<major_schedule>; ... ;<major_schedule>
```

Each major schedule consists of one or more *minor schedules* separated by commas:

```
<minor_schedule>,<minor_schedule>, ... ,<minor_schedule>
```

Each minor schedule consists of a positive integer called its *maximum length*, followed by either `"+"` or `"-"`, representing a Boolean value called its *may revisit* attribute. The maximum length may have the special value `"u"`, meaning unlimited. For example, schedule string

```
"1+;u-"
```

contains two major schedules. The first, `"1+"`, has one minor schedule, with 1 for maximum length and `true` for may revisit; the second, `"u-"`, also has one minor schedule, this time with unlimited for maximum length and `false` for may revisit.

The entire main loop of the algorithm, which repeatedly tries to augment out of each defect of the solution until no further improvements can be found, is repeated once for each major schedule in order. Within each iteration of the main loop, the augment for one defect is tried once for each minor schedule of the current major schedule, until an augment succeeds in reducing the cost of the solution or all minor schedules have been tried.

The maximum length determines the maximum number of repairs allowed on one chain. Value 0 allows no repairs at all and is forbidden. Value 1 allows augment calls from the main loop, but prevents them from making recursive calls, producing a kind of hill climbing. Value 2 allows the calls made from the main loop to make recursive calls, but prevents those calls from recursing. And so on.

When the may revisit attribute is `false`, each part of the solution may be changed by at most one of the recursive calls made while repairing a defect; when it is `true`, each part may be changed by any number of them, although only once along any one chain.

It is not good to set the maximum length to a large value and may revisit to `true` in the same minor schedule, because the algorithm will then usually take exponential time. But setting the maximum length to a small constant, or setting may revisit to `false`, or both, guarantees polynomial time. Another interesting idea is *iterative deepening*, in which several maximum lengths are tried. For example,

```
"1+;2+;3+;u-"
```

tries maximum length 1, then 2, then 3, and finishes with unlimited length.

A mentioned earlier, in between calling `KheEjectorMakeBegin` and `KheEjectorMakeEnd`, the augment functions need to be loaded. They are written by the user, as described in Section 13.6, and passed to the ejector by calls to

```
void KheEjectorAddAugment(KHE_EJECTOR ej, KHE_MONITOR_TAG tag,
  KHE_EJECTOR_AUGMENT_FN augment_fn, int augment_type);
void KheEjectorAddGroupAugment(KHE_EJECTOR ej, int sub_tag,
  KHE_EJECTOR_AUGMENT_FN augment_fn, int augment_type);
```

The first says that defects which are non-group monitors with tag `tag` should be handled by `augment_fn`; the second says that defects which are group monitors with sub-tag `sub_tag` should be handled by `augment_fn`. Here `sub_tag` must be between 0 and 29 inclusive. Any values not set are handled by doing nothing, as though an unsuccessful attempt was made to repair them. Ejectors handle the polymorphic dispatch by defect type. The `augment_type` parameter is used by statistics gathering (Section 13.5), and may be 0 if statistics are not wanted.

## 13.3. Ejector solving

Once an ejector has been set up, the ejection chain algorithm may be run by calling

```
bool KheEjectorSolve(KHE_EJECTOR ej, KHE_GROUP_MONITOR start_gm,
  KHE_GROUP_MONITOR continue_gm, KHE_AUGMENT_OPTIONS ao,
  KHE_OPTIONS options);
```

This runs the main loop of the ejection chain algorithm once for each major schedule, returning `true` if it reduces the cost of the solution monitored by `start_gm` and `continue_gm`.

The main loop repairs only the defective child monitors of `start_gm`, and the recursive calls repair only the defective child monitors of `continue_gm`. These two group monitors could be equal, and either or both could be an upcast solution. Although it is not required, in practice every child monitor of `start_gm` would also be a child monitor of `continue_gm`.

Parameter `ao` contains options for the detailed control of augment functions. The ejector object does not know what these are; it just passes `ao` back to the user's augment functions, which use it to control their detailed behaviour (Section 13.6.4).

Just as an ejector is constructed by a sequence of calls enclosed in `KheEjectorMakeBegin` and `KheEjectorMakeEnd`, so a solve is carried out by a sequence of calls beginning with

```
void KheEjectorSolveBegin(KHE_EJECTOR ej, KHE_GROUP_MONITOR start_gm,
  KHE_GROUP_MONITOR continue_gm, KHE_AUGMENT_OPTIONS ao,
  KHE_OPTIONS options);
```

and ending with

```
bool KheEjectorSolveEnd(KHE_EJECTOR ej);
```

`KheEjectorSolveEnd` does the actual solving. Function `KheEjectorSolve` above just calls `KheEjectorSolveBegin` and `KheEjectorSolveEnd` with nothing in between.

The only functions callable between `KheEjectorSolveBegin` and `KheEjectorSolveEnd` (at least, the only ones that change anything) are

```
void KheEjectorAddMonitorCostLimit(KHE_EJECTOR ej,
  KHE_MONITOR m, KHE_COST cost_limit);
void KheEjectorAddMonitorCostLimitReducing(KHE_EJECTOR ej,
  KHE_MONITOR m);
```

The call `KheEjectorAddMonitorCostLimit(ej, m, cost_limit)` says that for a chain to end successfully, not only must the solution cost be less than the initial cost, but `KheMonitorCost(m)` must be no larger than `cost_limit`. `KheEjectorAddMonitorCostLimitReducing(ej, m)` is the same except that the cost limit is initialized to `KheMonitorCost(m)`, and if a successful chain is found and applied which reduces `KheMonitorCost(m)` to below its current limit, that limit is reduced to the new `KheMonitorCost(m)` for subsequent chains.

To visit these *limit monitors*, call

```
int KheEjectorMonitorCostLimitCount(KHE_EJECTOR ej);
void KheEjectorMonitorCostLimit(KHE_EJECTOR ej, int i,
  KHE_MONITOR *m, KHE_COST *cost_limit, bool *reducing);
```

The returned values are the monitor, its current cost limit, and whether that limit may be reduced. Any number of limit monitors may be added, but large numbers will not be efficient.

Each time the ejector enters the main loop, it makes a copy of `start_gm`'s list of defects and sorts the copy by decreasing cost. Ties are broken differently depending on the value of the solution's diversifier. If the `es_limit_defects` option is set to some integer (not to `"unlimited"`), defects are dropped from the end of the sorted list to ensure that there are no more than `es_limit_defects` of them.

Consider a defect *d* that the main loop of the ejection chain solver is just about to attempt to repair. Suppose that the most recent change either to the solution or to the major schedule occurred before the most recent previous attempt to repair *d*. Then, if the repair is deterministic, the current attempt to repair *d* is certain to fail like the previous attempt did. Accordingly, it is skipped. The implementation of this optimization uses visit numbers stored in monitors.

In practice, repairs are not deterministic, since, for diversity, KHE's augment functions vary the starting points of their traversals of lists of repairs between calls. However, the author carried out an experiment on a large instance (NL-KP-03), in which this optimization was turned off but a check was made to see whether there were any cases where repairs which it would have caused to be skipped were successful. Over 8 diversified solves there were 15 cases.

The following functions may be called while `KheEjectorSolve` is running (that is, from within augment functions):

```
KHE_GROUP_MONITOR KheEjectorStartGroupMonitor(KHE_EJECTOR ej);
KHE_GROUP_MONITOR KheEjectorContinueGroupMonitor(KHE_EJECTOR ej);
KHE_OPTIONS KheEjectorOptions(KHE_EJECTOR ej);
KHE_SOLN KheEjectorSoln(KHE_EJECTOR ej);
KHE_COST KheEjectorTargetCost(KHE_EJECTOR ej);
bool KheEjectorCurrMayRevisit(KHE_EJECTOR ej);
int KheEjectorCurrLength(KHE_EJECTOR ej);
int KheEjectorCurrAugmentCount(KHE_EJECTOR ej);
bool KheEjectorCurrDebug(KHE_EJECTOR ej);
int KheEjectorCurrDebugIndent(KHE_EJECTOR ej);
```

`KheEjectorStartGroupMonitor`, `KheEjectorContinueGroupMonitor`, and `KheEjectorOptions` are `start_gm`, `continue_gm`, and `options` from `KheEjectorSolve`.

`KheEjectorSoln` is `start_gm`'s and `continue_gm`'s solution. `KheEjectorTargetCost` is the cost that the chain needs to improve on (`c` in the abstract code above): the cost of the solution when `Augment` was most recently called from the main loop. `KheEjectorCurrMayRevisit` is the `may_revisit` attribute of the current minor schedule.

`KheEjectorCurrLength` is 1 when the augment function was called from the main loop, 2 when it was called from an augment function called from the main loop, etc.

`KheEjectorCurrAugmentCount` is the number of augments since this solve began. `KheEjectorCurrDebug` returns `true` when `ej` is currently debugging, because it is trying to repair a main loop defect in the monitor stored in the `es_debug_monitor` option of `options`. It seems to work well for each repair to generate a one-line description of itself when `KheEjectorCurrDebug` is `true`. `KheEjectorCurrDebugIndent` is the current amount by which debug prints should be indented; this is twice the current length.

For completeness we just mention two other functions:

```
void KheEjectorRepairBegin(KHE_EJECTOR ej);
bool KheEjectorRepairEnd(KHE_EJECTOR ej, int repair_type, bool success);
```

Each repair must be bracketed in calls to these. They will be discussed in Section 13.6.4.

`KheEjectorSolveBegin` is affected by `options`. It extracts the following options from its `options` argument and uses them to control the ejector during solving:

`es_max_augments`
: An integer upper limit on the number of augments (including recursive calls) tried on each attempt to repair one main loop defect. The default value is `120`, which may be better than a larger number: when there is a time limit, it reduces time wasted on lost causes.

`es_max_repairs`
: An integer upper limit on the number of repairs tried on each attempt to repair a main loop defect. The default value is `INT_MAX`.

`es_limit_defects`
: An option whose value is either `"unlimited"` or an integer. This integer is a limit on the number of defects handled by the main loop of the ejector. Each time the main list of defects is copied and sorted, if its size exceeds this limit, defects are dropped from the end until it doesn't. When the option is not set, or its value is `"unlimited"`, no limit is applied.

`es_fresh_visits`
: A Boolean option which, when `true`, instructs an ejector to make fresh visits (Section 13.4.2).

`es_max_beam`
: An integer option which gives the maximum number of monitors that may appear in a beam (Section 13.4.5). The default value is 1, producing ejection chains rather than beams.

`gs_debug_monitor_id`
: A string option, also used by other solvers, which instructs the ejector object to generate debug output when repairing the monitor with this Id.

`es_whynot_monitor_id`
: A string option which instructs the ejector object to end its run with an augment of the monitor with this Id with debugging on (unless its cost is 0). This will show why this monitor was not able to be repaired. (Just occasionally, this attempt will succeed, and then the improvement is incorporated into the solution returned.)

`KheEjectorMakeBegin` and `KheEjectorMakeEnd` are not affected by `options`.

## 13.4. Variants of the ejection chains idea

This section presents some variants of the basic ejection chains idea.

### 13.4.1. Defect promotion

*This feature has been withdrawn to streamline the implementation. It has had some successes but not enough, in the author's judgment, to justify a place in the ejector framework.*

Successful chains begin by repairing a defect which is one of `start_gm`'s children, and they continue by repairing defects which are children of `continue_gm`. The intention is that `start_gm` should monitor some region of the solution that has only just been assigned, so that there has been no chance yet to repair its defects, while `continue_gm` monitors the entire solution so far, or the part of it that is relevant to repairing the defects of `start_gm`. These two regions may be the same, which is why `start_gm` and `continue_gm` may be the same group monitor; but when they are different, the difference is important, as the following argument shows.

Suppose only `start_gm` is used. Then the ejector sets out to repair the right defects, but is unable to follow chains of repairs into parts of the solution that have been assigned previously. Or suppose only `continue_gm` is used. If the children of `continue_gm` are a superset of the children of `start_gm`, as is always the case in practice, this does allow a full search, but at the cost of trying again to repair many defects for which a previous repair attempt failed (those in `continue_gm` which are not also in `start_gm`). This can waste a lot of running time.

At this point, however, an unexpected issue enters. Suppose a successful chain is found which causes some child `d` of `continue_gm` to become defective, but which nevertheless terminates without repairing `d` because it improves the overall solution cost. Here is a new defect, a child of `continue_gm` not known to have been repaired previously, and thus worthy of being targeted for repair; but if it is not also a child of `start_gm`, it won't be.

*Defect promotion* addresses this issue. When an ejection chain is declared successful, the ejector examines the defects created by that chain's last repair that are children of `continue_gm`. It makes any of these that are not children of `start_gm` into children of `start_gm`: they get dynamically added to the set of defects targeted by the current solve. Of course, when `start_gm` and `continue_gm` are the same, it does nothing.

Defect promotion is optional, controlled by option `es_no_promote_defects`, whose default value is `false`. On one run it reduced the final solution cost from 0.04571 to 0.03743, while increasing running time from 286.84 seconds to 490.21 seconds—a substantial amount, but nothing like what would have occurred if `start_gm` had been replaced by `continue_gm`.

### 13.4.2. Fresh visit numbers for sub-defects

It is common for a monitor to monitor several points in the solution. For example, a prefer times monitor monitors several meets, all those derived from one point of application of the corresponding prefer times constraint (one event). Arguably, having one monitor for each meet would make more sense; but there is a problem with this, at least when the cost function is not `Linear`, because then there is no well-defined value of the cost of such a monitor. A cost is only defined after all the deviations of the *sub-defects* at all the monitored points are added up.

The usual way to repair a defective monitor which monitors several points is to visit each point, determine whether that point is a sub-defect, and try some repairs if so. When the repair is of a main loop defect (when the current length is 1), it makes sense for the augment function to give a fresh visit number to each sub-defect, so that the repair at each sub-defect is free to search the whole solution, as in this template:

```
for( i = 0;  i < KheMonitorPointCount(m);  i++ )
{
  p = KheMonitorPoint(m, i);
  if( KheMonitorPointIsDefective(p) )
  {
    if( KheEjectorCurrLength(ej) == 1 )
      KheSolnNewGlobalVisit(soln);
    if( KheMonitorPointTryRepairs(p) )
      return;
  }
}
```

Calling `KheSolnNewGlobalVisit` opens up the whole solution for visiting. This is what would happen if the monitor was broken into smaller monitors, one for each point. It is important, however, not to call `KheSolnNewGlobalVisit` at deeper levels, since that amounts to allowing revisiting, so it leads to exponential time searches.

Fresh visit numbers are *not* assigned in this way within the augment functions supplied with KHE. Instead, a more radical version of the idea is offered by the `es_fresh_visits` option. When set to `true`, it causes

```
if( KheEjectorCurrLength(ej) == 1 )
  KheSolnNewGlobalVisit(soln);
```

to be executed within each call to `KheEjectorRepairBegin`, opening up the entire solution, not just to each sub-defect at length 1, but to each repair of each sub-defect at length 1.

### 13.4.3. Ejection trees

*The variant described here has been withdrawn. Ejection beams (Section 13.4.5) perform a similar function more simply.*

An *ejection tree* is like an ejection chain except that at each level below the first, instead of repairing one newly introduced defect, it tries to repair several (or all) of the newly introduced defects, producing a tree of repairs rather than a chain.

Ejection trees are not likely to be useful often. It is true that the run time of an ejection tree is limited as usual by the size of the solution, but its chance of success is lower than usual, because it must repair several defects at the lower level to succeed at the higher level. If repairing the first defect produces two new defects, repairing each of those produces two more, and so on, then the result is a huge number of defects that must all be repaired successfully. And to make a repair which introduces a defect and then repair that defect using an ejection tree is to spend a lot of time on a defect that can be removed much more easily by undoing the initial repair.

However, when the original solution has a very awkward defect, the best option may be a complex repair which usually introduces several new defects. For example, the best way to repair a cluster busy times overload defect may be to unassign every meet on one of the problem days. In that case, it makes sense to use an ejection tree at that level alone: that is, to try a repair that introduces several defects, then try to repair them by finding an ejection chain for each.

The `max_sub_chains` parameter of `KheEjectorRepairEndLong` (*withdrawn*) allows for ejection trees, by specifying the maximum number of defects introduced by that repair that are to be repaired. Different repairs may have different values of `max_sub_chains`. For example, the complex cluster busy times repair could be tried only when `KheEjectorCurrLength(ej)` is 1, with `max_sub_chains` set to `INT_MAX`. All other repairs could be given value 1 for `max_sub_chains`, producing ordinary chains elsewhere.

A set of defects now has to be repaired, not necessarily just one. One option would have been to change the interface of `Augment` to pass this set to the user. This was not done, because it would be a major change from the targeted repairs used by ejection chains. Instead, just as the framework handles the dynamic dispatch by defect type, so it also accepts a whole set of defects for repair and passes them one by one to conventional `Augment` calls.

The remainder of this section explains the implementation of ejection trees (and indeed ejection chains) by presenting a more detailed description of the `Augment` function than the one given at the start of this chapter.

To begin with, it was stated earlier that the main loop tries an augment for every defective child of `start_gm`. In fact, main loop augments are tried only for defects `d` such that

```
Potential(d) = KheMonitorCost(d) - KheMonitorLowerBound(d)
```

is positive. Clearly, when `Potential(d) == 0` there is no chance of improvement.

We also need to consider monitor cost limits, which require that the solution not change so as to cause the cost of some given monitors to exceed given limits (Section 13.3). To handle them, the interface of `Augment` is changed to

```
bool Augment(Solution s, Cost c, Limits x, Defect d);
```

where `x` is a set of monitor cost limits. `Augment` returns `true` if the value of `s` afterwards is such that `s`'s cost is less than `c` and the limits `x` are all satisfied. This condition is evaluated by

```
bool Success(Solution s, Cost c, Limits x)
{
  return cost(s) < c && LimitsAllSatisfied(s, x);
}
```

The precondition of `Augment(s, c, x, d)` is changed to

```
!Success(s, c, x) && cost(s) - GPotential(d) < c
```

That is, the solution must not have already reached the target cost and limits, and the *generalized potential* of `d`, `GPotential(d)`, is large enough to suggest that repairing `d` might get it there. Its postcondition is `Success(s, c, x)` if `true` is returned, and ‘`s` is unchanged’ otherwise.

For a main loop defect, `cost(s)` is `c`, `GPotential(d)` is `Potential(d)`, and `Augment` may be called exactly when it *is* called—when `Potential(d) > 0`. For defects at lower levels, `GPotential(d)` is the amount that the cost of `d` increased when the repair that produced `d` occurred, as returned by `KheTraceMonitorCostIncrease` (Section 6.9.3). Often, the cost beforehand will be `KheMonitorLowerBound(d)`, so that this increase will just be `Potential(d)` as before; but if `d` was already defective beforehand it will be smaller, making `d` less likely to be augmented. The point is that we can only realistically hope to remove the new cost added when the previous repair was made, not the pre-existing cost. (KHE used `Potential(d)` here for many years; the switch to `GPotential(d)` produced better cost on average, and better running time; a few individual instances had marginally worse cost.)

The new defects chosen for repair must be *open defects*: defects whose generalized potential is positive, according to `KheTraceMonitorCostIncrease`. The `max_sub_chains` open defects of largest generalized potential, or all open defects if fewer than `max_sub_chains` open defects are reported by the trace, are selected. In the code below, this selection is made by line

```
{d1, ..., dn} = SelectOpenDefects(new_defect_set, MaxSubChains(r));
```

This is implemented by a call to `KheTraceReduceByCostIncrease` (Section 6.9.3).

Here is the more detailed implementation of `Augment`:

```
bool Augment(Solution s, Cost c, Limits x, Defect d)
{
  repair_set = RepairsOf(d);
  for( each repair r in repair_set )
  {
    new_defect_set = Apply(s, r);
    if( Success(s, c, x) )
      return true;
    if( NotAtLengthLimit() )
    {
      {d1, ..., dn} = SelectOpenDefects(new_defect_set, MaxSubChains(r));
      for( i = 1;  i <= n;  i++ )
      {
        sub_c = c + GPotential(d(i+1)) + ... + GPotential(dn);
        sub_x = (i < n ? {} : x);  /* empty limit set except at end */
        if( Success(s, sub_c, sub_x) )
          continue;
        if( cost(s) - GPotential(di) >= sub_c )
          break;
        if( !Augment(s, sub_c, sub_x, di) )
          break;
        if( Success(s, c, x) )
          return true;
      }
    }
    reset s to its state just before Apply(s, r);
  }
  return false;
}
```

As before, all of this except the loop that iterates over and applies repairs is hidden in calls to

KheEjectorRepairBegin and KheEjectorRepairEnd. It is easy to verify that this satisfies the revised postcondition. The reset near the end is carried out by a call to KheMarkUndo.

After the usual test for success immediately after the repair, if the length limit has not been reached the new code selects n open defects for repair, then calls Augment recursively on each in turn. The complicating factor is the choice of a target cost and set of limits for each recursive call, denoted sub_c and sub_x above. Using the original c and x, as is done with ejection chains, would wrongly place the entire burden of improving the solution onto the first recursive call.

When repairing d1, the right cost target to shoot for is

```
sub_c = c + GPotential(d2) + ... + GPotential(dn);
```

The best that can be hoped for from repairing d2 is GPotential(d2), the best that can be hoped for from repairing d3 is GPotential(d3), etc. So if the first recursive Augment cannot reduce cost(s) below the given value of sub_c, there is little hope that after all the recursive augments it will be reduced below c. The same idea is applied for each of the di.

When a recursive call to Augment changes the solution, some GPotential(di) values may change. So this code re-evaluates sub_c from scratch on each iteration of the inner loop, rather than attempting to save time by adjusting the previous value of sub_c.

The choice of sub_x causes limits to be ignored except when carrying out the last augment. This is in accord with the intention of monitor cost limits, which is to only check them at the end. It would be a mistake to check them earlier. For example, the repair of the cluster busy times defects described above is likely to violate a monitor cost limit when it deassigns meets. These do need to be reassigned by the end, but they will not all be reassigned earlier.

After defining sub_c and sub_x but before the call to Augment, the code executes

```
if( Success(s, sub_c, sub_x) )
  continue;
if( cost(s) - GPotential(di) >= sub_c )
  break;
```

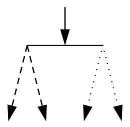
These lines ensure that the precondition of the recursive Augment call holds at the time it is made. If Success(s, sub_c, sub_x) holds, then the aim of that call has already been achieved, so the algorithm moves on to the next one. It does not matter that it skips the Success(s, c, x) test further on, because there has been such a test since the last time the solution changed. If cost(s) - GPotential(di) >= sub_c holds, then the algorithm has no real hope of beating sub_c by repairing di, and so no real hope of success at all, so it abandons the current repair.

Success(s, c, x) implies Success(s, sub_c, sub_x) throughout Augment, because sub_c >= c and sub_x is a subset of x. This cannot be used to simplify Augment, but it does have one or two interesting consequences. For example, it applies transitively down through all active calls to Augment, so while Success(s, sub_c, sub_x) is false at any level of recursion, the original aim of the ejection tree cannot be satisfied.

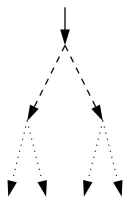
When repairing dn, sub_c == c and sub_x == x. This gives confidence that Augment could succeed, and shows that it reduces to the original Augment when MaxSubChains(r) == 1, except for the different expression of how one open defect is selected.

The method described here finds the first chain that repairs d1, fixes it, and moves on to d2.

Representing the higher path by a solid arrow, the chains (successful or not) that repair `d1` by dashed arrows, and the chains (successful or not) that repair `d2` by dotted arrows, the picture is



Another possibility is to find the first chain that repairs `d1`, then try to find chains for `d2`, but if that fails, to continue searching for other chains for `d1`:



This approach is implementable within the current framework, but it has not been tried. Ejection beams (Section 13.4.5) do something of this kind.

### 13.4.4. Sorting repairs

*The variant described here never performed very well, and it has now been withdrawn.*

Each repair is usually followed immediately by recursive calls which extend the chain, where applicable. Setting the `save_and_sort` parameter of `KheEjectorRepairEndLong` (*withdrawn*) to `true` invokes a different arrangement. Paths representing the repairs are saved in the ejector without recursion. After the last repair they are sorted into increasing order of the cost of the solutions they produce, and each is tried in turn, just as though they had occurred in that sorted order [5].

In practice, `save_and_sort` would be given the same value for every repair of a given defect. However, it is legal to use a mixture of values. Those given value `true` will be saved, those given value `false` will be recursed on immediately in the usual way. If any of those lead to success, that chain is accepted and any saved repairs are forgotten.

Only repairs with some hope of success are saved: those for which

```
Success(s, c, x) || (NotAtLengthLimit() &&
  cost(s) - (GPotential(d1) + ... + GPotential(dn)) < c)
```

holds after the repair, in the terminology of Section 13.4.3.

The author's experience with `save_and_sort` has been disappointing. Chains can end successfully anywhere in the search tree, and low solution cost at an intermediate point is not a good predictor of a successful end. Every saved repair is executed once before sorting to establish the solution cost after it, then undone. If the repair is tried later, it is executed again (by a path redo). The significant benefit needed to justify this extra work does not seem to be there.

### 13.4.5. Ejection beams

*Ejection beams* are yet another variant of the basic ejection chains idea. They are similar to the now-withdrawn ejection trees in that they aim to allow the algorithm to carry on when more than one defect is introduced by a repair.

An *ejection beam*, or just *beam*, is a non-empty set of monitors `m`, each with its *cost*, denoted `cost(m)`, and its *target*, denoted `targ(m)`. The value of `cost(m)` is the cost of `m` in the current solution, as usual. The value of `targ(m)` will be specified shortly; it is a value that the algorithm aspires to reduce `cost(m)` to.

A beam is a true set: no monitor may appear in it twice. Its cardinality (its number of monitors) is at least 1 and at most some small integer `K` (a parameter of the algorithm) whose value must be at least 1. `K` might be 2 or 3, for example.

Instead of passing a single defective monitor, the basic call passes an entire beam `B`:

```
bool Augment(Solution s, Cost c, Beam B)
```

As usual, if `Augment` can find a way to reduce the cost of `s` to less than `c`, it does so and returns `true`, otherwise it changes nothing and returns `false`. But now it has a set of monitors, all those in `B`, to repair. When `Augment` is called, each of these monitors `m` is unvisited and satisfies

```
cost(m) > targ(m) >= 0
```

Furthermore, along with the usual `cost(s) >= c`, the condition

```
Open(s, c, B) = cost(s) - sum[m in B](cost(m) - targ(m)) < c
```

holds initially. So one way for `Augment` to succeed would be to reduce the cost of every monitor in `B` down to its target without introducing any new defects.

Like the ejection chain algorithm, the ejection beam algorithm has a main loop that tries to repair each top-level defect `m` whose cost exceeds its lower bound. To do this it first marks all monitors unvisited, then it calls `Augment`, setting `s` to the current solution, `c` to the cost of the current solution, and `B` to a beam consisting of a single element, `m`, setting `targ(m)` to `m`'s lower bound. This clearly satisfies the initial conditions of `Augment`.

Here is the implementation of `Augment`. We've added a `Limits` parameter which handles the limits imposed by `KheEjectorAddMonitorCostLimit` (Section 13.3):

```
bool Augment(Solution s, Cost c, Limits x, Beam B)
{
  choose an arbitrary element d of B;
  MonitorSetVisited(d);
  repair_set = RepairsOf(d);
  for( each repair r in repair_set )
  {
    new_defect_set = Apply(s, r);
    if( Success(s, c, x) )
      return true;
    if( LimitsNotReached() && BeamMerge(s, c, B, new_defect_set, K, &B2) )
    {
      if( Augment(s, c, x, B2) )
        return true;
    }
    reset s to its state just before Apply(s, r);
  }
  return false;
}
```

Here `Success` compares the solution cost with `c` and checks the limits:

```
bool Success(Solution s, Cost c, Limits x)
{
  return cost(s) < c && LimitsAllSatisfied(s, x);
}
```

while `LimitsNotReached` returns `true` if the various limits, on number of augments and so on, have not yet been reached.

Apart from a few minor details, this version of `Augment` is clearly the original one with a set of monitors `B` instead of a single defect `d`. It demands nothing additional from the user except a single value for `K` at the start.

The key new step is `BeamMerge`. It returns a new non-empty beam `B2` which is the set union of `B` and `new_defect_set`, sorted into decreasing `cost(m) - targ(m)` order. Monitors are omitted when they are visited, and they are dropped from the end of the sorted list until just before dropping another would make `Open(s, c, B2) false`. If the number of monitors remaining is less than 1 or more than `K`, `MakeBeam` returns `false`. So if the recursive call to `Augment` is made, `B2` is a beam of legal size which satisfies the precondition of `Augment`.

The author was surprised to discover that beams of size 0 could emerge from `BeamMerge`. They occur when the cost has been reduced to a new best but limit monitors prevent the current solution from being declared a success.

A new monitor `m` entering `B2` has its `targ(m)` value set to its cost before `r` was applied. Clearly, `cost(m) > targ(m) >= 0`, because `targ(m)` is `cost(m)` before `r` was applied, and it was `r` that caused `m` to appear in `new_defect_set`, meaning that `m`'s cost increased. The algorithm aspires to remove the new defects introduced by `r`, which in the case of `m` means returning its cost to what it was before `r`, so this is a suitable value for `targ(m)`.

A monitor `m` could lie in both `B` and `new_defect_set`, if it was a defect before applying `r`, and `r` made it worse. That is fine; it will appear only once in `B2`, with its original `targ(m)`.

Ejectors always run the ejection beam code, but the default value of `K` is 1, producing ejection chains. `K` is determined by the `es_max_beam` option (Section 13.3), so setting this to a value larger than 1 is the way to get ejection beams.

## 13.5. Gathering statistics

Ejectors gather statistics about their performance. This takes a negligible amount of time, as the author has verified by comparing run times with preprocessor flag `KHE_EJECTOR_WITH_STATS` in the ejector source file set to 0 (no statistics) and 1 (all statistics). On two typical instances, the increase in overall run time caused by gathering statistics was less than 0.1 seconds.

### 13.5.1. Options for choosing ejectors and schedules

Each ejector holds its own statistics, independently of other ejectors. Some statistics accumulate across the entire lifetime of an ejector; they are never reset. This makes it possible, for example, to measure the performance of time repair ejection chains and resource repair ejection chains over an entire set of instances, by carrying out all time repairs in all instances using one ejector and all resource repairs in all instances using another.

To facilitate this, options objects usually contain two ejectors, under names `"es_ejector1"` and `"es_ejector2"`, as explained in Section 13.6.1; they could contain more.

The next question is what schedules to give to these ejectors. A set of schedules is an option, so the `options` object has option `es_schedules` for it, whose value is a string. Its default value is `"1+,u-"`, for the meaning of which see Section 13.2.

Setting the schedule string does not set any ejector schedules, it merely sets one option of `options`, to a fresh copy of the string it is given. User code must set the actual schedules, by passing a schedule string to function `KheEjectorMakeBegin` (Section 13.2).

### 13.5.2. Statistics for analysing Kempe meet moves

The ejector itself does not maintain statistics for analysing Kempe meet moves. These are stored in `kempe_stats` objects, one of which is conveniently available from option `ts_kempe_stats` (Section 10.2.2). This object is passed to the calls to `KempeMeetMove` made by the augment functions described in this chapter. Only Kempe meet moves which are complete repairs on their own are passed this object, not Kempe meet moves combined with other operations (meet splits and merges, for example). So by the end of an ejction chain run, statistics about these Kempe meet moves will have been accumulated in the `ts_kempe_stats` option of the `options` object passed to the ejection chain repairs.

### 13.5.3. Statistics describing a single solve

The statistics presented in this section make sense only for one call to `KheEjectorSolveEnd`. So they are available only until the next call to `KheEjectorSolveEnd`, when they are reset.

An *improvement* is an ejection chain or tree, rooted in a defect examined by the main loop, which is applied to the solution and reduces its cost. Each time an improvement is applied, four facts about it are recorded. The number of improvements applied is returned by

```
int KheEjectorImprovementCount(KHE_EJECTOR ej);
```

and the four facts about the `i`th improvement (counting from 0 as usual) are returned by

```
int KheEjectorImprovementNumberOfRepairs(KHE_EJECTOR ej, int i);
float KheEjectorImprovementTime(KHE_EJECTOR ej, int i);
KHE_COST KheEjectorImprovementCost(KHE_EJECTOR ej, int i);
int KheEjectorImprovementDefects(KHE_EJECTOR ej, int i);
```

These return the number of repairs in the `i`th improvement (this tends to increase with `i`), the time from the moment when `KheEjectorSolveEnd` was called to the moment after the improvement was applied, the solution cost afterwards, and the number of defects of `start_gm` afterwards. Times are measured in seconds, to a precision much better than one second. There are also

```
KHE_COST KheEjectorInitCost(KHE_EJECTOR ej);
int KheEjectorInitDefects(KHE_EJECTOR ej);
```

which return the cost and number of defects when `KheEjectorSolve` began.

### 13.5.4. Statistics describing multiple solves

The statistics presented in this section make sense across multiple calls to `KheEjectorSolveEnd`. They are initialized when the ejector is created and never reset.

It is interesting to see how many repairs make up one improvement. Each time an improvement occurs on any solve during the lifetime of the ejector, one entry in a histogram of numbers of repairs is incremented. This histogram can be accessed at any time by calling

```
int KheEjectorImprovementRepairHistoMax(KHE_EJECTOR ej);
int KheEjectorImprovementRepairHistoFrequency(KHE_EJECTOR ej,
  int repair_count);
```

`KheEjectorImprovementRepairHistoMax` returns the maximum, over all improvements *x*, of the number of repairs that make up *x*, or 0 if there have been no improvements. `KheEjectorImprovementRepairHistoFrequency` returns the number of improvements with the given number of repairs. Also, functions

```
int KheEjectorImprovementRepairHistoTotal(KHE_EJECTOR ej);
float KheEjectorImprovementRepairHistoAverage(KHE_EJECTOR ej);
```

use this same basic information to find the total number of improvements, and the average number of repairs per improvement when there is at least one improvement.

Another histogram, again with one element for each improvement, records the number of calls to `Augment` since the most recent one in the main loop:

```
int KheEjectorImprovementAugmentHistoMax(KHE_EJECTOR ej);
int KheEjectorImprovementAugmentHistoFrequency(KHE_EJECTOR ej,
  int augment_count);
int KheEjectorImprovementAugmentHistoTotal(KHE_EJECTOR ej);
float KheEjectorImprovementAugmentHistoAverage(KHE_EJECTOR ej);
```

This is helpful, for example, in deciding whether it would be useful to terminate a search after some number of augments has failed to find an improvement. A method of doing this is built into ejectors, but not offered as an official option at the moment.

Another interesting question is how successful the various augment functions and repairs are. There are methodological issues here, however. For example, if one kind of repair is tried before another, it has more opportunities to both succeed and fail than the other. If there are several alternatives to choose from, the best test would be to compare the results of several complete runs, one for each alternative. No statistical support is needed for that. But even after the best alternatives are chosen, there remains the question of whether each component is pulling its weight. The statistics to be described now attempt to answer this question.

An *augment type* is a small integer representing one kind of augment function. A *repair type* is a small integer representing one kind of repair. Functions `KheEjectorAddAugment` and `KheEjectorAddGroupAugment` assign an augment type to each augment function, and thus to each call on an augment function. Each repair is followed by a call to `KheEjectorRepairEnd` (Section 13.6.4), and its `repair_type` parameter assigns a repair type to that repair. Based on this information, the ejector records the following statistics:

1. For each distinct `augment_type`, the number of repairs made by calls on augment functions with that augment type;

2. For each distinct `(augment_type, repair_type)` pair, the number of repairs of that repair type made by calls on augment functions with that augment type;

3. For each distinct `augment_type`, the number of successful repairs made by calls on augment functions with that augment type;

4. For each distinct `(augment_type, repair_type)` pair, the number of successful repairs of that repair type made by calls on augment functions with that augment type.

Only repairs with a `true` value for the `success` parameter of `KheEjectorRepairEndLong` are counted. For the purposes of statistics gathering, a repair is considered successful if it causes its enclosing `Augment` function to return `true`, whether this happens immediately, or after recursion, or after saving and sorting. The statistics may be retrieved at any time by calling

```
int KheEjectorTotalRepairs(KHE_EJECTOR ej, int augment_type);
int KheEjectorTotalRepairsByType(KHE_EJECTOR ej, int augment_type,
  int repair_type);
int KheEjectorSuccessfulRepairs(KHE_EJECTOR ej, int augment_type);
int KheEjectorSuccessfulRepairsByType(KHE_EJECTOR ej, int augment_type,
  int repair_type);
```

where `augment_type` and `repair_type` are arbitrary non-negative integers. Based on these numbers, a reasonable analysis of the effectiveness of the augment functions and their repairs can be made. For example, the effectiveness of an augment function can be measured by the ratio of the third number to the first. Adding up the result of `KheEjectorTotalRepairsByType` over all values of `repair_type` produces the result of `KheEjectorTotalRepairs`, and adding up the result of `KheEjectorSuccessfulRepairsByType` over all values of `repair_type` produces the result of `KheEjectorSuccessfulRepairs`.

### 13.5.5. Organizing augment and repair types

`KheEjectorAddAugment` and `KheEjectorAddGroupAugment` accept any `augment_type` values. The user should define these values using an enumerated type. The following function may be called any number of times during the ejector's setup phase, to tell it what values to expect:

```
void KheEjectorAddAugmentType(KHE_EJECTOR ej, int augment_type,
  char *augment_label);
```

This tells `ej` to expect calls to `KheEjectorAddAugment` and `KheEjectorAddGroupAugment` with the given value of `augment_type`, and associates a label with that augment type. Labels must be non-`NULL`; copies are stored, not originals. No checks are made that the values passed via `KheEjectorAddAugment` and `KheEjectorAddGroupAugment` match those declared by `KheEjectorAddAugmentType`. But if they do, then making tables of statistics is simplified by calling the following functions afterwards.

To visit all the augment types declared by calls to `KheEjectorAddAugmentType`, call

```
int KheEjectorAugmentTypeCount(KHE_EJECTOR ej);
int KheEjectorAugmentType(KHE_EJECTOR ej, int i);
```

To retrieve the label corresponding to an augment type, call

```
char *KheEjectorAugmentTypeLabel(KHE_EJECTOR ej, int augment_type);
```

In this way, suitable values for passing to `KheEjectorTotalRepairs` and the other statistics functions above can be generated, along with labels to identify the statistics.

The same functionality is offered for repair types. `KheEjectorRepairBegin` may be passed any values for `repair_type`, but the user knows which values will be passed, and the following function may be called any number of times during the ejector's setup phase to tell it this:

```
void KheEjectorAddRepairType(KHE_EJECTOR ej, int repair_type,
  char *repair_label);
```

`KheEjectorAddRepairType` declares that `ej` can expect calls to `KheEjectorRepairBegin` with the given value of `repair_type`, and associates a label with that repair type. Labels must be non-`NULL`; copies are stored, not originals. No checks are made that the values passed via `KheEjectorRepairBegin` match those declared by `KheEjectorAddRepairType`. But if they do, then making tables of statistics is simplified by calling the following functions afterwards.

To visit all the repair types declared by calls to `KheEjectorAddRepairType`, call

```
    int KheEjectorRepairTypeCount(KHE_EJECTOR ej);
    int KheEjectorRepairType(KHE_EJECTOR ej, int i);
```

To retrieve the label corresponding to a repair type, call

```
    char *KheEjectorRepairTypeLabel(KHE_EJECTOR ej, int repair_type);
```

There is no way to declare which combinations of augment type and repair type to expect. The author handles this by ignoring cases where `KheEjectorTotalRepairsByType` returns 0.

## 13.6. Using ejection chains in practice

In the remaining sections of this chapter, we focus on the practical side of getting ejectors to repair solutions. The code presented in these sections, and found in files `khe_se_focus.c` and `khe_se_solvers.c`, is more volatile than the code in `khe_se_ejector.c` defining ejector objects—more subject to change as new ideas come along.

### 13.6.1. Top-level ejection chains functions

KHE offers several top-level functions for repairing solutions using ejection chains:

```
    bool KheEjectionChainNodeRepairTimes(KHE_NODE parent_node,
      KHE_OPTIONS options);
    bool KheEjectionChainLayerRepairTimes(KHE_LAYER layer,
      KHE_OPTIONS options);
    bool KheEjectionChainRepairResources(KHE_TASKING tasking,
      KHE_OPTIONS options);
```

These are all packaged up ready for the end user to call. `KheEjectionChainNodeRepairTimes` repairs the assignments of the meets of the descendants of the child nodes of `parent_node`, and `KheEjectionChainLayerRepairTimes` repairs the assignments of the meets of the descendants of the child nodes of `layer`. This is for repairing the time assignments of a layer immediately after they are made, without wasting time on earlier layers where repairs have already been tried and are very unlikely to succeed. `KheEjectionChainRepairResources` repairs the assignments of the tasks of `tasking`.

These top-level functions are all much the same. They use the same augment functions, for example. They differ in how some options are set, and in how `start_gm` is defined.

All three functions make assignments as well as change them, so may be used to construct solutions as well as repair them. However, there are better ways to construct solutions.

Here is function `KheEjectionChainRepairResources`:

```
bool KheEjectionChainRepairResources(KHE_TASKING tasking,
  KHE_OPTIONS options)
{
  KHE_EJECTOR ej;  KHE_GROUP_MONITOR kempe_gm, start_gm, limit_gm;
  KHE_SOLN soln;  bool res;  KHE_AUGMENT_OPTIONS ao;

  /* get an ejector */
  ej = KheEjectionChainEjectorOption(options, "es_ejector2");

  /* set the control options */
  KheOptionsSetBool(options, "es_repair_times", false);
  KheOptionsSetObject(options, "es_limit_node", NULL);
  KheOptionsSetBool(options, "es_repair_resources", true);

  /* build the required group monitors and solve */
  soln = KheTaskingSoln(tasking);
  KheGroupCorrelatedMonitors(soln, options);
  kempe_gm = KheKempeDemandGroupMonitorMake(soln);
  start_gm = KheTaskingStartGroupMonitorMake(tasking);
  ao = KheAugmentOptionsMake(soln, options, KheEjectorArena(ej));
  KheEjectorSolveBegin(ej, start_gm, (KHE_GROUP_MONITOR) soln, ao,
    options);
  res = KheEjectorSolveEnd(ej);

  /* clean up and return */
  KheGroupMonitorDelete(kempe_gm);
  KheGroupMonitorDelete(start_gm);
  KheUnGroupCorrelatedMonitors(soln);
  return res;
}
```

We've simplified it slightly to emphasize the main points. We'll go through it now line by line.

`KheEjectionChainEjectorOption` retrieves an ejector object from `options`, as described just below. This avoids wasting time by creating ejector objects over and over again, but its main purpose is to allow statistics over the whole run to be accumulated in just a few ejector objects—currently `"es_ejector1"` for time repair, and `"es_ejector2"` for resource repair.

Here is `KheEjectionChainEjectorOption` and another function that it may call:

```
KHE_EJECTOR KheEjectionChainEjectorMake(KHE_OPTIONS options,
  HA_ARENA a);
KHE_EJECTOR KheEjectionChainEjectorOption(KHE_OPTIONS options,
  char *key);
```

`KheEjectionChainEjectorMake` makes an ejector object. It starts with a call to `KheEjectorMakeBegin`, ends with a call to `KheEjectorMakeEnd`, and between them calls `KheEjectorAddAugmentType`, `KheEjectorAddRepairType`, `KheEjectorAddAugment`, and `KheEjectorAddGroupAugment` many times over, to load the augment functions described in

Sections 13.7 and 13.8, together with augment types and repair types that allow detailed statistics to be gathered.

KheEjectionChainEjectorOption retrieves an ejector object from options, stored under the given key. If there is no such object, it creates one by calling KheEjectionChainEjectorMake, stores it under key, and returns it.

Next, options "es_repair_times", "es_limit_node", and "es_repair_resources" are set to values appropriate to resource repair. These values will be consulted later by augment functions and will cause them to limit their repairs to resource repairs.

As discussed in Section 8.7.3, it is important for the effective use of ejection chains to group correlated monitors. This is done by the call to KheGroupCorrelatedMonitors, and undone at the end by KheUnGroupCorrelatedMonitors (Section 8.7.4). There is no need here to group monitors concerned with time assignment, but it is simpler to just group everything.

Two other group monitors, of the kind described as *focus groupings* in Section 8.7.2, are needed by KheEjectionChainRepairResources. The first is kempe_gm, needed to support repairs that include Kempe moves; the second is start_gm, needed to limit the main loop of the ejector to repairs of event resource and resource defects whose resource type is that of tasking. We'll have more to say about limiting the scope of repairs later on.

It is reasonable to worry about the time it takes to group monitors, so the author ran just the group monitor setup and removal parts of KheEjectionChainNodeRepairTimes 10000 times on a typical instance (BGHS98) and measured the time taken. This was 31.35 seconds, or about 0.003 seconds per setup/remove. This is not significant if it is done infrequently.

KheAugmentOptionsMake constructs an object of type KHE_AUGMENT_OPTIONS by filling in its fields based on values it retrieves from options. It is basically an optimization: it allows augment functions to access the options they need much more quickly that via a cumbersome retrieval from an options object. We'll look at this more closely in Section 13.6.3.

We are nearly done. The calls to KheEjectorSolveBegin and KheEjectorSolveEnd do the actual solve. We've omitted some code here which optionally installs a limit monitor in between these two calls. Then, in accordance with our general policy concerning group monitors (Section 8.7.1), the group monitors created by this function are deleted. Finally, the earlier call to KheGroupCorrelatedMonitors is undone, and true is returned if the solution was improved.

### 13.6.2. Focus groupings for ejection chains

Top-level ejection chain functions need focus groupings for their start group monitors (but not their continue group monitors, since they use the solution object for that), and for their limit monitors. This section describes the public functions, defined in khe_se_focus.c, which return these group monitors.

KheEjectionChainNodeRepairTimes uses the group monitor returned by

```
KHE_GROUP_MONITOR KheNodeTimeRepairStartGroupMonitorMake(KHE_NODE node);
```

as its start group monitor. The result has sub-tag KHE_SUBTAG_NODE_TIME_REPAIR. Its children are monitors, or correlation groupings of monitors where these are already present, of two kinds. First are all assign time, prefer times, spread events, order events, and ordinary demand monitors that monitor the meets of node and its descendants, plus any meets whose assignments are fixed, directly or indirectly, to them. Second are all resource monitors. Only preassigned resources are assigned during time repair, but those assignments may cause resource defects which can only be repaired by changing time assignments, just because the resources involved are preassigned.

`KheEjectionChainLayerRepairTimes` chooses one of the group monitors returned by

```
KHE_GROUP_MONITOR KheLayerTimeRepairStartGroupMonitorMake(
  KHE_LAYER layer);
KHE_GROUP_MONITOR KheLayerTimeRepairLongStartGroupMonitorMake(
  KHE_LAYER layer);
```

as its start group monitor, depending on option `es_layer_repair_long`. The result has sub-tag `KHE_SUBTAG_LAYER_TIME_REPAIR`, with the same children as before, only limited to those that monitor the meets and resources of `layer`, or (if `es_layer_repair_long` is `true`) of layers whose index number is less than or equal to `layer`'s.

`KheEjectionChainRepairResources` uses the group monitor returned by

```
KHE_GROUP_MONITOR KheTaskingStartGroupMonitorMake(KHE_TASKING tasking);
```

for its start group monitor. The result has sub-tag `KHE_SUBTAG_TASKING`, and its children are the following monitors (or correlation groupings of those monitors, where those already exist): the assign resource, prefer resources, and avoid split assignments monitors, and the resource monitors that monitor the tasks and resources of `tasking`. If the tasking is for a particular resource type, only monitors of entities of that type are included.

To allow an ejection chain to unassign meets temporarily but prevent it from leaving meets unassigned in the end, a limit monitor is imposed which rejects chains that allow the total cost of assign time defects to increase. This monitor is created by calling

```
KHE_GROUP_MONITOR KheGroupEventMonitors(KHE_SOLN soln,
  KHE_MONITOR_TAG tag, KHE_SUBTAG_STANDARD_TYPE sub_tag);
```

passing `KHE_ASSIGN_TIME_MONITOR_TAG` and `KHE_SUBTAG_ASSIGN_TIME` as tag parameters.

To prevent the number of unmatched demand tixels from increasing, when that is requested by the `resource_invariant` option, the group monitor returned by function

```
KHE_GROUP_MONITOR KheAllDemandGroupMonitorMake(KHE_SOLN soln);
```

is used as a limit monitor. Its sub-tag is `KHE_SUBTAG_ALL_DEMAND`, and its children are all ordinary and workload demand monitors. Correlation groupings are irrelevant to limit monitors, so these last two functions take no account of them.

### 13.6.3. Augment options

Ejection chain code consults many options. Rather than clumsy retrievals of these options from an `options` object each time they are needed, the ejection chain code retrieves each option just once, and stores its value in an object of type `KHE_AUGMENT_OPTIONS`. This object is passed to the ejector as an argument of function `KheEjectorSolveBegin`, which passes it back to each augment function, making the options immediately available to them.

Type `KHE_AUGMENT_OPTIONS` is defined in `khe_solvers.h` by

```
typedef struct khe_augment_options_rec *KHE_AUGMENT_OPTIONS;
```

and made concrete at the start of `khe_se_solvers.c`:

```
struct khe_augment_options_rec {

  /* time repair options */
  bool                          vizier_node;
  bool                          layer_repair_long;
  bool                          nodes_before_meets;
  KHE_OPTIONS_KEMPE             use_kempe_moves;
  bool                          use_fuzzy_moves;
  bool                          no_ejecting_moves;

  /* resource repair options */
  bool                          widening_off;
  bool                          reversing_off;
  bool                          balancing_off;
  int                           swap_widening_max;
  int                           move_widening_max;
  int                           balancing_max;
  bool                          full_widening_on;
  bool                          optimal_on;
  int                           optimal_width;

  /* options that are set by functions, not by the user */
  bool                          repair_times;
  bool                          use_split_moves;
  KHE_KEMPE_STATS               kempe_stats;
  KHE_NODE                      limit_node;
  bool                          repair_resources;
  KHE_SOLN                      soln;
  KHE_FRAME                     frame;
  KHE_EVENT_TIMETABLE_MONITOR   etm;
  KHE_MTASK_FINDER              mtask_finder;
};
```

The reader can see now why we describe `khe_se_solvers.c` as volatile: these options control various promising ideas that may come and go. The options will come and go with the ideas.

A few of these options are more permanent than the others and deserve special notice. Option `repair_times` is `true` when augment functions are permitted to change the assignments of meets, and `repair_resources` is `true` when they are permitted to change the assignments of tasks. At least one of these must be `true`, since otherwise the augment functions can change nothing. It is also acceptable for both to be `true`, although that can lead to slow runs.

Option `soln` is the solution under repair; `frame` is the common frame; `etm` is an event timetable monitor, from which an augment function can retrieve the tasks running at a given time; and `mtask_finder` is an mtask finder object, as in Section 11.9.

A `KHE_AUGMENT_OPTIONS` object is created by a call to `KheAugmentOptionsMake`. We won't present its implementation; it simply creates an object and fills its fields with values retrieved from the `options` object.

Here now is the full list of options that control the top-level ejection chain functions. These options are quite separate from the options used to control ejector objects, which are listed elsewhere (Section 13.3).

First come options that do not need to be stored in the `KHE_AUGMENT_OPTIONS` object, because they affect only the main functions, not the augment functions:

`rs_ejector_off`
A Boolean option which, when `true`, causes `KheEjectionChainRepairResources` to do nothing.

`es_ejector1`, `es_ejector2`
These two options hold ejector objects. `KheEjectionChainNodeRepairTimes` and `KheEjectionChainLayerRepairTimes` use the ejector object stored under key `es_ejector1`, while `KheEjectionChainRepairResources` uses the ejector object stored under key `es_ejector2`. There is no need for the user to set these options, since they will be set the first time they are needed; but retrieving them at the end of the solve can be useful, since they will then contain statistics on the performance of the ejection chain algorithm.

`es_schedules`
The value here is a string describing the schedules to apply to an ejector. The default value is `"1+,u-"`. For the meaning of this, consult Section 13.2.

Next come those options retrieved from `options` by `KheAugmentOptionsMake` and stored in the `KHE_AUGMENT_OPTIONS` object. First, those concerned with time assignment:

`es_vizier_node`
A Boolean option which, when `true`, instructs `KheEjectionChainNodeRepairTimes` and `KheEjectionChainLayerRepairTimes` to insert a vizier node (Section 9.5.4) temporarily while they run.

`es_layer_repair_long`
A Boolean option which, when `true`, instructs `KheEjectionChainLayerRepairTimes` to target every layer up to and including the current layer when repairing the current layer. Otherwise only the current layer is targeted.

`es_nodes_before_meets`
A Boolean option which, when `true`, instructs augment functions that try both node swaps and meet moves to try the node swaps first.

`es_kempe_moves`
This option determines whether augment functions that move meets use Kempe moves in addition to ejecting and basic ones (Section 10.2.2). Its possible values are `true`, meaning to use them, `false`, meaning to not use them, and `large_layers` (the default), meaning to

use them when moving the meets of nodes that lie in layers of large duration relative to the cycle duration, reasoning that swaps are usually needed when such meets are moved.

`es_fuzzy_moves`
A Boolean option which, when `true`, instructs augment functions that move meets to try fuzzy meet moves (Section 10.7.4) in addition to the other kinds of meet moves. If they do, to conserve running time they only do so when repairing a defect of the current best solution, not when repairing a defect introduced by a previous repair. At present the `width`, `depth`, and `max_meets` arguments passed to `KheFuzzyMeetMove` are fixed constants.

`es_no_ejecting_moves`
A Boolean option which, when `true`, instructs augment functions that assign and move meets to not use ejecting moves, only basic ones (Section 10.2.2).

Next come options concerned with resource assignment:

`es_widening_off`, `es_reversing_off`, `es_balancing_off`
Boolean options which, when `true`, cause widening, reversing, and balancing to be omitted from task move repairs.

*Note: Task move repairs treat tasks* `t` *for which* `KheTaskNeedsAssignmentHasCost(t)` *is* `false` *as free time. Inserting this note in the right place is still to do.*

`es_move_widening_max`
When widening is in effect (when `es_widening_off` is `false`), this integer option determines the maximum number of frame time groups that widening covers when moves are being tried, in addition to whatever frame time groups the mtask or mtask set being moved already covers. The default value is 4. Value 0 turns move widening off.

`es_swap_widening_max`
When widening is in effect (when `es_widening_off` is `false`), this integer option determines the maximum number of frame time groups that widening covers when swaps are being tried, in addition to whatever frame time groups the mtask or mtask set being swapped already covers. The default value is 16. Value 0 turns swap widening off.

`es_balancing_max`
When balancing is in effect (when `es_balancing_off` is `false`), this integer option determines the maximum number of repairs that move a task set in the other direction to the main move. The default value is 12.

`es_full_widening_on`
A boolean option which, when `true`, includes full widening in task move repairs. This swaps timetables from the task being moved back to the start or forward to the end of the cycle. It is effective, but its slowness typically increases cost on solves limited by time.

`es_optimal_on`
A boolean option which, when `true`, causes optimal widened task set moves to be used. In that case, `es_optimal_width` just below is consulted, as is `es_balancing_off` above, but the other options from `es_widening_off` down to here are ignored.

`es_optimal_width`
When `es_optimal_on` is `true`, this integer option determines how many extra days to include in the optimal move. The default value, 6, adds 6 extra days. When these are added to the one day typically involved in the original move, this covers a full week.

The following options are set within `KheEjectionChainRepairResources` and similar functions, making it futile for the user to set them:

`es_repair_times`
A Boolean option which, when `true`, lets augment functions change meet assignments. `KheEjectionChainNodeRepairTimes` and `KheEjectionChainLayerRepairTimes` set it to `true`, while `KheEjectionChainRepairResources` sets it to `false`.

`es_split_moves`
A Boolean option which, when `true`, instructs augment functions that move meets to try split meet moves in addition to other kinds of meet moves. `KheGeneralSolve2024` sets this option to `true` when the instance contains soft split events or distribute split events constraints, reasoning that they may not have been satisfied by structural solvers.

`es_limit_node`
This option holds a node object. When it is non-`NULL`, it causes augment functions that assign and move meets to limit their repairs to the descendants of that node. This option is set by all three functions.

`es_repair_resources`
A Boolean option which, when `true`, lets augment functions change task assignments. `KheEjectionChainNodeRepairTimes` and `KheEjectionChainLayerRepairTimes` set it to `false`, while `KheEjectionChainRepairResources` sets it to `true`.

As mentioned earlier, one can expect this list to change as good ideas come and go.

### 13.6.4. How to write an augment function

An augment function has type

```
bool (*KHE_EJECTOR_AUGMENT_FN)(KHE_EJECTOR ej,
  KHE_AUGMENT_OPTIONS ao, KHE_MONITOR d);
```

The parameters are the ejector `ej` passed to `KheEjectorSolve`, a set of options `ao` which may be consulted to influence the detailed behaviour of the augment function, and one defect `d` that the augment function is supposed to repair. When ejection beams are in use the ejector still only expects the user's augment functions to repair a single defect. Thus one can switch from chains to beams and back again without changing any augment functions.

It is a precondition that `d`'s cost exceeds its lower bound, and that reducing its cost to its lower bound would be a step towards a successful augment. The return value says whether the augment successfully reduced the solution cost or not, and must be `true` if some call to `KheEjectorRepairEnd` (see below) returns `true`, or `false` otherwise. The ejector relies on this value and it must be right.

Augment functions often look like this, although not necessarily exactly:

```
bool ExampleAugment(KHE_EJECTOR ej, KHE_AUGMENT_OPTIONS ao, KHE_MONITOR d)
{
  KHE_ENTITY e;  bool success;  REPAIR r;
  e = SomeSolnEntityRelatedTo(d);
  if( !KheEntityVisited(e) )
  {
    KheEntityVisit(e);
    for( each r in RepairsOf(e) )
    {
      KheEjectorRepairBegin(ej);
      success = Apply(r);
      if( KheEjectorRepairEnd(ej, 0, success) )
        return true;
    }
    if( KheEjectorCurrMayRevisit(ej) )
      KheEntityUnVisit(e);
  }
  return false;
}
```

Function `SomeSolnEntityRelatedTo` uses `d` to identify some entity (node, meet, task, etc.) that will be changed by the repairs, but that should only be changed if it has not already been visited (tested by calling `KheMeetVisited` etc. from Section 4.2.5). After the visit, the boiler-plate code

```
if( KheEjectorCurrMayRevisit(ej) )
  KheEntityUnVisit(e);
```

marks the entity unvisited if revisiting is allowed.

Each application of one repair must be bracketed by calls to `KheEjectorRepairBegin` and `KheEjectorRepairEnd`, as shown. We'll return to these two functions shortly.

The ejector object checks the time limit. Augment functions do not need to, and should not, since that could cause option `es_whynot_monitor_id` (Section 13.3) to stop too soon.

Since writing the above, it has occurred to the author that the visited entity related to `d` could be `d` itself. The code above that relates to visiting `e` is now applied to `d` behind the scenes in the ejector object. This has the usual effect of prohibiting exponential searches, and it means that the writer of an augment function can safely drop all visiting code, producing this:

```
bool ExampleAugment(KHE_EJECTOR ej, KHE_AUGMENT_OPTIONS ao, KHE_MONITOR d)
{
  bool success;  REPAIR r;
  for( each r in RepairsOf(d) )
  {
    KheEjectorRepairBegin(ej);
    success = Apply(r);
    if( KheEjectorRepairEnd(ej, 0, success) )
      return true;
  }
  return false;
}
```

The augment function is also conceptually simpler this way: it searches a graph whose nodes are defects and whose edges are repairs, without revisiting any node. Of course, the search is different, and the question of whether it is better or worse must be decided empirically. The author has successfully removed all visiting code from his nurse rostering augment functions: running times are often larger, but cost is nearly always lower, often significantly lower.

Function `RepairsOf` builds a set of alternative repairs `r` of `e` or `d`, and `Apply(r)` stands for the code that applies repair `r`. In practice, repairs just need to be iterated over and applied; an explicit set is not needed. Nor is there any need for the augment function to follow any particular structure; anything that generates a sequence of pairs of calls to `KheEjectorRepairBegin` and `KheEjectorRepairEnd` is acceptable.

For example, some expensive repairs are only worth trying when repairing a defect that is really present in the solution, not introduced by a previous repair. This can be effected by

```
if( KheEjectorCurrLength(ej) == 1 )
{
  KheEjectorRepairBegin(ej);
  ... apply expensive repair ...
  if( KheEjectorRepairEnd(ej, 0, success) )
    return true;
}
```

This works because, as mentioned earlier, `KheEjectorCurrLength(ej)` returns 1 when the augment function was called from the main loop, 2 when the augment function was called by an augment function called from the main loop, and so on.

Functions `KheEjectorRepairBegin` and `KheEjectorRepairEnd` are supplied by KHE:

```
void KheEjectorRepairBegin(KHE_EJECTOR ej);
bool KheEjectorRepairEnd(KHE_EJECTOR ej, int repair_type, bool success);
```

Calls to them must occur in unnested matching pairs. A call to `KheEjectorRepairBegin` informs `ej` that a repair is about to begin, and the matching call to `KheEjectorRepairEnd` informs it that that repair has just ended. The repair is undone and redone as required behind the scenes by `KheEjectorRepairEnd`, using marks and paths, so undoing is not the user's concern.

The `repair_type` parameter of `KheEjectorRepairEnd` is used to gather statistics about the solve (Section 13.5). It may be 0 if statistics are not wanted.

The `success` parameter tells the ejector whether the caller thinks the current repair was successful (that is, ran to completion). If it is `false`, the ejector undoes the partially completed repair and forgets that it ever happened. If it is `true`, the ejector checks whether the repair reduced the cost of the solution, whether there is a single new defect worth recursing on, and so on. The writer of an augment function can forget that all this is happening behind the scenes.

If `KheEjectorRepairEnd` returns `true`, the ejector has found a successful chain. When this happens, the rule is that the augment function should return `true` immediately. It does not matter whether any entity is marked unvisited or not before exit.

However, there is an exception to this rule. We illustrate this by the following example.

Suppose that, in order to encourage ejection chains to remove a cluster busy times defect, some days when the resource will be busy are chosen, all meets assigned the resource outside those days are unassigned, and repairs are tried which move those meets to the chosen days.

While the repairs are underway, it is desired to limit the domains of the resource's meets to the chosen days, to keep the repairs on track. So the repair altogether consists of unassigning some of the resource's meets and adding a meet bound to each of the resource's meets.

Whether the repair is successful or not, after it and the chains below it are finished, the meet bound must be removed from the resource's meets, since the domains of the meets should not be restricted permanently. If the repair is unsuccessful, the meet bound is removed by the ejector as part of undoing the repair. But if the repair is successful there is a problem, because the repair is not undone. So in this case, in between receiving the `true` result from `KheEjectorRepairEnd` and returning `true` itself, the augment function should remove the meet bounds that it added.

In general, this kind of cleanup should not change the cost of the solution. It might remove meet or task bounds, unfix meets or tasks, and so on. Because it is done after a successful chain, it cannot be assumed that the solution is in the same state as when the repair began, although the user may be able to prove that certain aspects of it cannot have changed, based on his knowledge of what the augment functions do. In the example, if no repairs remove meet bounds other than those they add themselves, then the meet bound will still be present and it is safe to delete it.

*Repairs, multi-repairs, and augment functions.* The author has made several attempts to organize the code that makes up augment functions hierarchically. At present the hierarchy has three levels: repairs, multi-repairs, and augment functions.

A *repair* is a code fragment that makes one or more changes to the solution, aiming to repair some defect. The changes work together; they are not alternatives. In concrete terms, a repair is the code lying between one call to `KheEjectorRepairBegin` and the matching call to `KheEjectorRepairEnd`. Often we include those two calls in the repair, but sometimes we speak of the repair as just the code between them. A repair is often packaged into a function, which is then called a *repair function.* It may be defined elsewhere (`KheMeetMove`, `KheTaskAssign`, etc.). We also refer to the changes made to the solution by this code as one repair.

A *multi-repair* is a code fragment that tries a sequence of zero or more alternative repairs, each enclosed as usual in `KheEjectorRepairBegin` and `KheEjectorRepairEnd`. A *multi-repair function* is a function whose body is a multi-repair.

An *augment function* is a multi-repair function of type `KHE_EJECTOR_AUGMENT_FN` which is passed to an ejector. It typically calls repair functions and other multi-repair functions.

### 13.6.5. Limiting the scope of changes

Ejection chains work best when they are free to follow chains into any part of a solution, and make any repairs that help. This freedom can conflict with the caller's desire to limit the scope of the changes they make, typically because initial assignments have not yet been made to some parts of the solution, and an ejection chain repair should not anticipate them.

For example, suppose resource *r* is preassigned to some tasks, and there are others it could be assigned to. The preassigned tasks go into *r*'s timetable when their meets are assigned times, and could then create resource defects that an ejection chain time repair algorithm knows about. Suppose a limit busy times underload defect is created (quite likely when the workload on some day first becomes non-zero), and its augment function tries (among other things) to assign more tasks to *r* to increase its workload on that day. This is not done at present, but it is plausible. Then there will be an unexpected burst of resource assignment during time assignment.

One romantic possibility is to 'let a thousand flowers bloom' and just accept such repairs. The problem with this is that a carefully organized initial assignment can be much better than the result of a set of uncoordinated individual repairs.

Another possibility is to fix the assignments of all variables beyond the scope of the current phase of the solve to their current values, often null values. This is a very reliable approach, and arguably the most truthful, because it says to the ejection chain algorithm, in effect, 'for reasons beyond your comprehension, you are not permitted to change these variables.' But it suffers from a potentially severe efficiency problem: a large amount of time could be spent in discovering a large number of repairs, which all fail through trying to change fixed variables.

Yet another possibility is to have one ejector object for each kind of call (one for repairing time assignments, another for repairing resource assignments, and so on), with different augment functions. The augment functions for time repair would never assign a task, for example. This was the author's original approach, but as the code grew it became very hard to maintain.

At present the author is using the following somewhat ad-hoc ideas to limit the scope of changes. They do the job well at very little cost in code and run time.

The start group monitor is one obvious aid to restricting the scope of a call. For example, time repair calls do not include event resource monitors in their start group monitors.

Many repairs move meets and tasks, but do not assign them. It seems that once a meet or task has been assigned, it is always reasonable to move it during repair. So the danger areas are augment functions that assign meets and tasks, not augment functions that merely move them.

Augment functions for assign time and assign resource defects must contain 'dangerous' assignments. But suppose that the assign time or assign resource monitor for some meet or task is not in the start group monitor. Then a repair of that monitor cannot occur first on any chain; and if the meet or task is unassigned to begin with, it cannot occur later either, since the monitor starts off with maximum cost, so its cost cannot increase, and only monitors whose cost has increased are repaired after the first repair on a chain. So assign time and assign resource augment functions can be included without risk of the resulting time and resource assignments being out of scope. This is just as well, since they are needed after ejecting meet and task moves.

If it can be shown, as was just done, that certain events will remain unassigned, then they can have no other event defects, since those require the events involved to be assigned. Similarly, unassigned event resources will never give rise to other event resource defects.

Another idea is to add options to the options object that control which repairs are tried. This is as general as different ejector objects with different augment functions are, but, if the options are few and clearly defined, it avoids the maintenance problems. If many calls on augment functions achieve nothing because options prevent them from trying things, that would be an efficiency problem, but there is no problem of that kind in practice.

We've already seen that the `KHE_AUGMENT_OPTIONS` object contains a `repair_times` field, which when `true` allows repairs that assign and move meets, and a `repair_resources` field, which when `true` allows repairs that assign and move tasks. It takes virtually no code or time to consult these options; often, just one test at the start of an augment function is enough.

When moving a meet, its chain of assignments is followed upwards, trying moves at each level. But if the aim is to repair only a small area (one runaround, say), then even if a repair starts within scope, it can leave it as it moves up the chain. This has happened and caused problems. So the `KHE_AUGMENT_OPTIONS` object contains a `limit_node` field, whose value is a node. If it is non-`NULL`, meet assignments and moves are not permitted outside its proper descendants.

`KheEjectionChainNodeRepairTimes` and `KheEjectionChainLayerRepairTimes` set option `es_repair_times` to `true`, `es_repair_resources` to `false`, and `es_limit_node` to the parent node, or to `NULL` if it is the cycle node. The `false` value for `es_repair_resources` solves the hypothetical problem, given as an example at the start of this section, of limit busy times repairs assigning resources during time assignment.

`KheEjectionChainRepairResources` sets option `es_repair_times` to `false`, option `es_repair_resources` to `true`, and option `es_limit_node` to `NULL`. Setting `es_repair_times` to `true` here would also be reasonable, although slow; it would allow the repairs to try meet moves while repairing task assignments.

## 13.7. Repairing time assignments

This section is concerned with the augment functions supplied by KHE for repairing defects in the assignments of times to meets. We'll start by presenting the full list of repairs used by the KHE augment functions that do this.

*Meet bound repairs* reduce the domains of all meets assigned a given resource `r` to a subset of a given time group `tg`, by adding `tg` as a meet bound to all those meets. Any meets assigned times outside `tg` are unassigned first. This is used to fix resource overload problems by moving meets away from over-used times; but it gets very little use in practice, because both resource repair and time repair have to be enabled if it is to be called.

*Node meet swaps* call repair function `KheNodeMeetSwap` (Section 10.2.1) to swap over the assignments of the meets of two nodes. If the `nodes_before_meets` option is `true`, then if node swaps are tried at all, they are tried before (rather than after) meet moves.

*Meet moves* call repair function `KheTypedMeetMove` (Section 10.2.2) to move the assignments of meets, performing either a basic move, an ejecting move, or a Kempe move as requested. Where it is stated below that a Kempe meet move is tried, it is in fact tried only when the `kempe_moves` option is `true`. Where it is stated below that an ejecting meet assignment or move is tried, a basic meet assignment or move is tried instead when `no_ejecting_moves` is `true`.

*Kempe/ejecting meet moves* are multi-repairs consisting of one or two alternative repairs, first a Kempe meet move, then an ejecting meet move with the same parameters. The ejecting meet move is omitted when the Kempe meet move reports that it did only what a basic meet move would have done, since in that case the ejecting move is identical to the Kempe move. This sequence is similar to making an ejecting move and then, on the next repair, favouring a particular reassignment of the ejected meet(s) which is likely to work well.

*Fuzzy meet moves* move meets by calling repair function `KheFuzzyMeetMove` (Section 10.7.4). Fuzzy meet moves are not mentioned below, but they are tried after Kempe and ejecting meet moves, although only when the `fuzzy_moves` option is `true` and the current length is 1.

*Split moves* split a meet into two and Kempe-move one of the fragments. Conversely, *merge moves* Kempe-move one meet to adjacent to another and merge the two fragments. Split and merge moves are used similarly to fuzzy meet moves: although not mentioned below, they are tried after Kempe and ejecting meet moves, but only when the `split_moves` option is `true` and the current length is 1. These Kempe moves are not influenced by the `kempe_moves` option.

*Meet-and-task moves* Kempe-move a meet at the same time as moving one of its tasks, succeeding only if both moves succeed.

Most of these repairs are packaged into function `KheMeetMultiRepair`, which tries, using whatever means the options settings allow, to move the assignment of a given meet.

Following is a description of what each augment function does when given a non-group monitor with non-zero cost. When given a group monitor (which must come from a correlation grouping) with non-zero cost, since the elements of the group all monitor the same thing, the augment function takes any individual defect from the group and repairs that defect.

Wherever possible, augment functions change the starting points of the sequences of alternative repairs that they try. For example, when trying alternative times, the code might be

```
for( i = 0;  i < KheTimeGroupTimeCount(tg);  i++ )
{
  index = (KheEjectorCurrAugmentCount(ej) + i) % KheTimeGroupTimeCount(tg);
  t = KheTimeGroupTime(tg, index);
  ... try a repair using t ...
}
```

The first time tried depends on the number of augments so far, an essentially random number. This simple idea significantly decreases final cost and run time.

*Split events and distribute split events defects.* Most events are split into meets of suitable durations during layer tree construction, but sometimes the layer tree does not remove all these defects, or a split move introduces one. In those cases, the split analyser (Section 9.7.1) from the `options` object is used to analyse the defects and suggest splits and merges which correct them. For each split suggestion, for each meet conforming to the suggestion, a repair is tried which splits the meet as suggested. For each merge suggestion, for each pair of meets conforming to the suggestion, four combined repairs are tried, each consisting of, first, a Kempe meet move which brings the two meets together, and second, the merge of the two meets. The four Kempe moves move the first meet to immediately before and after the second, and the second to immediately before and after the first, where possible.

*Assign time defects.* For each monitored unassigned meet, all ejecting meet moves to a parent meet and offset that would assign the meet to a time within its domain are tried.

*Prefer times defects.* For each monitored meet assigned an unpreferred time, all Kempe/ejecting meet moves to a parent meet and offset giving a preferred time are tried.

*Spread events defects.* For each monitored meet in an over-populated time group, all Kempe/ejecting moves of the meet to a time group that it would not over-populate are tried; and for each under-populated time group, for each meet whose removal would not under-populate its time group, all Kempe/ejecting moves of it to the under-populated time group are tried.

*Link events defects.* These are not repaired; they are expected to be handled structurally.

*Order events defects.* These are currently ignored, because, as far as the author is aware, there are no instances containing order events constraints. It will not be difficult to find suitable meet moves when such instances appear.

## 13.8. Repairing resource assignments

Solutions have two kinds of assignments: assignments of times to meets, and of resources to tasks. Repairs that change assignments of times to meets were treated in Section 13.7. Repairs that do both are possible in high school timetabling, but in this section we consider only repairs that change assignments of resources to tasks, in high school timetabling and nurse rostering.

### 13.8.1. High school timetabling and nurse rostering

In high school timetabling, the main issue that arises in resource assignment is avoiding split assignments (also called creating resource constancy). If split assignments are allowed, then resource assignment is essentially room assignment, which is well known to be easy in practice. Idle times may become an issue if there are many part-time teachers.

In nurse rostering, each resource is assigned to at most one task per day, and there are many constraints that relate what a resource does on one day to what it does on adjacent days. So it makes sense to use repairs that move sets of tasks from adjacent days from one resource to another. Many authors of nurse rostering solvers have come to this conclusion.

Accordingly, we distinguish between the *basic repair*, which does the minimum needed to remove some defect, and the *widened repair*, which is a basic repair enlarged by changes on adjacent days. For example, if resource $r$ is overloaded, reassigning any of $r$'s tasks from $r$ to any other resource will remove (or reduce) the defect, making one basic repair for each task assigned $r$. Widened versions of these repairs reassign several tasks assigned $r$ on adjacent days.

The same code is used for repairing resource assignments in both high school timetabling and nurse rostering. But there are two differences in setup between the two:

1. All repairs assume that each resource can be assigned to at most one task per day. The days are represented by the time groups of the common frame. For nurse rostering, these time groups come from the hard constraints that specify that each nurse can take at most one shift per day. For high school timetabling, when the solver realizes that there are none of these constraints, it sets the common frame in the only way consistent with the assumption, which is to make each individual time into one day. So in this case the solver distinguishes nurse rostering from high school timetabling without any intervention by the user.

2. High school timetabling uses basic repairs, and nurse rostering uses widened repairs. The `KHE_AUGMENT_OPTIONS` object has options for controlling widened repairs. By setting option `widening_off` to `true`, or by setting option `widening_max` to 0, one can restrict the solver to basic repairs. In this case the user is expected to set these options appropriately for nurse rostering or high school timetabling. There is nothing to prevent widened repairs being applied in high school timetabling, but they are not likely to be useful there.

Our focus will be on nurse rostering's widened repairs. High school timetabling repairs are the same, only (as just explained) with a different definition of one day, and with widening off.

In the following subsections, we'll work bottom-up, from individual repairs to multi-repairs (Section 13.6.4) to augment functions.

### 13.8.2. Repairs

This section presents the individual repairs used when repairing resource assignments.

But first, there is a question we must answer. All resource repairs use mtasks (Section 11.9.3). When we repair a defect `d` by reassigning some task `t`, we will be reassigning `t`'s enclosing mtask, not `t` itself. The question is: will `d` will be repaired by this mtask reassignment?

It may not matter if `d` is not repaired. The specification of the ejection chains `Augment` function does not say that `d` must be repaired, merely that a certain cost target must be reached. The augment function uses `d` as a hint, nothing more.

It might seem that there is a problem arising from the decision to make monitors the sole entities that are marked visited. If `d` is unrepaired, then a recursive call might attempt to repair `d` again, which will fail because `d` will have been marked visited on the first attempt to repair it.

But in fact this is not a problem. What happens is that some task `t2` from `mt` receives the reassignment intended for `t`. An examination of cases will show that `t2` must have needed the repair at least as much as `t` did, so other defects are repaired even if `d` is not. The repair causes further defects to appear, and these are used to continue the chain. The fact that `d` cannot now be visited does not matter, any more than it matters if `d` is repaired. This answers our question.

A repair is said to *fail* if it cannot be carried out for any reason; otherwise it *succeeds*. (Success is not the same as finding a successful ejection chain; for that, we need a whole sequence of successful repairs, ending in an improved solution. Nor is failing the same as aborting; an abort should never happen.) When we say here that some condition must hold, we mean that the repair will fail if it doesn't hold. When a repair discovers that it must fail, it abandons the rest of the repair, passes `false` for the `success` argument of `KheEjectorEnd`, and returns `false`.

An *interval* is a sequence of consecutive days (time groups of the common frame). It is represented by type `KHE_INTERVAL` (Section 8.12), which is a pair of integers. These are used here to represent the index in the common frame of the first day and of the last day. An interval can be empty (when the first index is one larger than the second), although that is uncommon.

Like many solvers, resource repair has two fundamental repair functions: *move* and *swap*. It would be better to call the move operation an *ejecting reassignment*, for reasons that will become clear; but that name is too long for daily use. Both functions implement one repair: each calls `KheEjectorRepairBegin` once, on entry, and `KheEjectorRepairEnd` once, on exit. Both guarantee that if no resource attends two tasks on the same day beforehand, then no resource attends two tasks on the same day afterwards. We'll see shortly how this comes about.

The move repair function has header

```
bool KheMoveRepair(KHE_EJECTOR ej, KHE_AUGMENT_OPTIONS ao,
  KHE_INTERVAL in, KHE_RESOURCE from_r, KHE_RESOURCE to_r,
  KHE_MTASK from_mt);
```

`KheMoveRepair` finds a maximal set of mtasks lying in interval `in` which are assigned `from_r`, and changes all those assignments to `to_r`, which must be different from `from_r`. It returns `true` when all this succeeds. The mtasks it reassigns must lie entirely within `in` and must be reassignable from `from_r` to `to_r`. They must run during the first and last times of `in` at least. This last rule is included so that repairs which could and should have been tried in a smaller interval are not reattempted in a larger interval. Assuming that `in` is non-empty, this rule also implies that the move cannot succeed by doing nothing.

If `from_r != NULL`, then when `KheMoveRepair` succeeds it guarantees that `from_r` is completely free on the days of `in`. It does this by failing if it finds an mtask assigned `from_r` which lies partly inside and partly outside `in` (it could include such an mtask in the reassignment, but that could cause problems for `to_r`), and by failing if any mtask assigned `from_r` lying entirely within `in` cannot be reassigned to `to_r` for any reason.

If `from_r == NULL`, then `to_r != NULL`, and the move is an *assignment*. The days of `in` are searched for mtasks that can be assigned `to_r`. A maximal set of these mtasks which do not overlap on any days is chosen and assigned `to_r`. This is the only case where `from_mt` is used: it must be non-`NULL`, and it is included in the set. Preference is given to mtasks which start at the same time of day as `from_mt` (all morning shifts, all night shifts, etc.).

If `to_r == NULL`, then `from_r != NULL`, and the move is an *unassignment*. The guarantee that `from_r` will be completely free after a successful repair still applies, but the chosen mtasks will be unassigned rather than assigned another resource.

When `from_r` is non-`NULL`, it is clear that it can only lose assignments, not gain them, and so if it does not attend two tasks on the same day beforehand, it will not attend two tasks on the same day afterwards. But as things stand, there is no such guarantee for `to_r`. This brings us to our final wrinkle, the one that makes this operation an *ejecting* reassignment: if `to_r` is non-`NULL`, then before the main reassignment of `from_r` to `to_r`, `KheMoveRepair` carries out another reassignment, of `to_r` to `NULL`. If this is successful (and if it isn't, the whole operation fails), it guarantees to clear out `to_r`'s timetable throughout `in`. Since the main reassignment only moves mtasks that lie entirely within `in` to `to_r`, if `to_r` does not attend two tasks on the same day beforehand, it will not attend two tasks on the same day afterwards.

The move from `to_r` to `NULL` differs from the main move in two ways. It unassigns mtasks that lie partly within `in` as well as mtasks that like wholly within `in`. And it does not care whether the mtasks it unassigns cover `in` completely, indeed it is happy to unassign no mtasks at all.

The implementation searches for mtasks to assign to `to_r` before carrying out the initial ejecting step. The order of these two steps does not matter when `from_r != NULL`, because the choice of mtasks to move is determined by `from_r`, but it matters when `from_r == NULL`, because unassigning `to_r` from an mtask may make that mtask available for assignment. Whatever the value of `from_r`, it is quite possible for the ejecting step to unassign `to_r` from some mtask, and then for the main step to assign `to_r` back to that same mtask again.

The swap repair function has header

```
bool KheSwapRepair(KHE_EJECTOR ej, KHE_AUGMENT_OPTIONS ao,
  KHE_INTERVAL in, KHE_RESOURCE from_r, KHE_RESOURCE to_r,
  /* KHE_MTASK from_mt, */ KHE_TIME_GROUP blocking_tg);
```

`KheSwapRepair` reassigns the mtasks initially assigned `from_r` in `in` to `to_r`, and reassigns the mtasks initially assigned `to_r` in `in` to `from_r`. For this to succeed, every mtask assigned `from_r` or `to_r` lying wholly or partially within `in` must lie wholly within `in` and must be reassignable to the other resource. So each resource gains a fresh timetable within `in`, but there are no changes outside `in`. The interval covered by the mtasks that move must equal `in`, for the usual reason that we do not want to retry repairs that could and should have been tried in smaller intervals.

A swap can be interpreted as two moves, one in each direction, provided the mtasks to be reassigned are identified before any assignments are changed. (The ejecting aspect is irrelevant, because the ejected tasks are assigned the other resource.) We can use this interpretation to work out what a swap would mean when one of the resources, `from_r` say, is `NULL`. The move from `to_r` to `NULL` unassigns the mtasks initially assigned `to_r`, and the move from `NULL` to `to_r` finds a set of suitable unassigned mtasks from `in` (including `from_mt` as usual) and assigns `to_r` to them. Apart from a slight difference at the edges of `in`, this is just what a move from `NULL` to `to_r` would do. Accordingly, `KheSwapRepair` requires `from_r` and `to_r` to be non-`NULL` as well as distinct. This is why, unlike in `KheMoveRepair`, there is no `from_mt` parameter.

`KheSwapRepair(in, from_r, to_r)` and `KheSwapRepair(in, to_r, from_r)` (we are abbreviating the argument lists) should have identical effect. And so they do, except that the last parameter, `blocking_tg`, applies only to the move from `to_r` to `from_r`. Its purpose is to prevent fruitless operations. If `blocking_tg != NULL`, and the move from `to_r` to `from_r` would make `from_r` busy during `blocking_tg`, then the swap fails.

During a swap, an mtask `mt` may be a part of the move from `from_r` to `to_r`, and at the same time it may be part of the move from `to_r` to `from_r`. This may happen if `mt` is initially assigned both `from_r` and `to_r`. The effect should be to not change `mt`. This is indeed what the two moves accomplish: the first reassigns a task of `mt` initially assigned `from_r` to `to_r`, and the second reassigns a task assigned `to_r` (possibly the same task) to `from_r`. Altogether, nothing changes except possibly the order in which the resources appear within the mtask, which does not matter.

It seems like good policy to try swaps only between non-`NULL` resources, because swaps involving `NULL` are very similar to moves, as we have seen. And it seems like good policy to try moves only when one of the resources is `NULL`, because a move from one non-`NULL` resource to another will often eject several tasks, leaving them unassigned, when a swap would have reassigned them. The author's solvers adopt both policies.

The reader might wonder whether moves and swaps that follow these policies could be unified into a single operation. Indeed they could, simply by delegating calls with one `NULL` argument to `KheMoveRepair`. But those who want to interpret this as more than mere relabelling face a serious obstacle. The literature contains good evidence that moves are only worth trying over small intervals, while swaps are worth trying over much larger intervals. Accordingly, two separate options, `es_move_widening_max` and `es_swap_widening_max`, are supplied for the maximum width of a move and of a swap. Their default values are 4 and 16 respectively. Any deep unification of moves with swaps is doomed to shipwreck on this important difference.

### 13.8.3. Avoiding repeated repairs

Before considering multi-repairs, we need to tackle the issue of avoiding repeated repairs.

A defect `d` invokes a set of alternative repairs. When repairing resource assignments, these are calls to `KheMoveRepair(in, from_r, to_r)` and `KheSwapRepair(in, to_r, from_r)` for various values of `in`, `from_r`, and `to_r`. Regardless of how each repair comes to be included, we want to avoid trying the same repair twice when repairing `d`, because that wastes time.

The perfect solution is to cache, for each defect `d`, the argument lists of the repairs of `d` that have been tried so far, and to check the cache before each repair. But that is very clumsy. We need something simpler that produces the same effect, although it does not have to be perfect.

We start with a few miscellaneous points. `KheSwapRepair(in, to_r, from_r)` repeats `KheSwapRepair(in, from_r, to_r)`, so we'll need to be careful not to try every swap twice. The author uses moves only to and from `NULL`, and swaps only between non-`NULL` resources, so no swap repeats any move. Two moves with the same `in`, `from_r`, and `to_r` can be different when `from_r == NULL`, since `from_mt` is used then. We've arranged for moves and swaps to fail when they don't use all of `in`, ensuring that repairs over different intervals are genuinely different.

Some iterations are based on calls on functions `KheMTaskFinderMTasksInTimeGroup` and `KheMTaskFinderMTasksInInterval` (Section 11.9.3). They use sorting to uniqueify the mtask sets they return, and so avoid all revisiting.

The main cause of repeated repairs is repeated intervals. For example, suppose resource `r` is overloaded within time groups `tg1` and `tg2`. We will try repairs that move tasks assigned `r` in several intervals based on `tg1`, and in several intervals based on `tg2`. The set of repairs based on `tg1` can easily be organized to avoid repeats, as can the set based on `tg2`. But if `tg1` and `tg2` are chronologically close, these two sets are likely to include repairs with the same interval.

We use caching to prevent repeats in this cases, but we don't cache complete argument lists; instead, the cache holds a set of first days. When generating intervals, we skip intervals whose first day is already in the cache. What one first day represents, in effect, is a large set of argument lists, all those whose interval's first day is that day, and whose last day, `from_r`, `to_r`, and `from_mt` could be anything at all.

We represent the set of cached first days by an interval of day indexes. This can lead to problems when the time groups (or whatever) iterated over are not in chronological order. We handle this by caching only the most recent set of first days. So each set of repairs which is known to not generate repeats within itself avoids repeating repairs generated by the previous such set of repairs. This is enough to avoid all repeated repairs, except in unlikely cases.

These ideas are implemented using type

```
typedef struct khe_exclude_days_rec {
  KHE_INTERVAL                unassign_in;
  KHE_INTERVAL                swap_in;
} *KHE_EXCLUDE_DAYS;
```

The intervals tried by moves are different from those tried by swaps, so two intervals are kept, one representing the first days of the previous set of moves (in fact, unassignments), the other the first days of the previous set of swaps. One of these intervals is passed to each interval iterator, which uses it to avoid repeats. A typical call on an interval iterator looks like this:

```
KheIntervalIteratorInit(&swap_ii_rec, ao, kernel_in,
  ao->swap_widening_max, &ed->swap_in);
KheForEachInterval(&swap_ii_rec, in)
{
  ... try swaps in interval in ...
}
```

The iterator object, `swap_ii_rec`, generates all intervals that enclose `kernel_in`, cover up to `ao->swap_widening_max` extra days, and do not begin on any of the days of `ed->swap_in`. It updates `ed->swap_in` with its own first days, but uses the original value itself.

### 13.8.4. Multi-repairs

We now move up a level to some multi-repair functions (Section 13.6.4). Here is the first of two that underlie all augment functions for resource monitors:

```
bool KheDecreaseLoadMultiRepair(KHE_EJECTOR ej,
  KHE_AUGMENT_OPTIONS ao, KHE_RESOURCE r, KHE_TIME_GROUP tg,
  bool require_zero, KHE_EXCLUDE_DAYS ed);
```

Here `r` is any non-`NULL` resource, and `tg` is any non-empty set of times.

`KheDecreaseLoadMultiRepair` is called when `r` is too busy during `tg`; it looks for repairs that reduce the number of tasks assigned `r` during `tg`. If `require_zero` is `true`, only repairs that make `r` completely free during `tg` are of any use; otherwise, any reduction is useful. As explained in Section 13.8.3, when iterating over intervals, intervals whose first day appears in `ed` are to be skipped over, because they have already been tried.

`KheDecreaseLoadMultiRepair` has to convert this specification into a set of moves and swaps, passing each an interval and two resources. It does this as follows.

Step 1. If `require_zero` is `false`, each mtask assigned `r` in `tg` needs a separate repair. So the code iterates over those mtasks, and for each it determines the interval of days it occupies and the set of resources it could be reassigned to (including `NULL`, meaning unassignment). These we call the *kernel interval* and the *domain*. For each mtask it then applies Step 2.

If `require_zero` is `true`, the mtasks assigned `r` in `tg` need to be moved as a block. The interval containing all of them becomes the sole kernel interval, and the set of resources that any one of them could be moved to becomes the sole domain. Then Step 2 is applied just once.

Step 2. Here we have a kernel interval and domain, and the aim is to swap or move the tasks assigned `r` in the kernel interval from `r` to any element of the domain. We'll present the code for this first; it is packaged into function `KheDoDecreaseLoad`:

```
bool KheDoDecreaseLoad(KHE_EJECTOR ej, KHE_AUGMENT_OPTIONS ao,
  KHE_RESOURCE r, KHE_TIME_GROUP tg, KHE_EXCLUDE_DAYS ed,
  KHE_INTERVAL kernel_in, KHE_RESOURCE_GROUP domain)
{
  struct khe_interval_iterator_rec unassign_ii_rec, swap_ii_rec;
  struct khe_resource_iterator_rec other_ri_rec;
  KHE_INTERVAL in;  KHE_RESOURCE other_r;

  /* try widened swaps */
  KheIntervalIteratorInit(&swap_ii_rec, ao, kernel_in,
    ao->swap_widening_max, false,
    ao->full_widening_on && KheEjectorCurrLength(ej) == 1, &ed->swap_in);
  KheResourceIteratorInit(&other_ri_rec, ao, domain, NULL, r, false);
  KheForEachInterval(&swap_ii_rec, in)
    KheForEachResource(&other_ri_rec, other_r)
      if( KheSwapRepair(ej, ao, in, true, r, other_r, /* NULL, */ tg) )
        return true;

  /* try widened unassignments */
  KheIntervalIteratorInit(&unassign_ii_rec, ao, kernel_in,
    ao->move_widening_max, false, false, &ed->unassign_in);
  KheForEachInterval(&unassign_ii_rec, in)
    if( KheMoveRepair(ej, ao, in, r, NULL, NULL) )
      return true;

  /* no luck */
  return false;
}
```

Notice that `tg` is passed to `KheSwapRepair`, causing swaps that leave `r` busy during `tg` to fail.

We are now ready to see `KheDecreaseLoadMultiRepair`:

```
bool KheDecreaseLoadMultiRepair(KHE_EJECTOR ej, KHE_AUGMENT_OPTIONS ao,
  KHE_RESOURCE r, KHE_TIME_GROUP tg, bool require_zero, KHE_EXCLUDE_DAYS ed)
{
  KHE_RESOURCE_GROUP domain;  KHE_RESOURCE_TYPE rt;  KHE_RESOURCE r2;
  KHE_INTERVAL kernel_in;  KHE_MTASK from_mt;
  struct khe_resource_mtask_iterator_rec rmi_rec;
  struct khe_resource_task_iterator_rec rti_rec;

  rt = KheResourceResourceType(r);
  if( require_zero )
  {
    /* move all of r's tasks running during tg together away from r */
    if( ao->repair_resources && !KheResourceTypeDemandIsAllPreassigned(rt)
        && KheGetIntervalAndDomain(ej, ao, r, tg, &kernel_in, &domain)
        && KheDoDecreaseLoad(ej, ao, r, tg, ed, kernel_in, domain) )
      return true;
  }
  else
  {
    /* move each of r's tasks running during tg separately away from r */
    if( ao->repair_resources && !KheResourceTypeDemandIsAllPreassigned(rt) )
    {
      /* iterate over each separate mtask that it would be good to move */
      KheResourceMTaskForEach(&rmi_rec, ao, r, tg, from_mt)
        if( !KheMTaskIsPreassigned(from_mt, &r2) &&
            !KheMTaskAssignIsFixed(from_mt) &&
            KheDoDecreaseLoad(ej, ao, r, tg, ed, KheMTaskInterval(from_mt),
              KheMTaskDomain(from_mt)) )
          return true;
    }
  }
  return false;
}
```

If `require_zero` is `true`, this code calls `KheGetIntervalAndDomain` (not shown) to find a suitable kernel interval and domain for all of `r`'s tasks running during `tg`, then calls `KheDoDecreaseLoad` to swap or move them away. If `require_zero` is `false`, it does much the same thing, only for each mtask that `r` is assigned to during `tg` separately.

Our second multi-repair function is a kind of inverse of our first:

```
bool KheIncreaseLoadMultiRepair(KHE_EJECTOR ej,
  KHE_AUGMENT_OPTIONS ao, KHE_RESOURCE r, KHE_TIME_GROUP tg,
  bool allow_zero, KHE_EXCLUDE_DAYS ed);
```

Once again, `r` is any non-`NULL` resource, and `tg` is any non-empty set of times.

`KheIncreaseLoadMultiRepair` is called when `r` is not busy enough during `tg`; it looks for repairs that increase the number of tasks assigned `r` during `tg`. If `allow_zero` is `true`, reducing the number of tasks assigned `r` during `tg` to zero is an odd but acceptable alternative. As usual, when iterating over intervals, intervals whose first day appears in `ed` are to be skipped over, because they have already been tried.

Once again, `KheIncreaseLoadMultiRepair` has to convert this specification into a set of moves and swaps, passing each an interval and two resources. It does this as follows.

First, if `allow_zero` is `true`, `KheIncreaseLoadMultiRepair` calls

```
KheDecreaseLoadMultiRepair(ej, ao, r, tg, true, ed)
```

to try to clear out `r` completely during `tg`.

Next, it tries swaps that might increase the load:

```
/* try widened swaps */
day_in = KheTimeGroupInterval(tg, ao);
for( i = KheIntervalFirst(day_in);  i <= KheIntervalLast(day_in);  i++ )
{
  KheIntervalIteratorInit(&swap_ii_rec, ao, KheIntervalMake(i, i),
    ao->swap_widening_max, &ed->swap_in);
  KheResourceIteratorInit(&other_ri_rec, ao,
    KheResourceTypeFullResourceGroup(rt), NULL, r, false);
  KheForEachInterval(&swap_ii_rec, in)
    KheForEachResource(&other_ri_rec, other_r)
      if( KheSwapRepair(ej, ao, in, r, other_r, NULL, NULL) )
        return true;
}
```

`KheTimeGroupInterval` returns the smallest interval of days whose first and last elements intersect with `tg`. For each of these days, this code iterates over all intervals `in` with this one day as kernel, and for each resource `other_r` of `r`'s type except for `r`, it tries a swap with `other_r` in `in`. Repeated intervals are almost certain here, but `ed->swap_in` avoids them as usual.

Finally, `KheIncreaseLoadMultiRepair` tries to assign unassigned mtasks:

```
/* try widened assignments */
for( need = 1;  need >= 0;  need-- )
{
  KheAllMTaskIteratorTimeGroupInit(&ami_rec, ao, rt, tg);
  KheAllMTaskForEach(&ami_rec, mt)
    if( need == (int) KheMTaskNeedsAssignment(mt) )
    {
      KheIntervalIteratorInit(&assign_ii_rec, ao, KheMTaskInterval(mt),
        ao->move_widening_max, false, false, NULL);
      KheForEachInterval(&assign_ii_rec, in)
        if( KheMoveRepair(ej, ao, in, NULL, r, mt) )
          return true;
    }
}
```

This iterates over all mtasks of the right type lying wholly or partly in tg, first those that need assignment, then those that don't, and tries moving each of those, plus other mtasks that come from the widening, to r. There is no reference here to ed, because each mtask mt is distinct and makes for a different repair. (Some mt might intersect with two time groups tg, and then repairs with mt as kernel will be repeated. This cannot be prevented by using ed, since several mtasks may run on the same day or days.)

It was remarked earlier that there was some danger of trying each swap twice, because KheSwapRepair(in, to_r, from_r) repeats KheSwapRepair(in, from_r, to_r). We can see, however, that there is no danger of that here, because each swap has r for its first resource and something else for its second resource. Indeed, entire sequences of arbitrarily intermixed calls to KheDecreaseLoadMultiRepair and KheIncreaseLoadMultiRepair with the same r parameter are safe from this kind of repetition for this reason.

The key multi-repair function for repairing event resource defects is

```
bool KheEventResourceMultiRepair(KHE_EJECTOR ej,
  KHE_AUGMENT_OPTIONS ao, KHE_EVENT_RESOURCE er,
  KHE_RESOURCE_ITERATOR to_ri, KHE_EXCLUDE_DAYS ed,
  KHE_MTASK *prev_mt, KHE_RESOURCE *prev_r)
```

Parameter to_ri is a *resource iterator*, a set of resources with some fancy features: one can specify a resource group of resources to include, and optionally a resource group and resource to exclude. One can also specify whether to include NULL, meaning not assigned.

For each task t derived from er which is not already assigned a resource from to_ri, KheEventResourceMultiRepair tries to reassign t to a resource from to_ri. If t lies in mtask from_mt and is assigned resource from_r, it tries similar widened swaps and unassignments to those tried by KheDecreaseLoadMultiRepair:

```
/* try widened swaps */
KheIntervalIteratorInit(&swap_ii_rec, ao, KheMTaskInterval(from_mt),
  ao->swap_widening_max, &ed->swap_in);
KheForEachInterval(&swap_ii_rec, in)
  KheForEachNonNullResource(to_ri, other_r)
    if( KheSwapRepair(ej, ao, in, from_r, other_r, NULL, NULL) )
      return true;

/* try widened unassignments, if allowed by to_ri */
if( KheResourceIteratorContainsResource(to_ri, NULL) )
{
  KheIntervalIteratorInit(&unassign_ii_rec, ao,
    KheMTaskInterval(from_mt), ao->move_widening_max, &ed->unassign_in);
  KheForEachInterval(&unassign_ii_rec, in)
    if( KheMoveRepair(ej, ao, in, from_r, NULL, NULL) )
      return true;
}
```

If t is unassigned (but still lying in mtask from_mt), it tries widened assignments:

```
/* try widened assignments */
KheIntervalIteratorInit(&assign_ii_rec, ao, KheMTaskInterval(from_mt),
  ao->move_widening_max, NULL);
KheForEachInterval(&assign_ii_rec, in)
  KheForEachNonNullResource(to_ri, other_r)
    if( KheMoveRepair(ej, ao, in, NULL, other_r, from_mt) )
      return true;
```

Repeated attempts to reassign the same resource from the same mtask are avoided; this is what `*prev_mt` and `*prev_r` are used for. Intervals forbidden by `ed` are skipped as usual, except that when assigning, the distinct values of `from_mt` make this unnecessary and indeed unwanted.

Again, we want to avoid trying swaps twice because `KheSwapRepair(in, to_r, from_r)` repeats `KheSwapRepair(in, from_r, to_r)`. There seems to be no easy proof that repetitions of this kind cannot happen here. However, it is hard to construct cases where they do happen.

### 13.8.5. Augment functions

We are now ready to implement the augment functions for event resource and resource monitors `m`, by calling functions `KheDecreaseLoadMultiRepair`, `KheIncreaseLoadMultiRepair`, and `KheEventResourceMultiRepair` from Section 13.8.4, as follows.

*Assign resource monitor.* This monitors one event resource `er`, and it is repaired by one call to `KheEventResourceMultiRepair`. The set of resources to try is just the domain of `er`.

*Prefer resources monitor.* This also monitors one event resource `er`, and is repaired by one call to `KheEventResourceMultiRepair`. The set of resources to try is the monitor's set of preferred resources, plus unassignment, which also removes a prefer resources defect.

*Avoid split assignments monitor.* This has a different character from the other event resource monitors, and it is handled in a different way. Documenting it is *still to do.*

*Limit resources monitor.* This monitors a set of event resources. For each of them it makes one call to `KheEventResourceMultiRepair`. If there are not enough tasks assigned resources from `rg`, the monitor's set of resources of interest, then the set of resources passed to `KheEventResourceMultiRepair` is just `rg`. If there are too many tasks assigned resources from `rg`, then the set of resources passed to `KheEventResourceMultiRepair` is the set of resources acceptable to each event resource plus unassignment but excluding `rg`.

*Avoid unavailable times monitor*. This is handled by a single call to

```
KheDecreaseLoadMultiRepair(ej, ao, r, tg, false, ed)
```

where `tg` is the set of times that `m` requires `r` to be free.

This call to `KheDecreaseLoadMultiRepair` does not convert `tg` into an interval. This is just as well, because here the times of `tg` may be scattered at random through the cycle, and repairing the interval covering them all would be mad. Instead, each interval containing a task running when `r` should be free becomes one kernel interval, as we have seen.

*Cluster busy times monitor*. If `m`'s defect is too many active time groups, then the repair visits each active time group `tg` of `m` and tries to make it inactive, by calling

```
    KheDecreaseLoadMultiRepair(ej, ao, r, tg, true, ed)
```

when `tg` is positive, and

```
    KheIncreaseLoadMultiRepair(ej, ao, r, tg, false, ed)
```

when it is negative. Here `require_zero` is `true` because to make a positive `tg` inactive one must reduce the number of tasks running during `tg` to zero. If `m`'s defect is too few active time groups, then the repair visits each inactive time group `tg` of `m` and tries to make it active, by calling

```
    KheIncreaseLoadMultiRepair(ej, ao, r, tg, false, ed)
```

when `tg` is positive, and

```
    KheDecreaseLoadMultiRepair(ej, ao, r, tg, true, ed)
```

when it is negative. Once again `require_zero` must be `true`. The usual method of avoiding repeated intervals, by keeping track of first days in variable `ed`, is used.

If the monitor's `allow_zero` flag is set, then another way to handle too few active time groups is to reduce the number of active time groups to zero. The repair tries this too, using calls to `KheDecreaseLoadMultiRepair` and `KheIncreaseLoadMultiRepair` similar to those just given, but only when there is exactly one active time group. Making several active time groups inactive is awkward to implement under current arrangements, and unlikely to succeed.

*Limit busy times monitor*. Limit busy times constraints can be converted into cluster busy times constraints. The repair does what would be done if this conversion were carried out.

*Limit workload monitor*. These are like limit busy times monitors, except that they monitor total workload rather than busyness. Accordingly, the same repairs are used for them.

*Limit active intervals monitor*. Suppose limit active intervals monitor `m` for resource `r` has a sequence of consecutive busy time groups `s` which is too long. The multi-repairs for this are a call to `KheDecreaseLoadMultiRepair` which make `s`'s first time group free for `r`, and a call to `KheDecreaseLoadMultiRepair` which does the same for `s`'s last time group. These are widened in the same way as for a cluster busy times monitor, including remembering first days.

If `m` monitors sequences of free time groups rather than busy ones, we switch to calls to `KheIncreaseLoadMultiRepair`, to make these time groups busy rather than free. Again this is what cluster busy times repairs do. If we have a sequence of busy time groups which is too short, we treat that as though the adjacent sequences of free time groups are too long, leading to repairs to the first free time group to the left, and the first free time group to the right.

When a sequence is too short, another option is to remove it altogether by trying repairs which clear out its time groups (or make them busy for sequences of free time). We try this too, but only when the time groups are positive.

### 13.8.6. Enumeration of basic repairs

In this section we investigate whether moves and swaps are the only possible repairs. Of course, if we allow any number of mtask sets and resources to be affected by a repair, then the number of repairs is unlimited. But if we limit repairs to neighbourhoods similar to the ones we have already decided to use, the choices become correspondingly more limited.

Accordingly, we define a *simple repair operation* to be one satisfying these conditions:

- The operation has one or two mtask set parameters, called `mts1` and (optionally) `mts2`. These will usually contain all the mtasks lying in a given interval, but we are not concerned here with where these mtask sets come from. When there are two mtask sets, we assume here that their values are disjoint.

- The operation has one or two resource parameters, called `r1` and (optionally) `r2`. Their values may be `NULL`. When there are two, their values must be different.

- When there is one mtask set parameter, the operation is expressed by

  ```
  KheMTaskSetResourceReassign(mts1, ?, ?)
  ```

  When there are two mtask set parameters, it is expressed by

  ```
  KheMTaskSetResourceReassign(mts1, ?, ?) &&
  KheMTaskSetResourceReassign(mts2, ?, ?)
  ```

  Here `?` stands for `r1`, or `r2` (when available), or `NULL`.

`KheMoveRepair` and `KheSwapRepair` are simple repair operations, for example. We ask what operations satisfying these conditions there can be.

Suppose first that the operation has one mtask set parameter. There are three choices for the first `?` and three for the second, making nine altogether. Dropping three whose first and second `?` are the same, which causes `KheMTaskSetResourceReassign` to fail, we get six possibilities:

```
KheMTaskSetResourceReassign(mts1, r1, r2)

KheMTaskSetResourceReassign(mts1, r1, NULL)

KheMTaskSetResourceReassign(mts1, r2, r1)

KheMTaskSetResourceReassign(mts1, r2, NULL)

KheMTaskSetResourceReassign(mts1, NULL, r1)

KheMTaskSetResourceReassign(mts1, NULL, r2)
```

However, expressions that do not mention `r1` can be dropped, because there is always an `r1` parameter and not using it is not reasonable. This brings us back to four possibilities:

```
KheMTaskSetResourceReassign(mts1, r1, r2)

KheMTaskSetResourceReassign(mts1, r1, NULL)

KheMTaskSetResourceReassign(mts1, r2, r1)

KheMTaskSetResourceReassign(mts1, NULL, r1)
```

The third is symmetrical with the first: it can be obtained from the first by reordering arguments. Dropping it leaves three cases, all offered by `KheMoveRepair` if we ignore its ejecting aspect.

The same method will work for repair functions with two mtask set parameters. There are 6 legal combinations for the first two parameters, and 6 legal combinations for the second two parameters, making 36 cases in all.

Here are the cases starting with `KheMTaskSetResourceReassign(mts1, r1, r2)`:

1.
   ```
   KheMTaskSetResourceReassign(mts1, r1, r2) &&    /* union case */
   KheMTaskSetResourceReassign(mts2, r1, r2)
   ```

2.
   ```
   KheMTaskSetResourceReassign(mts1, r1, r2) &&
   KheMTaskSetResourceReassign(mts2, r1, NULL)
   ```

3.
   ```
   KheMTaskSetResourceReassign(mts1, r1, r2) &&
   KheMTaskSetResourceReassign(mts2, r2, r1)
   ```

4.
   ```
   KheMTaskSetResourceReassign(mts1, r1, r2) &&
   KheMTaskSetResourceReassign(mts2, r2, NULL)
   ```

5.
   ```
   KheMTaskSetResourceReassign(mts1, r1, r2) &&
   KheMTaskSetResourceReassign(mts2, NULL, r1)
   ```

6.
   ```
   KheMTaskSetResourceReassign(mts1, r1, r2) &&
   KheMTaskSetResourceReassign(mts2, NULL, r2)
   ```

We can rule out the first of these because it is what we will call a *union case*: it is equivalent to a move on the union of `mts1` and `mts2`. The other five seem to be genuinely distinct.

Here are the cases starting with `KheMTaskSetResourceReassign(mts1, r1, NULL)`:

7.
   ```
   KheMTaskSetResourceReassign(mts1, r1, NULL) &&  /* symm. with (2) */
   KheMTaskSetResourceReassign(mts2, r1, r2)
   ```

8.
   ```
   KheMTaskSetResourceReassign(mts1, r1, NULL) &&  /* union case */
   KheMTaskSetResourceReassign(mts2, r1, NULL)
   ```

9.
   ```
   KheMTaskSetResourceReassign(mts1, r1, NULL) &&  /* symm. with (4) */
   KheMTaskSetResourceReassign(mts2, r2, r1)
   ```

10.
    ```
    KheMTaskSetResourceReassign(mts1, r1, NULL) &&
    KheMTaskSetResourceReassign(mts2, r2, NULL)
    ```

11.
    ```
    KheMTaskSetResourceReassign(mts1, r1, NULL) &&
    KheMTaskSetResourceReassign(mts2, NULL, r1)
    ```

12.
    ```
    KheMTaskSetResourceReassign(mts1, r1, NULL) &&
    KheMTaskSetResourceReassign(mts2, NULL, r2)
    ```

The cases with adjacent comments can be ruled out for the reasons given.

Cases starting with `KheMTaskSetResourceReassign(mts1, r2, r1)` are symmetrical with cases starting with `KheMTaskSetResourceReassign(mts1, r1, r2)`, so we skip them.

Cases starting with `KheMTaskSetResourceReassign(mts1, r2, NULL)` are symmetrical with cases starting with `KheMTaskSetResourceReassign(mts1, r1, NULL)`, so we skip them.

Here are the cases starting with `KheMTaskSetResourceReassign(mts1, NULL, r1)`:

25. 
```
KheMTaskSetResourceReassign(mts1, NULL, r1) &&   /* symm. with (5) */
KheMTaskSetResourceReassign(mts2, r1, r2)
```

26. 
```
KheMTaskSetResourceReassign(mts1, NULL, r1) &&   /* symm. with (11) */
KheMTaskSetResourceReassign(mts2, r1, NULL)
```

27. 
```
KheMTaskSetResourceReassign(mts1, NULL, r1) &&   /* symm. with (6) */
KheMTaskSetResourceReassign(mts2, r2, r1)
```

28. 
```
KheMTaskSetResourceReassign(mts1, NULL, r1) &&   /* symm. with (12) */
KheMTaskSetResourceReassign(mts2, r2, NULL)
```

29. 
```
KheMTaskSetResourceReassign(mts1, NULL, r1) &&   /* union case */
KheMTaskSetResourceReassign(mts2, NULL, r1)
```

30. 
```
KheMTaskSetResourceReassign(mts1, NULL, r1) &&
KheMTaskSetResourceReassign(mts2, NULL, r2)
```

There is only one genuinely new operation here.

Cases starting with `KheMTaskSetResourceReassign(mts1, NULL, r2)` are symmetrical with cases starting with `KheMTaskSetResourceReassign(mts1, NULL, r1)`, so we skip them.

That ends the enumeration. Altogether we have found nine distinct simple repairs with two mtask set parameters:

```
KheMTaskSetResourceReassign(mts1, r1, r2) &&
KheMTaskSetResourceReassign(mts2, r1, NULL)

KheMTaskSetResourceReassign(mts1, r1, r2) &&   /* KheSwapRepair */
KheMTaskSetResourceReassign(mts2, r2, r1)

KheMTaskSetResourceReassign(mts1, r1, r2) &&   /* KheMoveRepair */
KheMTaskSetResourceReassign(mts2, r2, NULL)

KheMTaskSetResourceReassign(mts1, r1, r2) &&
KheMTaskSetResourceReassign(mts2, NULL, r1)

KheMTaskSetResourceReassign(mts1, r1, r2) &&
KheMTaskSetResourceReassign(mts2, NULL, r2)

KheMTaskSetResourceReassign(mts1, r1, NULL) &&
KheMTaskSetResourceReassign(mts2, r2, NULL)

KheMTaskSetResourceReassign(mts1, r1, NULL) &&
KheMTaskSetResourceReassign(mts2, NULL, r1)

KheMTaskSetResourceReassign(mts1, r1, NULL) &&
KheMTaskSetResourceReassign(mts2, NULL, r2)
```

```
    KheMTaskSetResourceReassign(mts1, NULL, r1) &&
    KheMTaskSetResourceReassign(mts2, NULL, r2)
```

`KheSwapRepair` and `KheMoveRepair` are here, although the latter is actually

```
    KheMTaskSetResourceReassign(mts2, r2, NULL) &&
    KheMTaskSetResourceReassign(mts1, r1, r2)
```

Note the change of order of the two calls: order may be important when `mts1` and `mts2` are not disjoint, a detail that we have not tried to capture here. It is an interesting exercise to look through the other seven operations and ponder what they do and whether that might be useful in practice. To this author, none of them stand out.

It may seem that we have also omitted to consider the various kinds of mtask sets. The main ones are sets whose mtasks are assigned a particular resource `r`, and sets whose mtasks contain at least one task that needs assignment. But these kinds are implicit in the calls above: moving `mts1` from `r1` to `r2` implies that the mtasks of `mts1` are initially assigned `r1`, and so on.

# Appendix A. Modules Packaged with KHE

This chapter documents several modules packaged with KHE and used by it behind the scenes. By including their header files the user may also use these modules.

## A.1. Arenas, arena sets, and arrays

This section describes the Ha module, which implements arenas, arena sets, and extensible arrays. It is used throughout KHE in place of `malloc` to obtain heap memory. Its header file is `howard_a.h`. This module was originally flown in from another project of the author's, called Howard. For a general overview of arenas and arena sets, see Section 1.3.

### A.1.1. Arenas

An arena is an object (a pointer to a private struct) of type `HA_ARENA`. It represents an unlimited amount of *arena memory*: heap memory held in an arena so that it can be freed all at once later. All heap memory allocated by KHE is arena memory.

Every arena belongs to exactly one *arena set*, an object of type `HA_ARENA_SET` representing a set of arenas, from which the arena obtains its memory and to which it returns its memory when it is no longer needed. Function

```
HA_ARENA HaArenaMake(HA_ARENA_SET as)
```

creates an arena belong to arena set `as`. And

```
void HaArenaDelete(HA_ARENA a);
```

deletes `a`, returning all its arena memory to `a`'s arena set, where it becomes available for use by the other arenas of `as`.

In practice, functions `KheSolnArenaBegin` and `KheSolnArenaEnd` (Section 4.2.2) are the best way to create and delete arenas. They call `HaArenaMake` and `HaArenaDelete`.

Operations

```
void *HaAlloc(HA_ARENA a, size_t size);
void HaMake(X res, HA_ARENA a);
```

allocate memory. `HaAlloc` returns a pointer to at least `size_t` bytes of arena memory from `a`, aligned suitably for any data. Macro `HaMake` sets `res` (which may have any pointer type `X`) to point to at least `sizeof(*res)` bytes of memory obtained from `HaAlloc`. These objects may not be resized. For resizable objects, see Section A.1.4.

An arena obtains its memory from its arena set, which obtains it from `malloc`. As long as `malloc` can supply memory, an arena will supply memory to the user. If a request for memory from `malloc` fails, then the arena set will make a long jump using a jump environment passed to it by `KheArenaJmpEnvBegin` (Section A.1.2) if that is available, otherwise it will abort.

The memory pointed to by a variable `a` of type `HA_ARENA` is arena memory from a private arena held by `a`'s arena set. A deleted arena has its memory reclaimed, but the arena object itself is saved in a free list in its arena set, where it is available to later calls to `HaArenaMake`. This makes the cost of calling `HaArenaMake` and `HaArenaDelete` small enough to allow many small arenas to come and go. Section A.1.4 has more detail on this.

For maximum efficiency, arena memory is not initialized to zero. This can cause two problems. First, an uninitialized object field can cause a program to behave differently each time it runs, which is a nightmare to debug. The user must be very disciplined about initializing objects, which basically means enclosing every call to `HaMake` in a function something like this:

```
THING ThingMake(FIELD1 f1, FIELD2 f2, HA_ARENA a)
{
  THING res;
  HaMake(res, a);
  res->field1 = f1;
  res->field2 = f2;
  return res;
}
```

where every field of `THING` is initialized within `ThingMake`. The second problem is that `HaArrayContains` (Section A.1.3) compares array elements using `memcmp`. In the unlikely case where the elements are structs with gaps in them, the user needs to call `memset` to zero out the struct before its fields are initialized and it is copied into the array.

Finally,

```
void HaArenaDebug(HA_ARENA a, int verbosity, int indent, FILE *fp);
```

produces a debug print of `a` onto `fp` with the given verbosity and indent.

### A.1.2. Arena sets

An *arena set* is a set of arenas. It is also where the memory allocated by deleted arenas is stored and made available to other arenas, and where a stack of `longjmp` environments is kept, allowing the arena set to perform a long jump when memory runs out.

To create a new, empty arena set, call

```
HA_ARENA_SET HaArenaSetMake(void);
```

Memory for the arena set object is taken from a private arena kept by the arena set for its own purposes. If this call cannot obtain memory from `malloc` for this private arena, it aborts; but that will never happen if all arena sets are created near the start of the run, as happens in practice.

To create and delete an arena belonging to arena set `as`, the calls are `HaArenaMake(as)` and `HaArenaDelete(a)`, as we saw previously. Each arena belongs to exactly one arena set and it knows which arena set it belongs to.

There should be one arena set per thread. When a thread terminates, its memory needs to be passed on to the arena set of the parent thread. To do this, the parent thread should call

```
void HaArenaSetMerge(HA_ARENA_SET dest_as, HA_ARENA_SET src_as);
```

It moves the memory used by `src_as` to `dest_as`, destroying `src_as`. Also,

```
void HaArenaSetDelete(HA_ARENA_SET as);
```

deletes `as` and all its arenas, returning all their memory to the operating system via calls to `free`. In practice there is no need to do this.

When an arena is asked for memory but has none to give, it asks its arena set for more. When the arena set cannot give any more (when memory runs out), by default the arena set will cause an abort. But there is a more graceful alternative, which is to pass a `longjmp` environment to the arena set; it will then execute a long jump instead of aborting. The calls for this are

```
void HaArenaSetJmpEnvBegin(HA_ARENA_SET as, jmp_buf *env);
void HaArenaSetJmpEnvEnd(HA_ARENA_SET as);
```

These must occur in matching pairs; the second is made when the jump environment becomes unavailable. These pairs of calls may be nested; the arena set holds a stack of jump environments and makes its long jump using the environment on top of the stack.

In the unlikely case where there is no memory to store `env` within `as`, the action taken depends on what is on the jump environment stack before the call to `HaArenaSetJmpEnvBegin`. That is, the new jump environment takes effect only after `HaArenaSetJmpEnvBegin` returns.

An important problem with exception handling is giving up resources—closing files and so on. Ha does not pretend to offer a complete solution to this problem, but it does handle one major part of it: just before the long jump is taken, it deletes each arena created in `as` since the most recent call to `HaArenaSetJmpEnvBegin` in `as` but not yet deleted. This does not mean that all memory consumed since the most recent call to `HaArenaSetJmpEnvBegin` is freed, only memory in arenas created in `as` since that call. At times the user may have a choice of creating an arena before or after a call to `HaArenaSetJmpEnvBegin`; the choice will be determined by whether that arena should continue in use after an exception.

Here is an example using these functions. Suppose we have a solver `MySolve` which might use a lot of memory, and we want to abandon it gracefully when memory runs out. We do this:

```
#include <setjmp.h>
...
bool MySolve(KHE_SOLN soln, ...)
{
  jmp_buf env;  bool success;
  ...
  if( setjmp(env) == 0 )
  {
    /* get here on direct call; do the solve */
    KheSolnJmpEnvBegin(soln, &env);
    success = DoMySolve(soln);
    KheSolnJmpEnvEnd(soln);
  }
  else
  {
    /* get here by calling longjmp; abandon the solve */
    fprintf(stderr, "MySolve abandoned (out of memory)\n");
    success = false;
    KheSolnJmpEnvEnd(soln);
  }
  return success;
}
```

The call to `KheSolnJmpEnvBegin` just calls `HaArenaSetJmpEnvBegin` on `soln`'s arena set, and `KheSolnJmpEnvEnd` just calls `HaArenaSetJmpEnvEnd`.

A thread which takes a long jump will have previously consumed all available memory, leaving none for the other threads. To avoid this problem, one can call

```
void KheArenaSetLimitMemory(HA_ARENA_SET as, size_t limit);
```

This limits the total amount of memory consumed by `as` to `limit` bytes. If satisfying some request for memory would exceed this limit, the long jump or abort is taken, as though memory had run out completely. If `limit` is set to the amount of available memory divided by the number of threads, calling this function ensures that each thread gets its fair share of available memory. `KheArchiveParallelSolve` (Section 8.5) has a `ps_avail_mem` option which does this.

Finally, function

```
void HaArenaSetDebug(HA_ARENA_SET as, int verbosity, int indent,
  FILE *fp);
```

produces a debug print of `as` onto `fp` with the given verbosity and indent.

### A.1.3. Arrays

Like C's native arrays, Ha's arrays are *generic*: they may have elements of any one type, of any width, and the C compiler will report an error if there is a type mismatch. But, unlike C's arrays, Ha's arrays are *extensible*: they may grow to any length during use.

The type of an extensible generic array must be declared using a `typedef` invoking macro `HA_ARRAY`. For example, the following declarations already appear within `howard_a.h`:

```
typedef HA_ARRAY(bool)         HA_ARRAY_BOOL;
typedef HA_ARRAY(char)         HA_ARRAY_NCHAR;
typedef HA_ARRAY(wchar_t)      HA_ARRAY_CHAR;
typedef HA_ARRAY(short)        HA_ARRAY_SHORT;
typedef HA_ARRAY(int)          HA_ARRAY_INT;
typedef HA_ARRAY(int64_t)      HA_ARRAY_INT64;
typedef HA_ARRAY(void *)       HA_ARRAY_VOIDP;
typedef HA_ARRAY(char *)       HA_ARRAY_NSTRING;
typedef HA_ARRAY(wchar_t *)    HA_ARRAY_STRING;
typedef HA_ARRAY(float)        HA_ARRAY_FLOAT;
typedef HA_ARRAY(double)       HA_ARRAY_DOUBLE;
```

Create your own array type by placing any type at all between the parentheses.

To gain access to `wchar_t` and `int64_t`, `howard_a.h` includes header files `<wchar.h>` and `<stdint.h>`. Use of `long` just leads to trouble, in the author's experience, since its width varies across platforms, so `int64_t`, a standard 64-bit signed integral type, is used instead.

A variable of any of these types is a struct (not a pointer to a struct) with three fields: a typed pointer to arena memory holding the elements, the number of elements that that memory *can* hold, and the number of elements that it currently *does* hold. Structs are used rather than pointers to structs because extensible arrays are mainly used as aids to the implementation of other abstractions, and are thus usually private to one class or function, not shared. So there is no problem in having their structs lie directly in class objects or on the call stack, rather than in arena memory at the end of a pointer; and it is more efficient this way.

An array may be a field of an object that lies in one arena, while the array's arena memory lies in a different arena. But that would be unusual, since the array would normally have the same lifetime as the object, and thus would naturally belong in the same arena.

When an array is initialized, it contains no elements and no arena memory is allocated for it. Its pointer to arena memory points to a shared empty array in its arena. As the array grows, arena memory for it is allocated and reallocated, but always from the same arena. Each reallocation approximately doubles the number of elements that the array can hold, ensuring that another reallocation will not be needed soon, while wasting at most as much space as is used. Memory freed by a reallocation becomes available to hold other resizable objects in the same arena.

If one array is assigned to another using the C = operator or parameter passing, the arrays will have separate copies of their three fields, yet share their elements. This is only safe when the original array is not used afterwards, or the array's length remains constant thereafter.

Ha's array operations are macros, necessarily so since they are generic. They take structs as parameters, not pointers to structs. This encourages the user to think of arrays as opaque objects, like file pointers and so on. A disadvantage of macros is that their parameters may be evaluated more than once during a call. Unless explicitly stated otherwise, the user should assume that all parameters of all array operations are evaluated more than once. In many cases they are.

The first operation on any array must be to initialize it by a call to

```
    void HaArrayInit(ARRAY_X a, HA_ARENA arena);
```

This sets `a` to empty and specifies the arena which will supply its memory when elements are added later. Here and throughout this section, array operations are presented as though they are functions, even though they are really macros, and `ARRAY_X` stands for the type created by

```
    typedef HA_ARRAY(X) ARRAY_X;
```

for any type `X`. To find the arena that an initialized array `a` lies in, call

```
    HA_ARENA HaArrayArena(ARRAY_X a);
```

In general, memory allocated by Howard's functions can only be reclaimed by deleting the arena. However, resizable objects such as arrays are an exception, and function

```
    void HaArrayFree(ARRAY_X a);
```

frees the arena memory used by `a`, if any. This does not free `a` itself; `a` is not a pointer. It frees the memory holding the elements of `a`, making it available to other resizable objects in `a`'s arena.

To find the number of elements currently stored in an array, call

```
    int HaArrayCount(ARRAY_X a);
```

The elements have indexes from `0` to `HaArrayCount(a) - 1` inclusive, as usual in C. For efficiency, array bounds are not checked by any Ha operation. To access the element with index `i`, or the first element, or the last element, call

```
    X HaArray(ARRAY_X a, int i);
    X HaArrayFirst(ARRAY_X a);
    X HaArrayLast(ARRAY_X a);
```

`HaArray` and `HaArrayFirst` evaluate their parameters only once, and all three operations can be used as variables as well as values. So one can write, for example,

```
    HaArray(frequencies, i)++;
```

to increment the element of `frequencies` whose index is `i`, or

```
    do_something(&HaArrayFirst(a))
```

to pass a pointer to an element.

To add one element to an array, the operations are

```
    X HaArrayAdd(ARRAY_X a, int i, X x);
    X HaArrayAddFirst(ARRAY_X a, X x);
    X HaArrayAddLast(ARRAY_X a, X x);
```

`HaArrayAdd` adds `x` to `a` at index `i`, which may range from `0` to `HaArrayCount(a)` inclusive. It makes room for `x` by shifting elements up one place, including reallocating arena memory if necessary. It returns `x`. `HaArrayAddFirst(a, x)` is equivalent to `HaArrayAdd(a, 0, x)`, and `HaArrayAddLast(a, x)` is a faster version of `HaArrayAdd(a, HaArrayCount(a), x)`.

```
void HaArrayFill(ARRAY_X a, int len, X x);
```

adds x 0 or more times to the end of a, stopping when HaArrayCount(a) is at least len.

```
X HaArrayPut(ARRAY_X a, int i, X x);
```

replaces the value at index i with x and returns x. It evaluates its parameters only once. And

```
void HaArrayMove(ARRAY_X a, int dest_i, int src_i, int len);
```

uses the C memmove function to move (that is, copy with overlapping allowed) the len elements starting at index src_i to index dest_i. It assumes without checking that len >= 0 and that src_i and dest_i are at least 0 and at most HaArrayCount(a) - len. It is used by HaArrayAdd above and HaArrayShiftRight, HaArrayShiftLeft, and HaArrayDeleteAndShift below to do their shifting.

For searching an array there is

```
bool HaArrayContains(ARRAY_X a, X x, int *pos);
```

It returns true if a contains x, setting *pos to the index of its first occurrence; otherwise it returns false, leaving *pos unchanged. The individual comparisons are made by memcmp.

Two operations shift the entire contents of an array to the right or left:

```
void HaArrayShiftRight(ARRAY_X a, int n, X x);
void HaArrayShiftLeft(ARRAY_X a, int n);
```

HaArrayShiftRight shifts the elements of a to the right by n places. Afterwards, the array has n more elements than it did before. The first n places, opened up by the shift, are each initialized to x. It is up to the caller to ensure that 0 <= n. HaArrayShiftLeft shifts the elements of a to the left by n places. Afterwards, the array has n fewer elements than it did before. It is up to the caller to ensure that 0 <= n and n <= HaArrayCount(a).

Two operations delete the ith element, offering two ways to fill the gap it leaves behind:

```
void HaArrayDeleteAndShift(ARRAY_X a, int i);
void HaArrayDeleteAndPlug(ARRAY_X a, int i);
```

HaArrayDeleteAndShift shifts the elements after i down one place; HaArrayDeleteAndPlug assigns the last element to position i, then deletes the last element. Operations

```
bool HaArrayFindDeleteAndShift(ARRAY_X a, X x, int *pos);
bool HaArrayFindDeleteAndPlug(ARRAY_X a, X x, int *pos);
```

call HaArrayContains, returning what it returns but also using HaArrayDeleteAndShift or HaArrayDeleteAndPlug to delete the element it found, if any. There are also

```
void HaArrayDeleteLast(ARRAY_X a);
void HaArrayDeleteLastSlice(ARRAY_X a, int n);
void HaArrayClear(ARRAY_X a);
```

for deleting the last element, deleting the last n elements (which can be done very efficiently), and deleting the last `HaArrayCount(a)` elements, leaving the array empty. And

```
X HaArrayLastAndDelete(ARRAY_X a);
```

returns the last element of `a` and also deletes it from `a`. Deleting elements does not free any memory. The vacated memory remains available to the array, should it decide to grow again.

Here are some more complex operations that change the contents of arrays.

```
void HaArraySwap(ARRAY_X a, int i, int j, X tmp);
```

Swap the elements of `a` at positions `i` and `j`. Parameter `tmp` is a variable used to hold an element temporarily while swapping.

```
void HaArrayWholeSwap(ARRAY_X a, ARRAY_X b, ARRAY_X tmp);
```

Swap two whole arrays, that is, swap the contents of their structs, using `tmp` as a temporary.

```
void HaArrayAppend(ARRAY_X dest, ARRAY_X source, int i);
```

Append the elements of `source` to the end of `dest`, leaving `source` unchanged. Parameter `i` is a variable used as an external cursor when scanning `source`.

```
void HaArraySort(ARRAY_X a, int(*compar)(const void *, const void *));
```

Sort `a` by means of a call to `qsort`, using `compar` as the comparison function.

```
void HaArraySortUnique(ARRAY_X a,
  int(*compar)(const void *, const void *));
```

Like `HaArraySort`, except that after sorting, elements are deleted until no two adjacent elements return 0 when compared using `compar`. If this is done purely for uniqueifying, it is common to implement `compar` as a mere subtraction of two pointers. However, on a 64-bit architecture this yields a 64-bit integer, and merely returning this cast to `int`, the return type of `compar`, does not work. Use a conditional expression returning `-1`, `0`, or `1` instead.

Finally, Ha offers iterator macros for traversing arrays:

```
HaArrayForEach(ARRAY_X a, X x, int i)
HaArrayForEachReverse(ARRAY_X a, X x, int i)
```

These iterate over the elements of `a`, in forward or reverse order. Within each iteration, `x` is one element of `a` and `i` is the index of `x` in `a`. For example,

```
HaArrayForEach(strings, str, i)
  fprintf(stdout, "string %d: %s\n", i, str);
```

prints the elements of array `strings`. Like all Howard's iterators, both macros expand to

```
for( ... ; ... ; ... )
```

and may be used syntactically in any way that this construct may be.

### A.1.4. Howard's memory allocator

This section contains more information about Howard's memory allocator than the user is likely to need. It explains how memory is aligned, presents the operations for allocating resizable arena memory, and describes how the allocator works. What it does not do is explain the rationale behind the various features. They are all concerned with trying to avoid problems that memory allocators are prone to, as indeed is the basic design of arena sets containing arenas.

Howard's memory allocator promises to return memory aligned correctly for any kind of data. However, there seems to be no standard way to find out what that alignment is. So file `howard_a.h` includes a typedef of a type `HA_ALIGN_TYPE`, and the allocator assumes that memory aligned with this type aligns with all types. By default this typedef is

```
typedef void *HA_ALIGN_TYPE;
```

but it may be changed to any type whose size is at least the size of a pointer. `HaArenaSetMake` checks this condition and aborts if it does not hold, since the implementation depends on it.

*Resizable arena memory* is arena memory that can be resized. It is usually accessed via resizable arrays and symbol tables, but it can also be accessed directly, using these functions:

```
void *HaResizableAlloc(HA_ARENA a);
void *HaResizableReAlloc(void *resizable, size_t size);
void HaResizableFree(void *resizable);
```

`HaResizableAlloc` returns a pointer to `0` bytes of resizable arena memory from arena `a`. This may seem useless, but experience shows that it produces the most convenient initial value. All pointers to 0 bytes from `a` are shared, so there is no memory cost. `HaResizableReAlloc` assumes that `resizable` points to resizable arena memory, and begins by finding its arena and size. If `size` is no larger than this old size, `resizable` is returned. If `size` is larger, a pointer to `size` or more bytes of resizable arena memory from the same arena is returned. Its first old size bytes are copied from `resizable` using `memcpy`, and `resizable` is reclaimed for re-use by other calls for resizable memory from the same arena (unless its size is 0). `HaResizableFree` reclaims `resizable` just as `HaResizableReAlloc` does, but without allocating new memory.

The user can find the arena and size in bytes of a block of resizable arena memory:

```
HA_ARENA HaResizableArena(void *resizable);
size_t HaResizableSize(void *resizable);
```

`HaResizableSize` may be larger than the size requested when `resizable` was allocated. Like ordinary arena memory, resizable arena memory is aligned suitably for any kind of data.

The remainder of this section describes the implementation of the arena memory allocator.

An arena obtains its memory from its arena set, which obtains it from `malloc`. A piece of memory given to an arena set by `malloc` and passed on to an arena will be called a *chunk*; a piece of memory given to the user by an arena will be called a *block*.

Let $A$ be `sizeof(HA_ALIGN_TYPE)`. Since the memory returned has to align, every block might as well contain (and does contain) a number of bytes which is a multiple of $A$. If the number requested is not a multiple of $A$, it is increased to the next multiple of $A$. The resulting wasted memory is called the *alignment overhead*. It will be negligible in practice, and often zero.

One block of memory of size $A$, suitably aligned, will be called one *word*.

An arena cannot satisfy all the block requests it receives out of one chunk. So it calls on its arena set more than once, and maintains a linked list of the chunks it receives. The arena object contains a pointer to the most recently obtained chunk; this chunk begins with a pointer to the next most recently obtained chunk, and so on. The chunk also records the initial number of words available for allocation to end users in the chunk, the current number of words still available for allocation, and another integer, called $k$ below, which is roughly the base 2 logarithm of the number of words in the chunk. Most chunks are large, so this *chunk overhead* is negligible.

The linked list serves two purposes. First, when the arena is deleted, its memory is freed by traversing the list and returning the chunks to the arena set. The arena object itself lies in a separate arena (the arena set's private arena), but the block list header objects described below lie within chunks like user blocks do, and so are freed when the chunks are freed. Second, when the end user requests a new block, the first step is to try to obtain it from the first chunk on the list. Later chunks may not be entirely used up, but they are never tried.

When one chunk holds many blocks, arena allocation is much better than general allocation. Blocks are allocated contiguously within chunks, with no memory overhead other than the alignment overhead. Unless a new chunk is needed, allocating a block is very fast: just round up the requested size, test whether memory is available in the first chunk, and make two assignments.

All chunks cannot be the same size. If they were, for memory efficiency one would want that size to be large; but a large chunk would be wasteful if the arena remains small. Also, a request for a block whose size is larger than the chunk size could not be satisfied.

Accordingly, the chunks obtained from `malloc` vary in size, as follows. Each has size $2^k - c$ for some $k$ and $c$. A value for $c$ is chosen which is Ha's memory overhead per chunk plus the amount of `malloc`'s overhead, which seems to usually be 16 bytes. When a chunk of this size is requested, hopefully this will cause `malloc` to request a block of size $2^k$ from the operating system, which should work well.

Whenever an arena needs a new chunk, it requests one from the arena set of size $2^{k+1} - c$ words, given that its previous chunk has size $2^k - c$ words. Or if this is not enough memory to cover the current request from the user, it keeps doubling until the request is large enough. The arena set uses $k + 1$ to index an array of free chunks to find a list of free chunks of size $2^{k+1} - c$ words. If this list is empty it calls `malloc`.

When an arena is deleted, its chunks are moved into the free chunk lists of its arena set. Each chunk contains $k$ so this is a simple array indexing operation for each chunk. When an arena set is deleted, all its arenas except its private arena should already have been deleted. So it simply passes its free chunks, and the chunks of its private arena, to `free`.

It remains to describe how resizable blocks are handled. The size of each resizable block is $R_n A$ for some $n \geq 0$, where $R_0 = 0$ and $R_n = 3 \cdot 2^n - 1$ for $n \geq 1$. These numbers (0, 5, 11, 23, 47, ...) make good hash table sizes. From 5 onwards, each is obtained from its predecessor by doubling and adding one.

Growing out of each arena object is a linked list of *block list header* objects. The first block list header contains $R_0$ and a pointer to a singly linked list of all free blocks of size $R_0A$ (this particular pointer is always `NULL`); the second contains $R_1$ and a pointer to a singly linked list of all free blocks of size $R_1A$; and so on. Each block list header also contains a pointer to its arena and a pointer to the block list header for the next larger size. Initially, only the first block list header is present.

In addition to the $R_n A$ bytes passed to the user, a resizable block has $A$ bytes, just in front of the pointer returned to the user, holding a pointer to the block list header holding $R_n$. If the block is free, its second $A$ bytes holds a pointer to the next free resizable block of that size.

Given a user's pointer to a resizable block, one can find its block list header by going back $A$ bytes and following the pointer. The block list header gives access to the block's arena and size, and to the free block list of blocks of that size.

A resizable block of at least a given size can be obtained by searching the block list header list for the first block list header whose block size is sufficiently large. New block list headers are added if required as the search proceeds. Once the appropriate block list header is reached, its first free block is returned to the user; or if it has no free blocks, a fresh block is obtained from `HaAlloc`, a pointer to the block list header is placed in its first $A$ bytes, and a pointer to its $(A + 1)$th byte is returned to the user. `HaResizableReAlloc` begins its search for a block list header from `resizable`'s block list header. Most calls to `HaResizableReAlloc` request blocks about double the old size, so most traversals of the list of block list headers visit only one block list header, ensuring that the time taken to find a new resizable block is usually a small constant.

The memory overhead is $A$ bytes per allocated block (holding the pointer to the block list header), plus the space occupied by the block list headers (negligible once the blocks grow to even moderate size), plus the free blocks, plus any unused space within allocated blocks.

The worst case is elicited by an arena containing a single extensible array that grows one element at a time. (This case can be duplicated by growing two arrays in parallel.) Now, resizable blocks are needed just because the application cannot predict how much memory will be needed. Thus, the application might as well ask for sizes of the form $R_n A$, and the extensible array module does this. As the array grows, it leaves a trail of freed blocks behind it of sizes $(5 + 1)A$, $(11 + 1)A$, $(23 + 1)A$, and so on. Their total size is less than half the current block size. The current block may itself be only half full, so at worst, three times as much memory is allocated as is used. But none of this memory is completely lost: half of it is available for further growth of the array, the other half is available for other arrays, and all of it is freed when the arena is deleted.

## A.2. Strings and symbol tables

*Note – the Hn library has been flown in from another project of the author's, called Howard. Three libraries, Hw, Hn, and Ho, are documented here, but only the Hn library is included in KHE. Its header file is* `howard_n.h`.

This section describes Howard's Hw library, which provides operations on wide strings (type `wchar_t *`), and symbol tables whose keys are wide strings. It also documents Howard's Hn library, which is the same except that its strings are narrow (`char *` instead of `wchar_t *`).

### A.2.1. Strings

One handy use for extensible arrays is to build up strings piece by piece in arena memory, similarly to `open_memstream` from POSIX-2008:

```
void HwStringBegin(HA_ARRAY_CHAR ac, HA_ARENA a);
void HwStringAdd(HA_ARRAY_CHAR *ac, wchar_t *format, ...);
wchar_t *HwStringEnd(HA_ARRAY_CHAR ac);
```

`HwStringBegin` and `HwStringEnd` are in fact macros. `HwStringAdd` is a function; note that a reference to the array is passed, not the array itself. `HA_ARRAY_CHAR` is defined by Ha and holds an extensible array of wide characters. `HwStringBegin` initializes this array to empty (like `HaArrayInit`); `HwStringAdd` appends a formatted string to the growing array; and `HwStringEnd` adds the final `L'\0'` and returns the string. The string returned by `HwStringEnd` (call it `str`) may be freed by calling either `HaArrayFree(ac)` or `HaResizableFree(str)`. There is also

```
void HwStringVAdd(HA_ARRAY_CHAR *ac, wchar_t *format, va_list args);
```

which is to `HwStringAdd` what `vwprintf` is to `wprintf`.

Thanks to a robust implementation, there is no limit on the size of any one of the formatted strings added to `*ac` by these functions. There is an unchecked limit of `INT_MAX - 1` on the total length of the string, because type `HA_ARRAY_CHAR` stores an array length in an `int` field.

In between the calls to `HwStringBegin` and `HwStringEnd`, ordinary array operations may be applied to `ac` as usual. For example,

```
HaArrayFill(ac, 80, L' ');
```

pads out `ac` to length 80 with blanks.

For the convenience of applications which sometimes need to build a string and sometimes need to write to a file, functions

```
void HwStringAddOrPrint(HA_ARRAY_CHAR *ac, FILE *fp,
  const wchar_t *format, ...);
void HwStringVAddOrPrint(HA_ARRAY_CHAR *ac, FILE *fp,
  const wchar_t *format, va_list args);
```

are defined. These are like `HwStringAdd` and `HwStringVAdd` when `ac != NULL`, and like `fwprintf` and `vfwprintf` when `ac == NULL` (so `fp` had better be non-`NULL` in that case).

Hw offers three other functions that create strings in arena memory:

```
wchar_t *HwStringCopy(wchar_t *s, HA_ARENA a);
wchar_t *HwStringSubstring(wchar_t *s, int start, int len, HA_ARENA a);
wchar_t *HwStringMake(HA_ARENA a, const wchar_t *format, ...);
```

These are functions, not macros. The arena memory remains allocated until the arena is freed. `HwStringCopy` returns a copy of `s`, like the Linux `wcsdup`. `HwStringSubstring` returns the substring of `s` which begins at position `start`, counting from 0, and has length `len`, or less if `s` ends before then. `HwStringMake` returns a formatted string:

```
name = HwStringMake(a, L"%ls/%ls_%d", dir_name, file_name, version);
```

Thanks to a robust implementation, `HwStringMake` imposes no length limits. There is also

```
wchar_t *HwStringVMake(HA_ARENA a, const wchar_t *format, va_list args);
```

which is to `HwStringMake` what `vwprintf` is to `wprintf`.

Howard is written on the assumption that strings stored in memory will generally be wide strings. Even so, some conversion is needed when interfacing with the operating system, so Hw offers two functions that convert from and to narrow strings:

```
wchar_t *HwStringFromNarrow(char *s, HA_ARENA a);
char *HwStringToNarrow(wchar_t *s, HA_ARENA a);
```

For example, `HwStringFromNarrow` is useful for converting a command-line argument, which is a narrow string, to a string, and `HwStringToNarrow` is useful for converting a file name to the narrow string format required by `fopen`. These strings may be freed immediately by passing them to `HaResizableFree`, or kept until the arena is deleted later.

The standard C library offers several functions which query strings (`wcscmp`, `wcsstr`, etc.). These may be used on Hw's strings. Hw supplements these functions with a few others:

```
int HwStringCount(wchar_t *s);
bool HwStringIsEmpty(wchar_t *s);
```

Return the length of `s`, like `wcslen`; return `true` if `s` has count 0.

```
bool HwStringEqual(wchar_t *s1, wchar_t *s2);
```

Return `true` if `s1` and `s2` are equal.

```
bool HwStringContains(wchar_t *s, wchar_t *substr, int *pos);
```

If `substr` occurs within `s`, return `true` with `*pos` set to the starting position of the first occurrence of `substr` within `s`. Otherwise return `false` with `*pos` not set.

```
bool HwStringBeginsWith(wchar_t *s, wchar_t *prefix);
bool HwStringEndsWith(wchar_t *s, wchar_t *suffix);
```

Return `true` if `prefix` occurs within `s` at the start, or if `suffix` occurs within `s` at the end.

### A.2.2. Abort and assert

Hw offers two functions for checking assertions:

```
void HwAbort(wchar_t *fmt, ...);
void HwAssert(bool cond, wchar_t *fmt, ...);
```

`HwAbort`'s parameters are the same as `wprintf`'s, but it prints onto `stderr` and then calls `abort`. `HwAssert` does nothing if `cond` is `true`, and it does what `HwAbort` does if `cond` is `false`. It is a function, not a macro, so its parameters must be well-defined whether `cond` is true or not.

### A.2.3. Symbol tables

A symbol table is a set of *entries*, each consisting of a *key*, which is a string, and a *value*, whose type is the same for all entries but is otherwise arbitrary. One table may contain any number of entries. Entries may be added, deleted, and retrieved by key.

As for Ha's arrays, and for the same reasons, Hw's symbol tables are structs, not pointers to structs, and the operations are macros. The implementation is a linear probing hash table, which is essentially just an array (actually two arrays, one for keys, one for values). At any moment, not all of the array's elements contain entries. The table doubles in size when it becomes 80% full.

To define a symbol table type whose keys are strings of type `wchar_t *` and whose values have type `X`, where `X` is any type, write this:

```
typedef HW_TABLE(X) TABLE_X;
```

From now on, `TABLE_X` stands for any type defined by a typedef like this one, and `X` stands for the type between the parentheses in that typedef. To initialize a symbol table, call

```
void HwTableInit(TABLE_X table, HA_ARENA a);
```

To find the arena containing a given table, call

```
HA_ARENA HwTableArena(TABLE_X table);
```

When the symbol table is no longer needed, its memory may be reclaimed by

```
void HwTableFree(TABLE_X table);
```

This does not free `table` itself (`table` is not a pointer). It frees the memory used to hold the arrays of keys and values, although not the keys and values themselves.

To add an entry to a symbol table, call

```
void HwTableAdd(TABLE_X table, wchar_t *key, X value);
bool HwTableAddUnique(TABLE_X table, wchar_t *key, X value, X other);
```

`HwTableAdd` adds a new entry with the given key and value to the table, even if that causes the table to contain two or more entries with the same key. `HwTableAddUnique`, on the other hand, first checks whether there is already an entry with the given key. If so, it sets `other` to the value of an existing entry with the given key and returns `false` without changing the table. If not, it adds the new entry and returns `true` without setting `other`.

Two variants of `HwTableAdd` and `HwTableAddUnique` are offered:

```
void HwTableAddHashed(TABLE_X table, int hash_code, wchar_t *key,
  X value);
bool HwTableAddUniqueHashed(TABLE_X table, int hash_code, wchar_t *key,
  X value, X other);
```

These are the same as the originals, except for parameter `hash_code`, which is assumed to be the hash code of `key` (before reduction modulo the table size), as returned by `HwTableHash`:

```
int HwTableHash(wchar_t *key);
```

Passing the hash code explicitly saves time when inserting the same entry into several tables.

Retrieval has three forms. The first is the 'contains' form, which merely reports whether an entry with the given key is present:

```
bool HwTableContains(TABLE_X table, wchar_t *key, int pos);
bool HwTableContainsHashed(TABLE_X table, int hash_code, wchar_t *key,
  int pos);
bool HwTableContainsNext(TABLE_X table, int pos);
```

`HwTableContains` returns `true` if `table` contains an entry with the given key, setting `pos` to its position in the table, or `false` if there is no such entry, in which case `pos` is an empty position. `HwTableContainsHashed` is the same, except that it assumes that `hash_code` is the hash code of `key` (before reduction modulo the table size). `HwTableContainsNext` assumes that `pos` is a non-empty position of `table`; it searches the table beyond that point (wrapping around to the front if necessary) for an entry with the same key as the one at that point. Like `HwTableContains`, it returns `true` or `false` depending on whether it finds such an entry, and it changes `pos` to its position, or to an empty position.

The second form of retrieval is the 'contains value' form, which reports whether an entry with the given key and value is present:

```
void HwTableContainsValue(TABLE_X table, wchar_t *key, X value,
  int pos);
void HwTableContainsValueHashed(TABLE_X table, int hash_code,
  X value, int pos);
```

`HwTableContainsValue` hashes the key and then compares values along the table using the C '==' operation, instead of comparing keys. It runs very quickly since it executes no string comparisons. Owing to problems behind the scenes it does not return a Boolean result. Instead, it is syntactically a `for` statement which sets `pos` to the position of the entry if present. Function `HwTableOccupied`, defined below, may be used to determine the outcome, like this:

```
HwTableContainsValue(table, "fred", fred, pos);
if( HwTableOccupied(table, pos) )
{
  /* fred is present at position pos */
}
```

`HwTableContainsValueHashed` is the same, except that it avoids hashing the key as usual. In fact it does not need to know the key, so the usual `key` parameter is omitted.

The third form of retrieval is the 'retrieve' form, which sets a `value` parameter to the value associated with the given key if found, and leaves `value` untouched if not:

```
bool HwTableRetrieve(TABLE_X table, wchar_t *key, X value, int pos);
bool HwTableRetrieveHashed(TABLE_X table, int hash_code, wchar_t *key,
  X value, int pos);
bool HwTableRetrieveNext(TABLE_X table, X value, int pos);
```

Apart from setting `value`, these are the same as the corresponding 'contains' versions.

The `pos` parameters of retrieval functions have several uses. They are needed to ensure that concurrent retrievals do not interfere with each other. They can be passed to

```
bool HwTableOccupied(TABLE_X table, int pos);
wchar_t *HwTableKey(TABLE_X table, int pos);
X HwTableValue(TABLE_X table, int pos);
```

which return `true` if position `pos` is occupied (has an entry), and if so, the key and value of the entry at position `pos`. And they are used by the operations to be defined next.

Assuming that there is an entry at position `pos`,

```
void HwTableReplace(MTABLE_X table, int pos, X value);
```

replaces the entry's value, and

```
void HwTableDelete(TABLE_X table, int pos);
```

deletes the entry. For example,

```
if( HwTableContains(table, L"fred", pos) )
  HwTableDelete(table, pos);
```

deletes an entry with key `L"fred"`, if there is one. Function

```
void HwTableClear(TABLE_X table);
```

deletes every entry in the table, leaving it empty.

For traversal there are iterator macros in the usual style:

```
void HwTableForEachWithKey(TABLE_X table, wchar_t *key, X value, int pos)
void HwTableForEachWithKeyHashed(TABLE_X table, int hash_code,
  wchar_t *key, X value, int pos)
```

These visit each entry with a given key. `HwTableForEachWithKeyHashed` is the same as `HwTableForEachWithKey` except that the user supplies the hash code as well as the key, as for `HwTableRetrieveHashed`. For example, to visit every person called `L"fred"` in table `people`:

```
HwTableForEachWithKey(people, L"fred", person, pos)
{
   ... visit person ...
}
```

On each iteration, this code sets `person` to a person with name `L"fred"`, and `pos` to the position of that person in the table. A similar iterator macro visits every entry of the table:

```
void HwTableForEach(TABLE_X table, wchar_t *key, X value, int pos)
```

The entries will be visited in an essentially random order, as usual with hash tables. For example, the following code counts the number of entries in `table`:

```
count = 0;
HwTableForEach(table, key, value, pos)
  count++;
```

This number is not maintained automatically. Another fairly useless number is

```
int HwTableSize(TABLE_X table);
```

which is the current array size. It will be somewhat larger than the current number of entries.

### A.2.4. Narrow strings and symbol tables

This section describes Howard's Hn library. It is the same as Hw except that its strings are *narrow* (have type `char *` instead of `wchar_t *`), so the description is brief.

For creating narrow strings in arena memory there are functions

```
void HnStringBegin(HA_ARRAY_NCHAR anc, HA_ARENA a);
void HnStringAdd(HA_ARRAY_NCHAR *anc, char *format, ...);
char *HnStringEnd(HA_ARRAY_NCHAR anc);
void HnStringVAdd(HA_ARRAY_NCHAR *anc, const char *format, va_list args)
```

(`HnStringBegin` and `HnStringEnd` are macros). For either adding to a string in memory or adding to a file, use functions

```
void HnStringAddOrPrint(HA_ARRAY_NCHAR *anc, FILE *fp,
  const char *format, ...);
void HnStringVAddOrPrint(HA_ARRAY_NCHAR *anc, FILE *fp,
  const char *format, va_list args);
```

Other functions which create strings in arena memory are

```
char *HnStringCopy(char *s, HA_ARENA a);
char *HnStringSubstring(char *s, int start, int len, HA_ARENA a);
char *HnStringMake(HA_ARENA a, const char *format, ...);
char *HnStringVMake(HA_ARENA a, const char *format, va_list args);
char *HnStringFromWide(wchar_t *s, HA_ARENA a);
wchar_t *HnStringToWide(char *s, HA_ARENA a);
```

For querying strings there are

```
int HnStringCount(char *s);
bool HnStringIsEmpty(char *s);
bool HnStringEqual(char *s1, char *s2);
bool HnStringContains(char *s, char *substr, int *pos);
bool HnStringBeginsWith(char *s, char *prefix);
bool HnStringEndsWith(char *s, char *suffix);
```

For handling white space there are

```
bool HnStringIsWhiteSpaceOnly(char *s);
char *HnStringCopyStripped(char *s, HA_ARENA a);
```

`HnStringIsWhiteSpaceOnly` returns `true` if `s` is `NULL` or consists of white space characters only (including when `s` is empty), and `HnStringCopyStripped` returns a copy of `s` with any white space characters at the beginning or end removed; if `s` is `NULL` or there are no non-white space characters it returns the empty string.

For abort and assert there are

```
void HnAbort(char *fmt, ...);
void HnAssert(bool cond, char *fmt, ...);
```

A symbol table is defined by

```
typedef HN_TABLE(X) TABLE_X;
```

and initialized, its arena returned, and freed by

```
void HnTableInit(TABLE_X table, HA_ARENA a);
HA_ARENA HnTableArena(TABLE_X table);
void HnTableFree(TABLE_X table);
```

Entries are added with

```
void HnTableAdd(TABLE_X table, char *key, X value);
bool HnTableAddUnique(TABLE_X table, char *key, X value, X other);
```

plus the two variants

```
void HnTableAddHashed(TABLE_X table, int hash_code, char *key,
  X value);
bool HnTableAddUniqueHashed(TABLE_X table, int hash_code, char *key,
  X value, X other);
```

Hash codes are calculated with

```
int HnTableHash(char *key);
```

Retrievals are carried out with

```
bool HnTableContains(TABLE_X table, char *key, int pos);
bool HnTableContainsHashed(TABLE_X table, int hash_code, char *key,
  int pos);
bool HnTableContainsNext(TABLE_X table, int pos);

void HnTableContainsValue(TABLE_X table, char *key, X value,
  int pos);
void HnTableContainsValueHashed(TABLE_X table, int hash_code,
  X value, int pos);

bool HnTableRetrieve(TABLE_X table, char *key, X value, int pos);
bool HnTableRetrieveHashed(TABLE_X table, int hash_code, char *key,
  X value, int pos);
bool HnTableRetrieveNext(TABLE_X table, X value, int pos);
```

The positions returned by the retrieve operations may be used in

```
bool HnTableOccupied(TABLE_X table, int pos);
char *HnTableKey(TABLE_X table, int pos);
X HnTableValue(TABLE_X table, int pos);
```

To replace a value, delete an entry, or clear the table, call

```
void HnTableReplace(TABLE_X table, int pos, X value);
void HnTableDelete(TABLE_X table, int pos);
void HnTableClear(TABLE_X table);
```

To iterate over all entries with a given key, use iterator macros

```
void HnTableForEachWithKey(TABLE_X table, char *key, X value, int pos)
void HnTableForEachWithKeyHashed(TABLE_X table, int hash_code,
  char *key, X value, int pos)
```

To iterate over all entries, use

```
void HnTableForEach(TABLE_X table, char *key, X value, int pos)
```

Finally,

```
int HnTableSize(TABLE_X table);
```

returns the size of the hash table.

### A.2.5. Pointer tables and groups

A *pointer table* is like an object table in that it is a hash table indexed by non-`NULL` void pointers rather than strings. However, the value of what the pointer points to is used to index the table. To make this work, the table needs to be given two functions, one to hash what the pointer points to, and one to compare two of those things to decide whether they are equal. So the user must supply two functions with these signatures:

```
int KeyHash(void *p);
bool KeyEqual(void *p1, void *p2);
```

Then Howard does the rest.

Two equal keys must have the same hash value. In other words, if `KeyEqual(p1, p2)` returns `true`, then `KeyHash(p1)` and `KeyHash(p2)` must be equal. Without this the pointer table will not work as expected.

There is an optional third user-defined function,

```
void KeyDebug(void *p, FILE *fp);
```

When present, it is used by `HpTableDebug` below to produce a debug print of key `p` onto file fp.

This section describes Howard's Hp library, which implements pointer tables, offering operations analogous to the symbol table operations from Hw and Hn.

A pointer table whose keys are what is pointed to by void pointers and whose values have type `X`, for any `X`, is defined by

```
typedef HP_TABLE(X) TABLE_X;
```

It is initialized, its attributes are returned, and it is freed by

```
void HpTableInit(TABLE_X table, HP_HASH_FN key_hash_fn,
  HP_EQUAL_FN key_equal_fn, HP_DEBUG_FN key_debug_fn, HA_ARENA a);
HP_HASH_FN HpTableKeyHashFn(TABLE_X table);
HP_EQUAL_FN HpTableKeyEqualFn(TABLE_X table);
HP_DEBUG_FN HpTableKeyDebugFn(TABLE_X table);
HA_ARENA HpTableArena(TABLE_X table);
void HpTableFree(TABLE_X table);
```

where `HP_HASH_FN`, `HP_EQUAL_FN`, and `HP_DEBUG_FN` are the types of the three user-supplied functions described above. The value of `key_debug_fn` may be `NULL`, but the other two have to really hash a key and compare two keys for equality.

Entries are added with

```
void HpTableAdd(TABLE_X table, void *key, X value);
bool HpTableAddUnique(TABLE_X table, void *key, X value, X other);
```

plus the two variants

```
void HpTableAddHashed(TABLE_X table, int hash_code, void *key,
  X value);
bool HpTableAddUniqueHashed(TABLE_X table, int hash_code, void *key,
  X value, X other);
```

Retrievals are carried out with

```
bool HpTableContains(TABLE_X table, void *key, int pos);
bool HpTableContainsHashed(TABLE_X table, int hash_code, void *key,
  int pos);
bool HpTableContainsNext(TABLE_X table, int pos);

void HpTableContainsValue(TABLE_X table, void *key, X value,
  int pos);
void HpTableContainsValueHashed(TABLE_X table, int hash_code,
  X value, int pos);

bool HpTableRetrieve(TABLE_X table, void *key, X value, int pos);
bool HpTableRetrieveHashed(TABLE_X table, int hash_code, void *key,
  X value, int pos);
bool HpTableRetrieveNext(TABLE_X table, X value, int pos);
```

The positions returned by the retrieve operations may be used in

```
bool HpTableOccupied(TABLE_X table, int pos);
void *HpTableKey(TABLE_X table, int pos);
X HpTableValue(TABLE_X table, int pos);
```

To replace a value, delete an entry, or clear the table, call

```
    void HpTableReplace(TABLE_X table, int pos, X value);
    void HpTableDelete(TABLE_X table, int pos);
    void HpTableClear(TABLE_X table);
```

To iterate over all entries with a given key, use iterator macros

```
    void HpTableForEachWithKey(TABLE_X table, void *key, X value, int pos)
    void HpTableForEachWithKeyHashed(TABLE_X table, int hash_code,
      void *key, X value, int pos)
```

To iterate over all entries, use

```
    void HpTableForEach(TABLE_X table, void *key, X value, int pos)
```

or

```
    void HpTableForEachValue(TABLE_X table, X value, int pos)
```

to omit retrieving each key (which can produce unwanted error messages about unused variables). Function

```
    int HpTableSize(TABLE_X table);
```

returns the size of the table. Function

```
    float HpTableProbeLength(TABLE_X table);
```

returns the average, taken over all calls to `HpTableAddUnique`, `HpTableAddUniqueHashed`, `HpTableContains`, `HpTableContainsHashed`, `HpTableRetrieve`, `HpTableRetrieveHashed`, and `HpTableContainsNext`, of the number of probes of non-empty table entries made by those calls. The result should be less than about 3; higher values are a sign that the hash function is not working effectively. `HpTableProbeLength` requires macro `HP_DEBUG_PROBE_LENGTH`, defined at the top of file `howard_p.h`, to have value `1`; when it has value `0`, `HpTableProbeLength` returns `-1.0`. This is also the value returned when no calls to the functions have been made.

Finally, function

```
    void HpTableDebug(TABLE_X table, int indent, FILE *fp);
```

produces a debug print of `table` onto file `fp` with the given indent. This can be long. It uses parameter `key_debug_fn` to produce a debug print of each key, unless `key_debug_fn` is `NULL`, in which case it prints the address of each key

Hp also offers a version of the pointer table idea in which the keys have no corresponding values. This is useful when the need is merely to build a set of objects and find out whether a given object is present in the set or not. Hp calls this data structure a *pointer group*.

A pointer group is not generic; its unique type is `HP_GROUP`. It is initialized, its arena returned, and freed by

```
void HpGroupInit(HP_GROUP group, HP_HASH_FN key_hash_fn,
  HP_EQUAL_FN key_equal_fn, HP_DEBUG_FN key_debug_fn, HA_ARENA a);
HP_HASH_FN HpGroupKeyHashFn(HP_GROUP group);
HP_EQUAL_FN HpGroupKeyEqualFn(HP_GROUP group);
HP_DEBUG_FN HpGroupKeyDebugFn(HP_GROUP group);
HA_ARENA HpGroupArena(HP_GROUP group);
void HpGroupFree(HP_GROUP group);
```

Entries are added with

```
void HpGroupAdd(HP_GROUP group, void *key);
bool HpGroupAddUnique(HP_GROUP group, void *key);
```

plus the two variants

```
void HpGroupAddHashed(HP_GROUP group, int hash_code, void *key);
bool HpGroupAddUniqueHashed(HP_GROUP group, int hash_code, void *key);
```

Note the absence of values.

Retrievals are carried out with

```
bool HpGroupContains(HP_GROUP group, void *key, int pos);
bool HpGroupContainsHashed(HP_GROUP group, int hash_code, void *key,
  int pos);
bool HpGroupContainsNext(HP_GROUP group, int pos);
```

There are no `ContainsValue` or `Retrieve` operations. The positions returned by the contains operations may be used in

```
bool HpGroupOccupied(HP_GROUP group, int pos);
void *HpGroupKey(HP_GROUP group, int pos);
```

To delete an entry or clear the group, call

```
void HpGroupDelete(HP_GROUP group, int pos);
void HpGroupClear(HP_GROUP group);
```

To iterate over all entries with a given key, use iterator macros

```
void HpGroupForEachWithKey(HP_GROUP group, void *key, int pos)
void HpGroupForEachWithKeyHashed(HP_GROUP group, int hash_code,
  void *key, int pos)
```

To iterate over all entries, use

```
void HpGroupForEach(HP_GROUP group, void *key, int pos)
```

Finally,

```
int HpGroupSize(HP_GROUP group);
```

returns the size of the group.

## A.3. Sets of integers

### A.3.1. Variable-length bitsets

KHE comes with a C module called LSet for managing variable-length sets of smallish unsigned integers implemented as bit vectors. The module consists of header file `khe_lset.h` and implementation file `khe_lset.c`. These are stored and compiled with KHE, but they can also be used separately. KHE formerly used LSet extensively behind the scenes (all its time groups, resource groups, and event groups were represented both as arrays of elements and LSets of element index numbers), although now SSets (Appendix A.3.3) are used instead. LSet may be useful when writing helper functions and solvers. To use it, simply include `khe_lset.h`. Including `khe_solvers.h` does not automatically include `khe_lset.h` as well.

File `khe_lset.h` begins with these two type definitions:

```
typedef struct lset_rec *LSET;
typedef HA_ARRAY(LSET) ARRAY_LSET;
```

The first defines the type of an LSet, and the second defines an array of LSets, as usual.

Internally, an LSet is represented by a pointer to a `struct` containing a length followed by the bit vector itself. When an element needs to be added that would overflow the currently allocated memory, the whole LSet is freed and a new one is returned. This is not particularly convenient for the user of LSet but it is the most efficient way.

Functions

```
LSET LSetNew(void);
void LSetFree(LSET s);
```

create a new, empty LSet and free an LSet;

```
LSET LSetCopy(LSET s);
```

creates a fresh new LSet with the same value as `s`. Function

```
void LSetShift(LSET s, LSET *res, unsigned int k,
  unsigned int max_nonzero);
```

takes two existing LSets, `s` and `*res`, and replaces the current value of `*res` by `s` with `k` added to each of its elements, except that elements which would thereby have value greater than `max_nonzero` are omitted. The old `*res` will be freed and a new one allocated if necessary. This arcane function is used behind the scenes to calculate shifted time domains. Function

```
void LSetClear(LSET s);
```

clears `s` back to the empty set, and

```
void LSetInsert(LSET *s, unsigned int i);
void LSetDelete(LSET s, unsigned int i);
```

insert element `i` (changing nothing if `i` is already present) and delete it (changing nothing if `i` is already absent). The value of `i` is arbitrary but very large values are obviously undesirable, since the bit vectors then become very large.

```
    void LSetAssign(LSET *target, LSET source);
```

replaces the current value of `*target` with the value of `source`, reallocating `*target` if necessary. The value is a copy, there is no sharing anywhere in the LSet module.

The next three functions implement the set operations of union, intersection, and difference, replacing their first parameter's value with the result of the operation:

```
    void LSetUnion(LSET *target, LSET source);
    void LSetIntersection(LSET target, LSET source);
    void LSetDifference(LSET target, LSET source);
```

The usual Boolean operations are available on LSets:

```
    bool LSetEmpty(LSET s);
    bool LSetEqual(LSET s1, LSET s2);
    bool LSetSubset(LSET s1, LSET s2);
    bool LSetDisjoint(LSET s1, LSET s2);
    bool LSetContains(LSET s, unsigned int i);
```

These return `true` when `s` is empty, when `s1` and `s2` are equal, when `s1` is a subset of `s2`, when `s1` and `s2` are disjoint, and when `s` contains `i`. Functions

```
    unsigned int LSetMin(LSET s);
    unsigned int LSetMax(LSET s);
```

return the smallest and largest elements of `s` respectively, using an efficient table lookup on the first or last non-zero byte. Both functions abort if `s` is empty. Function

```
    int LSetLexicalCmp(LSET s1, LSET s2);
```

returns a negative, zero, or positive result depending on whether `s1` is lexicographically less than, equal to, or greater than `s2`. Function

```
    void LSetExpand(LSET s, ARRAY_SHORT *add_to)
```

assumes that `*add_to` is an initialized array, and adds the elements of `s` to the array in increasing order by repeated calls to `HaArrayAddLast`. Function

```
    char *LSetShow(LSET s);
```

returns a display of `s` in static memory (so it is not thread-safe, but it does keep four separate buffers, allowing it to be called several times in one line of debug output). Finally,

```
    void LSetTest(FILE *fp);
```

tests the module and prints its results onto file `fp`.

### A.3.2. Sorted sets

Many of KHE's object collections are represented by a set of sorted integers, which are indexes into arrays of objects. Sets of times are represented using *shiftable sets* (Appendix A.3.3), for efficient calculation of time group neighbourhoods. Other collections are represented by an extensible array of integers, sorted into increasing order. These *KHE sets* are described here.

The KHE set module consists of header file `khe_set.h` and implementation file `khe_set.c`. These are stored and compiled with the KHE platform, but they can also be used separately. To use KHE sets, simply include `khe_set.h`. Including `khe_platform.h` or `khe_solvers.h` does not automatically include `khe_set.h`.

File `khe_set.h` contains this definition of type `KHE_SET`, representing one KHE set:

```
typedef struct khe_set_rec { ... } KHE_SET;
```

We've omitted the contents, but that is just an extensible array of integer indexes, stored in increasing order. KHE sets are sets, not multisets—there are no duplicates among the items.

Type `KHE_SET` is a struct, not a pointer to a struct, because `KHE_SET` is intended as an aid to implementing other modules, and values of type `KHE_SET` are expected to be private fields of these other modules' structs. Structs are better than pointers to structs in these cases, because they save memory and avoid one level of indirection.

To pass a `KHE_SET` as a parameter it is always best to pass its address, not the struct itself. The following functions appear to violate this rule, but they are in fact macros which insert the address-of operators for you. For example, the function given as

```
void KheSetUnion(KHE_SET to_s, KHE_SET from_s);
```

below is really macro

```
#define KheSetUnion(to_s, from_s) KheSetImplUnion(&(to_s), &(from_s))
```

and thus passes its `KHE_SET` parameters by reference.

To initialize KHE set `s` to an empty set, using memory from arena `a`, call

```
void KheSetInit(KHE_SET s, HA_ARENA a);
```

To make a new KHE set `to_s` in arena `a` by copying another set `from_s`, call

```
void KheSetCopy(KHE_SET to_s, KHE_SET from_s, HA_ARENA a);
```

To copy the elements of `from_s` to an existing set `to_s`, first clearing `to_s`, call

```
void KheSetCopyElements(KHE_SET to_s, KHE_SET from_s);
```

To clear `s`, call

```
void KheSetClear(KHE_SET s);
```

To insert `item` into `s`, call

```
void KheSetInsert(KHE_SET s, int item);
```

This does nothing if `item` is already present. To delete `item` from `s`, call

```
void KheSetDelete(KHE_SET s, int item);
```

This does nothing if `item` is not present. To delete the last (largest) element, assuming there is one, call

```
void KheSetDeleteLast(KHE_SET s);
```

There are operations to replace `to_s` by its union, intersection, and difference from `from_s`:

```
void KheSetUnion(KHE_SET to_s, KHE_SET from_s);
void KheSetIntersect(KHE_SET to_s, KHE_SET from_s);
void KheSetDifference(KHE_SET to_s, KHE_SET from_s);
```

and operations to return the cardinality of the union, intersection, difference, and symmetric difference, without changing `to_s`:

```
int KheSetUnionCount(KHE_SET to_s, KHE_SET from_s);
int KheSetIntersectCount(KHE_SET to_s, KHE_SET from_s);
int KheSetDifferenceCount(KHE_SET to_s, KHE_SET from_s);
int KheSetSymmetricDifferenceCount(KHE_SET to_s, KHE_SET from_s);
```

For finding the number of elements, the `i`th element, and the last element, there are

```
int KheSetCount(KHE_SET s);
int KheSetGet(KHE_SET s, int i);
int KheSetGetLast(KHE_SET s);
```

Then there are the Boolean queries

```
bool KheSetEmpty(KHE_SET s);
bool KheSetEqual(KHE_SET s1, KHE_SET s2);
bool KheSetSubset(KHE_SET s1, KHE_SET s2);
bool KheSetDisjoint(KHE_SET s1, KHE_SET s2);
bool KheSetContains(KHE_SET s, int item);
```

which return `true` when `s` is empty, when `s1` is equal to, a subset of, and disjoint from `s2`, and when `s` contains `item`. For sorting an array of sets to bring equal sets together there is

```
int KheSetTypedCmp(KHE_SET s1, KHE_SET s2);
```

Also offered are

```
int KheSetMin(KHE_SET s);
int KheSetMax(KHE_SET s);
```

which return the minimum (first) and maximum (last) elements of `s`.

For iterating over sets there are the iterator macros

```
KheSetForEach(KHE_SET s, int t, int i)
KheSetForEachReverse(KHE_SET s, int t, int i)
```

These work like `HaArrayForEach` and `HaArrayForEachReverse`, repeatedly setting `t` to the `i`th item of `s`, as `i` increases or decreases. Finally,

```
char *KheSetShow(KHE_SET s);
```

returns a string representation of `s`. The result lies in static memory and will be overwritten by the next call to `KheSetShow`. More precisely, `KheSetShow` holds four static memory buffers, each 200 characters long. Any result which should be longer than 200 characters is safely truncated to 200 characters (including the final `'\0'`). The fifth call to `KheSetShow` re-uses the buffer used by the first call, the sixth call re-uses the buffer used by the second call, and so on.

There are tables whose entries have sets for keys and generic pointers for values. These are implemented by tries, so the integer elements of the sets should not be very large.[1] To create a new table using memory from a given arena, and to insert and retrieve in it, call

```
KHE_SET_TABLE KheSetTableMake(HA_ARENA a);
void KheSetTableInsert(KHE_SET_TABLE st, KHE_SET s, void *val);
bool KheSetTableRetrieve(KHE_SET_TABLE st, KHE_SET s, void *val);
```

One can obtain a debug print of a table from

```
void KheSetTableDebug(KHE_SET_TABLE st, int indent, FILE *fp);
```

At present these are the only table operations offered.

### A.3.3. Shiftable sets

KHE has a C module called SSet for managing *shiftable sets* of integers. These are sets which hold an integer *shift* which is added to each value, allowing shifted copies to be created very efficiently, as needed when implementing time group neighbourhoods. A shifted copy may also be *sliced*, that is, trimmed at each end to produce a subset of the original set.

The module consists of header file `sset.h` and implementation file `sset.c`. These are stored and compiled with KHE, but they can also be used separately. To use SSet, simply include `sset.h`. Including `khe_solvers.h` does not automatically include `sset.h` as well.

File `sset.h` contains this definition of type `SSET`, representing one shiftable set:

```
typedef struct sset_rec { ... } SSET;
```

We've omitted the contents, but they include an array of items, the shift, and a few other things. The items are stored as themselves (as integers) in increasing order. SSets are sets, not multisets—there are no duplicates among the items.

Type `SSET` is a struct, not a pointer to a struct, because `SSET` is intended as an aid to implementing other modules, and values of type `SSET` are expected to be private fields of these other modules' structs. Structs are better than pointers to structs in these cases, because they save memory and avoid one level of indirection.

[1]The first level of the trie is indexed by the first element, the second level is indexed by the second element minus the first element, the third level by the third element minus the second, and so on. This helps to keep the trie arrays short. KHE sets may contain arbitrary integers, but KHE sets used as table indexes may only contain non-negative integers.

To pass an `SSET` as a parameter it is always best to pass its address, not the struct itself. The following functions appear to violate this rule, but they are in fact macros which insert the address-of operators for you. For example, the function given as

```
void SSetUnion(SSET to_ss, SSET from_ss);
```

below is really macro

```
#define SSetUnion(to_ss, from_ss) SSetImplUnion(&(to_ss), &(from_ss))
```

and thus passes its SSet parameters by reference.

Each SSet object contains a `finalized` flag which, when set, prohibits further changes to the value of the set (although the set can be re-initialized). This has been included to prevent the user from changing a set after slicing it, since that could change and indeed invalidate its slices.

Each SSet object also contains a `slice` flag which is `true` when the SSet is a shifted version, and perhaps a slice, of another set. This is used only when freeing an SSet: when an SSet is freed, the memory used to hold its items is freed only when the `slice` flag is `false`, avoiding freeing that memory multiple times. Of course, freeing an SSet invalidates all its shifted and sliced versions. In the KHE application they are held nearby and freed at the same time.

To initialize (or re-initialize) an SSet to an unfinalized empty set with shift 0, call

```
void SSetInit(SSET ss, HA_ARENA a);
```

Memory for the SSet will be taken from arena `a`. As usual with arenas, there is no operation to free this memory; instead, it will be freed when the arena is deleted. To change the value of an unfinalized SSet, use these functions:

```
void SSetClear(SSET ss);
void SSetInsert(SSET ss, int item);
void SSetDelete(SSET ss, int item);
void SSetUnion(SSET to_ss, SSET from_ss);
void SSetIntersect(SSET to_ss, SSET from_ss);
void SSetDifference(SSET to_ss, SSET from_ss);
```

These clear `ss` back to the empty set, insert `item` (or do nothing if `item` is already present), delete `item` (or do nothing if `item` is not present), and change the value of `to_ss` to its union, intersection, or difference with `from_ss`. When `to_ss` and `from_ss` are the exact same object, `SSetUnion` and `SSetIntersect` do nothing, which is the mathematically correct thing to do, but `SSetDifference` aborts, as a sanity measure.

Once these changes are complete, a call to

```
void SSetFinalize(SSET ss);
```

finalizes `ss`. This causes later attempts to change it to abort with an error message. Function

```
bool SSetIsFinalized(SSET ss);
```

returns `true` when `ss` has been finalized.

Function

```
void SSetInitShifted(SSET to_ss, SSET from_ss, int shift);
```

initializes (or re-initializes) to_ss to a finalized SSet holding the items of from_ss with shift added to each item. The shift is stored separately, allowing to_ss to share from_ss's item memory. Here from_ss must be finalized. Function

```
void SSetInitShiftedAndSliced(SSET to_ss, SSET from_ss, int shift,
  int lower_lim, int upper_lim);
```

first carries out the same shift, but then it trims to_ss at each end, removing all items with value less than lower_lim, and all items with value larger than upper_lim. Again, from_ss must be finalized and the item memory is shared with from_ss.

The following functions perform queries on SSets without changing their values:

```
int SSetCount(SSET ss);
int SSetGet(SSET ss, int i);
int SSetMin(SSET ss);
int SSetMax(SSET ss);
```

They return the cardinality of ss; its ith element, counting from 0 as usual, with the items stored and thus returned in increasing order; its first (smallest) element; and its last (largest) element. The last three functions are tiny macros and do not check that the calls are valid.

The following more complex queries are also offered:

```
bool SSetEmpty(SSET ss);
bool SSetEqual(SSET ss1, SSET ss2);
bool SSetSubset(SSET ss1, SSET ss2);
bool SSetDisjoint(SSET ss1, SSET ss2);
bool SSetContains(SSET ss, int item);
```

These return true when ss is empty, when ss1 is equal to, a subset of, or disjoint from ss2, and when ss contains item.

The current shift is returned by

```
int SSetShift(SSET ss);
```

However, calling this is unlikely to be a good idea, because it goes behind the abstraction.

For convenience, iterator macros are defined which expand to for loops:

```
SSetForEach(SSET ss, int *item, int *i)
SSetForEachReverse(SSET ss, int *item, int *i)
```

These iterate over the items of ss, setting *item and *i to each item and its index in turn. For example, to sum the elements one would write

```
int total, item, i;
total = 0;
SSetForEach(ss, &item, &i)
  total += item;
```

`SSetForEachReverse` is like `SSetForEach` except that it iterates in reverse order.

Function

```
char *SSetShow(SSET ss);
```

returns a string stored in static memory showing the value of `ss`, for example `"{0, 3-5}"`. When the set is finalized an asterisk is appended to the string. A long result is neatly elided to fit into the 200-character buffer set aside to hold it. Actually there are four such buffers, and `SSetShow` may be called up to four times before one of its previous results is overwritten.

Function

```
void SSetTest(FILE *fp);
```

carries out a fixed set of tests on this module, writing its results to `fp`.

The SSet module also offers tables indexed by SSets, as follows:

```
SSET_TABLE SSetTableMake(HA_ARENA a);
void SSetTableInsert(SSET_TABLE st, SSET ss, void *val);
bool SSetTableRetrieve(SSET_TABLE st, SSET ss, void **val);
void SSetTableDebug(SSET_TABLE st, int indent, FILE *fp);
void SSetTableTest(FILE *fp);
```

`SSetTableMake` returns a new, empty table. `SSetTableFree` frees the memory used by `st`. `SSetTableInsert` inserts an entry with key `ss` (actually `&ss`, and there is no copying of the SSet) and value `val` into `st`. It aborts with an error message if an entry with an equal key is already present. It would be disastrous to change `ss` after it has been inserted into a table, but `SSetTableInsert` does not actually require `ss` to be finalized. `SSetTableRetrieve` retrieves the entry with key `ss` from `st`, setting `*val` to its value and returning `true` on success, and setting `*val` to `NULL` and returning `false` on failure. Finally, `SSetTableDebug` produces a debug print of `st` onto `fp` with the given indent, and `SSetTableTest` tests the table code, with output to `fp`.

The table is implemented by a trie structure; each item is used to index an extensible array. Actually, for items after the first, the difference between the item and the previous item (always non-negative because items are held in increasing order) is used. Sets whose items are large integers should not be stored in these tables, because they will lead to excessively long arrays.

## A.4. Priority queues

When a solver needs to visit things in priority order, it is easiest to just put them in an array and sort them. Occasionally, however, their priorities change as solving proceeds, and then, since resorting after every change is not efficient, a priority queue is needed.

KHE comes with a C priority queue module called PriQueue, consisting of header file `khe_priqueue.h` and implementation file `khe_priqueue.c`. These are stored and compiled with KHE, but can also be used separately. To use PriQueue, simply include `khe_priqueue.h`. Including `khe.h` does not automatically include `khe_priqueue.h` as well. The implementation uses a Floyd-Williams heap with back indexes. Each operation takes $O(\log(n))$ time at most.

File `khe_priqueue.h` begins with these type definitions:

```
typedef struct khe_priqueue_rec *KHE_PRIQUEUE;

typedef int64_t (*KHE_PRIQUEUE_KEY_FN)(void *entry);
typedef int (*KHE_PRIQUEUE_INDEX_GET_FN)(void *entry);
typedef void (*KHE_PRIQUEUE_INDEX_SET_FN)(void *entry, int index);
```

The first defines the type of a PriQueue as a pointer to a private record in the usual way. The others define the types of callback functions stored within the PriQueue and called by it.

An *entry* is one element of a priority queue. PriQueue is generic: its entries are represented by void pointers and may have any type consistent with that. Each entry has a *key*, which is its priority in the priority queue, and an *index*, which is used internally by PriQueue to point to its position in the priority queue. A typical entry type would look like this:

```
typedef struct my_entry_rec {
  int64_t       key;                    /* PriQueue key */
  int           index;                  /* PriQueue index */
  ...
} *MY_ENTRY;
```

where `...` stands for other fields. PriQueue needs to retrieve the key, and to retrieve and set the index, which is what the three callback functions are for. Here they are for type `MY_ENTRY`:

```
int64_t MyEntryKey(void *entry)
{
  return ((MY_ENTRY) entry)->key;
}

int MyEntryIndex(void *entry)
{
  return ((MY_ENTRY) entry)->index;
}

void MyEntrySetIndex(void *entry, int index)
{
  ((MY_ENTRY) entry)->index = index;
}
```

PriQueue sets the value of an entry's index field to a positive integer during an insertion, and to zero during a deletion. Accordingly, the user should initialize it to zero, and then it can be used to determine whether the entry is currently in a priority queue or not.

To create a new PriQueue, call

```
KHE_PRIQUEUE KhePriQueueMake(KHE_PRIQUEUE_KEY_FN key,
  KHE_PRIQUEUE_INDEX_GET_FN index_get,
  KHE_PRIQUEUE_INDEX_SET_FN index_set, HA_ARENA a);
```

For the example above, the call would be

```
KhePriQueueMake(&MyEntryKey, &MyEntryIndex, &MyEntrySetIndex, a);
```

Initially the queue is empty. There is no operation to delete a priority queue; instead, it is deleted when arena `a` is deleted. To test whether a priority queue is empty or not, call

```
bool KhePriQueueEmpty(KHE_PRIQUEUE p);
```

To insert an entry, call

```
void KhePriQueueInsert(KHE_PRIQUEUE p, void *entry);
```

making sure that the entry's key is defined beforehand; the index need not be, since it will be set by PriQueue. Functions

```
void *KhePriQueueFindMin(KHE_PRIQUEUE p);
void *KhePriQueueDeleteMin(KHE_PRIQUEUE p);
```

return an entry with minimum key, assuming that `p` is not empty, and `KhePriQueueDeleteMin` removes the entry from the queue at the same time. Function

```
void KhePriQueueDeleteEntry(KHE_PRIQUEUE p, void *entry);
```

deletes `entry` from `p`; it must lie in `p`. There is also

```
void KhePriQueueClear(KHE_PRIQUEUE p);
```

which calls `KhePriQueueDeleteEntry` repeatedly until `p` is empty, choosing at each step the entry that is easiest to delete (the last one in the Floyd-Williams heap).

To update the priority of an entry, first change its key and then call

```
void KhePriQueueNotifyKeyChange(KHE_PRIQUEUE p, void *entry);
```

to inform `p` that it has changed. This will change `entry`'s order in the queue, moving it forwards or backwards as required. Finally,

```
void KhePriQueueTest(FILE *fp);
```

tests the module and prints its results onto file `fp`.

## A.5. XML handling with KML

KML is a C module for reading and writing XML. It consists of a header file called `kml.h`, and implementation files called `kml.c` and `kml_read.c`. These are stored and compiled with the KHE platform, and `khe_platform.h` includes `kml.h`. They can also be abstracted from it and used separately, although they do use the `Ha` memory module (Appendix A.1).

KHE uses KML to read and write XML. The KHE user encounters KML in exactly one place: when reading an archive, an object of type `KML_ERROR` is returned if there is a problem.

### A.5.1. Representing XML in memory

Type `KML_ELT` represents one node in an XML tree structure, including its label, attributes, and children. The operations for querying a `KML_ELT` object are

```
int KmlLineNum(KML_ELT elt);
int KmlColNum(KML_ELT elt);
char *KmlLabel(KML_ELT elt);
KML_ELT KmlParent(KML_ELT elt);
char *KmlText(KML_ELT elt);
```

`KmlLineNum` and `KmlColNum` return a line number and column number stored in the element, presumably recording its position in some input file somewhere. `KmlLabel` returns the label of the element, and `KmlParent` returns its parent element in the tree structure, or `NULL` if none. `KmlText` returns the text content of `elt`, or `NULL` if none.

For querying the attributes of `elt` the operations are

```
int KmlAttributeCount(KML_ELT elt);
char *KmlAttributeName(KML_ELT elt, int index);
char *KmlAttributeValue(KML_ELT elt, int index);
bool KmlContainsAttributePos(KML_ELT elt, char *name, int *index);
bool KmlContainsAttribute(KML_ELT elt, char *name, char **value);
```

`KmlAttributeCount` returns the number of `elt`'s attributes, and `KmlAttributeName` and `KmlAttributeValue` return its `index`'th attribute's name and value. The first attribute has index 0. Negative indexes are allowed: `-1` means the last attribute, `-2` the second last, and so on. `KmlContainsAttributePos` returns `true` if `elt` contains an attribute with the given name, setting `*index` to its index if so; otherwise it returns `false` and sets `*index` to `-1`. `KmlContainsAttribute` has the same return value, but it sets `*value` to the attribute's value if found, and to `NULL` otherwise.

For querying the children of `elt` the operations are

```
int KmlChildCount(KML_ELT elt);
KML_ELT KmlChild(KML_ELT elt, int index);
bool KmlContainsChildPos(KML_ELT elt, char *label, int *index);
bool KmlContainsChild(KML_ELT elt, char *label, KML_ELT *child_elt);
```

`KmlChildCount` returns the number of children, and `KmlChild` returns the `index`'th child, again counting from 0 with negative indices allowed. `KmlContainsChildPos` returns `true` if `elt` contains a child with the given label, setting `*index` to the index of the first such child if so; otherwise it returns `false` and sets `*index` to `-1`. `KmlContainsChild` has the same return value, but it sets `*child_elt` to the first such child if found, and to `NULL` otherwise.

There are operations for constructing `KML_ELT` objects directly:

```
KML_ELT KmlMakeElt(int line_num, int col_num, char *label, HA_ARENA a);
void KmlAddAttribute(KML_ELT elt, char *name, char *value);
void KmlAddChild(KML_ELT elt, KML_ELT child);
void KmlDeleteChild(KML_ELT elt, KML_ELT child);
void KmlAddText(KML_ELT elt, char *text);
void KmlAddFmtText(KML_ELT elt, char *fmt, ...);
```

`KmlMakeElt` creates a new element with the given line number, column number, and label, using memory from arena `a`; `KmlAddAttribute` adds an attribute; `KmlAddChild` adds a child;

KmlDeleteChild deletes a child; and KmlAddText and KmlAddFmtText add text, either as given or formatted using sprintf (with no risk of overflow). They may be called repeatedly on one elt, in which case the successive texts are concatenated. All these functions store copies, kept in arena a, of the strings they are passed, not the original strings.

As usual throughout KHE, there is no operation for freeing the memory used by an element. Instead, it is freed when arena a is deleted. Typically, a whole tree is built in one arena, so that it can be freed very efficiently by deleting the arena.

It is not safe to retrieve a string from an element, delete the enclosing arena, and then attempt to use the string. Such strings must be copied into a longer-lived arena. KHE's operations all do this, so there is no danger when KHE converts elements into archives, instances, etc.

### A.5.2. Error handling and format checking

KML does not print any error messages; instead it reports an error by returning an object of type KML_ERROR, containing the line number and column number of the point of error, plus a message explaining what the problem was:

```
int KmlErrorLineNum(KML_ERROR ke);
int KmlErrorColNum(KML_ERROR ke);
char *KmlErrorString(KML_ERROR ke);
```

These objects can form the basis of error messages printed by the user.

KML's operations for reading a file check only for well-formedness, not for conformance to a legal document type definition, nor for high-level semantic constraints. During the conversion from KML_ELT to the user's own data structure, other errors may be uncovered, and it is convenient to be able to report those as objects of type KML_ERROR also. Accordingly, operation

```
KML_ERROR KmlErrorMake(HA_ARENA a, int line_num, int col_num,
  char *fmt, ...);
```

is provided. It creates a new object of type KML_ERROR in arena a, initializes it with the given line number, column number, and formatted text (as for printf), and returns it. There is also

```
KML_ERROR KmlVErrorMake(HA_ARENA a, int line_num, int col_num,
  char *fmt, va_list ap);
```

which is to KmlErrorMake what vprintf is to printf, and

```
bool KmlError(KML_ERROR *ke, HA_ARENA a, int line_num, int col_num,
  char *fmt, ...);
```

which is like KmlErrorMake except that it sets *ke to the object it makes, and always returns false. This is convenient for uses such as

```
if( bad_thing_discovered )
  return KmlError(ke, a, line_num, col_num, "bad %s thing", str);
```

which bails out of a function that returns a boolean indicating whether all is well. There is also

```
KML_ERROR KmlErrorCopy(KML_ERROR ke, HA_ARENA a);
```

which returns a fresh copy of `ke` in arena `a`.

To check whether a `KML_ELT` object conforms to a document type definition, call:

```
bool KmlCheck(KML_ELT elt, char *fmt, KML_ERROR *ke);
```

If `elt` conforms to the definition expressed by `fmt`, then `true` is returned; otherwise, `false` is returned and `*ke` is set to an object recording the nature of the error, including a line and column number taken from either `elt` itself or one of its children, as appropriate.

Parameter `fmt` describes the attributes and children of `elt`—not the label of `elt`, which will have already been checked by the time `elt` is examined, nor the children's children, which may be checked by the user during a recursive traversal of `elt`'s children. For example,

```
"+Reference : #Value"
```

says that `elt` has an optional attribute whose name is `Reference`, and exactly one child whose label is `Value` and whose body must contain text denoting an integer (no children). The part before the colon specifies attributes, and the part after it (if there is a colon at all) specifies children. An initial + means optional, and an initial `*` means zero or more; neither means exactly one. After that, an initial `$` means text (no children), and an initial # means text representing an integer (again, no children); neither means that there may be children. Here is a longer example:

```
"Reference : +#Duration +Time +Resources"
```

The element must have exactly one attribute, `Reference`. It has up to three children, an optional integer `Duration`, followed by an optional `Time`, and finally an optional `Resources`. As mentioned, the structure of the children may be checked by subsequent calls to `KmlCheck`.

## A.5.3. Reading XML files

The simple way to read an XML file is to call

```
bool KmlReadFile(FILE *fp, FILE *echo_fp, KML_ELT *res, KML_ERROR *ke,
  HA_ARENA a);
```

`KmlReadFile` reads `fp`, which must be open for reading UTF-8. If `echo_fp != NULL`, it writes everything it reads to `echo_fp`, as a debugging aid. If there were no problems with the read, `*res` is set to a new `KML_ELT` object representing the XML that was found, and `true` is returned. The operations of Appendix A.5.1 may be used to traverse `*res`. Otherwise, `*ke` is set to an error object (Appendix A.5.2) describing the first error (reading stops there), and `false` is returned.

`KmlReadFile` skips over any prolog, then reads exactly one element (including its descendants) from `fp`, from the first tag in `fp` to the matching end tag, then skips over any epilog (trailing comments, etc.) which involves skipping white space as well to see if epilog elements are there. After `KmlReadFile` ends, `fp` remains open, leaving it to the caller to either close it or keep reading from it. At that point, either end of file will have been reached, or else the next character read will be the first character that could not be part of the epilog, pushed back using `ungetc`.

All memory consumed by `KmlReadFile`, including memory for `*res` and its descendants, and for `*ke` if needed, comes from arena `a`. After everything useful has been extracted from `*res` and its descendants, `a` may be deleted or recycled as usual.

XML files can be large, and it may be better to read and process them one piece, or *segment*, at a time. A segment is defined by an element called its *root*. It consists of its root plus its root's descendants, excluding elements which are the roots of other segments, and their descendants.

There is a *root segment* whose root element is the overall root. So every element lies in one segment, the one defined by its nearest ancestor (possibly itself) that is the root of a segment.

Reading in segments requries several steps. The first step is to call

```
KML_READER KmlReaderMake(void *impl, HA_ARENA_SET as, HA_ARENA a);
```

This creates a `KML_READER` object in arena `a`. The `impl` parameter is a pointer back to the user's data structures, and `as` is an arena set which is the source of any arenas, additional to `a`, that may be needed, of which more later. Functions

```
void *KmlReaderImpl(KML_READER kr);
HA_ARENA_SET KmlReaderArenaSet(KML_READER kr);
HA_ARENA KmlReaderArena(KML_READER kr);
```

return the three attributes of `kr`.

While the file is being read (while function `KmlReaderReadFileSegmented` below is running), callbacks are made to user code, which might detect a semantic error which should abort the whole read. For this there is

```
void KmlReaderFail(KML_READER kr, KML_ERROR ke);
```

which uses a C long jump to return early from `KmlReaderReadFileSegmented` with error `ke`.

There is no operation to reclaim the memory consumed by a `KML_READER` object. As usual, it is freed when its arena is deleted.

The second step is to make matching pairs of calls to these functions:

```
void KmlReaderDeclareSegmentBegin(KML_READER kr, char *path_name,
  KML_SEGMENT_FN segment_begin_fn);
void KmlReaderDeclareSegmentEnd(KML_READER kr,
  KML_SEGMENT_FN segment_end_fn);
```

These give the path names of the elements which are to be the roots of segments. For example, suppose that the file structure is

```
HighSchoolTimetableArchive
    +Instances
        *Instance
    +SolutionGroups
        *SolutionGroup
            *Solution
```

where + means optional, `*` means zero or more, and indenting indicates nesting, and suppose that each `Instance`, `SolutionGroup`, and `Solution` is to be one segment. Then the calls are

```
KmlReaderDeclareSegmentBegin(kr, "HighSchoolTimetableArchive", &fn1);
  KmlReaderDeclareSegmentBegin(kr, "Instances/Instance", &fn2);
  KmlReaderDeclareSegmentEnd(kr, &fn3);
  KmlReaderDeclareSegmentBegin(kr, "SolutionGroups/SolutionGroup", &fn4);
    KmlReaderDeclareSegmentBegin(kr, "Solution", &fn5);
    KmlReaderDeclareSegmentEnd(kr, &fn6);
  KmlReaderDeclareSegmentEnd(kr, &fn7);
KmlReaderDeclareSegmentEnd(kr, &fn8);
```

using indenting to show the structure. They mimic the structure of the file. Each path name is a sequence of one or more element names separated by slashes, and is relative to the enclosing segment, except at the root. As a special case, an element name may be `"*"`, and then it will match with any name.

In cases like those for `Instance` and `Solution` above, where there are no inner segments, `segment_begin_fn` is called immediately before `segment_end_fn`, as will be explained below. In that case two callbacks are not needed, and so KML offers

```
void KmlReaderDeclareSegment(KML_READER kr, char *path_name,
  KML_SEGMENT_FN segment_fn);
```

to replace `KmlReaderDeclareSegmentBegin` and `KmlReaderDeclareSegmentEnd`:

```
KmlReaderDeclareSegmentBegin(kr, "HighSchoolTimetableArchive", &fn1);
  KmlReaderDeclareSegment(kr, "Instances/Instance", &fn2);
  KmlReaderDeclareSegmentBegin(kr, "SolutionGroups/SolutionGroup", &fn3);
    KmlReaderDeclareSegment(kr, "Solution", &fn4);
  KmlReaderDeclareSegmentEnd(kr, &fn5);
KmlReaderDeclareSegmentEnd(kr, &fn6);
```

There is no substantial difference.

A path name can also be a sequence of path names separated by colons, like this:

```
"HighSchoolTimetableArchive:EmployeeScheduleArchive"
```

Then elements indicated by all paths are the roots of segments, with the same inner segments.

The third step is to actually read the file, by calling

```
bool KmlReaderReadFileSegmented(KML_READER kr, FILE *fp, FILE *echo_fp,
  KML_ERROR *ke);
```

`KmlReaderReadFileSegmented` is similar to `KmlReadFile`, except that no `KML_ELT` is returned. It can be called multiple times on one `KML_READER`, although not in parallel.

As `KmlReaderReadFileSegmented` reads the file, it calls callback functions `segment_begin_fn` and `segment_end_fn` at the beginning and end of each segment. In the syntax that the user would use to declare these functions, they are

```
void segment_begin_fn(KML_SEGMENT ks)
{
   ... process ks ...
}
```

This allows the user access to each segment, at the start of the segment and again at the end.

The call on `segment_begin_fn` does not occur at the moment its element begins in the input file. That would not be useful, because none of the element's content is available then. Instead, the callback is delayed until the first inner segment is about to begin, or if there are no inner segments, until the segment is about to end. At that point, the segment's root contains data that can be processed into an initial value for the corresponding object on the user side.

The call on `segment_end_fn` occurs as the segment's root element is ending, and can be used to finalize the corresponding user data structure. Either or both of `segment_begin_fn` and `segment_end_fn` may be `NULL`, and then the corresponding callback is omitted.

The final step is to write the callback functions. Within each function, the user has access to segment `ks`, to which the following functions may be applied:

```
KML_ELT KmlSegmentRoot(KML_SEGMENT ks);
KML_READER KmlSegmentReader(KML_SEGMENT ks);
HA_ARENA KmlSegmentArena(KML_SEGMENT ks);
```

`KmlSegmentRoot` returns the root of the segment. From there one can explore the children, their children, and so on, insofar as they exist at the moment that the callback occurs. One can never reach the elements of any inner segments in this way, not even from the callback at the end of the segment, because such elements are not made children of their (logical) parent elements in the usual way. The same fact looked at from the other side means that the root element has no parent, so there is no way to reach elements in the enclosing segment.

`KmlSegmentReader` returns the `KML_READER` object passed to the enclosing call to `KmlReaderReadFileSegmented`. This is useful for reaching user data structures via `KmlReaderImpl`, ending the read early with failure via `KmlReaderFail`, and so on.

`KmlSegmentArena` returns the segment's arena. This holds the segment object itself, its root element, and the root element's decendants. Care is needed not to create objects, for example error objects, in a segment's arena that are intended to outlast the segment. An alternative arena that will outlast the segment is `KmlReaderArena(KmlSegmentReader(ks))`.

The use of arenas in segmented file reading is somewhat complex, in that the root segment is a special case. Its arena is the arena passed to `KmlReaderMake`. That arena holds both the reader object and the root segment, and is not deleted by KML. The user should delete or recycle it after the whole read is over. Each of the other segments has its own arena, taken from the arena set `as` passed to `KmlReaderMake` (or created, as usual, if `as` is empty). This arena is deleted, or rather recycled through `as`, immediately after the segment's `segment_end_fn` returns. So the user must ensure that everything needed on the user side is extracted from the segment by that time. It is almost certainly a disastrous error to store the segment passed in the callback function, or any of its elements, in user-side data structures.

### A.5.4. Writing XML files

Writing an XML file begins with the creation of a `KML_FILE` object, by calling

```
    KML_FILE KmlMakeFile(FILE *fp, int initial_indent, int indent_step);
```

Pointer type `KML_FILE`, defined in `kml.h`, represents an XML file open for writing (never reading). It holds a file pointer and a few attributes describing the state of the write, including a current indent, used to produce neatly indented XML. File `fp` must be open for writing UTF-8 characters; `initial_indent` is the initial indent, typically 0, and `indent_step` is the number of spaces to indent at each level, typically 2 or 4.

When reading an XML file using KML it is necessary to first read the file into a `KML_ELT` object, and then build the user data structure that is really wanted, while traversing the `KML_ELT` object. The reverse procedure may be used for writing, by calling

```
    void KmlWrite(KML_ELT elt, KML_FILE kf);
```

`KmlWrite` writes `elt` and its attributes and children recursively to `kf`. But it is also possible to write directly to a file while traversing the user's data structure, without using `KML_ELT` objects. To do this, the operations are

```
    void KmlBegin(KML_FILE kf, char *label);
    void KmlAttribute(KML_FILE kf, char *name, char *value);
    void KmlPlainText(KML_FILE kf, char *text);
    void KmlFmtText(KML_FILE kf, char *fmt, ...);
    void KmlEnd(KML_FILE kf, char *label);
```

`KmlBegin` begins an object with the given label, and `KmlEnd` ends it. KML does not check that the labels match, even though they must. Immediately after calling `KmlBegin`, any number of calls to `KmlAttribute` are allowed; each adds one attribute, with the given name and value, to the object just begun. After that, `KmlPlainText` may be called to add some text as the body of the object, or `KmlFmtText` to add some formatted text as the body (where `fmt` and the following parameters are suitable for passing on to `fprintf`). `KmlPlainText` prints the characters `&<>'"` in their escape sequence forms (`&amp;` and so on); `KmlFmtText` does not, so it is best limited to tasks that cannot generate such characters (printing numbers, etc.). Alternatively, any number of nested calls to `KmlBegin` ... `KmlEnd` may precede the matching `KmlEnd`, to add children.

For convenience, three operations are offered which write an entire element in one call:

```
    void KmlEltAttribute(KML_FILE kf, char *label, char *name, char *value);
    void KmlEltPlainText(KML_FILE kf, char *label, char *text);
    void KmlEltFmtText(KML_FILE kf, char *label, char *fmt, ...);
```

These are simple combinations of the functions above, only writing on one line (except newlines in text). `KmlEltAttribute` writes an object with the given label and attribute, but no body. `KmlEltPlainText` and `KmlEltFmtText` write an object with the given label, no attributes, and a plain or formatted text body. A few other such functions are available, for which see `kml.h`.

# Appendix B. Implementation Notes

This chapter documents aspects of the implementation of KHE. It is included mainly for the author's own reference; it is not needed for using KHE.

## B.1. Source file organization

The KHE platform is organized in object-oriented style, with one C source file for each major type. A type's internals are visible only within its file; all access to them is via functions. Headers for some of these functions appear in `khe_platform.h`, making them available to the end user. Headers for others appear in `khe_interns.h`, making them available only to the platform.

Although this section applies to all source files, it is motivated by the problems of organizing the source files of types defining parts of solutions. Some of these are quite large. For example, file `khe_meet.c`, which holds the internals of type `KHE_MEET`, is about 5000 lines long.

There is a canonical order for the types representing parts of solutions: `KHE_SOLN`, `KHE_MEET`, `KHE_MEET_BOUND`, `KHE_TASK`, `KHE_TASK_BOUND`, `KHE_MARK`, `KHE_PATH`, `KHE_NODE`, `KHE_LAYER`, `KHE_ZONE`, `KHE_TASKING`. The intention of defining this order is that these types should be handled in this order whenever appropriate—in this Guide for example.

Source files are organized internally by dividing them into *submodules*, which are segments of the files separated by comments. Each submodule handles one aspect of the type. Here is a generic list of the submodules appearing in any one file, in their order of appearance:

*Type declaration*
*Simple attributes (back pointers, visit numbers, etc.)*
*Creation and deletion*
*Relations with objects of the same type (copy, split, etc.)*
*Relations with objects of different types*
*File reading and writing*
*Debug*

Simple attributes are easily handled attributes that are not closely related to any following categories. They may appear in separate submodules, or be grouped into one submodule. Each relation is one submodule (counting opposite operations, such as split and merge, as part of one relation), except that a large relation may be broken into several submodules. Relations with different types appear in the canonical order defined above.

An attempt has been made to keep the submodules in the same order as their functions are presented in this Guide, except for debugging. Some submodules have no defined position according to this rule, because they are present only to support other submodules, and offer no functions to the end user. Those are placed where they seem to fit best.

## B.2. Relations between objects

This section explains how KHE maintains relations between objects. Not every relation is maintained as explained here, but it is the author's aim to achieve that in time.

The most common relation, by far, is the *one-to-many* relation, in which one object is related to any number of objects of the same or another type: one node contains any number of meets, one meet contains any number of tasks, one meet is assigned any number of meets, and so on.

Let `KHE_A` be the type of the entity that there is one of, and `KHE_B` be the type of the entity that there are many of. KHE implements the relation by placing one attribute, of type `ARRAY_KHE_B`, in `KHE_A`, holding the many `KHE_B` objects related to `KHE_A`, and two in `KHE_B`:

```
KHE_A   a;
int     a_index;
```

holding the one `KHE_A` object related to this object, and this object's index in that object's array. Any attributes of the relation, such as the offset attribute of the meet assignment relation, appear alongside these two. In the `KHE_A` class file, functions

```
void KheAAddB(KHE_A a, KHE_B b);
void KheADeleteB(KHE_A a, KHE_B b);
```

are defined which add and delete elements of the relation, as well as the usual `KheABCount` and `KheAB` functions which iterate over the array. In the `KHE_B` class file, functions

```
KHE_A KheBA(KHE_B b);
void KheBSetA(KHE_B b, KHE_A a);
int KheBAIndex(KHE_B b);
void KheBSetAIndex(KHE_B b, int a_index);
```

get and set the `a` and `a_index` attributes of `b`, supporting constant time deletions. Instead of searching for `b` in `a`'s array, `a_index` is used to find it directly. It is overwritten by the entity at the end of the array, whose index is then changed. This assumes that the order of the array's elements may be arbitrary, as is usually the case. The setter functions are private to the platform.

This plan allows a `KHE_B` object to be unrelated to any `KHE_A` object (just set its `a` attribute to `NULL`), but does not support *many-to-many* relations, where a `KHE_B` object may be related to any number of `KHE_A` objects. On the rare occasions when KHE needs this kind of relation, it adapts the familiar edge lists implementation of graphs: it defines a type `KHE_A_REL_B` representing one element of the many-to-many relation, and installs one one-to-many relation from `KHE_A` to `KHE_A_REL_B`, and another from `KHE_B` to `KHE_A_REL_B`. This gives `KHE_A_REL_B` attributes

```
KHE_A   a;
int     a_index;
KHE_B   b;
int     b_index;
```

and places it in arrays in both `entity_a` and `entity_b`. Now the operations for adding and deleting an element of the relation must add or delete two one-to-many relations, as well as creating or deleting one `KHE_A_REL_B` object, which is done using a free list to save time.

## B.3. Kernel operations

The promises made in connection with marks and paths, that all operations that change a solution can be undone (except changes to visit numbers), and that undoing a deletion recreates the object at its original address, have significant implications for the implementation.

The KHE platform has an inner layer called the *solution kernel*, or just the *kernel*, consisting of a set of private operations, called *kernel operations*, which change a solution. Each kernel operation has a name of the form `KheEntityKernelOp`, where `Entity` is the data type and `Op` is the operation. It is the kernel operations that are stored in paths. All operations (except operations on visit numbers) change the solution only by calling kernel operations, so if those are correctly done, undone, and redone, all operations will be correctly done, undone, and redone.

For the record, here is the complete list of kernel operations:

```
KheMeetKernelSetBack                  KheTaskKernelSetBack
KheMeetKernelAdd                      KheTaskKernelAdd
KheMeetKernelDelete                   KheTaskKernelDelete
KheMeetKernelSplit                    KheTaskKernelSplit
KheMeetKernelMerge                    KheTaskKernelMerge
KheMeetKernelMove                     KheTaskKernelMove
KheMeetKernelAssignFix                KheTaskKernelAssignFix
KheMeetKernelAssignUnFix              KheTaskKernelAssignUnFix
KheMeetKernelAddMeetBound             KheTaskKernelAddTaskBound
KheMeetKernelDeleteMeetBound          KheTaskKernelDeleteTaskBound
KheMeetKernelSetAutoDomain

KheMeetBoundKernelAdd                 KheTaskBoundKernelAdd
KheMeetBoundKernelDelete              KheTaskBoundKernelDelete
KheMeetBoundKernelAddTimeGroup
KheMeetBoundKernelDeleteTimeGroup     KheNodeKernelSetBack
                                      KheNodeKernelAdd
KheLayerKernelSetBack                 KheNodeKernelDelete
KheLayerKernelAdd                     KheNodeKernelAddParent
KheLayerKernelDelete                  KheNodeKernelDeleteParent
KheLayerKernelAddChildNode            KheNodeKernelSwapChildNodesAndLayers
KheLayerKernelDeleteChildNode         KheNodeKernelAddMeet
KheLayerKernelAddResource             KheNodeKernelDeleteMeet
KheLayerKernelDeleteResource
                                      KheZoneKernelSetBack
                                      KheZoneKernelAdd
                                      KheZoneKernelDelete
                                      KheZoneKernelAddMeetOffset
                                      KheZoneKernelDeleteMeetOffset
```

Each `KheEntityKernelOp` function has a companion `KheEntityKernelOpUndo` function. `KheEntityKernelOp` carries out its operation and adds itself to the solution's path, if present. `KheEntityKernelOpUndo` undoes what `KheEntityKernelOp` did, only without removing itself from the solution's path, since it is called by a function that has already done that.

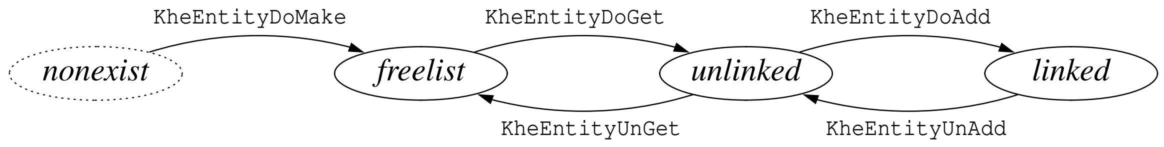
A redo must be identical to the original operation, because both can be inverted by calling `KheEntityKernelOpUndo` and removing one record from the solution path. So there are no `KheEntityKernelOpRedo` functions; `KheEntityKernelOp` functions are called instead.

Some operations come in opposing pairs (split and merge, fix and unfix, and so on), such that doing one is the same as undoing the other, except that a do or redo adds a record to the solution's path, whereas an undo does not. In these cases the implementation contains one private function called `KheEntityDoOp1` and another called `KheEntityDoOp2`, where `Op1` and `Op2` are opposing pairs. These functions carry out the two operations without touching the solution's path. Then `KheEntityKernelOp1`, `KheEntityKernelOp2`, `KheEntityKernelOp1Undo`, and `KheEntityKernelOp2Undo` are each implemented by one call on `KheEntityDoOp1` or `KheEntityDoOp2`, plus an addition to the solution's path if the operation is not `Undo`.

Operations that create and delete objects are awkward, as it turns out, so the rest of this section is devoted to them. The meet split and merge operations are particularly awkward, so we will start with the regular creation and deletion operations, generically named `KheEntityMake` and `KheEntityDelete`, and treat meet splitting and merging afterwards.

Solution objects are recycled through free lists held in the enclosing solution. When a new object is needed, it is taken from the free list, or from the solution's arena if the free list is empty. When an object is no longer needed, it is added to the free list. When the solution is deleted, and only then, the objects on the free list are deleted as part of the deletion of the arena. Free lists not only save time in handling the objects, they also save time in handling any extensible arrays within those objects: those arrays remain initialized while the object is on the free list.

An operation which obtains a new object from a memory allocator or free list cannot be a kernel operation, because then a redo would not re-create the object at its previous memory location. An operation which returns an object to a memory allocator or free list cannot be a kernel operation, because an undo would not re-create the object at its previous memory location. So only the part of `KheEntityMake` which initializes the object and links it into the solution is the kernel operation, and only the part of `KheEntityDelete` which unlinks the object from the solution is the kernel operation. This leads to this picture of the life cycle of a kernel object:



State *nonexist* means that the object does not exist; *freelist* means that it exists on a free list; *unlinked* means that it exists, not on a free list, not linked to the solution, but referenced from somewhere on some path; and *linked* means that it exists and is linked to the solution.

`KheEntityDoMake` obtains a fresh object from the memory allocator and initializes its private arrays. There is no corresponding `KheEntityUnMake` operation, because memory is freed only by deleting arenas, not directly.

`KheEntityDoGet` obtains a fresh object from the free list, or from `KheEntityDoMake` if the free list is empty. Either way, the object's arrays are initialized, although not necessarily empty. Objects returned by `KheEntityDoMake` do not actually enter the free list. `KheEntityUnGet` does the opposite, adding the object it is given to the free list.

`KheEntityDoAdd` initializes the unlinked object it is given, assuming that its private arrays are initialized, although not necessarily empty (it clears them), and links it into the solution. `KheEntityUnAdd` does the opposite, unlinking the object it is given from the solution.

The kernel operations `KheEntityKernelAdd` and `KheEntityKernelDelete` and their `Undo` companions are each implemented by one call to `KheEntityDoAdd` or `KheEntityUnAdd`, plus an addition to the solution path if the function is not an undo. `KheEntityKernelAdd` and `KheEntityKernelDelete` form an opposing pair, as defined above, except that `KheEntityKernelDelete` may include a call to `KheEntityUnGet` as explained below.

The public function that creates a kernel object, `KheEntityMake`, is `KheEntityDoGet` followed by `KheEntityKernelAdd`. The public function that deletes one, `KheEntityDelete`, begins with kernel operations that help to unlink the object (unassignments and so on), then ends with `KheEntityKernelDelete`.

An object can be referenced from the solution and from paths, and there is no simple rule saying when to call `KheEntityUnGet` to add it to the free list. To solve this problem, an integer reference count field is placed in each kernel object, counting the number of references to the object. Not all references are counted. References from paths at points where the object is added or deleted are counted. For example, in a path's record of a meet split or merge, the reference to the second meet is counted, but not the first. So reference counts increase when paths grow or are copied, and decrease when paths shrink or are deleted. Also, `KheEntityDoAdd` adds 1 to the count, and `KheEntityUnAdd` subtracts 1. This summarizes references from the solution generally in one unit of the count.

When the reference count falls to zero, `KheEntityUnGet` is called to return the object to the free list. This could happen during a call to `KheEntityUnAdd`, or when a path shrinks: during a call to `KhePathDelete`, or while undoing, which shrinks the solution's main path.

An *unlinked* object could have come from the free list, and so could contain no useful information. It would be a mistake for `KheEntityDoAdd` to assume that the object it is given has passed through `KheEntityUnAdd` and retains useful information from when it was previously linked. Instead, `KheEntityDoAdd` must initialize every field of the object it is given, assuming that its arrays are initialized, but not that they contain useful information.

An example of getting this wrong would be to try to preserve the list of tasks of a meet in its `tasks` array when it is unlinked, in a mistaken attempt to ensure that they remain available for when the meet is recreated. What really happens is that before deleting the meet, `KheMeetDelete` deletes its tasks, so records of those task deletions appear on the solution path just before the meet deletion. When an undo recreates the meet, it immediately goes on to recreate the tasks, without any need for their preservation in the dormant meet.

A meet split is similar to a creation of the second meet, and a meet merge is similar to a deletion of the second meet. The main new problem is that tasks need to be split and merged too. So separate kernel operations are defined for splitting the meet itself and for splitting one of its tasks, and conversely for merging two meets and for merging two of their tasks. The user operation for meet splitting does a kernel meet split followed by a sequence of kernel task splits, and the user operation for meet merging does the opposite.

The key advantage of doing it this way is that tasks are stored explicitly in paths, and their reference counters take account of this. So the usual method of handling the allocation and deallocation of entities generally, described above, applies without change to the tasks created and deleted by meet splitting and merging.

Meet bounds are related to meets in much the same way as tasks are. Once again, the kernel meet split operation does not make meet bounds for the split-off meet; instead, they are made by separate kernel meet bound creation operations, and thus will be undone before a meet split is undone. Task bounds are handled similarly.

Paths have negligible time cost compared with the operations they record; and their space cost is moderate, provided they are not used to record wandering methods like tabu search. Reference counting as implemented here also costs very little: in time, a few simple steps, only carried out when creating or deleting a kernel object, not each time the object is referenced; and in space, one integer per kernel object.

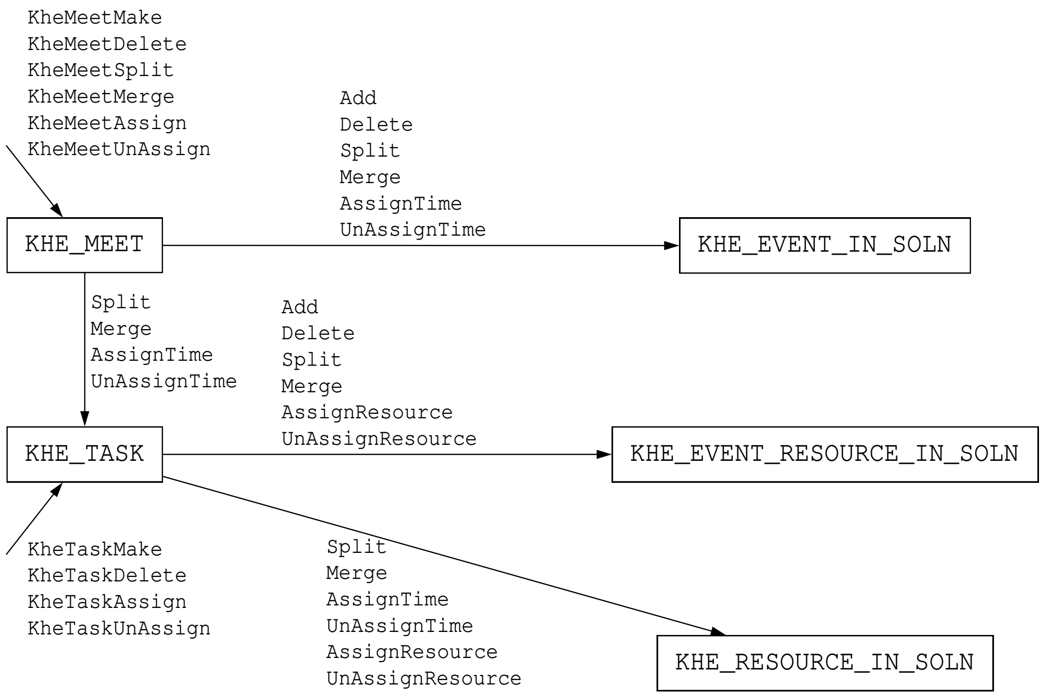
## B.4. Monitor updating

When the user executes an operation that changes the state of a solution, KHE works out the revised cost. For efficiency, this must be done incrementally. This section explains how it is done—but just for information: the functions defined here cannot be called by the user.

The monitors are linked into a network that allows state changing operations to flow naturally to where they need to go. Only attached monitors are linked in; detached ones are removed, so that no time is wasted on them. The full list of basic operations that affect cost is

```
KheMeetMake      KheMeetMerge      KheTaskMake
KheMeetDelete    KheMeetAssign     KheTaskDelete
KheMeetSplit     KheMeetUnAssign   KheTaskAssign
                                   KheTaskUnAssign
```

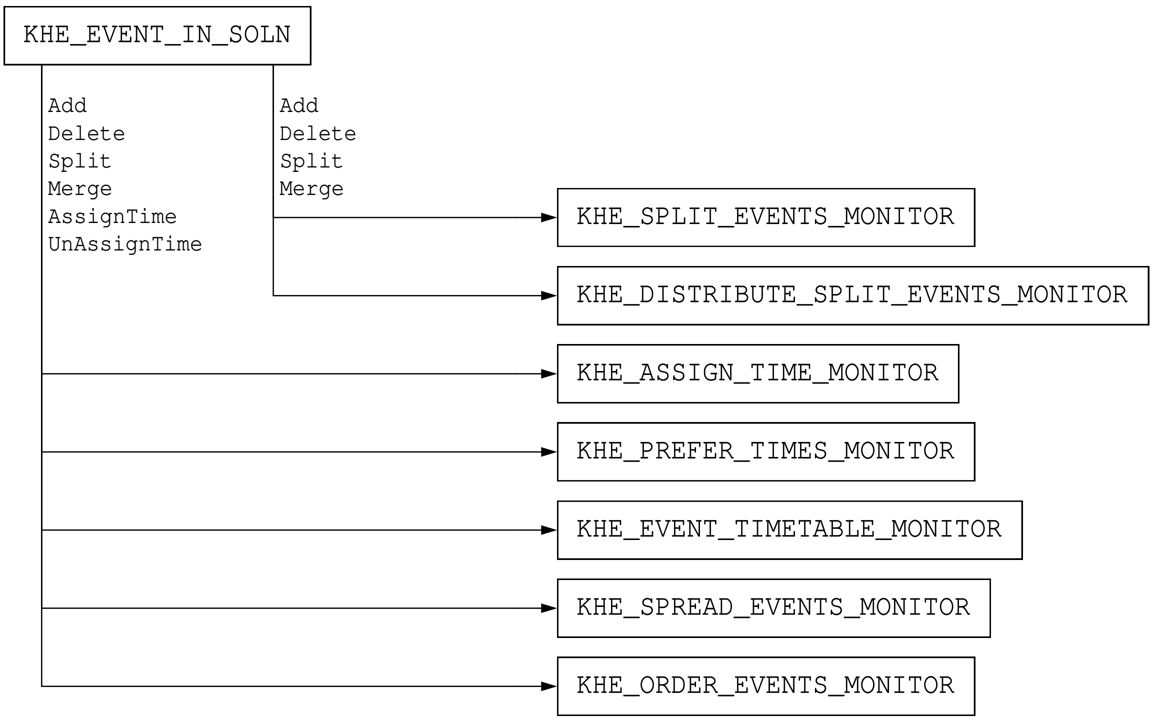
Six originate in `KHE_MEET` objects, four in `KHE_TASK` objects. From there their impulses flow to objects of three private types:



KHE_EVENT_IN_SOLN holds information about one event in a solution: the meets derived from it (where KheEventMeet gets its values from), a list of 'event resource in solution' objects, one for each of its event resources, and a list of monitors, possibly including a timetable (timetables are monitors). KHE_EVENT_RESOURCE_IN_SOLN holds information about one event resource in a solution: the tasks derived from it, and a list of monitors. KHE_RESOURCE_IN_SOLN holds information about one resource in a solution: the tasks it is currently assigned to, and a list of monitors, usually including a timetable.

The connections are fairly self-evident. For example, if KheMeetMake is called to make a meet derived from a given instance event, then that event's event in solution object needs to know this, and the Add operation (full name KheEventInSolnAddMeet) informs it. KheMeetAssign only generates an AssignTime call when the assignment links the meet, directly or indirectly, to a cycle meet, assigning a time to it. Event resource in solution objects are not told about time assignments and unassignments. Calls only pass from a task object task to a resource in solution object when task is assigned a resource.

The connections leading out of KHE_EVENT_IN_SOLN are as follows:



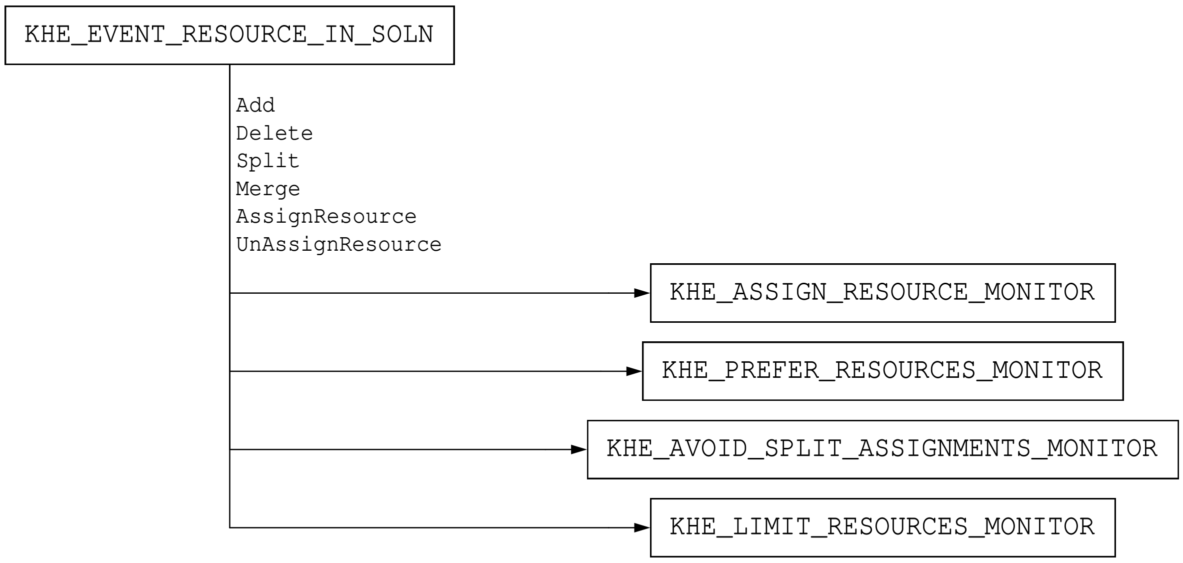
Split events and distribute split events monitors do not need to know about time assignment and unassignment. Based on the calls they receive, they keep track of meet durations and report cost accordingly. Assign time and prefer times monitors are even simpler; they report cost depending on whether the meets reported to them are assigned times or not.

Event timetables are used by link events constraints, which need to know the times when the event's meets are running, ignoring clashes, which is just what timetables offer.

A spread events monitor is connected to the event in solution objects corresponding to each of the events it is interested in. It keeps track of how many meets from those events collectively have starting times in each of its time groups, and calculates deviations accordingly. Spread events monitors are not attached to timetables because, although their monitoring is similar, there are significant differences: spread events monitor time groups come with upper and lower limits, making them not sharable in general, and the quantity of interest is the number of distinct meets that intersect each time group, not the number of busy times calculated by the time group monitors attached to timetables.

An order events monitor is connected to the two event in solution objects corresponding to the two events it is interested in. These keep track of the events' meets, including their number, and the monitor itself keeps track of the number of unassigned meets. So determining whether both events have at least one meet, and whether there are no unassigned meets, take constant time. If both conditions are satisfied, the monitor traverses both sets of meets to calculate the deviation and cost when a meet is added, deleted, or assigned a time. (In practice, events subject to order events constraints do not split, so this too takes constant time.) The other operations are faster: unassigning a time produces cost 0, and splitting and merging do not change the cost.

The connections leading out of `KHE_EVENT_RESOURCE_IN_SOLN` are

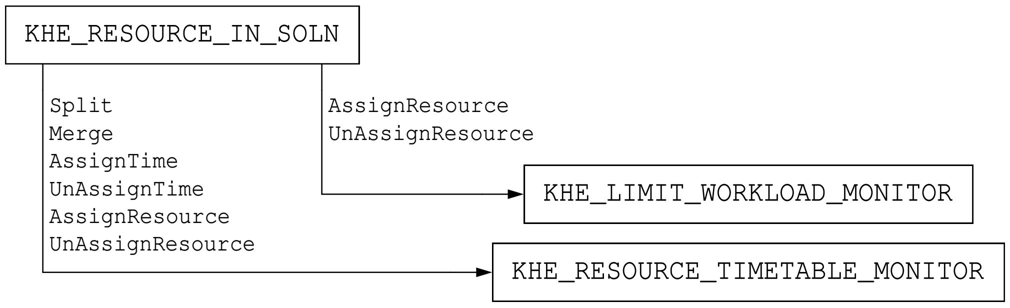

KHE_EVENT_RESOURCE_IN_SOLN
Add
Delete
Split
Merge
AssignResource
UnAssignResource
KHE_ASSIGN_RESOURCE_MONITOR
KHE_PREFER_RESOURCES_MONITOR
KHE_AVOID_SPLIT_ASSIGNMENTS_MONITOR
KHE_LIMIT_RESOURCES_MONITOR

None of these monitors cares about time assignments and unassignments. Assign resource monitors and prefer resources monitors are very simple, reporting cost depending on whether the tasks passed to them are assigned or not.

An avoid split assignments monitor is connected to one event resource in solution object for each event resource in its point of application. It keeps track of a multiset of resources, one element for each assignment to each task it is monitoring, and its cost depends on the number of distinct resources in that multiset.

A limit resources monitor is connected to one event resource in solution object for each event resource it monitors. It keeps count of the number of assignments of resources from its resource group.

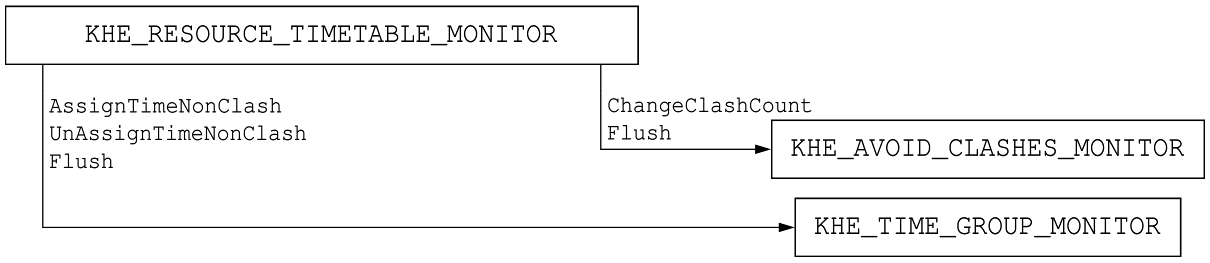
The connections leading out of `KHE_RESOURCE_IN_SOLN` are



Limit workload constraints do not need to know about time assignments, evidently, but they also do not need to know about splits and merges, since these do not change the total workload.

Calculating workloads is then very simple. Each meet receives a workload when it is created, and when a resource is assigned, the workload limit monitors attached to its resource in solution object are updated, and pass revised costs to the solution.

`KHE_RESOURCE_TIMETABLE_MONITOR` receives many kinds of calls. However, since it maintains a timetable containing tasks with assigned times, all these can be mapped to just two incoming operations, which we call `AddTaskAtTime` and `DeleteTaskAtTime`. For example, a split maps to one `DeleteTaskAtTime` and two `AddTaskAtTime` calls. The outgoing operations are

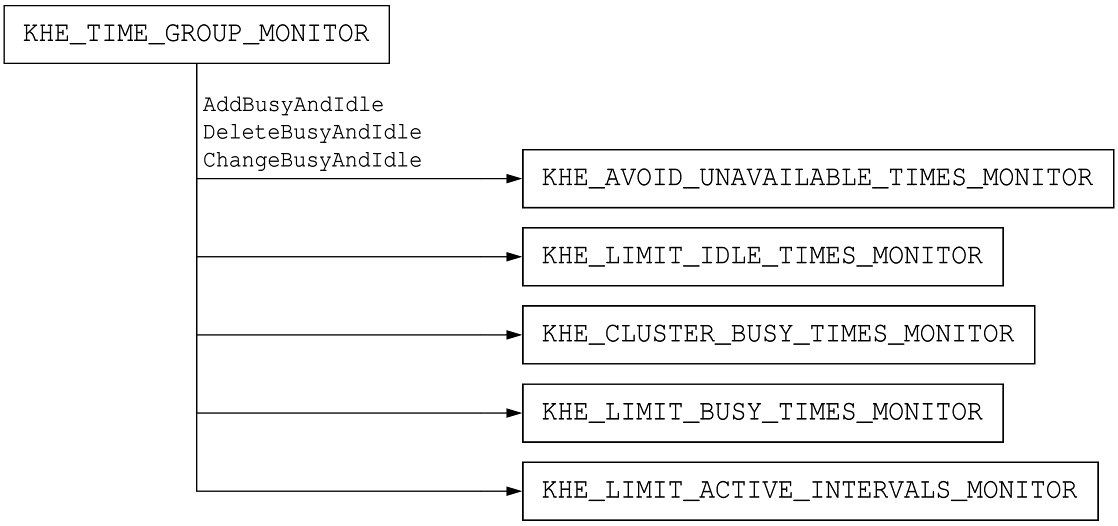


An avoid clashes monitor is notified whenever the number of meets at any one time increases to more than 1 or decreases from more than 1 (operation `ChangeClashCount` above). It uses these notifications to maintain its deviation. It updates the solution when a `Flush` is received from the timetable at the end of the operation.

The other monitors are attached to the timetable at each time they are interested in, and are notified when one of those times becomes busy (when its number of meets increases from 0 to 1) and when it becomes free (when its number of meets decreases from 1 to 0), by operations `AssignTimeNonClash` and `UnAssignTimeNonClash` above.

A time group monitor monitors one time group within one timetable. It is attached to its timetable at the times of its time group, so is notified when one of those times becomes busy or free. It keeps track of the number of busy and idle times in its time group. As an optimization, the number of idle times is calculated only when at least one limit idle times monitor is attached to the time group monitor; otherwise the number is taken to be 0.

Old and new values for the number of busy and idle times are stored, and when a flush is received they are propagated onwards via operation `ChangeBusyAndIdle`:


KHE_TIME_GROUP_MONITOR
AddBusyAndIdle
DeleteBusyAndIdle
ChangeBusyAndIdle
KHE_AVOID_UNAVAILABLE_TIMES_MONITOR
KHE_LIMIT_IDLE_TIMES_MONITOR
KHE_CLUSTER_BUSY_TIMES_MONITOR
KHE_LIMIT_BUSY_TIMES_MONITOR
KHE_LIMIT_ACTIVE_INTERVALS_MONITOR

When a monitor is attached, function `AddBusyAndIdle` is called instead, and when a monitor is detached, function `DeleteBusyAndIdle` is called instead.

An unavailable times monitor is connected to a time group monitor monitoring the unavailable times. It receives an updated number of busy times from `ChangeBusyAndIdle` and reports any change of cost to the solution.

A limit idle times monitor is connected to the time group monitors corresponding to the time groups of its constraint. It receives updated idle counts from each of them, and based on them it maintains its deviation.

A cluster busy times monitor is connected to the time group monitors corresponding to the time groups of its constraint. It is interested in whether the busy counts it receives from them change from zero to non-zero, or conversely.

A limit busy times monitor is connected to the time group monitors corresponding to the time groups of its constraint. It receives updated busy counts from each of them, and based on them it maintains its deviation.

A limit active intervals monitor is connected to the time group monitors corresponding to the time groups of its constraint. It is interested in whether the busy counts it receives from them change from zero to non-zero, or conversely. Using a data structure holding the current set of active intervals, it maintains its deviation by tracking changes in their lengths (Appendix B.6).

## B.5. Monitor attachment and unattachment

Monitor attachment and unattachment are constrained by some basic facts: they can occur at any time while a solver is running; unattachment is intended to save time, which means that an unattached monitor must be genuinely unlinked from the solution; and an unattached monitor has cost 0. Also, it is convenient to bring a monitor into existence in the unattached state and then attach it, because there is a lot of shared code between creation and attachment.

When a monitor is unattached, it is in the *unattached state.* Its cost is 0 by definition, and its `attached` flag is `false`. Any other attributes that change as the solution changes are in principle undefined, because an unattached monitor, including these attributes, is usually out of date. However a monitor's invariant is free to assign particular values to any of these attributes in the unattached state, if that is convenient.

A monitor becomes attached in two steps. The first step is to convert the unattached state into the *unlinked state*, which is the appropriate state for the monitor when it is formally attached but not yet linked in to the constraint propagation network. Its `attached` flag is `true`, and its attributes that change as the solution changes (including its cost) have well-defined values, and its cost has been reported to its parents. The second step is to call on each relevant part of the constraint propagation network, informing it that the monitor is now attached and wants to receive updates. Each such part will call back with an initial update, that the monitor uses to bring itself fully up to date.

It is true that one could take a different approach, in which the monitor's state is not well-defined, and cost is not reported to parents, until after the monitor is fully linked in to the constraint propagation network. However, linking to part of the solution or to a monitor often has the same effect on the monitor as a change of state in that part of the solution or monitor, and the approach taken here brings out that commonality.

Returning now to our two-step approach, we give some examples of unlinked states. To keep above the details we confine ourselves to the *unlinked cost*: the monitor's cost in the unlinked state. This is often 0, but not always. Here are a few examples.

The unlinked cost of an assign resource monitor is 0, because it is not linked to any event resources, and so it cannot be aware of any unassigned ones.

The unlinked cost of a limit busy times monitor is 0, because its cost is summed over its time groups, and initially it is linked to none.

The main causes of non-zero unlinked costs are minimum limits. Consider a limit workload monitor with a minimum limit. When it is unlinked, it has no evidence that its resource is assigned any work at all, and so its unlinked deviation is the cost of being assigned nothing.

In general, the process of attachment of monitor `m` looks like this:

```
m->attached = true;
if( unlinked_cost > 0 )
{
  m->cost = unlinked_cost;
  report to parents the cost change from 0 to m->cost;
}
add the links from the solution and other monitors to m;
```

As previously explained, the last step produces callbacks to `m` that further change its state, and so possibly its cost. Unattachment reverses what attachment did:

```
remove the links from the solution and other monitors to m;
assert(m->cost == unlinked_cost);
if( unlinked_cost > 0 )
{
  report to parents the cost change from m->cost to 0;
  m->cost = 0;
}
m->attached = false;
```

## B.6. The limit active intervals monitor

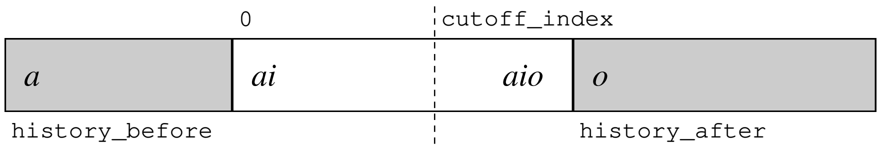
Monitors can be quite lengthy to implement, given the many state-changing operations they need to accept. However they are usually straightforward, once one understands the basic structure of taking a state change in, producing a new cost, and reporting it if it changed.

The limit active intervals monitor has a much longer and more complex implementation than the other monitors. Finding an efficient and coherent implementation was challenging, so this section documents that implementation in detail.

The basic data structure is a sequence of *time group info* objects, one for each time group, holding four fields: a pointer to the time group monitor for that time group, a polarity, and *state* and *interval* fields. A time group info object will be referred to here simply as a time group.

The state field contains the time group's state. The user is encouraged to believe that there are two states, active and inactive, but in fact there are three: active, inactive, and *open*, meaning that the monitor cannot assume that the time group is either active or inactive.

As Jeff Kingston's paper on history [10] explains, a limit active intervals monitor is actually a *projection* of a larger monitor spanning the full cycle. Its `history_before` attribute says how many active time groups there are immediately preceding the current monitor, and its `history_after` attribute says how many time groups (in any state) there are following it. The time group sequence is extended at each end to accommodate these *virtual* time groups:



This diagram illustrates several points. The *real* (non-virtual) time groups are represented by the white box. The first has index 0. But some indexes outside this range are permitted: down to `-history_before`, and up to `count + history_after - 1`, where `count` is the number of real time groups. Actual objects are not present in the two extended parts of the range, but nevertheless the monitor's functions accept these indexes. They behave as though each time group in the left part is active, and each time group in the right part is open. In this way, the virtuality of these time groups is hidden, except that it is not possible to change their state.

One could simplify the implementation by creating objects for all time groups, but that would be a mistake. It would be safe enough for `history_before`, but `history_after` could be very large, and creating all those extra time groups would waste time and memory.

A real time group can be active, inactive, or open. It is open when it lies at or after the cutoff index and its busy count is zero. Otherwise, it is either active or inactive, depending on its busy count and polarity in the usual way. A virtual time group is active when it lies in the `history_before` range, and open when it lies in the `history_after` range, except that all time groups (real and virtual) are inactive when the monitor is not attached.

This definition exposes the similarity between cutoff indexes and history after: both specify that some part of the cycle is not being solved, and hence that time groups there may be open.

There is an asymmetry in when a time group is open which needs explanation. Consider a real time group at or after the cutoff index. When it is busy (because of a preassignment, say, or task grouping), its activity or inactivity, depending on its polarity, is known, so considering it to be open, while possible, entails a loss of potentially valuable information. But when it is not busy, its activity or inactivity is not known, so its state must be open, at least by default.

`KheLimitActiveIntervalsMonitorSetNotBusyState` mitigates this by allowing the user to specify that when a given time group is at or after the cutoff index and is not busy, its state is to be active or inactive rather than open. It would perhaps be better to add to the platform an operation that declares that a certain resource will not be assigned anything at a certain time. But such an operation would be a major undertaking and it is not likely that it will ever be added.

We've reached a key point: we now know what time groups (real and virtual) there are, and how the state of each time group is defined. So we have a firm foundation to build intervals on. In doing so we will forget that some time groups are virtual. We will also forget why time groups have the states they have, and simply take those states as given.

An *interval* is a sequence of adjacent time groups. An *active interval*, or *a-interval*, is a maximal sequence of adjacent active time groups. Maximum limits are checked by comparing the lengths of the a-intervals with the limit. An *ao-interval* is a maximal sequence of adjacent time groups, each of which is either active or open. Minimum limits are checked by comparing the lengths of the ao-intervals with the minimum limit.

Actually the rules just given have a flaw. If an ao-interval is entirely open (if it contains no active time groups), then it is not defective, even if its length is less than the minimum limit. For now we will pretend that this flaw does not exist. We'll return to it and handle it later.

An interval is represented by an object in the usual way, containing indexes defining its endpoints, and its cost. One option would be to maintain a list of a-intervals when the monitor has a non-trivial maximum limit, and a list of ao-intervals when the monitor has a non-trivial minimum limit. However, given that many monitors have both and that open time groups are uncommon, this seems too expensive. What we actually do is maintain a list of ao-intervals only.

When a time group changes state for any reason, the sequence of ao-intervals is adjusted to take account of the change. The relevant ao-intervals are easily reached, because each active and each open time group contains a pointer to its enclosing interval. It may be necessary to lengthen an adjacent interval, or merge two intervals that become adjacent, or even to delete an interval (when the changing time group was its only element). Each interval that changes recalculates its cost and reports the change in cost to the monitor, which passes on the total change in cost.

New intervals come from a free list of interval objects in the monitor; deleted intervals return there. So once a solve is well under way there is little or no memory allocation.

It is trivial for an ao-interval to check itself against a minimum limit, because it knows its own length. (We are still ignoring the problem of ao-intervals with no active time groups.) To check itself against a maximum limit, it might seem that it needs to find all the a-intervals within itself and check them against the limit. This is potentially slow. Of particular concern are cases where there is a small cutoff index, and hence, potentially, a long ao-interval extending past it, with a-intervals (produced by preassignments, say) scattered along it.

Here, however, we use the fact that the cost of a limit active intervals monitor when there is a cutoff index is open to negotiation. We include the following in its definition: *violations of limits by active intervals that begin at or after the cutoff index attract no cost*. This is a plausible part of what it means to install a cutoff index; but its real reason for being there is efficiency.

Where then are the a-intervals whose costs we need to calculate? They must begin before the cutoff index, so they must lie in ao-intervals that begin before the cutoff index. They cannot be preceded by open time groups, because all open time groups lie at or after the cutoff index. So the a-intervals we need are exactly those whose first time group is the first time group of its enclosing ao-interval, which itself must begin before the cutoff index.

For each ao-interval, even at and after the cutoff index, let its *initial a-interval* be the a-interval (if any) whose first time group is the ao-interval's first time group. Record in each ao-interval the length of its initial a-interval, or 0 if there is no initial a-interval. When the ao-interval's first time group is before the cutoff index, compare this with the maximum limit and generate a cost.

These initial a-interval lengths are easy to maintain as time groups change state and intervals are merged and split. At the worst, when a time group changes from open to active just at the end of the initial a-interval, the time groups from there on must be scanned to see how much longer the initial a-interval has become.

Finally, we can now solve the problem of ao-intervals with no active time groups. Since open time groups can only occur at or after the cutoff index, such intervals always begin at or after the cutoff index. And we have just introduced a rule which requires all such intervals to have no cost. Problem solved.

## B.7. An arena and arena set plan

Arenas and arena sets can be used to allocate and free memory very efficiently. However, if their advantages are to be realized, a carefully worked out plan for them is needed.

Some basic facts constrain this plan. Although arenas are cheap to create, still their number should be minimized, since from one point of view every arena creation is an unproductive use of time and memory; and it calls `malloc` and thus may produce contention. Accordingly, all objects that are known to have the same lifetime should share an arena. For example, any significant solver will allocate memory while it is running, but after it ends its effect will be confined to changes in the solution it worked on. So it makes sense for all memory allocated by a solver to be kept in a single arena, and for that arena to be deleted or recycled as one of the final steps of that solver. Again, the objects making up one solution will all be deleted together (except the solution object itself, which may need to survive as a placeholder), so they should lie in one arena.

To maximize the re-use of memory, two rules are needed. First, there should be as few arena sets as possible, since then there will be as few idle arenas as possible. The minimum number of arena sets is one per thread, because arena sets have no locking and cannot be shared between threads. It would be possible to have locked arena sets and create just one global arena set that all arenas come from, but that approach has not been followed, because even though there would be very little contention for this arena set, still we prefer to avoid all unnecessary locking.

When a thread ends, its arena set is deleted, after moving its arenas across into the arena set of the parent thread, the one that will be continuing. Care is needed here when the thread's arena set is stored in other objects that are continuing, as KHE stores it in solution objects.

KHE stores the arena set in solution objects but nowhere else. When the thread ends, the arena set field of every solution that is being kept is set to the parent thread's arena set, leaving no trace of the thread arena set in any continuing object.

The second rule for maximizing re-use of memory is that every arena should be taken from an arena set and returned to an arena set when it is no longer needed: `HaArenaMake` and

`HaArenaDelete` should not be called directly. The main issue here is ensuring that an arena set is available at every point in the program. For the implementer of KHE, and for users who write their own multi-threaded programs, this takes some care; but for most users of KHE, it is trivial, because KHE supplies a suitable arena set with every solution, that the user can obtain arenas from via functions `KheSolnArenaBegin` and `KheSolnArenaEnd` (Section 4.2.2).

We also need to consider modules which assist solvers, such as the priority queue or weighted bipartite matching modules. Such modules might not wish to get their memory from `KheSolnArenaBegin` and `KheSolnArenaEnd`, because they want to be independent of KHE solution objects and rely only on Ha, or because they can use the same arena as the solver that calls them, saving arena creations. These modules typically accept an arena parameter, and offer no operation to delete themselves, that being done when the arena they are passed is deleted.

The remainder of this section analyses an issue that has puzzled the author. In general, it arises when it is not known whether a program is going to break into multiple threads or not.

When a solution is created afresh, it is clear that it is going to be solved, and it can be passed the arena set of the thread it is being solved by. Different solutions may thus have different arena sets. But when a solution is read, the read is part of a single thread that reads many solutions, and all solutions would naturally share a single arena set (and do so in the current implementation).

Now consider reading some solutions and resuming solving them in parallel. KHE offers no functions for doing this, but there is nothing to prevent it, except that there will be a contention problem in their shared arena set.

However, there is a way out. There is no contention within individual arenas, because each solution occupies a separate arena (in fact two, owing to the placeholder issue). So the answer is to create one new arena set for each solution and install it by calling `KheSolnSetArenaSet`. Then parallel solving can proceed without problems. Solutions can even be deleted in parallel: the arenas freed by deletions will be recycled into the new arena sets, not into the old one.

# Appendix C. Dynamic Programming Resource Reassignment: Theory

This Appendix presents the theory underlying the dynamic programming algorithm for resource reassignment (Section 12.6), including an overview of the algorithm, an analysis of its running time, and the algebra behind its cost calculations. Appendix D presents the implementation.

## C.1. Overview of the algorithm

This section gives an overview of the algorithm.

Let the days (time groups of the common frame) selected for reassignment, called the *open days*, be $\langle d_1, \ldots, d_n \rangle$ in chronological order. This is not the full sequence of days of the cycle; other days, the unselected ones, may occur before, between, and after the open days. Let the resources (nurses or other employees) selected for reassignment, called the *open resources*, be $\{r_1, \ldots, r_m\}$. Their order does not matter. Let the *shift types* (morning, afternoon, and so on) be $\{s_0, s_1, \ldots, s_a\}$, where $s_0$ is a special shift type denoting non-assignment (a free day).

Instead of using the terms 'shift' and 'shift type', whose exact meaning can be slippery, we prefer the following terms when we need to be precise. A *task* is a variable representing an indivisible piece of work. It can be unassigned, or else it can be assigned a single resource $r$, indicating that $r$ will carry out that work. A *multi-task*, or *mtask*, is a set of tasks which are similar enough to be interchangeable for most purposes; Section 11.9 has the exact definition. For example, the informal expression 'the morning shift on the second Monday of the cycle' can be formalized as an mtask. There is no precise equivalent for 'shift type', although one shift type on one day often defines one mtask.

An *assignment* is a triple $(d_i, r_j, C_{ik})$, indicating that on open day $d_i$, open resource $r_j$ is assigned to a task from mtask $C_{ik}$, whose tasks all run on day $d_i$. By convention, mtask $C_{i0}$ on day $d_i$ is a special mtask indicating that $r_j$ is free on $d_i$. It does not matter which task from $C_{ik}$ is assigned $r_j$, since the tasks of $C_{ik}$ are interchangeable. In the implementation, the assignment requests a task from $C_{ik}$, which returns the best unassigned one.

A *solution* $S$ is a set of assignments satisfying two conditions. First, at most one assignment containing a given $d_i$ and $r_j$ may appear in $S$, because $r_j$ may only do one thing on day $d_i$. If there is no assignment containing $d_i$ and $r_j$ in $S$, it does not mean that $r_j$ is free on $d_i$; rather, it means that no decision has been made about what $r_j$ is doing on $d_i$. This is a key point: on any given day, a resource is not either busy or free in $S$; rather, it is either busy, free, or undecided.

The second condition required for $S$ to be a solution is that although each $C_{ik}$ may appear multiple times in $S$, its number of occurrences may not exceed the number of tasks in $C_{ik}$. The mtask $C_{i0}$, representing a free day on $d_i$, may appear any number of times.

If assignments containing all pairs $(d_i, r_j)$ are present, $S$ is called a *complete solution.* We often deal with solutions which are complete up to and including a given day $d_k$; that is, with solutions which contain one assignment for every $d_i$ and $r_j$ such that $i \le k$, and no assignments for any $d_i$ and $r_j$ such that $i > k$. These we call *$d_k$-solutions.* Even though there is no day $d_0$, we call the empty solution a $d_0$-solution.

All solutions implicitly include many assignments from the *initial solution* (the one we are finding a reassignment for): all time assignments, all assignments of unselected resources, and all assignments of selected resources on unselected days.

An *extension* of a solution $S$ is a solution $S'$ such that $S \subseteq S'$. That is, $S'$ is a solution that contains all the assignments of $S$, plus possibly some others. A *complete extension* of a solution $S$ is an extension of $S$ which is a complete solution.

The implementation has *solution objects* which represent solutions. Rather than holding its full set of assignments, each solution object holds a few assignments plus a *parent pointer* which when non-`NULL` points to another solution object that this solution object is an extension of. So to find the full set of assignments one has to follow a path made by parent pointers.

As a step towards the dynamic programming algorithm, consider first a tree search algorithm which assigns a $C_{1k}$ to each $r_j$ on $d_1$, then a $C_{2k}$ to each $r_j$ on $d_2$, and so on. If there are $a$ mtasks on each day (in reality the number can vary), and we ignore the fact that each $C_{ij}$ can be chosen a limited number of times, then each day has $(a + 1)^m$ choices, so the tree tries up to $(a + 1)^{mn}$ timetables altogether. Clearly this will find the best timetable, but at the cost of exploring an infeasibly large number of alternatives in practice.

So let us move on to dynamic programming. Consider these two $d_5$-solutions, assuming that there is just one resource, $r_1$. Each box represents what $r_1$ is doing on one day:

$S_1$

| | | | | |
|---|---|---|---|---|
| $s_1$ | $s_1$ | $s_1$ | $s_0$ | $s_0$ |

$S_2$

| | | | | |
|---|---|---|---|---|
| $s_2$ | $s_2$ | $s_2$ | $s_0$ | $s_0$ |

For convenience we write $s_1$ for $C_{11}$, $C_{21}$, etc. If there is no constraint on the total number of $s_1$ shifts that may be assigned $r_1$, and none on the total number of $s_2$ shifts that may be assigned $r_1$, then $S_1$ and $S_2$ are indistinguishable from here on. If, say, $S_1$ has a smaller cost than $S_2$ (for example if there is an upper limit of 2 on the number of consecutive $s_2$ shifts, but not on the number of consecutive $s_1$ shifts), then it is safe to not explore the search tree rooted at $S_2$, because any timetable it leads to will be worse than the corresponding timetable in the search tree rooted at $S_1$. The dynamic programming algorithm exploits this idea.

Of course, one cannot simply ignore all but one solution for $d_k$. In the example, if there *are* upper limits on the number of $s_1$ and $s_2$ shifts, then the search trees rooted at both solutions must be explored, because the different numbers of $s_1$ and $s_2$ shifts may lead to different costs later.

Given two solutions $S_1$ and $S_2$, we say that $S_1$ *dominates* $S_2$ if the presence of $S_1$ allows $S_2$ to be dropped. Dominance works by associating an array of numbers with each solution, called its *signature.* Most of the numbers will be integers, but some can be floating-point numbers. The signature has one element for each constraint, usually, recording the constraint's state. For example, if there is a maximum limit on the number of shifts worked, the signature of solution $S$ will have one element recording the number of shifts worked in $S$. By comparing this element in $S_1$ and $S_2$ we can find out if $S_1$ dominates $S_2$ as far as this constraint goes. If $S_1$ dominates $S_2$

for all constraints, and its cost is not larger, then $S_1$ dominates $S_2$.

The solver implements several kinds of dominance. The simplest is *equality dominance*, which says that $S_1$ dominates $S_2$ when their signatures are equal at every position, and the cost of $S_1$ does not exceed the cost of $S_2$. Having equal signatures means that all costs incurred as the solve progresses beyond $d_k$ will be the same, so $S_1$'s cost advantage will never be lost. Another kind, *separate dominance*, makes a '$\leq$', '$\geq$', or '$=$' comparison at each element, depending on whether the constraint there has a maximum limit, a minimum limit, or both. There are other kinds of dominance as well, which we'll explore later.

The dynamic programming algorithm is as follows. Search the search tree in a breadth-first fashion, first finding the unique $d_0$-solution, then $d_1$-solutions, then $d_2$-solutions, and so on. For each day $d_k$, maintain $P_k$, a set of $d_k$-solutions. As each new $d_k$-solution $S$ is created, see whether $S$ is dominated by any existing solution in $P_k$ and drop it if so. If not, drop from $P_k$ any existing solutions that are dominated by $S$, then add $S$ to $P_k$. In this way, $P_k$ eventually contains all undominated $d_k$-solutions.

Suppose $P_k$ is all finished. To expand beyond there, for each solution $S$ in $P_k$, create one new solution for each combination of assignments of the $r_j$ to the $s_u$ on day $d_{k+1}$. Each of these new solutions is a $d_{k+1}$-solution (ignoring multi-day tasks), so each gets added to $P_{k+1}$, with dominance testing applied within $P_{k+1}$ just as within $P_k$. On the last day, $d_n$, all that is required for dominance is smaller cost, as we will see later. So $P_n$ contains at most one solution, and that is the result.

As we will see, each solution comes with a cost. Solution cost is non-decreasing along each path in the search tree, because no cost is added until it is certain that subsequent assignments will not remove it. (This is true even of constraints with minimum limits, whose cost might otherwise be expected to decrease as the solve proceeds.) So the algorithm compares the cost of each newly created solution $S$ with the cost of the initial solution, and deletes $S$ when its cost is not smaller. This is important: it prunes away many inferior alternatives.

Instead of proceeding from one day to the next, one can instead keep all the solutions in a priority queue and proceed from one solution to the next, always choosing an unchosen solution of minimum cost (this could be for any day). The advantage is that the algorithm can stop as soon as a solution is chosen which is for the last day; the disadvantage is the priority queue overhead. Some solves run more quickly when they use the priority queue; others run more slowly.

A clearer benefit is gained by grouping similar mtasks into larger structures that the author calls *shifts*, just as tasks are grouped into mtasks. This is a subject for later.

## C.2. Running time

We now prove the result stated in Section 12.6, that one solve runs in time $O(n(a+1)^m mn^{cm})$, where $m$ is the number of selected resources, $n$ is the number of selected days, $a$ is a constant, the number of shift types, and $c$ is another constant (usually 1 or 2), the number of constraints per resource whose maximum limits increase with $n$ (such as limits on the total number of shifts).

Let $P_k$ be the final set of stored undominated $d_k$-solutions, for $0 \leq k \leq n$. Let $W(k)$ be the number of solutions in $P_k$ when equality dominance is used.

A constraint only needs a presence in the signatures of $d_k$-solutions if its cost is affected by what happens on $d_k$ or earlier, and $d_k$ is not the last day it is affected by. This is because on

| | |
|---|---|
| $a_i$ | The history before value of an (unspecified) monitor $m$ |
| $c_i$ | The history after value of an (unspecified) monitor $m$ |
| $c(m, S)$ | The cost of monitor $m$ in solution $S$ |
| $c(S)$ | The cost of solution $S$ |
| $d_i$ | A selected (also called open) day |
| $(d_i, r_j, C_{ik})$ | An assignment of resource $r_j$ to a task of mtask $C_{ik}$ on open day $d_i$ |
| $[d_a, \ldots, d_b]$ | The active interval of an (unspecified) monitor $m$ |
| $d(m, S)$ | The determinant of monitor $m$ in solution $S$ |
| $dom(S)$ | The domain of solution $S$ |
| $f(x)$ | The cost function attribute of a monitor |
| $l(m, S)$, or just $l$ | The lower determinant of monitor $m$ in solution $S$ |
| $l(z)$ | The length of an a-interval or au-interval $z$ |
| $m$ | The number of selected (also called open) resources; or a monitor |
| $n$ | The number of selected (also called open) days |
| $r_i$ | A selected (also called open) resource |
| $s_i$ | A shift type |
| $u(m, S)$, or just $u$ | The upper determinant of monitor $m$ in solution $S$ |
| $v(x, k)$ | The number of distinct values of expression $x$ on day $d_k$ |
| $w$ | The weight attribute of a monitor, part of its cost function $f(x)$ |
| $x_i$ | The history value of an (unspecified) monitor $m$ |
| $z$ | An a-interval or au-interval of a sequence monitor |
| $A$ | The set of a-intervals of an (unspecified) sequence monitor $m$ |
| $AU$ | The set of au-intervals of an (unspecified) sequence monitor $m$ |
| $C_{ik}$ | An mtask (a set of interchangeable tasks) |
| $D_l$ | A minimum determinant—a lower limit on $l(m, S)$ |
| $D_u$ | A maximum determinant—an upper limit on $u(m, S)$ |
| $E_k$ | The set of expressions requiring an entry in day $d_k$ signatures |
| $L$ | The lower limit attribute of a monitor |
| $M$ | The set of all monitors for the current instance |
| $M_k$ | The set of monitors that are active on open day $d_k$ |
| $P_k$ | The set (possibly under construction) of undominated $d_k$-solutions |
| $S$ | A solution (a set of assignments), not necessarily complete |
| $S_0$ | The initial solution (the empty set of assignments) |
| $S(k)$ | The cardinality of the final value of $P_k$ using separate dominance |
| $U$ | The upper limit attribute of a monitor |
| $W(k)$ | The cardinality of the final value of $P_k$ using equality dominance |
| $Z$ | The allow zero flag of a monitor |
| $\delta(l, u)$ | The deviation of a counter monitor with determinants $l$ and $u$ |
| $\chi(c, v_1, v_2)$ | **if** $c$ **then** $v_1$ **else** $v_2$ |
| $\Delta(m, S_1, S_2)$ | A lower bound on $m$'s contribution to available cost |

**Figure C.1.** Notations used in this Appendix (excluding separate and tradeoff dominance).

earlier days there is nothing for the signature to remember, and on its last day its cost is finalized and added to the solution cost, so that again there is nothing to remember.

We'll see later that the solver represents constraints by expressions; so this analysis speaks of expressions, but the reader can safely take them to be constraints. Let $E_k$ be the set of expressions which need a presence in the signatures for day $d_k$. If $x \in E_k$, then $x$ contributes one value to the signature of each $d_k$-solution. Let $v(x, k)$ be the number of distinct values that could be stored on behalf of $x$ in the signature of a $d_k$-solution. For example, there are *OR* expressions representing the logical 'or' of their children. If $x$ is an *OR* expression, the stored value could be 0 or 1, so $v(x, k) = 2$.

The signature of a solution in $P_k$ is the concatenation of the values stored for the expressions $x \in E_k$, and under equality dominance the solutions of $P_k$ have distinct signatures, so

$$W(k) \le \prod_{x \in E_k} v(x, k)$$

The next questions are, how large could $E_k$ be, and how large could $v(x, k)$ be? To answer these questions, we will focus on nurse rostering instances that occur in practice, and we divide their constraints (strictly, monitors) into three classes.

In the first class lie all event resource constraints. In practice, each of these applies to a single day, and so its expressions do not lie in any $E_k$, and contribute nothing to the product.

In the second class lie resource constraints that concern local patterns, such as prohibiting a day shift following a night shift, or requiring both days of a weekend to be busy or neither. Clearly, for one resource on one day, there will be only a small constant number of these, say $b$, and $v(x, k)$ will also be a small constant, typically 2. So these constraints contribute about $2^b$ to the product per resource, or $2^{bm}$ over all $m$ selected resources.

In the third class lie resource constraints that concern global limits, for example on the total number of shifts worked. For each resource there will only be a small constant number of such constraints, say $c$, but for them $v(x, k)$ will be larger, on the order of $n$. For one resource this contributes about $n^c$ to the product, or $n^{cm}$ over all $m$ selected resources.

Putting these three cases together, and observing that the last term dominates, we get

$$W(k) \;\le\; \prod_{x \in E_k} v(x, k) \;=\; O(n^{cm})$$

where $c$ is the number of constraints per resource whose upper limits are on the order of $n$.

Given a solution in $P_k$, we may take the running time of assigning one more day's worth of shifts, including creating a new solution object, finding its signature and cost, and looking up the signature in the $P_{k+1}$ hash table, to be $m$. This is because the number of constraints per resource on a particular day is a constant. Our implementation does indeed do this in $O(m)$ time.

For each of the $W(k)$ solutions in $P_k$ we generate at most $(a+1)^m$ new solutions, where $a$ is the number of shift types (really mtasks that begin on day $d_k$), making a total running time of $(a+1)^m m W(k)$ to generate the $P_{k+1}$ hash table. So the overall running time is at most

$$\sum_{0 \le k < n} (a+1)^m m W(k) \;\le\; n(a+1)^m m W(n) \;=\; O(n(a+1)^m m n^{cm})$$

as advertised.

In some models, shifts have durations in minutes and there is a constraint on the total duration of the shifts taken by a nurse. This could lead to a very large value of $v(x,k)$, although the number should be manageable if all durations are multiples of, say, 30 or 60 minutes.

The author has not found any way to tighten up this analysis for other forms of dominance. It is easy to see that $S(k) \leq W(k)$, where $S(k)$ is the size of $P_k$ when separate dominance is used. This can be proved using induction on $k$ and the fact that every case of equality dominance is also a case of separate dominance. The running time for creating one solution and inserting it into $P_k$ must be multiplied by $S(k)$, to account for the cost of the pairwise dominance tests. (Equality dominance is much faster, merely requiring a retrieval of the signature in a hash table.) One would think that this would make separate dominance significantly slower, but testing suggests otherwise. Similar remarks apply to tabulated dominance.

Also important in practice is the extent to which solutions get pruned because their cost exceeds the cost of the initial solution. Again there seems to be no way to estimate this. Its effect will be larger as the initial solution improves, and also as the search approaches its end.

## C.3. Monitors, costs, and signatures

Each solution $S$ has an associated *cost*, written $c(S)$. When $S$ is complete, its cost must equal the cost of the solution as defined by XESTT. When $S$ is incomplete, XESTT does not define any cost, but we will define one ourselves.

Each solution also has a *signature*, which is an array of numbers (integers and floats) whose values represent the states in $S$ of the constraints of $S$'s instance. For example, if there is a constraint on the number of days that resource $r$ can be busy, there would be an entry in the signature of $S$ belonging to that constraint, recording the number of days that $r$ is busy in $S$.

This section shows how to calculate costs and signatures as solutions are extended.

A *monitor* (the term comes from KHE) is one point of application of one constraint, whose violation yields one soft or hard cost. For example, there might be a monitor requiring resource $r$ to be free on the second Monday of the cycle. If this is violated, a cost is incurred.

Where we have previously referred to constraints, we really meant monitors. One constraint in XESTT is a template from which many monitors may be derived—one for each resource subject to the constraint, typically. The monitors form a fixed set $M$, defined by the instance being solved, and common to all solutions. By definition, their total cost is the cost of the solution.

In XESTT, monitors are classified into three major types. *Event monitors* monitor the times assigned to events. Since all times are preassigned in nurse rostering, event monitors play no role here. *Event resource monitors*, often called *cover constraints* in nurse rostering, are concerned with ensuring that each event (shift) is assigned a suitable number of resources (nurses) with suitable skills. Finally, *resource monitors* monitor the timetables of individual resources, checking for undesirable patterns (such as a morning shift following a night shift), for a reasonable total workload, and so on.

Classified another way, nurse rostering monitors may be *counter monitors*, which constrain the number of occurrences of something (night shifts, busy weekends, and so on), *sum monitors*, which constrain the total amount of something (workload in minutes, and so on), or *sequence monitors*, which constrain the length of each non-empty sequence of consecutive somethings (night shifts, free days, and so on). All event resource monitors and most resource monitors are either counter monitors, or else they can easily be converted into counter monitors. We make these conversions. Sum monitors are derived only from XESTT limit busy times and limit workload constraints, although counter monitors can be viewed as special cases of sum monitors, where the values being summed are always 0 or 1. Sequence monitors are derived only from XESTT limit active intervals constraints. The rule that each monitor yields one cost is modified for sequence monitors: each non-empty sequence of consecutive somethings yields one cost.

As we will see in Appendix D, the solver represents monitors of all types by expression trees called *expressions*. We are concerned here with what happens in the root nodes of expression trees (this is where costs are calculated), so we will use 'monitor' rather than 'expression'.

We write $c(m,S)$ for the cost of monitor $m$ in solution $S$. If $S$ is a complete solution, we require $c(m,S)$ to be the cost of $m$ in $S$ as defined by the XESTT specification of $m$'s constraint. This satisfies the condition $c(m,S) \geq 0$, and larger values indicate worse violations.

The specifications of the constraints do not define $c(m,S)$ when $S$ is not complete. We'll be doing that ourselves later. We are free to choose any definition satisfying

$$0 \;\leq\; c(m,S) \;\leq\; c(m,S')$$

for all extensions $S'$ of incomplete solution $S$. That is, monitor costs are non-negative, and non-decreasing as solutions become more complete. We require this condition so that we can prune $S$ when its cost reaches the cost of a known complete solution, since at that point none of its complete extensions can improve on what we have.

It helps with pruning if our definition of $c(m,S)$ for an incomplete solution $S$ yields values that are as large as possible. That is why we do not define $c(m,S) = 0$ for incomplete solutions $S$, even though that is legal and very simple.

Then we define $c(S)$, the cost of solution $S$ (which may be complete or incomplete), to be

$$c(S) \;=\; \sum_{m \in M} c(m,S)$$

where $M$ is the fixed set of all monitors, as defined above. It is clear that this is non-negative, non-decreasing as we proceed from less complete to more complete solutions, and the same as the XESTT definition of cost when $S$ is complete.

The $c(m,S)$ values which sum to $c(S)$ are not stored individually. However, a monitor $m$ may own a position in $S$'s *signature*, an array of numbers (some integer, others floating-point) stored in $S$, in which the monitor stores its own state, which we call its *determinant*, written $d(m,S)$. For example, if $m$ monitors the number of days that resource $r$ is busy, $d(m,S)$ would be the number of days that $r$ is busy in $S$. If required, $c(m,S)$ can be calculated from $d(m,S)$.

Even before any assignments are made to open days, a monitor $m$ may be making a contribution to the total cost. For example, if $m$ imposes an upper limit of 10 busy days, and among the days that are not opened there are assignments that make 12 busy days, a cost caused by the 2 extra days is already being contributed. Call this cost $c(m,S_0)$, where $S_0$ denotes the initial empty solution. The solver ensures that

$$c(S_0) = \sum_{m \in M} c(m,S_0)$$

when solving begins. It does this by taking the cost of the initial solution and carefully updating it as resources and days are opened. The signature of $S_0$ is the empty array, because nothing needs to be remembered about the state of any monitor at that point. Each monitor does have an initial state, but it remembers that within itself.

The remainder of this section is concerned with how a newly created solution $S'$, assumed to be an extension of a solution $S$, obtains its correct cost and signature. Each monitor $m$ is responsible for calculating its own contribution: it needs to ensure that $c(m, S')$ is included in $c(S')$, and that $d(m, S')$ is appended to $S'$'s signature. Our method of calculating the cost is this:

$$\begin{aligned} c(S') &= \sum_{m \in M} c(m, S') \\ &= c(S) + \sum_{m \in M} [c(m, S') - c(m, S)] \end{aligned}$$

We call $c(m, S') - c(m, S)$ an *extra cost*. We have already seen that it is non-negative. We do it this way because it is faster and more convenient, as we will now show for $d_k$-solutions. Other types of solutions will be considered later.

Let $[d_a, \dots, d_b]$ be the largest sequence of consecutive open days such that what happens on $d_a$ and $d_b$ affects the cost of $m$. The days in the interior of the sequence may affect $m$ or not, but the two endpoints (which could be equal) definitely affect $m$, and nothing outside the interval affects $m$. As a special case, if $m$ is not affected by what happens on any open day, the interval is empty. A review of the XESTT monitors will show that there is no difficulty in calculating $[d_a, \dots, d_b]$ before solving begins. The solver does this.

We say that $m$ is *active* on the days of its interval, and we call the interval itself the *active interval* of $m$. (This is unrelated to the active intervals of the limit active intervals constraint.) For each open day $d_k$, the solver builds a set $M_k$ of the monitors that are active on that day. Monitor $m$ appears in $b - a + 1$ sets, possibly none (if $m$ is not affected by what happens on any open day).

As a solve proceeds, no work is done with $m$ on open days before $d_a$. This is because $m$'s cost remains constant at $c(m, S_0)$ on these days, so the extra cost is 0, and there is no need to remember a determinant in any signature.

Similarly, no work is done with $m$ on open days after $d_b$. This is because $m$ works out its final cost on $d_b$ and includes it in the solution cost that day. There is no need to remember a determinant from then on, because it would not be used. If $S'$ is a $d_k$-solution, $S$ is a $d_{k-1}$-solution, and $M_k$ is the set of active monitors on day $d_k$, we have just shown

$$c(S') = c(S) + \sum_{m \in M_k} [c(m, S') - c(m, S)]$$

This greatly reduces the number of monitors that we have to visit.

On a typical day when $m$ is active, $m$ retrieves its determinant $d(m, S)$ from the signature of $S$ and uses it to calculate $c(m, S)$. Then it calculates $d(m, S')$ based on $d(m, S)$ and other information it has available to it (the details depend on the monitor type so will be given later) and uses it to calculate $c(m, S')$. It then adds $c(m, S') - c(m, S)$ to $c(S')$ and appends $d(m, S')$ to $S'$'s signature.

On $m$'s first active day the procedure is a little different. There is nothing for $m$ to retrieve from $S$'s signature. Instead it initializes itself from its own internal data, then goes on to store a cost and signature value in $S'$ as usual.

On $m$'s last active day the procedure is different again. The usual cost calculation is made, but no signature value is appended to $S'$'s signature.

In practice, event resource monitors always seem to have a single active day. This means that they never need to store a determinant in the signature. However, the algorithm is quite open to multi-day event resource monitors. For example, if at least four of the seven night shifts of Week 1 must include a senior nurse, the monitor for that would store the number of night shifts with senior nurses so far as its signature value. Everything follows as usual.

## C.4. Dominance

An informal introduction to dominance was given in Section C.1. Formally, solution $S_1$ is said to *dominate* solution $S_2$ if for every complete extension $S_2'$ of $S_2$ there is a complete extension $S_1'$ of $S_1$ such that $c(S_1') \leq c(S_2')$. If $S_1$ dominates $S_2$, then $S_2$ can be dropped without increasing the cost of the best complete solution found, saving the time required to construct $S_2$'s extensions.

Finding as much dominance as possible is the key to low running time, so we will explore a sequence of increasingly effective (but increasingly complicated) dominance tests. A dominance test is needed for every expression that stores a value in the signatures, but we will not try to cover every type of expression here. Instead, we'll develop a repertoire of dominance tests, and leave it to the different expression types to choose the best test that works for them.

### C.4.1. Separate dominance

Perhaps the simplest form of dominance is *equality dominance*, in which $S_1$ dominates $S_2$ when $c(S_1) \leq c(S_2)$ and at each position along the signature the corresponding values are equal. Since these values hold the state of all monitors in play, their equality implies that as the same assignments are added to both solutions, the costs incurred will be the same, so that $S_1$'s current advantage over $S_2$ will never be lost. Equality dominance is implemented efficiently by the solver, using a hash table indexed by signature. But testing shows that it misses too many cases of dominance to be efficient overall.

This section presents *separate dominance*, so named because, like equality dominance, it handles each signature position independently of the others. The algebra gets a little messy, but the idea is simple and widely applicable.

Given solutions $S_1$ and $S_2$ with costs $c(S_1)$ and $c(S_2)$, we say that $S_1$ *separately dominates* $S_2$ when $c(S_1) \leq c(S_2)$, and at each position in the signature arrays of $S_1$ and $S_2$, $S_1$'s signature value separately dominates $S_2$'s signature value, in a sense to be defined now.

Just what constitutes dominance at position $i$ along a signature depends on what the value represents, but let's take the most common case, where the value is the determinant of a monitor $m$ that has a non-negative maximum limit $U$, a non-negative minimum limit $L$, and a Boolean allow zero flag $Z$. We require $L \leq U$, and if $Z$ is true we require $L \geq 2$.

We allow $L$, $U$, and the values at position $i$ to be integers or floating-point numbers; the algebra will work either way. If $m$ has no maximum limit, we set $U$ to any finite upper bound on the value of the determinant. Although we won't pause to prove it here, for all XESTT constraints such upper bounds are easy to find. If $m$ has no minimum limit, we set $L$ to 0.

Suppose the signature of solution $S_1$ contains value $l_1$ at position $i$, and the signature of solution $S_2$ contains value $l_2$ at position $i$. We ask whether $l_1$ dominates $l_2$, written $dom(l_1, l_2)$. Abstractly, $l_1$ dominates $l_2$ when these values ensure that, in each extension of $S_1$, the cost of $m$ is not greater than it is in the corresponding extension of $S_2$. We need to make this concrete.

Maximum limits affect dominance independently of minimum limits. Let $dom\_max(l_1, l_2)$ be the dominance condition for the maximum limit, and let $dom\_min(l_1, l_2)$ be the dominance condition for the minimum limit. Dominance requires both:

$$dom(l_1, l_2) \;=\; dom\_max(l_1, l_2) \textbf{ and } dom\_min(l_1, l_2)$$

Clearly, $dom\_max(l_1, l_2)$ is true when $l_1 \le l_2$, because as $S_1$ and $S_2$ are extended, corresponding solutions continue to have $l_1 \le l_2$ for $m$, and the cost associated with a maximum limit is always a monotone non-decreasing function of the determinant. But $dom\_max(l_1, l_2)$ is also true when $l_1$ is so small that $m$ cannot ever violate the maximum limit, either now or on subsequent days. This follows because then, in each extension of $S_1$, the cost of $m$ is 0, which cannot be greater than the cost of $m$ in $S_2$. We write this condition as $very\_small(l)$, and we get

$$dom\_max(l_1, l_2) \;=\; very\_small(l_1) \textbf{ or } l_1 \le l_2$$

Including $very\_small(l_1)$ may seem unimportant, but even a small increase in the chance of a dominance test succeeding can have a large cumulative effect on the number of solutions kept.

We require the formula for $very\_small(l)$ to be just $l \le a$ for some $a$. The choice of $a$ varies from one monitor type to another, so we'll postpone it for now; but here is an example. Suppose the instance covers four weekends and $m$ imposes a maximum limit of two busy weekends for some resource $r$. Suppose that the current solve has unassigned the first two weekends, that we have just finished assigning the first weekend, and that $r$ is free on the first weekend, busy on the third weekend, and free on the fourth weekend. Then $l_1$ is 1 and there is only one unassigned weekend remaining, so $m$ cannot be violated now or in the future, and so $very\_small(l_1)$ is true.

If it is clear that $very\_small(l)$ can never be true, we can choose $a = -1$, so that $l \le a$ is always false (because $l \ge 0$). So we don't make the inclusion of $very\_small(l)$ optional; it is always present, only possibly turned off by the choice of $a$.

Having $very\_small$ allows us to avoid treating the absence of a maximum limit as a special case: we set $a$ to any upper bound on the value of the determinant, making $very\_small(l_1)$ true, which in turn ensures that $dom\_max(l_1, l_2)$ is always true, in effect turning off the max test.

We turn now to minimum limits. Assuming for now that $Z$ is false, $l_1$ dominates $l_2$ when $l_1 \ge l_2$, because as $S_1$ and $S_2$ are extended, corresponding solutions continue to have $l_1 \ge l_2$ for $m$, and the cost associated with a minimum limit is always a monotone non-increasing function of the determinant. But $dom\_min(l_1, l_2)$ is also true when $l_1$ is so large that $m$ cannot ever violate the minimum limit, either now or on subsequent days, because then, in each extension of $S_1$, the cost of $m$ is 0, which cannot be greater than the cost of $m$ in $S_2$. We write this condition as $very\_large(l)$, and we get

$$dom\_min(l_1, l_2) \;=\; very\_large(l_1) \textbf{ or } l_1 \ge l_2$$

We require the formula for $very\_large(l)$ to be just $l \ge b$ for some $b$. Once again, we need to look into further details before we choose $b$.

If it is clear that *very_large*$(l)$ can never be true, we can set $b$ to any upper bound on the value of the determinant, plus one, so that $l \geq b$ is always false. So we don't make the inclusion of *very_large*$(l)$ optional; it is always present, only possibly turned off by the choice of $b$.

Having *very_large* allows us to avoid treating the absence of a minimum limit as a special case: we set $b$ to 0, making *very_large*$(l_1)$ true, which in turn ensures that *dom_min*$(l_1, l_2)$ is always true, in effect turning off the min test.

The next step is to incorporate $Z$, the allow zero flag, into our analysis. $Z$ has no effect on maximum limits, but it does modify the costs produced by minimum limits: value 0 can violate a minimum limit and produce a cost, but that cost disappears if the allow zero flag is set.

Assume that the allow zero flag is set. If *very_large*$(l_1)$ is true, the cost of $m$ is still 0 in all extensions of $S_1$, so $l_1$ dominates $l_2$. The remaining cases can therefore ignore *very_large*$(l_1)$. To be quite certain about them, we use brute force.

*Case 1*: $l_1 = 0$ and $l_2 = 0$. When the two values are equal in $S_1$ and $S_2$, they remain equal in all pairs of corresponding extensions. So $l_1$ dominates $l_2$ in this case (and $l_2$ dominates $l_1$).

*Case 2*: $l_1 = 0$ and $l_2 > 0$. At this moment, $m$ has no cost in $S_1$ but it may have a cost in $S_2$. Consider corresponding extensions $S'_1$ of $S_1$ and $S'_2$ of $S_2$, in both of which the value has increased by 1. There may be a cost in $S'_1$, and no cost in $S'_2$. So $l_1$ does not dominate $l_2$ in this case.

*Case 3*: $l_1 > 0$ and $l_2 = 0$. This is tricky, because here we have $l_1 \geq l_2$, which is enough for dominance when there is no allow zero flag. But suppose several days go by, these values do not change, and we reach the last day of the monitor. Then $m$ may have a cost in the first extension and it definitely has cost 0 in the second. So $l_1$ does not dominate $l_2$ in this case.

*Case 4*: $l_1 > 0$ and $l_2 > 0$. $Z$ has no effect here, either now or on subsequent days, because neither value is ever 0. So the analysis and formula for when $Z$ is false applies here.

We can also say of Case 1 that the formula (although not the analysis) for when $Z$ is false applies. So a concise expression of the test here is

$$\textit{very\_large}(l_1) \ \textbf{or} \ (l_1 \sim l_2 \ \textbf{and} \ l_1 \geq l_2)$$

defining $l_1 \sim l_2$ to be $Z \Rightarrow ((l_1 = 0) = (l_2 = 0))$. In words, $l_1 \sim l_2$ is true either when $Z$ is false, or when $Z$ is true and $l_1$ and $l_2$ are either both 0 or both non-zero (Cases 1 and 4). Overall,

$$\boxed{\begin{array}{l} \textit{dom}(l_1, l_2) \ = \ [l_1 \leq a \ \textbf{or} \ l_1 \leq l_2] \ \textbf{and} \ [l_1 \geq b \ \textbf{or} \ (l_1 \sim l_2 \ \textbf{and} \ l_1 \geq l_2)] \\ \qquad \text{where} \ \ l_1 \sim l_2 \ \ \text{stands for} \ \ Z \Rightarrow ((l_1 = 0) = (l_2 = 0)) \end{array}}$$

We box this to make it easy to refer back to. Here $a$, $b$, and $Z$ are constants; for a given monitor $m$, they may be different on different days, but on any one day they are the same for all tests.

We'll be using four simple equivalences:

| This | is equivalent to this |
|---|---|
| $l \le e_1$ **and** $l \le e_2$ | $l \le \min(e_1, e_2)$ |
| $l \le e_1$ **or** $l \le e_2$ | $l \le \max(e_1, e_2)$ |
| $l \ge e_1$ **and** $l \ge e_2$ | $l \ge \max(e_1, e_2)$ |
| $l \ge e_1$ **or** $l \ge e_2$ | $l \ge \min(e_1, e_2)$ |

The first part of the $dom(l_1, l_2)$ formula, $l_1 \le a$ **or** $l_1 \le l_2$, is equivalent to $l_1 \le \max(a, l_2)$. When $Z$ is false, the second part, $l_1 \ge b$ **or** $l_1 \ge l_2$, is equivalent to $l_1 \ge \min(b, l_2)$.

***Dominance and equality.*** There is a minor question whose answer is needed elsewhere: when is $dom(l_1, l_2)$ the same as $l_1 = l_2$ for all $l_1$ and $l_2$? The obvious answer is when $Z$ is false, $a = -1$, and $b$ is any upper bound on the value of the determinant, plus one, for then

$$\begin{aligned} dom(l_1, l_2) &= [l_1 \le a \textbf{ or } l_1 \le l_2] \textbf{ and } [l_1 \ge b \textbf{ or } (l_1 \sim l_2 \textbf{ and } l_1 \ge l_2)] \\ &= [\text{false} \textbf{ or } l_1 \le l_2] \textbf{ and } [\text{false} \textbf{ or } (\text{true} \textbf{ and } l_1 \ge l_2)] \\ &= l_1 = l_2 \end{aligned}$$

But, perhaps surprisingly, the case $Z$ is true, with $a$ and $b$ as before, also works:

$$\begin{aligned} dom(l_1, l_2) &= [l_1 \le a \textbf{ or } l_1 \le l_2] \textbf{ and } [l_1 \ge b \textbf{ or } (l_1 \sim l_2 \textbf{ and } l_1 \ge l_2)] \\ &= [\text{false} \textbf{ or } l_1 \le l_2] \textbf{ and } [\text{false} \textbf{ or } ((l_1 = 0) = (l_2 = 0) \textbf{ and } l_1 \ge l_2)] \\ &= l_1 \le l_2 \textbf{ and } (l_1 = 0) = (l_2 = 0) \textbf{ and } l_1 \ge l_2 \\ &= l_1 = l_2 \end{aligned}$$

since $l_1 = l_2$ implies $(l_1 = 0) = (l_2 = 0)$. Or to put it another way, if $(l_1 = 0) = (l_2 = 0)$ is false, causing $dom(l_1, l_2)$ to be false, we must have $l_1 \ne l_2$ anyway. So $a = -1$ and $b$ set to any upper bound on the value of the determinant plus one are sufficient.

***Reflexivity and transitivity.*** We can prove that $dom$ is reflexive very simply, by evaluating $dom(l, l)$. Proving transitivity is tedious, but important for assuring ourselves that our dominance test is well behaved. We have to show that $dom(l_1, l_2)$ and $dom(l_2, l_3)$ imply $dom(l_1, l_3)$, if $a$, $b$, and $Z$ are the same in all three formulas.

If relations $R_1$ and $R_2$ are transitive, then $R_1$ **and** $R_2$ is transitive. This easy result allows us to focus on a single signature value as we have been doing, knowing that when we build the full signature plus a cost, transitivity will be preserved. We also use it below.

First we show that $\sim$ is transitive: that $l_1 \sim l_2$ and $l_2 \sim l_3$ together imply $l_1 \sim l_3$. If $Z$ is false, all three conditions are true, so we can assume that $Z$ is true, reducing $x \sim y$ to $(x = 0) = (y = 0)$. Suppose $l_2 = 0$. Then $l_1 \sim l_2$ implies $l_1 = 0$, and $l_2 \sim l_3$ implies $l_3 = 0$, so $l_1 \sim l_3$ is true. Now suppose $l_2 \ne 0$. Then the same argument shows that $l_1 \ne 0$ and $l_3 \ne 0$, so again $l_1 \sim l_3$ is true.

Next we show that $dom\_max(l_1, l_2)$ and $dom\_max(l_2, l_3)$ imply $dom\_max(l_1, l_3)$. This is equivalent to showing that $l_1 \le \max(a, l_2)$ and $l_2 \le \max(a, l_3)$ imply $l_1 \le \max(a, l_3)$, which is easy: a valid substitution gives us $l_1 \le \max(a, \max(a, l_3)) = \max(a, l_3)$.

Finally, we show $dom\_min(l_1, l_2)$ and $dom\_min(l_2, l_3)$ imply $dom\_min(l_1, l_3)$. We are given

$$l_1 \geq b \textbf{ or } (l_1 \sim l_2 \textbf{ and } l_1 \geq l_2)$$

and

$$l_2 \geq b \textbf{ or } (l_2 \sim l_3 \textbf{ and } l_2 \geq l_3)$$

and we need to prove

$$l_1 \geq b \textbf{ or } (l_1 \sim l_3 \textbf{ and } l_1 \geq l_3)$$

When $l_1 \geq b$ the result is evidently true, so we can assume $l_1 < b$. The first formula then proves that $l_1 \geq l_2$, which gives us $l_2 < b$. So we have eliminated $b$, and our result will follow if, given

$$l_1 \sim l_2 \textbf{ and } l_1 \geq l_2$$

and

$$l_2 \sim l_3 \textbf{ and } l_2 \geq l_3$$

we can prove

$$l_1 \sim l_3 \textbf{ and } l_1 \geq l_3$$

But this is trivial, because $\sim$ and $\geq$ are transitive, so their conjunction is transitive. Then *dom*, being the conjunction of *dom_max* and *dom_min*, is also transitive.

### C.4.2. Tradeoff dominance

Suppose that solution $S_1$ fails to dominate solution $S_2$, but only at one point along the signature, and only by 1. Suppose that the monitor $m$ at that point has weight $w$. The failure means that $m$ could have a cost in some extension of $S_1$ which is $w$ more than its cost in the corresponding extension of $S_2$ (assuming the cost function is not quadratic), which is why dominance fails.

But now, if $c(S_1) + w \leq c(S_2)$, this extra $w$ cannot make any extension of $S_2$ cost less than the corresponding extension of $S_1$. In other words, $S_1$ still dominates $S_2$ even though separate dominance fails at this one point.

This idea easily extends to differences greater than 1, and to multiple points along the signature. It is simply a matter, as we proceed along the signature, of adding to $c(S_1)$ the cost of overlooking each violation of separate dominance. Then, if $c(S_1)$ exceeds $c(S_2)$ at any point, dominance has failed. We call this *tradeoff dominance*, because it trades off an initial difference in cost against individual failures of dominance.

There may be places along the signature which are not directly associated with any monitor, or where the monitor's cost function is quadratic. So tradeoff dominance needs to trade off where it can, and fall back on separate dominance where it can't. Dominance still holds if it is justified by tradeoff dominance at some points and by separate dominance at the rest.

The full dominance test works as follows, remembering that separate dominance applies at every position along the signature, but tradeoff dominance only applies at some positions. First, initialize the *available cost* to $c(S_2) - c(S_1)$. Fail immediately if this quantity is negative. Next, for each position $i$ along the signature, do the following. If the separate dominance test succeeds at $i$, then do nothing more at $i$. Otherwise, if tradeoff dominance applies at $i$, then subtract the tradeoff amount from the available cost and fail if the available cost goes negative. Otherwise, at $i$ we have found that separate dominance fails and tradeoff dominance does not apply, so fail.

### C.4.3. Tabulated dominance

In this section we present *tabulated dominance*, the method we prefer when it applies. It is just tradeoff dominance treated formally so as to squeeze as much dominance as possible out of it.

Let the *domain* of a set of assignments $S$, written $dom(S)$, be the set of pairs $(d_i, r_j)$ such that an assignment containing day $d_i$ and resource $r_j$ appears in $S$. Tabulated dominance applies to pairs of solutions $S_1$ and $S_2$ such that $dom(S_1) = dom(S_2)$. It tries to show that for every complete extension $S'_2$ of $S_2$, the set of assignments defined by the formula

$$S'_1 = S_1 \cup (S'_2 - S_2)$$

is a complete solution and satisfies

$$c(S'_2) - c(S'_1) \; = \; \sum_{m \in M} (c(m, S'_2) - c(m, S'_1)) \; \geq \; 0$$

As before, we calculate $c(S'_2) - c(S'_1)$ rather than $c(S'_1) - c(S'_2)$ so that the available cost is non-negative when dominance is possible.

Our definition of $S'_1$ takes the assignments that convert $S_2$ into $S'_2$ and adds them to $S_1$. Our first task is to investigate what is involved in proving that $S'_1$ is a complete solution. Clearly, it is a set of assignments. At most one assignment containing a given $d_i$ and $r_j$ may appear in it, because if such an assignment appears in $S_1$, then an assignment for $d_i$ and $r_j$ also appears in $S_2$, since we are assuming $dom(S_1) = dom(S_2)$, and therefore not in $S'_2 - S_2$. $S'_1$ is also easily seen to be complete, as follows. Taking domains,

$$dom(S'_1) \; = \; dom(S_1) \cup (dom(S'_2) - dom(S_2)) \; = \; dom(S'_2)$$

and $S'_2$ is assumed complete. But the last condition, that for each mtask $C_{ik}$, the number of occurrences of $C_{ik}$ in $S'_1$ may not exceed the number of tasks in $C_{ik}$, does not always hold. It holds when $S_1$ and $S_2$ are $d_k$-solutions, because then all occurrences of $C_{ik}$ in $S'_1$ come either from $S_1$ or from $S'_2$; but in other cases further investigation will be needed.

Suppose now that we have been able to show that $S'_1$ is a complete solution. We now make the same transition from costs to extra costs that we made previously. Since

$$c(S_1) - \sum_{m \in M} c(m, S_1) \; = \; 0$$

and

$$c(S_2) - \sum_{m \in M} c(m, S_2) \; = \; 0$$

we can write

$$\begin{aligned}
c(S_2') - c(S_1') &= \sum_{m \in M} (c(m,S_2') - c(m,S_1')) \\
&= c(S_2) - \sum_{m \in M} c(m,S_2) - c(S_1) + \sum_{m \in M} c(m,S_1) + \sum_{m \in M} (c(m,S_2') - c(m,S_1')) \\
&= c(S_2) - c(S_1) \;+\; \sum_{m \in M} \big[(c(m,S_2') - c(m,S_2)) - (c(m,S_1') - c(m,S_1))\big]
\end{aligned}$$

and it is this quantity that we have to prove to be non-negative. For each monitor $m$ we need to find the difference between two extra costs.

In practice we obviously cannot visit every complete extension $S_2'$ of $S_2$. Instead, we calculate a (possibly negative) quantity $\Delta(m,S_1,S_2)$ satisfying

$$\Delta(m,S_1,S_2) \;\le\; \big[(c(m,S_2') - c(m,S_2)) - (c(m,S_1') - c(m,S_1))\big]$$

for all complete extensions $S_2'$ of $S_2$. If we do this for every $m \in M$ and find that

$$c(S_2) - c(S_1) \;+\; \sum_{m \in M} \Delta(m,S_1,S_2) \;\ge\; 0$$

then we will have shown

$$\begin{aligned}
c(S_2') - c(S_1') &= c(S_2) - c(S_1) \;+\; \sum_{m \in M} \big[(c(m,S_2') - c(m,S_2)) - (c(m,S_1') - c(m,S_1))\big] \\
&\ge c(S_2) - c(S_1) \;+\; \sum_{m \in M} \Delta(m,S_1,S_2) \;\ge\; 0
\end{aligned}$$

as required. Our dominance test, then, begins by setting the available cost to $c(S_2) - c(S_1)$. Then for each monitor $m$ in $M$, it obtains a value for $\Delta(m,S_1,S_2)$ somehow and adds it to the available cost. If the final sum is non-negative, dominance is proved.

When $S_1$ and $S_2$ are $d_k$-solutions we can greatly reduce the number of monitors we have to visit. In fact, we only have to visit monitors that contribute a value to the day $d_k$ signatures. To prove this, let $m$ be any monitor which does not contribute to the signatures on day $d_k$. This means that $m$'s active interval, $[d_a,\dots,d_b]$, is such that either $k < a$ or $k \ge b$.

Suppose first that $k < a$. Then $m$ is unaffected by whatever happens on the days up to and including $d_k$, so $c(m,S_1) = c(m,S_2) = c(m,S_0)$, the initial cost of $m$. And the assignments that affect $m$ in $S_1'$ and $S_2'$ are the same, so $c(m,S_1') = c(m,S_2')$. So

$$\begin{aligned}
&(c(m,S_2') - c(m,S_2)) - (c(m,S_1') - c(m,S_1)) \\
&\quad = (c(m,S_2') - c(m,S_1')) - (c(m,S_2) - c(m,S_1)) \;=\; 0 - 0 \;=\; 0
\end{aligned}$$

and so there is no need to visit $m$.

Now suppose that $k \ge b$. Then $m$'s cost attains its final value on or before $d_k$, that is, within $S_1$ and $S_2$. So $c(m,S_1') = c(m,S_1)$ and $c(m,S_2') = c(m,S_2)$, and once again $m$'s contribution to the available cost is 0 and there is no need to visit $m$. That ends the proof.

So far we have merely formalized ideas already found in tradeoff dominance: subtracting tradeoffs from available cost, and visiting only monitors with a presence in the signature. But our algebra applies to all monitors $m$, not just those whose cost functions are not quadratic, so we are in a much better position to calculate tradeoffs, now represented by the $\Delta(m, S_1, S_2)$. And as we will see when we come to instantiate this method for counter and sequence monitors, it will be possible to tabulate values of $\Delta(m, S_1, S_2)$ in advance of solving, so that the appropriate change to available cost can be obtained very quickly by a table lookup.

### C.4.4. Early termination of the tabulated dominance test

Suppose we can prove that $\Delta(m, S_1, S_2) \leq 0$ for all $m$, $S_1$, and $S_2$. Then as we scan along the signature, if the available cost ever becomes negative we can rule out dominance immediately, because no subsequent additions could make it non-negative again.

The condition $\Delta(m, S_1, S_2) \leq 0$ holds most of the time but not always. When $S_1$ and $S_2$ are $d_k$-solutions, it always holds for counter monitors, as proved in Appendix C.5.5, and it holds most of the time, but not always, for sequence monitors, as proved in Appendix C.7.5.

By ruling out dominance immediately when the available cost becomes negative, as we choose to do, we may be failing to find dominance in some cases where it exists. This does not invalidate the overall algorithm, it merely means that the tables $P_k$ of undominated solutions might be somewhat larger than they need to be. Since there are always many fewer sequence monitors than counter monitors, and violations of the condition seem to be few and small, the time saved by returning from many dominance tests early more than compensates for any slight increase in the sizes of the tables $P_k$.

### C.4.5. Correlated dominance

Sometimes two expressions are *correlated*: they do not have independent values, because they measure related things. The chance of detecting dominance may be increased by recognizing and taking advantage of correlated expressions.

We give two real-world examples. The first is the case, occurring only on Saturdays, of the expression which has value 1 if resource $r$ is busy that day, and its parent, the monitor whose cost depends on the number of $r$'s busy weekends.

If we do not recognize that these two expressions are correlated, we have to use two separate tests. With correlation, we use the value of the first expression to form a more accurate estimate of the current number of busy weekends than just the second expression. There are two variants of this idea, that we call *Type 1* and *Type 2* correlations. Appendix C.5.6 has the details.

The second example concerns a monitor $m_1$ which limits the number of consecutive busy days, and a monitor $m_2$ which limits the number of consecutive free days. The limits do not have to be equal, nor do the weights.

The next assignment creates either a busy day or a free day, but not both. If we treat the two monitors independently, we could pay a cost in one for a busy day and a cost in the other for a free day, when only one of the two can occur. Again there are two variants of this idea; we call them *Type 3* and *Type 4* correlations. Appendix C.7.6 has the details.

## C.5. Counter monitors

This section is devoted to counter monitors: defining them; defining their cost, both in complete and in incomplete solutions; defining a suitable $\Delta(m, S_1, S_2)$ for them; and proving that $\Delta(m, S_1, S_2) \leq 0$. The following sections carry out the same programme for sum expressions and sequence monitors.

### C.5.1. Definition

We mentioned earlier that in the implementation, a monitor is the root node of an expression tree. A counter monitor is a root node with a set of children. Each child represents one shift, or weekend, or whatever is being counted, and has a value, which may be 0 or 1. Depending on a given solution $S$, each child is in one of three states:

1. It is *inactive* if its value in $S$ is known, and that value is 0;
2. It is *active* if its value in $S$ is known, and that value is 1;
3. It is *unassigned* if its value in $S$ is not known.

It will be unassigned if its value is affected by what happens on one of the selected days, and the search is not up to that day yet. We use *assigned* to mean inactive or active. As the search progresses, along any one search path the unassigned children gradually change to assigned.

XESTT has a feature, referred to generally as *history*, which helps to meld the solution for the time interval covered by the current instance with solutions for preceding and following time intervals. This is done by means of additional children, not given explicitly in the definition of the constraint, but rather represented implicitly by these *history attributes*:

- A non-negative integer *history before* value $a_i$, representing the number of children in the preceding interval.
- An integer *history* value $x_i$ satisfying $0 \leq x_i \leq a_i$, representing the number of children in the preceding interval which are active.
- A non-negative integer *history after* value $c_i$, representing the number of children in the following interval.

These are constant attributes: they are independent of any solution. The subscript $i$ means nothing here; it is the notation used by a previous work on history [10].

What happened in the preceding interval is assumed to be finalized, known, and expressed by $x_i$. What will happen in the following interval is assumed to be undecided; all that is known about that interval is the number of children it contains, $c_i$. For convenience we will assume that all monitors have $a_i$, $x_i$, and $c_i$; if they are absent, it changes nothing to add them with value 0.

A counter monitor $m$ places lower and upper limits on the number of active children in the full interval (preceding plus current plus following). The $x_i$ active children in the preceding interval are essentially the same as active children in the current interval, and the $c_i$ children in the following interval are essentially the same as unassigned children in the current interval. But even in a complete solution, the $c_i$ children in the following interval remain unassigned.

The *determinant* of counter monitor $m$ is the number compared with the limits to see if there is any excess or deficiency. The presence of unassigned children, when the solution is incomplete or $c_i > 0$, makes this rather indeterminate. Instead of trying to pin it down exactly, we bound it:

- The *lower determinant* of counter monitor $m$ in solution $S$, written $l(m,S)$ or just $l$, is the number of active children of $m$ in $S$ over the full interval (i.e. including $x_i$).
- The *upper determinant* of counter monitor $m$ in solution $S$, written $u(m,S)$ or just $u$, is the number of active and unassigned children over the full interval (i.e. including $x_i$ and $c_i$).

The point here is that $l(m,S)$ is a lower bound on the number of active children there could ever be, and $u(m,S)$ is an upper bound on this number. We have $l(m,S) \le u(m,S)$, in fact we have $l(m,S) \le u(m,S) - c_i$, and also

$$\begin{aligned} &l(m,S) \le l(m,S') && \text{for all extensions } S' \text{ of } S \\ &u(m,S) \ge u(m,S') && \text{for all extensions } S' \text{ of } S \end{aligned}$$

These conditions are natural because as solutions become more complete, the number of unassigned children decreases so our information about the determinant increases.

A counter monitor has many constant attributes in addition to its three history attributes:

- An integer *minimum determinant* $D_l$ which is a lower limit on $l(m,S)$. XESTT constraints do not give $D_l$ explicitly, but there is an obvious value to choose: $x_i$.
- An integer *maximum determinant* $D_u$ which is an upper limit on $u(m,S)$. Again, XESTT constraints do not give $D_u$ explicitly, but again there is an obvious value to choose: the number of children in the current interval, plus $x_i$, plus $c_i$.
- An integer *lower limit* $L$. A constraint without a lower limit is assigned lower limit $D_l$, and a lower limit below $D_l$ is increased to $D_l$. These changes do not change the meaning.
- An integer *upper limit* $U$. A constraint without an upper limit is assigned upper limit $D_u$, and an upper limit above $D_u$ is decreased to $D_u$. These changes do not change the meaning. The condition $L \le U$ must hold, otherwise the instance is invalid.
- A Boolean *allow zero flag* $Z$, indicating that if the determinant is 0, the cost should be 0 regardless of the limits $L$ and $U$.
- A *cost function* $f(x)$, which could be *linear* ($f(x) = wx$ for a given non-negative constant $w$), *quadratic* ($f(x) = wx^2$), or *step* ($f(x) = \chi(x = 0, 0, w)$). Here and elsewhere we use the notation $\chi(c, x, y)$ to mean '**if** $c$ **then** $x$ **else** $y$'.

All three cost functions $f$ have two key properties: $f(0) = 0$, and $f$ is monotone non-decreasing. Our work on counter and sum monitors assumes only these two properties, so it handles any cost function that has them. Later, when we come to sequence monitors, we will find an obscure case where we need to make a special arrangement when $f$ is a step function.

### C.5.2. Deviation and cost

This section defines the cost of a counter monitor, and shows that our definition has the right properties: it is always non-negative, it agrees with the XESTT definition for complete solutions, and it is monotone non-decreasing as solutions become more complete.

We generalize the well-known two-step procedure for calculating cost. The first step is to produce a *deviation* $\delta$, the amount by which the number of active children falls short of the lower limit $L$ or exceeds the upper limit $U$. Including the allow zero flag $Z$, which when set causes the deviation to be 0 when the number of active children is zero regardless of the limits, we have

$$\delta(l,u) \;=\; \chi(Z \wedge l = 0, 0, \max(0, L - u, l - U))$$

where $l$ is the lower determinant and $u$ is the upper determinant. The rationale is that $\delta(l,u)$ needs to be a lower bound on the deviation over the full interval (preceding, current, and following). When $Z \wedge l = 0$ is true, the allow zero flag is set and the number of active children we are certain about is 0, so we cannot justify any deviation larger than 0. When $Z \wedge l = 0$ is false, we can forget the allow zero flag. When finding the deviation from $L$, we need to assume that the number of active children is as large as possible, and when finding the deviation from $U$, we need to assume that the number of active children is as small as possible. $L$ and $U$ apply to the full interval.

The second step is to apply the cost function to obtain a cost:

$$c(m,S) = f(\delta(l,u))$$

The $c$ in $c(m,S)$ has nothing to do with the history value $c_i$. All values are non-negative integers.

In a complete solution the lower determinant is the number of active children in the current interval plus $x_i$, and the upper determinant is the lower determinant plus $c_i$. Given this, it is easy to check that our formula agrees with the XESTT one for complete solutions, as it must.

At most one of $L - u$ and $l - U$ can be positive. We prove this by showing that their sum is not positive: $(L - u) + (l - U) = (L - U) + (l - u) \leq 0$ since $L \leq U$ and $l \leq u$. Another useful result is that if $L - u \leq 0$ and $l - U \leq 0$, then $\delta(l,u) = 0$ and $f(\delta(l,u)) = 0$.

We may choose to assume $Z \Rightarrow L \geq 2$, for the following reason. If $Z$ is true and $L = 0$, we can change $Z$ to false without changing cost. If $Z$ is true and $L = 1$, we can change $Z$ to false and $L$ to 0 without changing cost. These changes never increase $L$, so they do not interact with $U$.

We now show that our cost formula $c(m,S) = f(\delta(l,u))$ is monotone non-decreasing as solutions become more complete. All cost functions $f$ are monotone non-decreasing, so we just have $\delta$ to consider. Omitting the allow zero flag for the moment, this is:

$$\delta(l,u) \;=\; \max(0, L - u, l - U)$$

As solutions become more complete, $l$ and hence $l - U$ is monotone non-decreasing, and $u$ is monotone non-increasing (see above) so $L - u$ is monotone non-decreasing.

Now consider the full formula, including the allow zero flag:

$$\delta(l,u) \;=\; \chi(Z \wedge l = 0, 0, \max(0, L - u, l - U)))$$

When $Z$ is false this is unchanged. When $Z$ is true, its effect is to possibly reduce $\delta(l,u)$ for the most incomplete solutions (those with $l = 0$), so $\delta(l, u)$ remains monotone non-decreasing as solutions become more complete.

During the construction of a new solution $S'$ by extending a given solution $S$, an active counter monitor $m$ needs to calculate $l(m, S)$, $u(m, S)$, $l(m, S')$, and $u(m, S')$. Given these four values, it is trivial to find the extra cost $c(m, S') - c(m, S)$ and the new saved value $l(m, S')$.

To find $l(m, S)$, we either use an initial value (if this is the first day that affects $m$), or retrieve it from the signature of $S$ (on days after the first). We find $u(m, S)$ by adding to $l(m, S)$ the number of unassigned children. This number is easily found because it depends on the domain of $S$, not its assignments: it is the number of children affected by days after $S$, plus $c_i$.

To find $l(m, S')$, we add to $l(m, S)$ the number of children that are unassigned in $S$ but active in $S'$. To find $u(m, S')$, we subtract from $u(m, S)$ the number of children that are unassigned in $S$ but inactive in $S'$. Both parts are carried out in the course of a single traversal of the children that are known to be unassigned in $S$ but assigned in $S'$. These children depend only on the domains of $S$ and $S'$, so they are easy to find.

### C.5.3. Separate dominance

In practice we prefer to use tabulated dominance for counter monitors, but for completeness we record here what needs to be done to support separate dominance.

Separate dominance depends on function $very\_small(l) = (l \leq a)$, which needs to be true when $l$, the value retrieved from the signature, is so small that there can be no extension in which it violates the maximum limit. We need to find a suitable value for $a$.

Now $l$ contains the number of active children in the current solution, plus the $x_i$ history children. Potentially it could increase by as much as $e$, the number of unassigned children still to come, a constant which is known when the dominance test is created, because it depends on which day we are up to. There is no prospect of exceeding the upper limit $U$ when $l + e \leq U$, and so we may take $a = U - e$ (this could be negative). This works when there is no upper limit, because then $U$ has value `INT_MAX` and the condition $l \leq a$ is always true, as required.

Separate dominance also requires a function $very\_large(l) = (l \geq b)$, which needs to be true when $l$ is so large that there can be no extension in which it violates a minimum limit. We need to find a suitable value for $b$.

Now $l$ contains the number of active children in the current solution, plus the $x_i$ history children. Potentially it might not increase at all, so there is no prospect of falling short of the minimum limit $L$ when $l \geq L$, and so we may take $b = L$. This works when there is no lower limit, because then $L$ has value 0 and the condition $l \geq b$ is always true, as required.

***OR and AND expressions.*** Counter monitors often have children which are *OR* (or *AND*) expressions, whose value is 1 when the resource is busy (or free) during some set of times. If the times span more than one day (for example, if they are the times of a weekend), these children will also have signature values.

*OR* and *AND* expressions are not suited to tradeoff dominance, because they have no cost. We prefer to use correlated dominance, but that is not always possible (in the case of *AND* expressions, because the analysis for it has not been a priority). So, as usual, we need to be able to fall back on separate dominance.

For an *OR* expression, the signature value is 1 if any of the expression's children are assigned with value 1, and 0 otherwise. For an *AND* expression, the signature value is 0 if any of the expression's children are assigned with value 0, and 1 otherwise.

Although these expressions are children of the counter monitor, not the monitor itself, a larger value for a child has much the same effect as a larger value for the parent. So if the parent has a maximum limit we want *very_small*($l$) to be false, so that dominance includes the test $l_1 \leq l_2$. Accordingly, we set $a$ to $-1$ when the counter monitor has a maximum limit, and to a very large number when it doesn't.

In the same way, if the parent has a minimum limit we want *very_large*($l$) to be false, so that dominance includes the test $l_1 \geq l_2$. So we set $b$ to a very large number when the counter monitor has a minimum limit, and to 0 when it doesn't.

### C.5.4. Tabulated dominance

A review of Section C.4.3 will show that what we need to do to incorporate a counter monitor $m$ into our tabulated dominance test is to define a $\Delta(m, S_1, S_2)$ satisfying

$$\Delta(m, S_1, S_2) \;\leq\; \left[c(m, S'_2) - c(m, S_2)\right] - \left[c(m, S'_1) - c(m, S_1)\right]$$

for all complete extensions $S'_2$ of $S_2$, where $S'_1 = S_1 \cup (S'_2 - S_2)$. Now assuming that $m$ is a counter monitor, we have

$$c(m, S) = f(\delta(l(m, S), u(m, S)))$$

for all solutions $S$, complete or incomplete. For readability, we define $l_1 = l(m, S_1)$, $l'_1 = l(m, S'_1)$, $u_1 = u(m, S_1)$, and $u'_1 = u(m, S'_1)$, and similarly for $S_2$. Here $u'_1 = l'_1 + c_i$ and $u'_2 = l'_2 + c_i$, because $S'_1$ and $S'_2$ are complete.

Now $S_1$ and $S_2$ have equal domains, so they leave the same children of $m$ unassigned. Let $e$ be the number of unassigned children of $m$ in $S_1$ (or equally in $S_2$) in the current interval (i.e. not counting history after children). We have $e \geq 0$ and $u_1 - l_1 = u_2 - l_2 = e + c_i$.

Now since $S'_1 - S_1$ and $S'_2 - S_2$ are equal, their contributions to the determinant are equal. (We are ignoring cases like busy weekends, where unassigned children are partly influenced by $S_1$ and $S_2$; we handle those in Appendix C.5.6.) This common contribution can be at most $e$, which occurs when every child left unassigned by $S_1$ and $S_2$ in the current interval (i.e. not including history after children) is made active. Letting this contribution be $y$, where $0 \leq y \leq e$, we get $l'_1 = l_1 + y$, $l'_2 = l_2 + y$, $u'_1 = l'_1 + c_i = l_1 + y + c_i$, and $u'_2 = l'_2 + c_i = l_2 + y + c_i$. So

$$\begin{aligned}\Delta(m, S_1, S_2) &\leq [c(m, S'_2) - c(m, S_2)] - [c(m, S'_1) - c(m, S_1)]\\ &= [f(\delta(l'_2, u'_2)) - f(\delta(l_2, u_2))] - [f(\delta(l'_1, u'_1)) - f(\delta(l_1, u_1))]\\ &= [f(\delta(l_2 + y, l_2 + y + c_i)) - f(\delta(l_2, l_2 + e + c_i))] - [f(\delta(l_1 + y, l_1 + y + c_i)) - f(\delta(l_1, l_1 + e + c_i))]\end{aligned}$$

for all $y$. We write this last formula as $\Psi_y(e, l_1, l_2)$. This is a reasonable notation because the other quantities that go into it ($c_i$ and the attributes that define $\delta$ and $f$) are constants. This gives us

$$\Delta(m, S_1, S_2) \ \leq \ \min_{0 \leq y \leq e} \Psi_y(e, l_1, l_2) \ = \ \Psi(e, l_1, l_2)$$

say. Finding an algebraic simplification for $\Psi$ is complicated and might require approximation. Instead, we actually evaluate it and use the result as our $\Delta(m, S_1, S_2)$. We build a cache holding these values, indexed by the triple $(e, l_1, l_2)$. These three indexes are easily obtained: $e$ is just the number of unassigned children in the current interval, which is the same for all solutions with the same domain, $l_1$ comes from the signature of $S_1$, and $l_2$ comes from the signature of $S_2$. So the dominance test at $m$ is very fast and simple: find these three numbers, use them to retrieve $\Psi(e, l_1, l_2)$ from the cache, and add that amount to the available cost.

We need to work out the ranges of the three indexes. Let $A$ be the number of children of $m$, excluding the $x_i$ history children and the $c_i$ history after children. Then $0 \leq e \leq A$. Then for each of these values of $e$, we have $x_i \leq l_1 \leq A + x_i - e$, since $l_1$ is a number of active children including the $x_i$ history children, and similarly $x_i \leq l_2 \leq A + x_i - e$. Taking care to offset the $l_1$ and $l_2$ indexing by $x_i$, the cache has size $O(A^3)$, which is acceptable given that we build the cache just once, when we initialize the solver.

The value of $e$ is the same for all $d_k$-solutions. So within the object representing day $d_k$ we can store the part of the cache that relates to $e$. Then when we consult the cache we just access this part of it, supplying it with index pair $(l_1, l_2)$.

### C.5.5. Early termination of the tabulated dominance test

Appendix C.4.4 promised that we would prove $\Delta(m, S_1, S_2) \leq 0$, so we do that now. This is equivalent to saying that for each $(e, l_1, l_2)$ triple there exists a $y$ in the range $0 \leq y \leq e$ such that

$$[f(\delta(l_2 + y, l_2 + y + c_i)) - f(\delta(l_2, l_2 + e + c_i))] - [f(\delta(l_1 + y, l_1 + y + c_i)) - f(\delta(l_1, l_1 + e + c_i))] \ \leq \ 0$$

Since the second main term in this formula is an extra cost, it is non-negative, and so it is sufficient to find a $y$ in the range $0 \leq y \leq e$ such that

$$f(\delta(l_2 + y, l_2 + y + c_i)) - f(\delta(l_2, l_2 + e + c_i)) \ \leq \ 0$$

This is what we will do.

In some cases we use a shortcut: we are able to find a $y$ in the range $0 \leq y \leq e$ such that $\delta(l_2 + y, l_2 + y + c_i) = 0$. That is all we need, because then $f(\delta(l_2 + y, l_2 + y + c_i)) = f(0) = 0$, and since $f(\delta(l_2, l_2 + e + c_i)) \geq 0$ we are done.

We prove our result using four cases. Let $K = L - l_2 - c_i$.

**Case 1:** $Z$ is true and $l_2 = 0$. Choose $y = 0$. This gives

$$\begin{aligned} \delta(l_2 + y, l_2 + y + c_i) \ &= \ \delta(l_2, l_2 + c_i) \\ &= \ \chi(Z \wedge l_2 = 0, 0, \max(0, L - l_2 - c_i, l_2 - U)) \\ &= \ 0 \end{aligned}$$

taking the first branch of the $\chi$. The shortcut applies so we are done for this case.

From now on we can ignore $Z$ and use the simpler formula

$$\delta(l,u) = \max(0, L-u, l-U)$$

We will do this in the remaining cases without further mention.

**Case 2:** $K \le 0$. Choose $y = 0$. This gives

$$\begin{aligned}\delta(l_2+y, l_2+y+c_i) &= \delta(l_2, l_2+c_i)\\ &= \max(0, L-l_2-c_i, l_2-U)\\ &= \max(0, l_2-U)\end{aligned}$$

because $L-l_2-c_i = K \le 0$. And

$$\begin{aligned}\delta(l_2, l_2+e+c_i) &= \max(0, L-l_2-e-c_i, l_2-U)\\ &= \max(0, l_2-U)\end{aligned}$$

because $L-l_2-e-c_i = K-e \le 0$. Putting these results together, we get

$$f(\delta(l_2+y, l_2+y+c_i)) - f(\delta(l_2, l_2+e+c_i)) = f(\max(0, l_2-U)) - f(\max(0, l_2-U)) = 0$$

So we have our result in Case 2.

**Case 3:** $K > 0$ and $e \le K$. Choose $y = e$. This gives

$$\begin{aligned}\delta(l_2+y, l_2+y+c_i) &= \delta(l_2+e, l_2+e+c_i)\\ &= \max(0, L-l_2-e-c_i, l_2+e-U)\\ &= \max(0, K-e, l_2+e-U)\\ &= \max(0, K-e)\end{aligned}$$

because

$$\begin{aligned}l_2+e-U &\le l_2+K-U\\ &= l_2+(L-l_2-c_i)-U\\ &= L-c_i-U\\ &\le 0\end{aligned}$$

since $L \le U$. And

$$\begin{aligned}\delta(l_2, l_2+e+c_i) &= \max(0, L-l_2-e-c_i, l_2-U)\\ &= \max(0, K-e, l_2-U)\\ &= \max(0, K-e)\end{aligned}$$

because $l_2 - U \le l_2 + e - U \le 0$ as we just saw. So

$$f(\delta(l_2+y, l_2+y+c_i)) - f(\delta(l_2, l_2+e+c_i)) \;=\; f(\max(0, K-e)) - f(\max(0, K-e)) \;=\; 0$$

So we have our result in Case 3.

**Case 4:** $K > 0$ and $e > K$. Choose $y = K$, which is legal since $0 < K < e$. This gives

$$\begin{aligned}\delta(l_2+y, l_2+y+c_i) &= \delta(l_2+K, l_2+K+c_i)\\ &= \max(0, L-l_2-K-c_i, l_2+K-U)\\ &= 0\end{aligned}$$

because we can simplify the second term: $L - l_2 - K - c_i = K - K = 0$; and we can simplify the third term: $l_2 + K - U = l_2 + (L - l_2 - c_i) - U = L - c_i - U \le 0$. The shortcut applies and we have our result in Case 4. This completes our proof that $\Delta(m, S_1, S_2) \le 0$.

## C.5.6. Correlated expressions

Here we work out the algebra for the Type 1 and Type 2 correlations from Appendix C.4.5, involving a monitor $m$ counting the number of busy weekends for some resource $r$, and its child reporting whether $r$ is busy on some Saturday. This analysis is relevant to any counter monitor's child whose value depends on more than one day, making it contribute a value to the signature. However for concreteness we will continue to think and speak of busy weekends and Saturdays. At the end of this section we will consider how general our analysis has turned out to be.

Suppose that the open days include an adjacent Saturday and Sunday. Let the signature values for $S_1$ on the Saturday be $(a_1, l_1)$, where $a_1$ is 1 if $r$ is busy on the Saturday, or else 0, and $l_1$ is the number of busy weekends before this *current weekend*. Let $(a_2, l_2)$ be the same values, only for $S_2$. Without correlation, the value we need for adding to the available cost is

$$\Psi(e, l_1, l_2) \;=\; \min_{0 \le y \le e} \Psi_y(e, l_1, l_2)$$

where $e$ is the number of unassigned children, that is, the number of weekends whose busyness is not yet known. On the Saturday, without correlation, that includes the current weekend. Referring back to Appendix C.5.4, we see that $\Psi_y(e, l_1, l_2)$ is

$$\begin{aligned}&[f(\delta(l_2+y, l_2+y+c_i)) - f(\delta(l_2, l_2+e+c_i))] - [f(\delta(l_1+y, l_1+y+c_i)) - f(\delta(l_1, l_1+e+c_i))]\\ &\quad= \Gamma_y(e, l_2) - \Gamma_y(e, l_1)\end{aligned}$$

say, letting $\Gamma_y(e, l) = f(\delta(l+y, l+y+c_i)) - f(\delta(l, l+e+c_i))$.

Adding correlation gives access to a child whose busyness is not completely known, showing that the condition $e \ge 0$ should be replaced by $e \ge 1$. Its values $a_1$ and $a_2$ in $S_1$ and $S_2$ become available when calculating $\Psi$. If either or both is 1 we have advance knowledge that the current weekend, whose busyness without correlation is unknown, is in fact busy.

Our previous algebra assumed that $y$, the number of busy weekends still to come, was the same in $S_1'$ and $S_2'$. But on Saturdays, this number can also be one greater in $S_1'$ than in $S_2'$ (when $a_1 = 1$ and $a_2 = 0$), or one greater in $S_2'$ than in $S_1'$ (when $a_2 = 1$ and $a_1 = 0$). So our first step is to divide $y$ into two variables: $y_1$, the number of busy weekends still to come in $S'_1$, and $y_2$, the number of busy weekends still to come in $S'_2$. We can express the correlated case by

$$\Psi(e, a_1, a_2, l_1, l_2) \;=\; \min_{\substack{a_1 \le y_1 \le e \\ a_2 \le y_2 \le e \\ y_1 - y_2 \in D(a_1, a_2)}} [\Gamma_{y_2}(e, l_2) - \Gamma_{y_1}(e, l_1)]$$

The conditions $a_1 \le y_1$ and $a_2 \le y_2$ say that if the current Saturday is busy, then there must be at least one busy weekend beyond the $l_1$ and $l_2$ previous busy weekends. The condition $e \ge 1$ is important here, because the iteration can be empty, leaving the value undefined, if $e = 0$.

Now $|y_1 - y_2| \le 1$, since only the current weekend can differ between $S'_1$ and $S'_2$. Also, $a_1 = a_2$ implies $y_1 = y_2$. The set $D(a_1, a_2)$ gives the precise choices for $y_1 - y_2$, depending on $a_1$ and $a_2$:

| $a_1$ | $a_2$ | $D(a_1, a_2)$ |
|---|---|---|
| 0 | 0 | $\{0\}$ |
| 0 | 1 | $\{-1, 0\}$ |
| 1 | 0 | $\{0, 1\}$ |
| 1 | 1 | $\{0\}$ |

For example, when $a_1 = 0$ and $a_2 = 1$, there could be one more busy weekend in $S'_2$ than in $S'_1$ (this will happen when the Sunday is free), and so $y_1 - y_2$ could be $-1$.

We build a table in the usual way to cache this version of $\Psi$. It has five indexes rather than three, but it is not a large table, because $a_1 \in \{0, 1\}$ and $a_2 \in \{0, 1\}$. For efficiency, initializing one entry is done by iterating over $y_1$ and the elements of $D(a_1, a_2)$, not over $y_1$ and $y_2$.

That ends our analysis. Now we return to the issue we raised at the start. How general is our analysis? Are there special cases or generalizations that we need to think about?

To begin with, there is a problem on the first open Saturday: the parent does not have a value in the signature on that day. But the problem can be overcome, as follows.

If the parent did have a value in the signature, it would be its initial value, which is its history value plus its number of busy weekends in parts of the cycle that have not opened. These values are stored in the parent, and the child can easily access them there. So for the first week (i.e. for any child that remains unmatched with its parent because the parent is not in the signature), the child can carry out the same test, obtaining $l_1$ and $l_2$ from its parent rather than from the signature. In this case, $l_1$ and $l_2$ are equal, but that does not seem to either help or hinder.

If the Saturday is the last open day, all signatures are empty and dominance testing is just simple cost comparison, so there is nothing to worry about. If the Sunday is the last open day, the Saturday is not a special case and our regular analysis applies. This just leaves the case where the Saturday is open, the Sunday is not open, but there are other open days later in the cycle. (Our algorithm allows several disconnected intervals within the cycle to be opened.)

This too is not a special case. The child has a value for the Saturday, and its Sunday part is constant, so it can and does report its final value to its parent on the Saturday. So such children have no value in the signature, and they are not candidates for correlation.

We also need to think about different kinds of children. Given that we are not dealing with arbitrary expression trees (because our trees are derived from XESTT constraints), a consideration of cases will show that the only possibilities are that the child could depend on more than two days, and that it could be testing whether the resource is free rather than busy.

A careful review will show that our analysis holds for a child that depends on more than two days, replacing ‘Saturday’ by ‘all days that have a value’ and ‘Sunday’ by ‘all days that do not yet have a value’, where both sets are non-empty, and the current day is one that has a value.

Our analysis does not hold for counting free weekends. Although a corresponding analysis could easily be constructed for that case, that has not been done, so the implementation does no correlating when the child reports free weekends rather than busy ones.

A *Type 1* correlation is one where both the child and the parent have an entry in the signature. A *Type 2* correlation is one where the child has an entry but the parent does not.

## C.6. Sum monitors

This section is devoted to *sum monitors*. These allow each assigned child to have a value which is any non-negative number. The determinant is the sum of the assigned children’s values; a deviation and cost are produced from the determinant in the usual way.

We also allow for *sum expressions*, which omit the cost calculation. Instead of a cost, sum expressions produce a value, which is the deviation. This form of the general idea is needed by the implementation of the XESTT limit busy times and limit workload monitors.

Sum monitors generalize counter monitors: a counter monitor is a sum monitor in which the value of each assigned child is limited to either 0 or 1. Much of the algebra for sum monitors follows the algebra for counter monitors. We have chosen not to subsume counter monitors into sum monitors because the limited range of counter monitor values makes counter monitors better suited to tabulated dominance than sum monitors are. Indeed the implementation uses tabulated dominance for counter monitors but separate dominance for sum monitors.

### C.6.1. Definition

A *sum monitor* is a monitor whose determinant is the sum of its children’s values. It calculates a deviation based on this determinant in the usual way, and optionally a cost from the deviation.

The value of each child, and the value of the sum, are *numeric*, by which we mean that they may have type `int` or `float`. The type is `float` when we use sum monitors to model XESTT limit workload monitors, and `int` otherwise. Deviations, however, are always integers, as the documentation of the limit workload constraint explains.

In the implementation, type `KHE_DRS_EXPR_SUM_INT` represents sum monitors with `int` values, and type `KHE_DRS_EXPR_SUM_FLOAT` represents sum monitors with `float` values. Apart from the value type these are identical, so this section describes both.

Within a given solution $S$, each child $x$ is either *assigned*, in which case it has a numeric *value* $v(x, S) \geq 0$, or else it is *unassigned*, in which case it has no value.

Although an unassigned child has no value, each child $x$ has a *value upper bound* $u(x)$ which is an upper bound on its value: $v(x, S) \leq u(x)$ for all $S$. For example, if we use a sum monitor to model a counter monitor, we would have $u(x) = 1$ for each child. If we are modelling a limit workload monitor, and each child represents what one nurse is doing at one time or on one day, we would set $u(x)$ to the largest possible workload available at that time or on that day. And so on. It turns out that for every child it is easy to calculate a reasonable value upper bound before solving begins. The solver does this.

We can include history in sum monitors by defining these generalizations of the three history values from counter monitors:

- A non-negative numeric *history before* value $a_i$, representing the sum of the value upper bounds of the children in the preceding interval.
- A numeric *history* value $x_i$ satisfying $0 \le x_i \le a_i$, representing the sum of the values of the children in the preceding interval. These children are assumed to be assigned, as usual.
- A non-negative numeric *history after* value $c_i$, representing the sum of the value upper bounds of the the children in the following interval. We can't sum their values because these children are assumed to be unassigned, as usual.

These definitions agree with those used with counter monitors, given that when using a value sum monitor to model a counter monitor we would set $u(x) = 1$ for each child $x$.

A sum monitor places lower and upper limits on the total value of the children in the full interval (preceding plus current plus following). The $x_i$ value from children in the preceding interval is essentially the same as the values of children in the current interval, and the $c_i$ value from the following interval is essentially the same as the value upper bounds of unassigned children in the current interval.

The *determinant* of sum monitor $m$ is the number compared with the limits to see if there is any excess or deficiency. As defined earlier, this is the sum of the values of the children. The presence of unassigned children, when the solution is incomplete or $c_i > 0$, makes this rather indeterminate, so instead of trying to pin it down exactly, we bound it:

- The *lower determinant* of sum monitor $m$ in solution $S$, written $l(m,S)$ or just $l$, is the sum of the values of the assigned children of $m$ in $S$, including $x_i$.
- The *upper determinant* of sum monitor $m$ in solution $S$, written $u(m,S)$ or just $u$, is $l(m,S)$ plus the sum of the value upper bounds of the unassigned children, including $c_i$.

Clearly $l(m,S)$ is a lower bound on the determinant, and $u(m,S)$ is an upper bound on the determinant. We have $l(m,S) \le u(m,S)$, in fact we have $l(m,S) \le u(m,S) - c_i$, and also

$$\begin{array}{ll} l(m,S) \le l(m,S') & \text{for all extensions } S' \text{ of } S \\ u(m,S) \ge u(m,S') & \text{for all extensions } S' \text{ of } S \end{array}$$

These conditions are natural because as solutions become more complete, the number of unassigned children decreases so our information about the determinant increases.

A sum monitor has many constant attributes in addition to its history attributes:

- A numeric *minimum determinant* $D_l$ which is a lower limit on $l(m,S)$. XESTT constraints do not give $D_l$ explicitly, but there is an obvious value to choose: $x_i$.

- A numeric *maximum determinant* $D_u$ which is an upper limit on $u(m, S)$. Again, XESTT constraints do not give $D_u$ explicitly, but again there is an obvious value to choose: the sum of the value upper bounds of the children in the current interval, plus $x_i$, plus $c_i$.

- A numeric *lower limit* $L$. A constraint without a lower limit is assigned lower limit $D_l$, and a lower limit below $D_l$ is increased to $D_l$. These changes do not change the meaning.

- A numeric *upper limit* $U$. A constraint without an upper limit is assigned upper limit $D_u$, and an upper limit above $D_u$ is decreased to $D_u$. These changes do not change the meaning. The condition $L \leq U$ must hold, otherwise the instance is invalid.

- A Boolean *allow zero flag* $Z$, indicating that if the determinant is 0, the cost should be 0 regardless of the limits $L$ and $U$.

- A *cost function* $f(x)$, which could be *linear* ($f(x) = wx$ for a given non-negative constant $w$), *quadratic* ($f(x) = wx^2$), or *step* ($f(x) = \chi(x = 0, 0, w)$). Here and elsewhere we use the notation $\chi(c, x, y)$ to mean '**if** $c$ **then** $x$ **else** $y$'. We sometimes need to calculate a deviation without also reporting a cost; in those cases it will be easy to omit the cost calculation.

All three cost functions $f$ satisfy $f(0) = 0$, and $f$ is monotone non-decreasing. Our work on sum monitors assumes only these two properties, so it handles any cost function that has them.

### C.6.2. Deviation and cost

This section defines the deviation and cost of a sum monitor precisely, and shows that they are always non-negative, have the expected values in complete solutions, and are monotone non-decreasing as solutions become more complete.

We proceed in the usual way, only allowing for `float` values. The first step is to produce a *deviation* $\delta$, the amount by which the sum of the values of the children falls short of the lower limit $L$ or exceeds the upper limit $U$. Including the allow zero flag $Z$, which when set causes the deviation to be 0 when the sum is zero regardless of the limits, we have

$$\delta(l, u) \;=\; \chi(Z \wedge l = 0, 0, \max(0, \lceil L - u \rceil, \lceil l - U \rceil))$$

where $l$ is the lower determinant, and $u$ is the upper determinant. Here $\lceil x \rceil$ is the ceiling function, returning the smallest integer greater than or equal to $x$; it is only relevant when `float` values are in use. As usual, the rationale for this formula is that $\delta(l, u)$ needs to be a lower bound on the deviation over the full interval (preceding, current, and following). When $Z \wedge l = 0$ is true, the allow zero flag is set and the value we are certain about is 0, so we cannot justify any deviation larger than 0. When $Z \wedge l = 0$ is false, we can forget the allow zero flag, leaving us with

$$\max(0, \lceil L - u \rceil, \lceil l - U \rceil))$$

When finding the deviation from $L$, we take the sum of the values of the children to be as large as possible, and when finding the deviation from $U$, we take the sum to be as small as possible. $L$, $U$, $l$, and $u$ apply to the full interval, that is, including history.

The second step (which is optional) is to apply the cost function to obtain a cost:

$$c(m, S) = f(\delta(l, u))$$

The $c$ in $c(m, S)$ has nothing to do with the history value $c_i$. All values are non-negative.

In a complete solution $l$ is the sum of the values of the children in the current interval plus $x_i$, and $u$ is $l + c_i$. Given this, it is easy to check that our formula agrees with the various XESTT cost formulas, as it needs to do.

At most one of $L - u$ and $l - U$ can be positive. We prove this by showing that their sum is not positive: $(L - u) + (l - U) = (L - U) + (l - u) \le 0$ since $L \le U$ and $l \le u$. Another useful result is that if $L - u \le 0$ and $l - U \le 0$, then $\delta(l, u) = 0$ and $f(\delta(l, u)) = 0$.

When values have type `int`, we may choose to assume $Z \Rightarrow L \ge 2$, for the following reason. If $Z$ is true and $L = 0$, we can change $Z$ to false without changing cost. If $Z$ is true and $L = 1$, we can change $Z$ to false and $L$ to 0 without changing cost. These changes never increase $L$, so they do not interact with $U$. When values have type `float`, there is no useful corresponding result.

We now show that our cost formula $c(m, S) = f(\delta(l, u))$ is monotone non-decreasing as solutions become more complete. All cost functions $f$ are monotone non-decreasing, so we just have $\delta$ to consider. Omitting the allow zero flag for the moment, this is:

$$\delta(l, u) \;=\; \max(0, \lceil L - u \rceil, \lceil l - U \rceil)$$

As solutions become more complete, $l$ is monotone non-decreasing so $\lceil l - U \rceil$ is too, and $u$ is monotone non-increasing (see above) so $\lceil L - u \rceil$ is monotone non-decreasing.

Now consider the full formula, including the allow zero flag:

$$\delta(l, u) \;=\; \chi(Z \wedge l = 0, 0, \max(0, \lceil L - u \rceil, \lceil l - U \rceil))$$

When $Z$ is false this is unchanged. When $Z$ is true, its effect is to possibly reduce $\delta(l, u)$ for the most incomplete solutions (those with $l = 0$), so $\delta(l, u)$ remains monotone non-decreasing as solutions become more complete.

During the construction of a new solution $S'$ by extending a given solution $S$, a value sum expression $m$ needs to calculate $l(m, S)$, $u(m, S)$, $l(m, S')$, and $u(m, S')$. Given these four values, it is trivial to find the extra cost $c(m, S') - c(m, S)$ and the new saved value $l(m, S')$.

To find $l(m, S)$, we either use an initial value (if this is the first day that affects $m$), or retrieve it from the signature of $S$ (on days after the first). We find $u(m, S)$ by adding to $l(m, S)$ the value upper bounds of all the unassigned children. This number can be tabulated before the solve begins, because tbe unassigned children in $S$ depend only on the domain of $S$, not on its assignments, and each child's value upper bound is independent of $S$.

To find $l(m, S')$, we add to $l(m, S)$ the values of the children that are unassigned in $S$ but assigned in $S'$. To find $u(m, S')$, we make these same additions (since $u$ includes $l$), but we also subtract from $u(m, S)$ the value upper bounds of these same children. These steps are carried out in the course of a single traversal of the children that are known to be unassigned in $S$ but assigned in $S'$. These children depend only on the domains of $S$ and $S'$, so they are easy to find.

### C.6.3. Upper bounds

We stated earlier that upper bounds $u(x)$ for each child $x$ are easy to find. We have to take account of the possibility that a child $x$ might itself be a sum expression, in which case we need an upper bound on a deviation. (In practice, it turns out that no sum expressions $x$ are used in contexts where $u(x)$ is needed, so the work done in this section is only for completeness.)

So suppose some expression $e$ has a value which is a deviation:

$$\delta(l,u) \;=\; \chi(Z \wedge l = 0, 0, \max(0, \lceil L - u \rceil, \lceil l - U \rceil))$$

for given fixed $Z, L,$ and $U$ such that $L \le U$. Assuming that $l$ and $u$ satisfy $0 \le l \le u \le M$ for some given $M$ (in fact, $M$ will be the sum of the value upper bounds of the children of $e$), how large could $\delta(l,u)$ be? The answer, as we'll prove in this section, is

$$\lceil \max(L, M - U) \rceil$$

This can be improved slightly to

$$\lceil \max(0, L - 1, M - U) \rceil$$

when values are integers and $Z = true$. Our method is a straightforward analysis of cases.

Let's start with the case $Z = false$, so that $\delta(l,u) = \max(0, \lceil L - u \rceil, \lceil l - U \rceil)$.

Suppose $l \le U$. Then $\max(0, \lceil L - u \rceil, \lceil l - U \rceil) = \max(0, \lceil L - u \rceil)$, which is a monotone non-increasing function of $u$ alone, so it reaches its maximum value when $u$ is as small as possible, that is, when $u = 0$, at which point its value is $\max(0, \lceil L \rceil) = \lceil L \rceil$.

Now suppose $u \ge L$. Then $\max(0, \lceil L - u \rceil, \lceil l - U \rceil) = \max(0, \lceil l - U \rceil)$, which is a monotone non-decreasing function of $l$ alone, so it reaches its maximum value when $l$ is as large as possible, that is, when $l = M$, at which point its value is $\max(0, \lceil M - U \rceil)$.

The remaining case, $l > U$ and $u < L$, cannot occur, because the two conditions imply $u < L \le U < l$ when in fact $l \le u$. Overall, then, the maximum value is

$$\max(\lceil L \rceil, \max(0, \lceil M - U \rceil)) \;=\; \lceil \max(L, M - U) \rceil$$

since $L \ge 0$.

We now consider the second main case, $Z = true$. We can divide this into two sub-cases: $l = 0$ and $l > 0$. When $l = 0$, the deviation is 0. When $l > 0$, the previous analysis applies, except that the additional condition $l > 0$ is present. When values are integers this implies $u \ge 1$ and leads to maximum value $\lceil \max(0, L - 1, M - U) \rceil$, which subsumes the $l = 0$ sub-case.

### C.6.4. Dominance

We do not use tabulated dominance with sum monitors. This is because the determinant of a sum monitor can have many different values, even floating-point values in some cases, making the table potentially much larger than for counter monitors. Also, indexing the table is a problem with `float`-valued indexes. So we fall back on separate dominance and tradeoff dominance.

The XESTT monitors that give rise to sum expressions (limit busy times and limit workload monitors) do not offer history. However, they easily could, for each of their time groups. So each sum expression has $a_i$, $x_i$, and $c_i$ history attributes, although currently they are set to 0 or 0.0.

Sum monitors are used in several contexts in the expression trees that model XESTT constraints. Dominance testing varies depending on this context, as follows.

*(1) Sum monitor at the root, with two or more children, each of which is a sum expression.* In this case, the sum monitor calculates an integer-valued determinant which is the sum of the values of its children, a deviation which is the amount by which the determinant exceeds zero (making it equal to the determinant), and a cost based on the determinant.

This is a very standard situation which can be handled with tradeoff dominance when the cost function is linear, and with separate dominance otherwise. Because the maximum limit is 0 we need $very\_small(l) = false$, so we set $a$ to $-1$. And because there is no minimum limit, we need $very\_large(l) = true$, so we set $b$ to 0.

*(2) Sum monitor at the root, with one or more* `int`*-valued children, none of which is a sum expression.* In this case the children all have value 0 or 1, as it turns out, so we use a counter monitor instead of a sum monitor, to allow tabulated dominance to be used.

*(3) Sum monitor at the root, with one or more* `float`*-valued children, none of which is a sum expression.* The determinant is the sum of the values of the children, and an integer-valued deviation is calculated, and based on that a cost. Because of the `float` values we don't use tabulated dominance or tradeoff dominance. But a `float`-valued separate dominance test may be used in the usual way.

If there is a maximum limit, we need a non-trivial value of $very\_small(l)$. The value $l$ retrieved from the signature is $x_i$ plus the values of the assigned children. This could increase by as much as $B$, the sum of the value upper bounds of the unassigned children. (These children are easily determined, because they just depend on which day the solve is up to; and their value upper bounds are fixed constants stored in the expressions.) So $very\_small(l)$ is true when $l + B \leq U$, and we may take $a = U - B$. This may be negative. This also works when there is no maximum limit, because $U$ is `FLT_MAX` then. A previous analysis for counter monitors is just this analysis when the value upper bounds of the unassigned children are all 1.

If there is a minimum limit, we need a non-trivial value of $very\_large(l)$. Clearly it holds when $l \geq L$, so we may take $b = L$. This is correct when there is no minimum limit, because $L = 0$ then. Again this generalizes an argument made previously for counter monitors.

*(4) Sum expression which is a child of a sum monitor.* In this case the determinant is the sum of the values of the children, possibly `float`-valued, and the value is the deviation from given limits. There is no cost, so separate dominance (possibly `float`-valued) is the only option.

We can find $a$ and $b$ based on the limits stored in the expression; the parent has trivial limits (maximum limit 0) which don't affect dominance at the children. The formulas to use are the same as those for case (3) above, although here they may be `int`-valued, not just `float`-valued.

## C.7. Sequence monitors

A sequence monitor monitors the lengths of non-empty sequences of consecutive things. This section defines sequence monitors, defines their costs in complete and incomplete solutions, shows that cost is non-negative and non-decreasing as solutions become more complete, and defines a suitable $\Delta(m, S_1, S_2)$ for use with tabulated dominance.

### C.7.1. Definition

This section defines sequence monitors. They have a lot in common with counter monitors. Where they are the same, we'll use the same notation and explain things only briefly.

Just as in counter monitors, a sequence monitor has some number of children, each representing one shift, or weekend, or whatever is being monitored, and having one of three states: active (value 1), inactive (value 0), or unassigned. As usual, a child is unassigned if its value is affected by what happens on one of the selected days, and the search is not up to that day yet; and as the search progresses, the unassigned children gradually change their state to assigned.

However, in sequence monitors the order of the children matters. So it makes sense to talk about the first or last child, the child just before or just after a given child, and so on. Any subset of the days may be unassigned initially, so as we proceed along the sequence of children we may encounter active, inactive, and unassigned children, arbitrarily intermingled. This intermingling adds considerable complexity to the implementation. It also occurs with counter monitors, but it does not matter there because the order of the children does not matter.

An *active interval*, or *a-interval*, is a non-empty maximal sequence of consecutive active children. A sequence monitor constrains the length (number of children) of each a-interval.

Sequence monitors have the usual history attributes, although $x_i$ is defined differently:

- A non-negative integer *history before* value $a_i$, representing the number of children in the preceding interval.
- An integer *history* value $x_i$ satisfying $0 \le x_i \le a_i$, representing the length of the a-interval whose last element is the history child immediately before the first true child, or 0 if $a_i = 0$ or the last history child is inactive.
- A non-negative integer *history after* value $c_i$, representing the number of children in the following interval.

The monitor constrains the a-intervals of the full interval (preceding plus current plus following, although there can be no a-intervals in the following interval), and understands that an a-interval can span across the boundary between the preceding interval and the current interval.

There is no single determinant for sequence monitors like there is for counter monitors; a separate determinant is calculated for each a-interval. We'll see the details in the next section. There are lower and upper limits $L$ and $U$, but here they constrain the length of any one a-interval. So $D_l$, the lower limit on $L$, is 1, while $D_u$, the upper limit on $U$, is as for counter monitors. There is no allow zero flag $Z$ and no need for it, because an a-interval always has length at least 1. The usual cost functions $f$, monotone non-increasing and satisfying $f(0) = 0$, are available.

### C.7.2. Deviation and cost

This section defines the cost of sequence monitors in complete and incomplete solutions. As usual, we need a definition of $c(m, S)$, the cost of sequence monitor $m$ in complete or incomplete solution $S$, such that when $S$ is complete the definition agrees with the XESTT definition; and we need this cost to be non-negative, and non-decreasing as solutions become more complete.

First recall the XESTT rule for complete solutions. An *interval* is a sequence of consecutive children. An *active interval*, or *a-interval*, is a maximal non-empty sequence of consecutive active children, as we already know. It is defined for a given monitor $m$ in a given solution $S$. If an a-interval includes the very first child, then it also includes $m$'s $x_i$ history children. Two a-intervals for $m$ in $S$ cannot overlap or abut, because then they would not be maximal.

Let $S$ be a complete solution, and let $A$ be the set of a-intervals of $m$ in $S$. XESTT defines

$$c(m,S) \;=\; \sum_{z \in A} f(\max(0, L - l(z), l(z) - U))$$

where $l(z)$ is the length of a-interval $z$. If $z$ includes the first child, $l(z)$ includes $x_i$, and if $z$ includes the last child, $L - l(z)$ must be replaced by $L - l(z) - c_i$.

We now define $c(m,S)$ for incomplete solutions. Our definition will work for arbitrarily intermingled active, inactive, and unassigned children, as it must; and it will be non-negative, and monotone non-decreasing as solutions become more complete.

An *au-interval* is a maximal non-empty sequence of consecutive children which are either active or unassigned. If an au-interval includes the very first child, then it also includes the $x_i$ history children, which are considered to be active; and if it includes the very last child, then it also includes the $c_i$ history after children, which are considered to be unassigned.

Two au-intervals cannot overlap or abut, because then they would not be maximal. Every a-interval is contained within an au-interval, but despite this intimate connection, we handle a-intervals and au-intervals separately.

Given a sequence monitor $m$, we can construct a corresponding counter monitor $m'$, by using the same children. The total length of the a-intervals of $m$ in solution $S$ equals the lower determinant $l(m', S)$ of $m'$ in $S$, and the total length of the au-intervals of $m$ in $S$ equals the upper determinant $u(m', S)$ of $m'$ in $S$. This correspondence between a-intervals and lower determinants on the one hand, and au-intervals and upper determinants on the other, can help to illuminate what is going on. However, we do not use it in any technical way, largely because although each a-interval lies within exactly one au-interval, an au-interval may contain several a-intervals.

As a search progresses, unassigned children become assigned. These changes cannot make an au-interval longer. All cost functions $f$ are monotone non-decreasing, so if we assign cost

$$f(\max(0, L - l(z)))$$

to au-interval $z$ with length $l(z)$, possibly including $x_i$ or $c_i$ or both as we stated above, this will be non-decreasing as the search progresses, and at the end it will be the cost contributed by $z$ if it violates the lower limit $L$.

Actually this argument has a flaw: if an au-interval has no active children at all, it might never lead to any a-interval (Section B.6), making $f(\max(0, L - l(z)))$ too large (just what we need to avoid) when $z$ contains no active children. So we correct the formula to

$$f(\chi(a(z) = 0, 0, \max(0, L - l(z))))$$

where $a(z)$ is the number of active children in au-interval $z$.

Let $AU$ be the set of au-intervals in solution $S$. We need to show that as the solve proceeds and unassigned children switch to being assigned, the sum

$$\sum_{z \in AU} f(\chi(a(z) = 0, 0, \max(0, L - l(z))))$$

is non-decreasing. Making an unassigned child inactive splits the au-interval containing that child into zero, one, or two smaller au-intervals. If the unsplit au-interval has no active children, it has cost 0 which cannot decrease. If it has at least one active child, then there will be a smaller au-interval containing that child which is shorter than the original, so again there can be no cost decrease. Making an unassigned child active has no effect on the set of au-intervals, but it may change the value of the condition $a(z) = 0$ from true to false. This again cannot decrease cost.

We can make a similar argument for a-intervals. Making an unassigned child assigned cannot make an a-interval shorter. So if we assign cost

$$f(\max(0, l(z) - U))$$

to a-interval $z$ with length $l(z)$, possibly including $x_i$, this will be non-decreasing as the search progresses, and at the end it will be the cost contributed by $z$ if it violates the upper limit $U$.

Once again, however, this argument has a flaw. Suppose that when an unassigned child becomes active, it causes two a-intervals $z_1$ and $z_2$ to be merged into one a-interval $z$ whose length $l(z)$ is $l(z_1) + l(z_2) + 1$. The change in cost is

$$f(\max(0, l(z) - U)) - f(\max(0, l(z_1) - U)) - f(\max(0, l(z_2) - U))$$

This is non-negative when $f$ is linear or quadratic, but it may be negative if $f$ is a step function, for example when $U = 1$ and $l(z_1) = l(z_2) = 2$. The step function counts the number of over-long a-intervals, and that number has decreased. We handle this by changing the formula so that when the cost function is a step function and there is an unassigned child adjacent on the left, the result is 0 instead of $\max(0, l(z) - U)$. Then $z_2$ contributes cost 0 before the switch, and the problem is solved. Our corrected cost formula for a-intervals is

$$f(\chi(u(z), 0, \max(0, l(z) - U)))$$

where $u(z)$ is true when the cost function is a step function and there is an unassigned child adjacent on the left of a-interval $z$.

Let $A$ be the set of a-intervals in solution $S$. We can show now that as the solve proceeds and unassigned children switch to being assigned, the sum

$$\sum_{z \in A} f(\chi(u(z), 0, \max(0, l(z) - U)))$$

is non-decreasing. Making an unassigned child inactive has no effect on the set of a-intervals. It may however change $u(z)$ in an immediately following a-interval $z$ from true to false. This cannot decrease the cost of $z$. Making an unassigned child active cannot decrease cost if it creates an a-interval of length 1, or increases the length of an existing a-interval by 1. We have already analysed what happens if it causes two a-intervals to merge into one.

So then, in any solution $S$, let $AU$ be the set of au-intervals for $m$ in $S$, and let $A$ be the set of a-intervals for $m$ in $S$. Define the cost of $m$ in $S$ to be

$$c(m,S) \;=\; \sum_{z \in AU} f(\chi(a(z) = 0, 0, \max(0, L - l(z)))) \;+\; \sum_{z \in A} f(\chi(u(z), 0, \max(0, l(z) - U)))$$

where $a(z)$ is the number of active children in au-interval $z$, $u(z)$ is true if there is an unassigned child adjacent to a-interval $z$ on the left, and $l(z)$ may include $x_i$ or $c_i$ or both as explained above. This is non-negative, and non-decreasing as solutions become more complete, as just shown.

To show that it equals the XESTT cost of $m$ in complete solutions, we argue as follows. First assume that there are no history after children. There are no unassigned children in complete solutions, so $a(z) > 0$ for every au-interval, $u(z)$ is false for every a-interval, and $A = AU$, giving

$$\begin{aligned} c(m,S) &= \sum_{z \in A} \big[ f(\max(0, L - l(z))) \;+\; f(\max(0, l(z) - U)) \big] \\ &= \sum_{z \in A} f(\max(0, L - l(z), l(z) - U)) \end{aligned}$$

as required. The second line follows because at most one of $L - l(z)$ and $l(z) - U$ can be positive, and $f(0) = 0$ for all cost functions $f$. The XESTT cost formula includes $x_i$ in $l(z)$ if $z$ includes the first child, and replaces $L - l(z)$ by $L - l(z) - c_i$ if $z$ includes the last child. These adjustments are easily seen to happen in our formula too.

One point we forgot to mention is that when $x_i > 0$ we are supposed to subtract cost

$$f(\max(0, L - x_i - b_i - c_i, x_i - U))$$

where $b_i$ is the number of non-history children of $m$. This happens automatically when we set the start cost to the solution cost as reported by KHE, since KHE performs this subtraction.

To see that our formula falls short of the ideal even when $f$ is not a step function, consider this example, which could arise in practice when reassigning weekends:

| Wed | Thu | Fri | Sat | Sun | Mon | Tue | Wed |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | ? | ? | 1 | 1 | 0 |

writing ‘0’ for inactive, ‘1’ for active, and ‘?’ for unassigned. Suppose $L = 3$ and $U = 5$. As the search proceeds, if both unassigned children become active we have one a-interval of length 6 and deviation 1; if both become inactive we have two a-intervals, each of length 2 and deviation 1; and if one becomes active we have two a-intervals, one of length 2 and deviation 1, the other of length 3 and deviation 0. The total deviation is at least 1, and ideally our formula would give this, but in fact it gives 0: there are two a-intervals, both under the maximum limit, and one au-interval, over the minimum limit. There seems to be no easy way to calculate the ideal deviation in such cases, so we don’t try to.

The implementation follows the cost formula quite literally. When a child changes from unassigned to assigned, it searches the child’s neighbourhood to find the enclosing au-interval and any neighbouring a-intervals, and applies the formula to each of these intervals. An extra cost is needed, so it first calculates the cost of the old nearby intervals and subtracts that away, then it calculates the cost of the new nearby intervals and adds that in.

The cost formula supports changing an arbitrary child from unassigned to assigned and vice versa, which is handy during the opening and closing parts of a solve. But during the main part, the implementation assumes that when a child becomes assigned, no unassigned children precede it. This simplifies things (for example, it allows the determinant to be a single integer) and make the cost calculations faster. Appendix D explains how the implementation guarantees this assumption, by delaying the switch in some children if necessary.

### C.7.3. Separate dominance

The signature value for sequence monitors is the number of active children in the sequence of active children immediately to the left of the point where the solve is up to. Although this is rather different from the signature value for counter monitors, the remarks about separate dominance for counter monitors (Appendix C.5.3) apply without change to sequence monitors, including the treatment of *OR* and *AND* children.

### C.7.4. Tabulated dominance

To incorporate a sequence monitor $m$ into our tabulated dominance test, we need to find a $\Delta(m, S_1, S_2)$ satisfying

$$\Delta(m, S_1, S_2) \;\leq\; [c(m, S_2') - c(m, S_2)] - [c(m, S_1') - c(m, S_1)]$$

for all complete extensions $S_2'$ of $S_2$, where $S_1' = S_1 \cup (S_2' - S_2)$ and

$$c(m, S) \;=\; \sum_{z \in AU} f(\chi(a(z) = 0, 0, \max(0, L - l(z)))) \;+\; \sum_{z \in A} f(\chi(u(z), 0, \max(0, l(z) - U)))$$

as defined in the previous section. We need to remember that inactive, active, and unassigned children may be arbitrarily intermingled.

Let a *cut* of $m$ be a partition of the sequence of $m$'s children into left and right parts, with the left part children preceding the right part children. Either part may be empty. We also refer to the point in the sequence of children where the left part ends and the right part begins as the cut.

The *current cut* of $m$ in $S$ is a particular cut of $m$. Its left part begins at the beginning and ends at the last assigned child before the first unassigned child. Its right part begins at the first unassigned child and ends at the end. If the first child is unassigned, the left part is empty (or one could consider the $x_i$ history children to lie in it) and all children lie in the right part. If there are no unassigned children, the left part contains all children and the right part is empty (or one could consider the $c_i$ history after children to lie in it).

If, as we assume, $S_1$ and $S_2$ have equal domains, then the current cut of $m$ in $S_1$ is the same as the current cut of $m$ in $S_2$. This is because the cut depends only on whether children are assigned or unassigned, not on whether assigned children are active or inactive. We call this cut the *common cut*. We also use it within $S_1'$ and $S_2'$: it is not the current cut there, but it is a cut.

An interval lies *strictly left of the common cut* when it lies in the left part of the common cut and does not include the last child of the left part. An interval lies *strictly right of the common cut* when it lies in the right part of the cut and does not include the first child of the right part.

The *neighbours* of an interval are the child immediately to its left, if any, and the child immediately to its right, if any. They are not part of the interval, but they have an important role in delimiting it. Also, the cost of an a-interval is influenced by the state of its left neighbour.

We consider intervals from different solutions to be distinct. We say that two a-intervals or two au-intervals $z$ and $z'$ form a *complementary pair* when they come from different solutions, contain the same children in the same states, and one's cost counts positively within the formula for $\Delta$ while the other's counts negatively. For a-intervals, we also require the state of the left neighbour to be the same for both intervals, to ensure $u(z) = u(z')$. The costs of the two intervals of a complementary pair are equal and cancel each other out, and in that sense those intervals do not contribute to $[c(m, S_2') - c(m, S_2)] - [c(m, S_1') - c(m, S_1)]$ at all.

We now group most of the intervals from the four solutions $S_1$, $S_2$, $S_1'$, and $S_2'$, into exactly one complementary pair each. There are four cases.

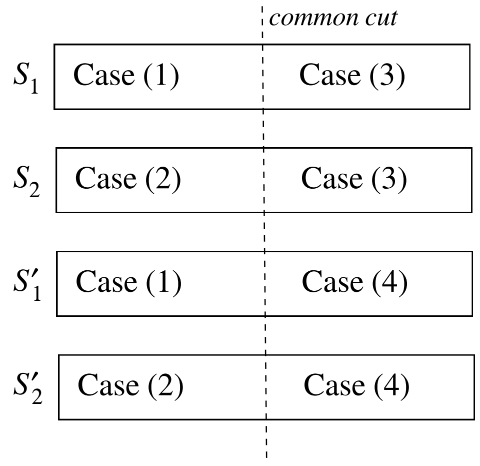
(1) Let $z$ be any a-interval or au-interval in $S_1$ lying strictly left of the common cut. Then $z$'s children, as well as its neighbours, are all assigned and will remain in the same state as $S_1$ is extended. So, since $S_1'$ is an extension of $S_1$, an interval $z'$ must appear in $S_1'$ such that $z$ and $z'$ form a complementary pair. Conversely, each a-interval or au-interval $z'$ in $S_1'$ lying strictly left of the common cut must derive from a $z$ in $S_1$. So this argument cancels out all a-intervals and au-intervals lying strictly left of the common cut in $S_1$ and $S_1'$.

(2) The same argument applied to $S_2$ and $S_2'$ cancels out all a-intervals and au-intervals lying strictly left of the common cut in $S_2$ and $S_2'$.

(3) Let $z$ be any a-interval or au-interval in $S_1$ lying strictly right of the common cut. Now $S_1$ includes no children that have switched from unassigned to active or inactive strictly right of the common cut, because the first child of the right part is unassigned, and, as we explained earlier, switching occurs strictly from left to right. So $z$ must have been present in the initial solution that $S_1$ was derived from. Therefore a complementary $z'$ must be present in $S_2$, since $S_2$ was derived from the same initial solution. The converse argument also works. So this cancels out all a-intervals and au-intervals in $S_1$ and $S_2$ that lie strictly right of the common cut.

(4) Let $z$ be any a-interval or au-interval in $S_1'$ lying strictly right of the common cut. Since $S_1'$ is complete, $z$'s children and the adjacent children are assigned, and their states are determined by the initial solution and by the assignments in $S_1' - S_1$. There must therefore be a complementary interval in $S_2'$, since $S_2'$ is derived from the same initial solution and the assignments of $S_2' - S_2$ are the same as the assignments of $S_1' - S_1$. So this cancels out all a-intervals and au-intervals lying strictly right of the common cut in $S_1'$ and $S_2'$.

Looking through the four cases, we see that they cover every a-interval and au-interval lying strictly left or right of the common cut in $S_1$, $S_2$, $S_1'$, and $S_2'$:



So the only intervals that contribute to $[c(m,S'_2)-c(m,S_2)]-[c(m,S'_1)-c(m,S_1)]$ are those not lying strictly left or right of the common cut. Call them *unpaired intervals.* We have shown

$$[c(m,S'_2)-c(m,S_2)]-[c(m,S'_1)-c(m,S_1)] \;=\; [c_u(m,S'_2)-c_u(m,S_2)]-[c_u(m,S'_1)-c_u(m,S_1)]$$

where each $c_u(m,S)$ is the cost of unpaired intervals in its solution.

Now there cannot be two unpaired au-intervals in one solution, because if there were, they would both not lie strictly to the left or right of the common cut, which means that they would have to overlap or abut, which au-intervals never do. For the same reason, there cannot be two unpaired a-intervals in one solution.

The next step is to find a formula which combines the cost of the single unpaired au-interval and the single unpaired a-interval of one solution $S$ into a single cost, $c_u(m,S)$. The contribution to $c_u(m,S)$ made by a non-existent interval needs to be 0.

Care is needed when including history after children in unpaired au-intervals. Let $p$ be the number of children to the right of the common cut, not counting history after children. Define

$$\overline{c_i}(x) \;=\; \chi(x<p,0,c_i)$$

Then if $x$ is the number of non-history children to the right of the cut in some au-interval, this will return $c_i$ when these children abut the history after children, and 0 when they don't.

In $S_1$, the left part of the common cut ends with (say) $l_1$ active children, possibly none, and possibly including the $x_i$ history children. Immediately to the left of these children there is either nothing or an inactive child, because there are no unassigned children to the left of the cut. The right part of the common cut begins with (say) $q$ unassigned children (not including history after children) and $r$ active children, arbitrarily intermingled. Immediately to the right of these children there is either nothing, or an inactive child, or the first history after child. The values $l_1$, $p$, $q$, and $r$ are always well-defined, independently of whether any unpaired intervals are present.

If there are no unassigned children to the right of the common cut, we have completed the last open day, and $p=q=r=0$. If it is not the last day, a result that will be important to us later is that $q>0$. This is because the first child to the right of the common cut is unassigned, so it counts towards $q$.

If there is an unpaired au-interval $z$ in $S_1$ then it consists of $l_1$ active children immediately to the left of the cut, followed by $q + r + \overline{c}_i(q + r)$ unassigned and active children immediately to the right of the cut. After all that comes nothing or an inactive child. The cost of $z$ is

$$f(\chi(a(z) = 0, 0, \max(0, L - l(z)))) \;=\; f(\chi(l_1 + r = 0, 0, \max(0, L - l_1 - q - r - \overline{c}_i(q + r))))$$

This formula is also correct when there is no unpaired au-interval, since then $l_1 = 0$ and $r = 0$.

Still in $S_1$, if there is an unpaired a-interval $z$, then it consists of the same $l_1$ active children immediately to the left of the cut, and nothing to the right, since immediately to the right of the cut is either nothing or an unassigned child (possibly a history after child). The cost of $z$ is

$$f(\chi(u(z), 0, \max(0, l(z) - U))) \;=\; f(\max(0, l_1 - U))$$

since $u(z)$ is false, because there can be no unassigned child to the left of the cut. Again, this is correct when there is no a-interval, since then $l_1 = 0$.

Our next step is to make this simple formula more complicated:

$$f(\max(0, l_1 - U)) \;=\; f(\chi(l_1 + r = 0, 0, \max(0, l_1 - U)))$$

This follows because if $l_1 + r = 0$, then this formula gives 0 and the simple formula does too, because $l_1 = 0$; while if $l_1 + r > 0$, this formula reduces to the simple formula.

We now combine the au-interval formula with the a-interval formula to get

$$\begin{aligned} c_u(m, S_1) \;=\;& f(\chi(l_1 + r = 0, 0, \max(0, L - l_1 - q - r - \overline{c}_i(q + r)))) \;+ \\ & f(\chi(l_1 + r = 0, 0, \max(0, l_1 - U))) \end{aligned}$$

Now if $l_1 + r = 0$ this gives 0, while if $l_1 + r > 0$ it gives

$$\begin{aligned} & f(\max(0, L - l_1 - q - r - \overline{c}_i(q + r))) \;+\; f(\max(0, l_1 - U)) \\ & \qquad = \; f(\max(0, L - l_1 - q - r - \overline{c}_i(q + r), l_1 - U)) \end{aligned}$$

since at most one of $L - l_1 - q - r - \overline{c}_i(q + r)$ and $l_1 - U$ is positive, and $f(0) = 0$. So the total cost of unpaired intervals in $S_1$ is

$$\begin{aligned} c_u(m, S_1) \;&=\; f(\chi(l_1 + r = 0, 0, \max(0, L - l_1 - q - r - \overline{c}_i(q + r), l_1 - U))) \\ &=\; f(\delta(l_1 + r, l_1, l_1 + q + r + \overline{c}_i(q + r))) \end{aligned}$$

writing

$$\delta(z, l, u) \;=\; \chi(z = 0, 0, \max(0, L - u, l - U))$$

This is similar to the $\delta$ function for counter monitors, omitting the allow zero flag $Z$ and allowing the test for the zero special case to depend on a separate parameter $z$. The same argument applies to $S_2$, replacing $l_1$ by $l_2$. Note that $p$ (used by $\overline{c}_i$) is the same for $S_1$ and $S_2$, as are $q$ and $r$ because only initial assignments determine the states of $m$'s children to the right of the cut.

We turn now to $S'_1$. Since $S'_1$ is complete, all $m$'s children are assigned in it (except history after children), and since it is an extension of $S_1$, each assigned child in $S_1$ preserves its state in $S'_1$. So there are $l_1$ active children immediately to the left of the common cut in $S'_1$, and some number $y$ such that $0 \le y \le q + r$ of active children immediately to the right of the cut. After them comes an inactive child, the first history after child, or nothing. Here $l_1$ and $y$ are always well-defined, independently of whether any unpaired intervals are present.

So if there is an unpaired au-interval $z$ in $S'_1$ then it consists of $l_1$ active children immediately to the left of the common cut, followed by some number $y$ such that $0 \le y \le q + r$ of active children to the right of the cut, possibly followed by the $c_i$ history after children. The cost is

$$f(\chi(a(z) = 0, 0, \max(0, L - l(z)))) \;=\; f(\chi(l_1 + y = 0, 0, \max(0, L - l_1 - y - \overline{c}_i(y))))$$

This is also correct when there is no au-interval, since then $l_1 = 0$ and $y = 0$.

Since $S'_1$ is complete, if there is an unpaired a-interval $z$ in $S'_1$ then it consists of the same children as the au-interval, although not the history after children. The cost is

$$f(\chi(u(z), 0, \max(0, l(z) - U))) \;=\; f(\max(0, l_1 + y - U))$$

since $u(z)$ is false as before. As usual, this is correct when there is no a-interval.

Once again our next step is to make this simple formula more complicated:

$$f(\max(0, l_1 + y - U)) \;=\; f(\chi(l_1 + y = 0, 0, \max(0, l_1 + y - U)))$$

which is clearly true. Then we combine the au-interval and a-interval formulas to get

$$\begin{aligned} c_u(m, S'_1) \;&=\; f(\chi(l_1 + y = 0, 0, \max(0, L - l_1 - y - \overline{c}_i(y)))) \;+ \\ &\qquad f(\chi(l_1 + y = 0, 0, \max(0, l_1 + y - U))) \end{aligned}$$

Now if $l_1 + y = 0$ this gives 0, while if $l_1 + y > 0$ it gives

$$\begin{aligned} &f(\max(0, L - l_1 - y - \overline{c}_i(y))) \;+\; f(\max(0, l_1 + y - U)) \\ &\qquad =\; f(\max(0, L - l_1 - y - \overline{c}_i(y), l_1 + y - U)) \end{aligned}$$

since at most one of $L - l_1 - y - \overline{c}_i(y)$ and $l_1 + y - U$ is positive, and $f(0) = 0$. So the total cost of unpaired intervals in $S'_1$ is

$$\begin{aligned} c_u(m, S'_1) \;&=\; f(\chi(l_1 + y = 0, 0, \max(0, L - l_1 - y - \overline{c}_i(y), l_1 + y - U))) \\ &=\; f(\delta(l_1 + y, l_1 + y, l_1 + y + \overline{c}_i(y))) \end{aligned}$$

using the $\delta$ function we defined above. The same argument applies to $S'_2$, changing $l_1$ to $l_2$. The values of $y$ and $\overline{c}_i(y)$ are the same for $S'_2$ as for $S'_1$, because they depend on assignments that are the same in both solutions.

Putting everything together, we find that

$$\Delta(m, S_1, S_2) \;\le\; [c(m, S'_2) - c(m, S_2)] - [c(m, S'_1) - c(m, S_1)]$$

$$\begin{aligned}
&= [c_u(m,S'_2) - c_u(m,S_2)] - [c_u(m,S'_1) - c_u(m,S_1)] \\
&= [f(\delta(l_2+y, l_2+y, l_2+y+\overline{c_i}(y))) - f(\delta(l_2+r, l_2, l_2+q+r+\overline{c_i}(q+r)))] \\
&\quad - [f(\delta(l_1+y, l_1+y, l_1+y+\overline{c_i}(y))) - f(\delta(l_1+r, l_1, l_1+q+r+\overline{c_i}(q+r)))]
\end{aligned}$$

for all $y$ such that $0 \le y \le q+r$. We define $\Psi_y(p,q,r,l_1,l_2)$ as a name for this last formula, and $\Psi(p,q,r,l_1,l_2)$ for its minimum value. This gives us

$$\Delta(m,S_1,S_2) \le \Psi(p,q,r,l_1,l_2) = \min_{0 \le y \le q+r} \Psi_y(p,q,r,l_1,l_2)$$

We will use $\Psi_y(p,q,r,l_1,l_2)$ in Appendix C.7.6.

All this is similar to what we had for counter monitors, and here too we can calculate the minimum over all $y$ and cache the results. For counter monitors the cache is indexed by the triple $(e,l_1,l_2)$, but here it is indexed by the quintuple $(p,q,r,l_1,l_2)$, where, within $S_1$ (or any solution with the same domain), $p$ is the number of current interval children to the right of the common cut, $q$ is the number of unassigned current interval children to the right of the common cut in the adjacent au-interval, and $r$ is the number of active children to the right of the common cut in that same interval. Letting $A$ be the number of children in the current interval, the index ranges are easily seen to be $0 \le p \le A$, $0 \le q \le p$, $0 \le r \le p-q$, $0 \le l_1 \le A-p+x_i$, and $0 \le l_2 \le A-p+x_i$.

The five indexes are all easy to find during dominance checking. There does not seem to be a way to reduce their number, because they appear in different combinations at different points in the formula. However, similarly to counter monitors, for a given $k$, all $d_k$-solutions have the same values for $p$, $q$, and $r$. So we can store a pointer to the part of the cache indexed by these numbers in the object representing $m$'s dominance test on day $d_k$, and pass just the pair $(l_1,l_2)$ to that part of the cache to retrieve the particular $\Psi(p,q,r,l_1,l_2)$ we need.

We need to give thought to how to calculate the correct $p$, $q$, and $r$. These are defined in terms of the current cut, but when we set up for solving, our starting point is $d_k$, the day that will have just been assigned in the $d_k$-solutions that we wish to test for dominance.

In a $d_k$-solution, then, the first unassigned child is the first child $x$ whose value depends on any of the open days $d_{k+1}, \dots, d_n$. If there is no such $x$, $d_k$ is $m$'s last open day, so we don't need a dominance test for $m$ on that day. Otherwise, this $x$ is the first child of the right part of the cut, and $p$ is the number of current interval children from $x$ inclusive to the end. For $q$ and $r$ we scan to the right, starting with $x$, counting the number of unassigned and active children we pass over, stopping at the first inactive child, or at the end of the current interval. We do this only when setting up for solving, not during solving.

## C.7.5. Early termination of the tabulated dominance test

Appendix C.4.3 stated that $\Delta(m,S_1,S_2) \le 0$ holds for sequence monitors in most cases but not always. We prove this now.

An example where the condition does not hold, the first example found by the author's checking code, is $L = 2$, $U = 5$, $p = 2$, $q = 0$, $r = 1$, $l_1 = 0$, and $l_2 = 5$. This gives range $0 \le y \le 1$, and for both of these values of $y$ the difference in extra cost is the positive value $f(1)$.

We now prove that $\Delta(m,S_1,S_2) \le 0$ when $r = 0$. This is most of the time in practice, because

$r$ is the number of active children in the au-interval to the right of the cut, so that the condition $r \geq 1$, while possible, is not common. The proof follows the proof for counter monitors closely. Two small adjustments are needed, one for when the determinant is 0, and the other for handling $\overline{c_i}$ instead of $c_i$. These seem to prevent us from simply re-using the earlier result.

Our task then is to prove that for each $(p, q, l_1, l_2)$ quadruple there exists a $y$ in the range $0 \leq y \leq q + r$ such that

$$[f(\delta(l_2 + y, l_2 + y, l_2 + y + \overline{c_i}(y))) - f(\delta(l_2 + r, l_2, l_2 + q + r + \overline{c_i}(q + r)))]$$
$$- [f(\delta(l_1 + y, l_1 + y, l_1 + y + \overline{c_i}(y))) - f(\delta(l_1 + r, l_1, l_1 + q + r + \overline{c_i}(q + r)))] \leq 0$$

assuming $r = 0$.

Since the second main term in this formula is an extra cost, it is non-negative. Using this and the condition $r = 0$ we see that it is sufficient to find a $y$ in the range $0 \leq y \leq q$ such that

$$f(\delta(l_2 + y, l_2 + y, l_2 + y + \overline{c_i}(y))) - f(\delta(l_2, l_2, l_2 + q + \overline{c_i}(q))) \leq 0$$

This is what we will do.

In some cases we use a shortcut: we are able to find a $y$ in the range $0 \leq y \leq q$ such that $\delta(l_2 + y, l_2 + y, l_2 + y + \overline{c_i}(y)) = 0$. This then gives us $f(\delta(l_2 + y, l_2 + y, l_2 + y + \overline{c_i}(y))) = f(0) = 0$, and since $f(\delta(l_2, l_2, l_2 + q + \overline{c_i}(q))) \geq 0$ we are done.

We prove our result using four cases. Let $K = L - l_2 - \overline{c_i}(0)$.

**Case 1:** $l_2 = 0$. Choose $y = 0$. This gives

$$\begin{aligned} \delta(l_2 + y, l_2 + y, l_2 + y + \overline{c_i}(y)) &= \delta(l_2, l_2, l_2 + \overline{c_i}(0)) \\ &= \chi(l_2 = 0, 0, \max(0, L - l_2 - \overline{c_i}(0), l_2 - U)) \\ &= 0 \end{aligned}$$

taking the first branch of the $\chi$. The shortcut applies so we are done for this case.

From now on we assume $l_2 > 0$. This means that in all cases the first argument of $\delta$ is strictly positive, and so we can ignore $\chi$ and use the simpler formula

$$\delta(z, l, u) = \max(0, L - u, l - U)$$

We will do this in the remaining cases without further mention.

**Case 2:** $l_2 > 0$ and $K \leq 0$. Choose $y = 0$. This gives

$$\begin{aligned} \delta(l_2 + y, l_2 + y, l_2 + y + \overline{c_i}(y)) &= \delta(l_2, l_2, l_2 + \overline{c_i}(0)) \\ &= \max(0, L - l_2 - \overline{c_i}(0), l_2 - U) \\ &= \max(0, l_2 - U) \end{aligned}$$

because $L - l_2 - \overline{c_i}(0) = K \leq 0$. And

$$\begin{aligned}\delta(l_2, l_2, l_2 + q + \overline{c}_i(q)) &= \max(0, L - l_2 - q - \overline{c}_i(q), l_2 - U) \\ &= \max(0, l_2 - U)\end{aligned}$$

because $\overline{c}_i(0) \le \overline{c}_i(q)$ and so

$$L - l_2 - q - \overline{c}_i(q) \;\le\; L - l_2 - q - \overline{c}_i(0) \;=\; K - q \;\le\; 0$$

Putting these results together, we get

$$\begin{aligned}&f(\delta(l_2 + y, l_2 + y, l_2 + y + \overline{c}_i(y))) \;-\; f(\delta(l_2, l_2, l_2 + q + \overline{c}_i(q))) \\ &\quad= f(\max(0, l_2 - U)) - f(\max(0, l_2 - U)) \\ &\quad= 0\end{aligned}$$

So we have our result in Case 2.

**Case 3:** $l_2 > 0$ and $K > 0$ and $q \le K$. Choose $y = q$. This gives

$$\begin{aligned}\delta(l_2 + y, l_2 + y, l_2 + y + \overline{c}_i(y)) &= \delta(l_2 + q, l_2 + q, l_2 + q + \overline{c}_i(q)) \\ &= \max(0, L - l_2 - q - \overline{c}_i(q), l_2 + q - U) \\ &= \max(0, L - l_2 - q - \overline{c}_i(q))\end{aligned}$$

because

$$\begin{aligned}l_2 + q - U &\le l_2 + K - U \\ &= l_2 + (L - l_2 - \overline{c}_i(0)) - U \\ &= L - \overline{c}_i(0) - U \\ &\le 0\end{aligned}$$

since $L \le U$. And

$$\begin{aligned}\delta(l_2, l_2, l_2 + q + \overline{c}_i(q)) &= \max(0, L - l_2 - q - \overline{c}_i(q), l_2 - U) \\ &= \max(0, L - l_2 - q - \overline{c}_i(q))\end{aligned}$$

because $l_2 - U \le l_2 + K - U \le 0$ as we just saw. So

$$\begin{aligned}&f(\delta(l_2 + y, l_2 + y, l_2 + y + \overline{c}_i(y))) \;-\; f(\delta(l_2, l_2, l_2 + q + \overline{c}_i(q))) \\ &\quad= f(\max(0, L - l_2 - q - \overline{c}_i(q)) \;-\; f(\max(0, L - l_2 - q - \overline{c}_i(q)) \\ &\quad= 0\end{aligned}$$

So we have our result in Case 3.

**Case 4:** $l_2 > 0$ and $K > 0$ and $q > K$. Choose $y = K$ (legal since $0 < K < q$). This gives

$$\begin{aligned}\delta(l_2+y, l_2+y, l_2+y+\overline{c}_i(y)) &= \delta(l_2+K, l_2+K, l_2+K+\overline{c}_i(K))\\ &= \max(0, L-l_2-K-\overline{c}_i(K), l_2+K-U)\\ &= 0\end{aligned}$$

because the second term of the 'max' is

$$\begin{aligned}L-l_2-K-\overline{c}_i(K) &= L-l_2-(L-l_2-\overline{c}_i(0))-\overline{c}_i(K)\\ &= \overline{c}_i(0)-\overline{c}_i(K)\\ &\leq 0\end{aligned}$$

since $\overline{c}_i(x)$ is a monotone non-decreasing function of $x$, and $K \geq 0$. And the third term is

$$\begin{aligned}l_2+K-U &= l_2+(L-l_2-\overline{c}_i(0))-U\\ &= L-\overline{c}_i(0)-U\\ &\leq 0\end{aligned}$$

The shortcut applies so Case 4 is done. This completes our proof that $\Delta \leq 0$ when $r = 0$.

### C.7.6. Correlated expressions

Here we work out the algebra for Type 3 and Type 4 correlations, as defined in Appendix C.4.5. We'll start with the definite case of a monitor $m_1$ which limits the number of consecutive busy days, and a monitor $m_2$ which limits the number of consecutive free days, then generalize it. Any limits and cost functions are allowed.

Appendix C.7.4 gives us this formula for any sequence monitor $m$:

$$\Delta(m, S_1, S_2) \leq \Psi(p,q,r,l_1,l_2) = \min_{0 \leq y \leq q+r} \Psi_y(p,q,r,l_1,l_2)$$

It will be convenient here to write this as

$$\Delta(m, S_1, S_2) \leq \Psi(m) = \min_{0 \leq y \leq u(m)} \Psi_y(m)$$

This is a reasonable notation because each monitor $m$ knows its own values for $p$, $q$, and $r$ on any given day, and also the common index of $l_1$ and $l_2$ in the signatures of $S_1$ and $S_2$.

Now correlating $m_1$ and $m_2$ amounts to merging the two monitors into a single monitor $\{m_1, m_2\}$ whose cost is the sum of the costs of $m_1$ and $m_2$. A little thought will show that

$$\Delta(\{m_1,m_2\}, S_1, S_2) \leq \min_{\substack{(y_1,y_2) \text{ s.t.}\\ 0 \leq y_1 \leq u(m_1)\\ 0 \leq y_2 \leq u(m_2)}} [\Psi_{y_1}(m_1) + \Psi_{y_2}(m_2)]$$

But now, $y_1$ represents some number of busy days immediately to the right of the common cut, bounded above by $u(m_1)$, while $y_2$ represents some number of free days immediately to the right of the same common cut, bounded above by $u(m_2)$. Since the first day after the common cut cannot be both busy and free for the same resource, at least one of $y_1$ and $y_2$ must be 0. Accordingly, $\{m_1, m_2\}$ needs to add $\min(D, E)$ to the available cost, where

$$\begin{aligned} D &= \Psi_0(m_1) + \min_{0 \le y_2 \le u(m_2)} \Psi_{y_2}(m_2) \\ E &= \min_{0 \le y_1 \le u(m_1)} \Psi_{y_1}(m_1) + \Psi_0(m_2) \end{aligned}$$

Using the terminology above, $D$ and $E$ can be expressed as

$$\begin{aligned} D &= \Psi_0(m_1) + \Psi(m_2) \\ E &= \Psi(m_1) + \Psi_0(m_2) \end{aligned}$$

We can implement this by storing $\Psi_0$ alongside $\Psi$ in each dominance test, and calculating the minimum of these two sums of table lookups. We could use a single lookup in a larger table, but given the number of indexes that does not seem to be a good idea. It is true that $p_1 = p_2$, that $l_1$ cannot be non-zero for both $m_1$ and $m_2$, and that $l_2$ cannot be non-zero for both $m_1$ and $m_2$, but that still leaves too many independent indexes to make a single table an attractive option.

The inequalities $\Psi_0(m_1) \ge \Psi(m_1)$ and $\Psi_0(m_2) \ge \Psi(m_2)$ follow trivially from the definitions. Using them we have

$$\begin{aligned} \min(D, E) &= \min(\Psi_0(m_1) + \Psi(m_2), \Psi(m_1) + \Psi_0(m_2)) \\ &\ge \min(\Psi(m_1) + \Psi(m_2), \Psi(m_1) + \Psi(m_2)) \\ &= \Psi(m_1) + \Psi(m_2) \end{aligned}$$

which is the quantity we add to the available cost when $m_1$ and $m_2$ are not correlated. This shows that using correlation is always at least as good as not using it.

We can generalize all this to any set of sequence monitors $\{m_1, \dots, m_k\}$, each with one child for each day, such that for each $i$, if $y_i > 0$ then $y_j = 0$ for every $j \ne i$. The available cost is

$$\min_{1 \le i \le k} \left( \Psi(m_i) + \sum_{\substack{1 \le j \le k \\ j \ne i}} \Psi_0(m_j) \right)$$

Assuming as usual that each resource can take at most one shift per day, an example of such a larger set is $\{m_1, m_2, m_3, m_4\}$, where $m_1$ limits the number of consecutive early shifts, $m_2$ limits the number of consecutive morning shifts, $m_3$ limits the number of consecutive day shifts, and $m_4$ limits the number of consecutive night shifts. The implementation uses a trivial transformation of this formula: it calculates

$$\Psi_z = \sum_{1 \le j \le k} \Psi_0(m_j)$$

and reports

$$\min_{1 \le i \le k} [\, \Psi(m_i) - \Psi_0(m_i) + \Psi_z] = \min_{1 \le i \le k} [\, \Psi(m_i) - \Psi_0(m_i)] + \Psi_z$$

for the available cost. In its final form this can be calculated in a single pass through the $m_i$, accumulating two values, one for each part of the formula, and adding them at the end. The loss of efficiency, compared with the uncorrelated calculation, is not large.

What we have done so far we call a *Type 3* correlation. In some cases we can refine this further into what we call a *Type 4* correlation, as follows.

If $m_1$ counts consecutive busy days and $m_2$ counts consecutive free days, then, since the next day is either busy or free, we have $y_1 \ge 1$ or $y_2 \ge 1$. (No such result holds for consecutive early, morning, day, and night shifts, because for them all the $y_i$ could be zero, when the next day is a free day. We can't add the consecutive free days monitor to them, because it is already linked with the consecutive busy days monitor, and we can't use it twice.) The easy way to add the new condition is to return to $D$ and $E$:

$$D = \Psi_0(m_1) + \min_{0 \le y_2 \le u(m_2)} \Psi_{y_2}(m_2)$$

$$E = \min_{0 \le y_1 \le u(m_1)} \Psi_{y_1}(m_1) + \Psi_0(m_2)$$

and require the $y_i$ that is not fixed to 0 to be at least 1:

$$D' = \Psi_0(m_1) + \min_{1 \le y_2 \le u(m_2)} \Psi_{y_2}(m_2)$$

$$E' = \min_{1 \le y_1 \le u(m_1)} \Psi_{y_1}(m_1) + \Psi_0(m_2)$$

This is undefined when $u(m_1) = 0$ or $u(m_2) = 0$. But those cases cannot happen, because $u(m) = q + r$, where $r \ge 0$, and $q > 0$ on every open day except the last (Appendix C.7.4). This cannot be the last day, because all signatures are empty then and dominance checking is just cost comparison. We define

$$\Psi_+(m) = \min_{1 \le y \le u(m)} \Psi_y(m)$$

and tabulate this with $\Psi$ and $\Psi_0$. We incorporate the new condition by replacing $\Psi$ with $\Psi_+$.

Real-world cases where Type 3 correlation is superior to no correlation are common, and indeed normal. However, at the time of writing the author has not observed a real-world case where the extra step from Type 3 to Type 4 was beneficial.

## C.8. Solution types

In this section we consider types of solutions beyond the usual $d_k$-solutions. These other types support optimizations that we will come to later. Here we introduce the different types and explain how dominance testing works for each.

A *solution* is a set of assignments. Each solution has a *type*: a set of rules that restrict the assignments. (The familiar $d_k$-solution is one type.) For any $k \geq 0$, an *assignment* is a triple

$$(V, \langle r_1, \ldots, r_k \rangle, \langle v_1, \ldots, v_k \rangle)$$

where $V$ is a set of tasks, each $r_i$ is an open resource, and each $v_i$ is a set of open tasks. It represents the assignment of each $r_i$ to an unspecified element of the corresponding $v_i$. When $k = 1$, the angle brackets may be omitted. A single task $t_i$ may replace a set of tasks $v_i$.

An assignment is *valid* when it satisfies two conditions. First, for each $i$ we require $v_i \subset V$. Second, for each $i$, we require $r_i$ to be a suitable resource for assignment to each task of $v_i$. For example, if $r_i$ is an ordinary nurse and some task of $v_i$ requires a senior nurse, an assignment that pairs $r_i$ with $v_i$ is invalid. As a reminder of these conditions we sometimes speak of *valid assignments*; but in fact invalid assignments are never created.

The initial $V$ serves to indicate in a general way what is being assigned. Although $V$ could be any set of tasks, in practice only these kinds of sets of tasks are used:

1. The set of all tasks which begin on a given day $d_i$. We denote this set by $d_i$, overloading the notation in an obvious way.

2. The set of all tasks of a shift $s_i$. A *shift* is a maximal set of tasks which have the same busy times and the same workload. This guarantees that whichever task of shift $s_i$ is assigned a resource $r$, the effect on $r$'s resource monitors is the same.

3. The set of all tasks of an mtask $c_i$. An mtask is a maximal set of tasks which have the same busy times and workload, and which satisfy other conditions defined elsewhere.

Each task lies in exactly one mtask, each mtask lies in exactly one shift, and each shift lies in exactly one day.

It is vital to distinguish between the case of a resource that is free on some day, and the case of a resource whose assignment on that day has not been decided on. To do this, each day contains a special shift called $s_0$, which contains one mtask called $c_0$, which contains an unlimited number of distinct tasks representing free time. Thus, to say that a resource has a free day, one has to assign it to one of these special sets or tasks. To say that a resource's assignment on some day has not been decided on, one has to ensure that no assignment of that resource to any task lying within that day appears in the solution.

Dominance testing between solutions is always carried out between pairs of solutions of the same type. In addition, there are usually additional rules, such as the one that states that two $d_k$-solutions can only be tested for dominance if they are for the same day.

The implementation uses a variety of type-dependent optimizations to reduce the memory consumed by assignments and solutions. For example, the object representing a $d_k$-solution holds a set of assignments for day $d_k$ plus a pointer to a $d_{k-1}$-solution. It formerly also held its day, but no longer does. During the expansion of a solution $S$, solution objects of several types are created whose lifetime is limited to the time that $S$ is being expanded. Although they are all extensions of $S$, they do not refer to $S$ explicitly.

### C.8.1. The $d_k$-solution

We are already familiar with $d_k$-solutions, so this section will be brief. A $d_k$-solution is a set of assignments of the form

$$(d_i, r, t)$$

indicating that on open day $d_i$ (standing for the set of tasks that begin on that day) resource $r$ is assigned to task $t$. For every open resource $r$ and every open day $d_i$ such that $1 \le i \le k$, a $d_k$-solution must include exactly one assignment containing $d_i$ and $r$, and at most one assignment containing any given task $t$. Also, there must be no assignments for days $d_i$ such that $i > k$.

Dominance testing is allowed between any two $d_k$-solutions for the same day.

### C.8.2. The $c_i$-solution

As we've seen, if there are $a$ mtasks and $m$ selected resources, then each solution for day $d_k$ gives rise to up to $(a+1)^m$ solutions for day $d_{k+1}$. Many of these new solutions will turn out to be dominated and will be deleted, but before that happens we will have wasted a lot of time generating them. For example, if $a = 4$ and $m = 5$, we generate up to $5^5 = 3125$ $d_{k+1}$-solutions for each undominated $d_k$-solution.

If we could take smaller steps, for example adding just one assignment before testing for dominance, we might be able to drop many solutions early and avoid generating those $(a+1)^m$ extensions. This is what $c_i$-solutions aim to achieve.

A $c_i$-solution is a solution of the form

$$S \cup \{(d_{k+1}, r, c_i)\}$$

where $S$ is a $d_k$-solution and $c_i$ is an mtask from day $d_{k+1}$. There are no other conditions: any $d_k$-solution, and any one assignment on day $d_{k+1}$, are acceptable.

Assigning to an mtask rather than to a single task amounts to saying that the properties of interest here are the same for all tasks of $c_i$. All algorithms for expanding $S$ begin by building one $c_i$-solution for each valid combination of an open resource $r$ and an open mtask $c_i$ (including the special mtask $c_0$ denoting a free day), calculating the effect on $r$'s monitors of assigning $r$ to any one task of $c_i$ (the effect is the same for all tasks of $c_i$), and building a signature holding that common effect. Ultimately a single task from $c_i$ must be chosen, but before then it saves a lot of time to do this work once for each mtask rather than once for each task.

We sometimes refer to $c_i$-solution objects as assignment objects. This is understandable because each contains an assignment and does not refer explicitly to $S$. However it is strictly incorrect because it ignores their signatures' dependence on $S$.

Now for dominance. Let $S$ be any $d_k$-solution. Consider these two $c_i$-solutions:

$$S_1 = S \cup \{(d_{k+1}, r, c_i)\}$$

and

$$S_2 = S \cup \{(d_{k+1}, r, c_j)\}$$

where $c_i \neq c_j$. As usual, we require both assignments to be valid ($r$ must be acceptable to both $c_i$ and $c_j$). It would be good if we could show that $S_1$ dominates $S_2$, say, because then, during the expansion of $S$, we can safely omit $(d_{k+1}, r, c_j)$ from $r$'s list of possible assignments (that is, we can rule out assigning $r$ to any task from $c_j$), saving a lot of time. For example, if $r$'s timetable in $S$ ends with a night shift, we might be able to show that adding one more night shift is at least as good as adding a morning shift.

When we apply our usual formula $S'_1 = S_1 \cup (S'_2 - S_2)$, an issue arises. Because $S'_2$ is an arbitrary extension of $S_2$, we cannot prevent $S'_2 - S_2$ from containing, say, $(d_{k+1}, r', c_i)$, but when we add this to $S_1$ we find that $c_i$ appears twice. This will be a problem when $c_i$ contains only one task, because there will be no way to refine the two assignments into assignments to different specific tasks. We'll be handling this by declaring that there is no dominance in this case. For the moment, though, we'll proceed as though it is not a problem.

Suppose, then, that we have $S_1$ and $S_2$ as defined above, that $S'_2$ is an arbitrary complete extension of $S_2$, and that $S'_1$ is valid and defined by $S'_1 = S_1 \cup (S'_2 - S_2)$. Our task now is to work out $c(S'_2) - c(S'_1)$ given that $S'_1$ and $S'_2$ differ only in the assignment to $r$ on $d_{k+1}$.

Suppose $m$ is a resource monitor for a resource other than $r$. Then $c(m, S'_2) - c(m, S'_1) = 0$ because $S'_1$ and $S'_2$ differ only in the one assignment to $r$, and that has no effect on $m$.

Now suppose $m$ is a resource monitor for $r$. Clearly, $m$ has the same cost and signature in $S_1$ as it has in any $d_{k+1}$-solution that includes $S$ and $(d_{k+1}, r, c_i)$, and similarly for $S_2$. So we can handle all such monitors $m$ by working out their extra costs and signatures just as we do when building a $d_{k+1}$-solution. We then add their extra costs to $c(S)$ to get $c(S_1)$ and $c(S_2)$, and perform the usual scan along the signature to test for dominance.

If we try to use signatures with event resource monitors we encounter problems. One is that one assignment does not produce a complete day's action on an event resource monitor $m$ like it does on a resource monitor: other assignments on the same day may affect $m$. But expressions are set up to be evaluated once per day. This is not a problem for $d_k$-solutions, because they add an entire day's worth of assignments at once.

Furthermore, an assignment is between an mtask and a resource. It has the same effect on resource monitors whichever task is chosen from the mtask, but its effect on event resource monitors varies with the task. This is again not a problem for $d_k$-solutions, because they make a whole day's worth of assignments at once, and that enables them to choose particular tasks from the mtasks. In principle we could carry out one dominance test for each pair $(x, y)$, where $x$ is an unassigned task from $c_i$ and $y$ is an unassigned task from $c_j$, using a signature containing one entry for each monitor that monitors $x$ or $y$, and declare that $S_1$ dominates $S_2$ when all of these tests succeed; but that seems like a bridge too far.

So although we make signatures for event resource monitors when building $d_k$-solutions, we will not do that here. Instead, we find lower bounds on available cost that suit all signatures.

It is a property of mtasks that the order in which resources are assigned to them does not matter. This allows us to make two key assumptions:

- In $S'_1$, $r$ is the last resource assigned to an unassigned task of $c_i$, after $p$ other open resources are assigned. Let $M_{i(p+1)}$ be the monitors that monitor the task assigned $r$.
- In $S'_2$, $r$ is the last resource assigned to an unassigned task of $c_j$, after $q$ other open resources are assigned. Let $M_{j(q+1)}$ be the monitors that monitor the task assigned $r$.

When an mtask represents a free day, the set of monitors is empty. The point here is that to get from $S'_1$ to $S'_2$, as we will be doing, we have to take the assignment of $r$ to a task of $c_i$ away, and we have to add an assignment of $r$ to a task of $c_j$. If these are the last assignments in their mtasks, the only event resource monitors affected by these changes are $M_{i(p+1)}$ and $M_{j(q+1)}$.

Let $x$ be the number of open resources. We previously called this $m$, but now $m$ is a monitor. In any solution, $p + q + 1 \leq x$, because the tasks of $c_i$ and $c_j$ all run on the same day, $d_{k+1}$, so they must be assigned distinct resources. This is true even when one of the mtasks represents a free day. However, we make the weaker assumptions $p \leq x - 1$ and $q \leq x - 1$. This allows us to keep the two mtasks separate, at the cost of a possibly weaker result overall.

Let $u(c_i)$ be the number of open tasks in $c_i$, and let $u(c_j)$ be the number of open tasks in $c_j$. These are infinite if the mtask represents a free day. Since $S'_2$ is given, we know that $q$ has a valid value, that is, $1 \leq q + 1 \leq u(c_j)$. But for $p$ we have the possibility that we mentioned earlier: if $p = u(c_i)$ we find that there is no task available for assignment to $r$ in $c_i$, and we have to declare that there is no dominance.

Putting all this together, we get that if $x - 1 \geq u(c_i)$ then $p = u(c_i)$ is possible and we must declare that there is no dominance. Otherwise we have $0 \leq p \leq \min(x - 1, u(c_i) - 1)$ and $0 \leq q \leq \min(x - 1, u(c_j) - 1)$.

For a given set of monitors $M$ and a given solution $S$, let

$$c(M,S) \;=\; \sum_{m \in M} c(m,S)$$

Assuming that $p$ and $q$ are as defined above, the only event resource monitors that differ in cost between $S'_1$ and $S'_2$ are $M_{i(p+1)}$ and $M_{j(q+1)}$, and so the cost difference we need is

$$\begin{aligned} & c(M_{i(p+1)} \cup M_{j(q+1)}, S'_2) \;-\; c(M_{i(p+1)} \cup M_{j(q+1)}, S'_1) \\ &\qquad = \; c(M_{i(p+1)}, S'_2) \;+\; c(M_{j(q+1)}, S'_2) \;-\; c(M_{i(p+1)}, S'_1) \;-\; c(M_{j(q+1)}, S'_1) \\ &\qquad = \; \big[c(M_{i(p+1)}, S'_2) \;-\; c(M_{i(p+1)}, S'_1)\big] \;+\; \big[c(M_{j(q+1)}, S'_2) \;-\; c(M_{j(q+1)}, S'_1)\big] \end{aligned}$$

The second line is not as trivial as it looks, because it is not impossible for some monitor $\overline{m}$ to lie in both $M_{i(p+1)}$ and $M_{j(q+1)}$. Then the first line counts the change in $\overline{m}$'s cost once, but the second line counts it twice. However, what changes for $\overline{m}$ is that one of its tasks loses $r$ while another gains it, and so $c(\overline{m}, S'_1) = c(\overline{m}, S'_2)$, assuming that the two tasks involved have equal durations. So it does not matter how often $\overline{m}$ is counted, because its contribution is 0 anyway.

Let $c^-(m, r)$ be a lower bound on the change in cost (i.e. cost after minus cost before) of $m$ when a task monitored by $m$ is unassigned resource $r$, and let $c^+(m, r)$ be a lower bound on the change in cost of $m$ when a task monitored by $m$ is assigned resource $r$. If $m \in M_{i(p+1)}$, then

$$c^-(m, r) \le c(m, S'_2) - c(m, S'_1)$$

because in proceeding from $S'_1$ to $S'_2$ what changes for the monitors of $M_{i(p+1)}$ is that the $(p + 1)$st task of $C_{(k+1)i}$ becomes unassigned from $r$. Similarly, if $m \in M_{j(q+1)}$, then

$$c^+(m, r) \le c(m, S'_2) - c(m, S'_1)$$

because in proceeding from $S'_1$ to $S'_2$ what changes for the monitors of $M_{j(q+1)}$ is that the $(q + 1)$st task of $C_{(k+1)j}$ becomes assigned $r$. Define

$$c^+(M, r) \;=\; \sum_{m \in M} c^+(m, r) \qquad \text{and} \qquad c^-(M, r) \;=\; \sum_{m \in M} c^-(m, r)$$

and choose for adding to the available cost the quantity

$$\Delta \;=\; \min_p c^-(M_{i(p+1)}, r) \;+\; \min_q c^+(M_{i(q+1)}, r)$$

This is the lower bound we need on the change in cost, because

$$\begin{aligned}
\Delta \;&=\; \min_p c^-(M_{i(p+1)}, r) \;+\; \min_q c^+(M_{j(q+1)}, r) \\
&\le\; \min_p [c(M_{i(p+1)}, S'_2) \;-\; c(M_{i(p+1)}, S'_1)] \;+\; \min_q [c(M_{j(q+1)}, S'_2) \;-\; c(M_{j(q+1)}, S'_1)] \\
&\le\; [c(M_{i(p+1)}, S'_2) \;-\; c(M_{i(p+1)}, S'_1)] \;+\; [c(M_{j(q+1)}, S'_2) \;-\; c(M_{j(q+1)}, S'_1)]
\end{aligned}$$

for all $p$ and $q$ in the range given above.

Defining $c^-$ and $c^+$ as lower bounds, and taking the minimum over all $p$ and $q$, is our alternative to the more precise but impractical method outlined above, of conducting a whole set of dominance tests using event resource monitor signatures.

Calculating $\Delta$ is straightforward. The dependence on $r$ is unavoidable, because prefer resources constraints largely determine whether dominance is possible in these cases. The dependence on $x$, the number of open resources (in the ranges of $p$ and $q$), is also unavoidable, because there may be some very undesirable open tasks in a mtask, and the only way to rule them out may be to notice that there are not enough open resources to require them to be used.

We now find $c^-(m, r)$ and $c^+(m, r)$ for each kind of event resource monitor $m$. We assume that $m$'s cost function is linear, that its weight is $w$, and that the task duration is $d$. Other cost functions can be and are handled, but the argument becomes more complicated.

If $m$ is an assign resource monitor, the costs are independent of $r$. Unassigning $r$ produces cost change $c^-(m, r) = wd$, and assigning it produces cost change $c^+(m, r_a) = -wd$.

If $m$ is a prefer resources monitor, the costs depend on whether $r$ is a preferred resource or not. If $r$ is a preferred resource, unassigning or assigning it produces no cost change. If $r$ is not a preferred resource, unassigning it produces cost change $c^-(m, r) = -wd$ and assigning it produces cost change $c^+(m, r) = wd$.

If $m$ is a limit resources monitor, the costs depend on whether $r$ is one of $m$'s resources of interest. If no, there is no change in cost. If yes, the change in cost depends on how the total number of assigned resources of interest compares with the limits. Since we want a lower bound, rather than trying to work this out in detail we simply take $c^-(m, r) = c^+(m, r) = -wd$. If instances arise where a better estimate would make a difference, we could revisit this.

Most instances contain only assign resource monitors and prefer resources monitors, both with linear cost functions. In those cases our choices for $c^-(m, r)$ and $c^+(m, r)$ are exact.

It is clearly possible to get $\Delta > 0$ in some cases. So we should add $\Delta$ to the available cost before failing any dominance test owing to negative available cost.

### C.8.3. The $c_ic_j$-solution type

A $c_ic_j$-solution is like a $c_i$-solution, except that it adds two assignments to $d_k$-solution $S$:

$$S_1 = S \cup \{(d_{k+1}, r_a, c_i), (d_{k+1}, r_b, c_j)\}$$

and

$$S_2 = S \cup \{(d_{k+1}, r_a, c_j), (d_{k+1}, r_b, c_i)\}$$

where $r_a \neq r_b$ and $c_i \neq c_j$ (without both conditions, $S_1$ and $S_2$ would be the same). For example, we might be comparing (say) assigning a morning shift to $r_a$ and a night shift to $r_b$ with assigning a night shift to $r_a$ and a morning shift to $r_b$. The assignments must be valid as usual.

The implementation does not define an object type representing one $c_ic_j$-solution. Instead, it uses two objects, a $c_i$-solution and a $c_j$-solution. It can do this because these solutions are used only for dominance testing, as follows.

We can be brief here, because much of the notation and analysis for $c_i$-solutions carries over to $c_ic_j$-solutions. We get from $S'_1$ to $S'_2$ by changing the assignment to the $(p+1)$st task of $c_i$ from $r_a$ to $r_b$, and changing the assignment to the $(q+1)$st task of $c_j$ from $r_b$ to $r_a$. There are no issues with not having a task to assign these resources to.

Let $c^{-+}(m, r_a, r_b)$ be a lower bound on the change in cost (i.e. cost after minus cost before) of $m$ when a task monitored by $m$ is unassigned resource $r_a$ and then assigned resource $r_b$. Let

$$c^{-+}(M, r_a, r_b) \;=\; \sum_{m \in M} c^{-+}(m, r_a, r_b)$$

as usual. Then following the same analysis that we used for $c_i$-solutions, we find that the value to add to the available cost is

$$\Delta \;=\; \min_p c^{-+}(M_{i(p+1)}, r_a, r_b) \;+\; \min_q c^{-+}(M_{j(q+1)}, r_b, r_a)$$

where $p$ and $q$ have the usual ranges.

If $m$ is an assign resource monitor, nothing that matters changes so $c^{-+}(m, r_a, r_b) = 0$.

If $m$ is a prefer resources monitor, it is fairly clear that $c^{-+}(m, r_a, r_b) = c^-(m, r_a) + c^+(m, r_b)$.

If $m$ is a limit resources monitor, if both resources are of interest to $m$, or both are not of interest to $m$, then $c^{-+}(m, r_a, r_b) = 0$. Otherwise we take $c^{-+}(m, r_a, r_b) = -wd$ as usual.

Now supposing that we have established that $S_1$ dominates $S_2$, our last job is to ensure that all $d_{k+1}$-solution extensions of $S_2$ are dropped. There will be many such solutions in general: all extensions of $S$ that contain $S_2$'s two $d_{k+1}$ assignments.

As we saw in the previous section, the solver has a type representing one $c_i$-solution, and if some $c_i$-solution is ruled out by dominance, its object is simply dropped from the list of its resource's possible assignments (we here exercise our right to refer to $c_i$-solution objects as assignment objects). This happens before $c_i c_j$-solution dominance testing begins. After that, all pairs of assignments that produce dominated solutions are calculated, and each assignment of each pair is placed in a list of assignments held in the other. For example, if

$$S_2 = S \cup \{(d_{k+1}, r_a, c_j), (d_{k+1}, r_b, c_i)\}$$

is dominated, $(d_{k+1}, r_a, c_j)$ goes into $(d_{k+1}, r_b, c_i)$'s list, and $(d_{k+1}, r_b, c_i)$ goes into $(d_{k+1}, r_a, c_j)$'s list.

If $S_1$ and $S_2$ dominate each other, either may be dropped, but care is needed to avoid dropping both. We iterate over all unordered pairs of solutions $\{S_1, S_2\}$, visiting each unordered pair once, not twice. For each such pair, we test whether $S_1$ dominates $S_2$, and take the appropriate action if so. But we only test whether $S_2$ dominates $S_1$ when $S_1$ does not dominate $S_2$.

Each assignment object contains a counter, initialized to zero. When the solver chooses to use assignment $(d_{k+1}, r_a, c_j)$, it increments the counter in every assignment in that assignment's list; and when it finishes with that assignment, it decrements all those counters. Then it skips any assignments whose counter is non-zero.

If the overall algorithm is expansion by resources, it suffices to place the first of the two assignments on the second's list, not each on the other's. However, this does not work for expansion by shifts, so for simplicity we always place each assignment on the other's list.

### C.8.4. The $s_i$-solution type

Define an equivalence relation between proper root tasks in which two proper root tasks are equivalent when they have the same busy times and the same workload, taking the tasks assigned to them (directly or indirectly) into account. We call one equivalence class of this relation a *shift*, because such a class will usually consist of the tasks of one shift (the day shift on Monday, or whatever). However, there are exceptions to this, notably when tasks are assigned to one another, so the reader should bear this definition in mind and not take the term 'shift' too literally.

Tasks in the same mtask are also required to be proper root tasks with the same busy times and total workload. So each mtask, considered as a set of tasks, is a subset of one shift. There are usually several mtasks in each shift, since the tasks of mtasks are required to satisfy other conditions, on their event resource monitors, which are not required by shifts.

An $s_i$-solution consists of a $d_k$-solution $S$ plus the assignment of a set of $n$ resources $R = \{r_1, \ldots, r_n\}$ to some of the open tasks of a shift $s_i$ whose tasks begin on day $d_{k+1}$:

$$S \cup \{(s_i, \langle r_1, \ldots, r_n \rangle, \langle t_1, \ldots, t_n \rangle)\}$$

We use a single *shift assignment* rather than $n$ separate assignments for an important reason: each shift assignment may assume that its resources are the only ones that will be assigned to tasks of $s_i$. Thus, the unmentioned open tasks of $s_i$ will remain unassigned, which may lead to event resource monitor costs which are included in the cost of the $s_i$-solution. (We saw a similar condition applied to $d_k$-solutions: all assignments on days up to $d_k$ inclusive are present.)

At present, dominance tests between $s_i$-solutions are used for one purpose: to find the set $U(S, s, R)$ of all undominated $s_i$-solutions for a given $S$, $s$, and $R$.

Since the tasks of shift $s$ have the same busy times and workloads, the signature value and cost of any resource monitor for a resource $r \in R$ will be the same in each of the solutions based on $(S, s, R)$. The signature value and cost of any resource monitor for a resource $r \notin R$ will also be the same, because it will just be its signature and cost in $S$. So the only monitors that matter for dominance testing are the event resource monitors that monitor the tasks of $s$, and so, to save time, the signature arrays and costs of the $s_i$-solutions we build reflect these monitors only.

$U(S, s, R)$ is created using a cut-down version of the straightforward method of expanding a $d_k$-solution $S$ into its $d_{k+1}$-solution extensions: first one resource is assigned in all possible ways, then for each of those ways the second resource is assigned in all possible ways, and so on. Instead of using all the $d_k$ mtasks, only the mtasks of $s$ are used; instead of using all the open resources, only the resources of $R$ are used; and instead of evaluating all constraints, only the event resource constraints affected by the tasks of $s$ are evaluated.

Most event resource constraints monitor tasks from just one shift. These produce a final cost once $s$ has been assigned the resources $R$, and add nothing to the signature. So in practice we can expect the signature array to be empty, meaning that $U(S, s, R)$ will contain just one solution. (This is partly why we expect to save time by using shifts.) However, the implementation handles arbitrary event resource constraints and any number of undominated solutions.

*what follows explains expansion by shifts and belongs elsewhere*

Here is how to expand a $d_k$-solution $S$ into its $d_{k+1}$-extensions with shifts.

The straightforward shift based method for this overall expansion is to assign the first open resource to each shift in turn (including a special shift denoting a free day), then for each of those cases to assign the second open resource to each shift in turn, and so on. This way, the number of combinations is easy to count. If there are $l$ shifts, plus one free day shift, there are $(l + 1)^m$ combinations. If each gives rise to just one undominated solution, as suggested above, this is significantly fewer than the $(a + 1)^m$ potential solutions from the mtask based method, where $a$ is the number of mtasks. This is because each shift usually contains several mtasks, making $l$ significantly smaller than $a$.

However, the straightforward method does not exploit the shift basis of the search very well, because until the bottom of the recursion is reached it is not clear which subset of the open resources is being assigned to each shift: the last resource could be assigned to any shift. The method we actually use comes to the same thing in the end, and so has the same number of combinations, but it fills each shift completely before moving on to the next, as follows.

Given a $d_k$-solution $S$, the first step is to calculate and cache the sets $U(S, s, R)$ for each day $d_{k+1}$ shift $s$ and each subset $R$ of the open resources. Each $U(S, s, R)$ will be needed many times while expanding $S$; calculating it just once saves a lot of time. Each $U(S, s, R)$ is represented by a simple list of solutions. The collection of all the $U(S, s, R)$ sets is stored in a trie, held in the object representing $s$, and indexed by the open resource indexes of $R$'s resources.

Given $\{s_0, s_1, s_2, \ldots, s_l\}$, the set of all shifts whose first busy time lies in day $d_{k+1}$ (with $s_0$

representing a free day), we first generate all subsets $R_1$ of open resources to assign to $s_1$. For each subset $R_1$ we traverse $U(S, s_1, R_1)$, and for each of those solutions we recurse to the second level, where we generate all subsets $R_2$ of $R - R_1$ to assign to $s_2$, and so on. Any resources unassigned after $R_l$ is assigned are then assigned to $s_0$, that is, they are given a free day. At the bottom of the recursion, the assignments of the current members of $U(S, s_1, R_1)$, $U(S, s_2, R_2)$, and so on are assembled into a $d_{k+1}$-solution which is added to the day $d_{k+1}$ table in the usual way.

The usual optimizations apply. Each shift has some minimum number of tasks that must be assigned, otherwise too high a cost will be paid, and we can prune a search path if it does not contain enough unconsumed open resources to cover all these tasks. Also, if assigning some set $R_i$ costs too much, typically because $R_i$ is too small or too large, we can prune there.

### C.8.5. The $s_i s_j$-solution type

The $s_i s_j$-solution type adds two shift assignments for day $d_{k+1}$ shifts to a $d_k$-solution $S$:

$$S \cup \{(s_a, R_a, T_a), (s_b, R_b, T_b)\}$$

We require $s_a \neq s_b$, and this forces $T_a$ and $T_b$ to be disjoint. We also require $R_a$ and $R_b$ to be disjoint, otherwise some resource is assigned twice on $d_{k+1}$, which is not allowed.

The idea here is the same as for $c_i c_j$-solutions, only applied to shifts: to find dominated pairs of shift assignments and avoid generating solutions that contain those pairs while expanding $S$. Shift pair dominance testing takes place between pairs of solutions of this form:

$$S_1 = S \cup \{(s_a, R_a, T_a), (s_b, R_b, T_b)\}$$

and

$$S_2 = S \cup \{(s_a, R_b, T'_a), (s_b, R_a, T'_b)\}$$

These two solutions start from the same $d_k$-solution $S$ and assign the same shifts $s_a$ and $s_b$ and the same resources $R_a$ and $R_b$, but the resources are assigned to opposite shifts, and the tasks they are assigned to may be different.

When $R_a$ and $R_b$ are both empty, $S_1$ and $S_2$ are valid $s_i s_j$-solutions but they are equal. Dominance testing between equal solutions is futile and could lead to the erroneous conclusion that the solution is dominated. So we allow each of $R_a$ and $R_b$ to be empty, but not both.

The tasks to which these resources are assigned in each shift are not constrained in any way, apart from needing to exist: if it is not possible to assign each of $R_a$ and $R_b$ to each of $s_a$ and $s_b$, then either or both of $S_1$ or $S_2$ cannot be constructed. However, this is not a problem in practice because we build all $s_i s_j$-solutions by taking all previously constructed $s_i$-solutions and merging pairs of them with disjoint shifts and resources in all possible ways. This produces many $s_i s_j$-solutions, but nowhere near as many as the $d_k$-solutions produced by expanding $S$ by shifts.

As usual with dominance testing, the main issue is which monitors have the same values in both solutions, allowing them to be ignored by the test. In this case, resource monitors for resources outside of $R_a \cup R_b$ can be omitted, and event resource monitors that do not monitor any tasks within $s_a$ or $s_b$ can be omitted. At the time these tests are carried out, the needed resource monitor signatures will have already been constructed, as usual; so the only expression evaluation and signature construction actually needed will be for the the event resource monitors of the two shifts, taking care that any expression dependent on both shifts is only evaluated once.

Although the signatures to be compared depend on everything in $S_1$ and $S_2$, the dominance tests themselves depend only on $d_k$, $s_a$, and $s_b$. There are relatively few pairs of shifts on any one day, and so it is quite feasible to construct the dominance test for each unordered pair of shifts $\{s_a, s_b\}$ just once, when the current solve opens.

*remainder (if any) still to do*

# Appendix D. Dynamic Programming Resource Reassignment: Implementation

*This Appendix is rather out of date. It should be up to date in the next release of KHE.* This Appendix presents the implementation of the dynamic programming algorithm for resource assignment. It is a companion to Appendix C, which describes the algorithm and the algebra underlying its cost calculations.

## D.1. Introducing the implementation

The implementation can be found in file `khe_sr_dynamic_resource.c`. This file is over 26,000 lines long, making it easily the largest of KHE's solvers. The code is presented here in the order that it appears in the file, with some exceptions.

The implementation defines over sixty types, not counting array types. We group them into ten 'major categories'. Here they are, with examples of their types:

| Category | KHE type | Dynamic resource solver (DRS) type |
|---|---|---|
| Times | `KHE_TIME_GROUP` (common frame) | `KHE_DRS_DAY` |
| Resources | `KHE_RESOURCE` | `KHE_DRS_RESOURCE` |
| Events | `KHE_TASK` | `KHE_DRS_TASK` |
| Signatures | - | `KHE_DRS_SIGNATURE` |
| Constraints | `KHE_CONSTRAINT` | `KHE_DRS_CONSTRAINT` |
| Expressions | `KHE_MONITOR` | `KHE_DRS_EXPR` |
| Solutions | `KHE_SOLN` | `KHE_DRS_SOLN` |
| Expansion | - | `KHE_DRS_EXPANDER` |
| Sets of solutions | `KHE_SOLN_GROUP` | `KHE_DRS_SOLN_SET` |
| Solvers | - | `KHE_DYNAMIC_RESOURCE_SOLVER` |

Seven of these categories resemble categories found in KHE: times, resources, events, constraints, expressions, solutions, and sets of solutions. The other three have no KHE equivalents.

The solver utilizes two kinds of trees: search trees, whose nodes represent partial solutions and have type `KHE_DRS_SOLN`, and expression trees, representing constraints, whose nodes have type `KHE_DRS_EXPR`. This makes the term 'node' ambiguous, so it will not be used. Instead, search tree nodes will be called solutions, and expression tree nodes will be called expressions.

`KheDynamicResourceSolverMake(soln, rt, options)` creates data structures for all of `soln` relevant to resource type `rt`. It saves time, when there are many solves, for as many objects as possible to be created just once like this, at the start. Objects that are created during individual solves come from free lists, recycled from previous solves.

When `KheDynamicResourceSolverMake` returns, all its objects are in the *closed* state, meaning that they are not part of any solve. Closed objects contain values that reflect the initial solution. At the start of each solve, a process called *opening* occurs, which identifies the DRS objects that are part of that solve. Opening also unassigns any KHE tasks affected by the solve that happen to be assigned initially. Its running time depends on the number of objects opened, not on the total number of objects.

At the end of each solve, an opposite process called *closing* occurs. It returns the open objects to the closed state, with values that reflect the new best solution found by the solve, if there is one, or the initial solution if there isn't. Closing also performs KHE task assignments to change the KHE solution into the new best solution, or return it to the initial solution.

In between opening and closing we build the search tree, a process we call *searching*. So the implementation has four main operations: *construction* of a solver object and its many associated objects (a slow but easy job which needs little documentation); opening; searching; and closing. The last three operations, carried out in sequence, make one *solve*.

Another key operation, part of searching, is *expansion*. This takes one $d_k$-solution $S$ as its main argument and *expands* it, that is, it adds one assignment for each open resource on day $d_{k+1}$ to $S$ in all possible ways, and stores the resulting $d_{k+1}$-solutions within the day $d_{k+1}$ table of solutions, after checking for dominance relations. Searching is in fact just a sequence of expansions, starting with the unique $d_0$-solution, then proceeding to the $d_1$-solutions, then to the the $d_2$-solutions, and so on.

The code is organized hierarchically. At the top level are the ten *major categories* given above, in the order shown above. For each of these, the second level contains one *submodule* for each of the types of that major category. Within each submodule are the operations on its type, typically organized into construction, simple queries, opening, closing, and other (including debugging), in that order. Helper functions appear wherever seems best.

Searching is mostly about expansion. The expansion algorithm is lengthy and distributed over many types, but is best understood as a whole. So its code appears in special submodules distributed through the implementation, but documented in one place here (Appendix D.10).

## D.2. Exactly what gets opened

Before diving into the code, we pause to explain exactly what gets opened. This question has puzzled the author, so here we set out a precise, complete answer.

We view that part of the initial solution that gets opened as a planning timetable, in which each column represents a selected day, and each row represents a selected resource:

| | Day 5 | Day 6 | Day 12 | Day 13 | Day 19 | Day 20 |
|---|---|---|---|---|---|---|
| Resource 3 | | | | | | |
| Resource 4 | | | | | | |
| Resource 7 | | | | | | |

This shows six selected days (presumably three weekends) and three selected resources. In any solution, for each selected resource $r$ and each selected day $d$, the table cell $(r, d)$ either contains a single proper root task, one that $r$ is assigned to and that (counting all the tasks assigned to it directly or indirectly) is running on day $d$, or else it is empty, indicating that $r$ is free on $d$. This is the view of the problem taken by the solver, which proceeds from left to right across the table, filling in the cells one day at a time.

Everything that gets opened lies within this table, but the converse is not quite true. Most things within this table get opened, but with the following exceptions. Let $r$ be any selected resource, and let $d$ be any selected day. The assignments referred to are in the initial solution, the solution passed in to the solver.

1. If $r$ is assigned to some proper root task on $d$, and the busy day range of that task (counting all the tasks assigned to it directly or indirectly) includes one or more unselected days, then the cell $(r, d)$ is not opened, that is, the solver will leave that assignment untouched.

2. If $r$ is preassigned to some proper root task on $d$, and also assigned to that task, then even if that assignment can be removed, it won't be removed, and again $(r, d)$ is not opened.

3. If $r$ is assigned to some proper root task on $d$, and that assignment cannot be removed (because `KheTaskUnAssign` returns `false`), then $(r, d)$ is not opened.

In the unlikely case that $r$ is preassigned to some task on $d$ but not assigned to it, cell $(r, d)$ is opened in the usual way, but the solver will try only one option for assigning it, namely the preassigned task. It will not try leaving the cell unassigned.

The data structure representing solutions contains one element for each cell $(r, d)$. A cell that is not opened is given a special `CLOSED` value, saying that the solver should not make and has not made an assignment to this cell. (The other special value, `NULL`, means that the solver has decided that $r$ should be free on $d$). After opening, the table might be

| | Day 5 | Day 6 | Day 12 | Day 13 | Day 19 | Day 20 |
|---|---|---|---|---|---|---|
| Resource 3 | `CLOSED` | | | | | |
| Resource 4 | | | | | | `CLOSED` |
| Resource 7 | | | `CLOSED` | `CLOSED` | | |

A `CLOSED` entry may appear anywhere. It represents a point where the solve may not change the initial state. Conceptually, the entire solution outside this table is filled with `CLOSED` entries.

An expression whose value depends directly on the timetable of some resource always depends on what that resource is doing on one specific day. It is open when the cell representing that resource on that day is not `CLOSED`.

An expression whose value depends directly on whether some proper root task is assigned or not, and possibly on what it is assigned to, is open when that task is open. A proper root task is open when its busy day range lies entirely within the set of selected days, its domain has a non-empty intersection with the set of selected resources, and it is either unassigned or else neither of the two impediments to removing its assignment (points (2) and (3) above) apply.

An expression whose value depends directly on the values of its children is open when at least one of its children is open. It may have some open and some closed children.

## D.3. Times

This section describes the two DRS types related to times. The first is `KHE_INTERVAL`, defined in file `khe_solvers.h` to represent an integer interval (Section 8.12). When used here it always represents an interval of days. The integers are indexes into the common frame, or into an array of open days.

The second time-related type is `KHE_DRS_DAY`. It represents one day, that is, one time group of the common frame:

```
typedef struct khe_drs_day_rec *KHE_DRS_DAY;
typedef HA_ARRAY(KHE_DRS_DAY) ARRAY_KHE_DRS_DAY;

struct khe_drs_day_rec {
  int                             frame_index;
  int                             open_day_index;
  KHE_TIME_GROUP                  time_group;
  ARRAY_KHE_DRS_SHIFT             shifts;
  KHE_DRS_SIGNER_SET              signer_set;
  KHE_DRS_SOLN_SET                soln_set;
  KHE_DRS_SOLN_LIST               soln_list;
  int                             soln_made_count;
  int                             solve_expand_count;
};
```

The `frame_index` field is the day's time group's index in the common frame. It is a fixed value, set when the day is created during `KheDynamicResourceSolverMake`. The `open_day_index` field is the day's index in the list of open days when it is open, and `-1` when it is closed. So it is fixed during any one solve, but may vary from one solve to the next. The `time_group` field holds the time group defining the day, taken from the common frame.

The `shifts` field holds a set of *shifts*. Each shift contains a set of similar mtasks; each mtask contains a set of similar tasks. The first time of every task in every mtask of every shift lies in this day's time group. Full details will be given when we come to these other types.

The `signer_set` field holds a set of *signers*, which are rather artificial objects representing templates for how to construct signatures and perform dominance testing for solutions which are $d_k$-solutions where $d_k$ is this day. Signatures and signers will be explained later (Section D.6). A signer set object is always present, but its value is set up afresh each time the day is opened.

The `soln_set` field is only defined when the day is open. It is initialized to empty at the start of each solve, but comes to hold the set of all undominated solutions within which resources are assigned tasks up to and including this day. This field was called $P_k$ in Appendix C.1.

The `soln_list` field holds a simple list of solutions. It is created by traversing `soln_set` after it is completed, and adding each solution found there to `soln_list`. Once the solutions are in the list, they are sorted by increasing cost, and then an optional daily limit on the number of solutions may be enforced, by removing solutions from the end until the limit is not exceeded.

The `soln_made_count` field holds the number of solutions created during the current solve that end on this day (not the number of undominated solutions). The `solve_expand_count` field is used during solving and will be explained later.

The operations on days are quite simple. Creation is easy. Opening sets open_day_index and clears the fields defined when the day is open. They are set properly later, when opening expressions, as explained below. Closing reverses opening, and also frees the solutions stored in the day. Searching adds solutions to the soln_set field but leaves the day object itself untouched.

There are also two operations, KheDrsDayExpandBegin and KheDrsDayExpandEnd, which are part of expansion. Following our policy, these have been placed into a separate submodule and documented elsewhere (Appendix D.10).

## D.4. Resources

This section describes the DRS types related to resources.

### D.4.1. The resource type

Type KHE_DRS_RESOURCE represents one resource. Here is its type definition:

```
typedef struct khe_drs_resource_rec *KHE_DRS_RESOURCE;
typedef HA_ARRAY(KHE_DRS_RESOURCE) ARRAY_KHE_DRS_RESOURCE;

struct khe_drs_resource_rec {
  KHE_RESOURCE                   resource;
  int                            open_resource_index;
  ARRAY_KHE_DRS_RESOURCE_ON_DAY  days;
  KHE_DRS_RESOURCE_EXPAND_ROLE   expand_role;
  ARRAY_KHE_DRS_SIGNATURE        expand_signatures;
  ARRAY_KHE_DRS_MTASK_SOLN       expand_mtask_solns;
  KHE_DRS_MTASK_SOLN             expand_free_mtask_soln;
  KHE_DRS_DIM2_TABLE             expand_dom_test_cache;
};
```

The last five fields are used by expansion and will be explained later (Appendix D.10). The other fields hold the corresponding KHE resource, the resource's index in the array of open resources when open (or -1 when closed), and an array of KHE_DRS_RESOURCE_ON_DAY objects, one for each day of the cycle, recording what the resource is doing on that day (Appendix D.4.2).

Resource and resource on day objects are easily built during the initialization of the solver. The most complex resource operation is the one for opening a resource on the selected days:

```
void KheDrsResourceOpen(KHE_DRS_RESOURCE dr, int open_resource_index,
  KHE_DRS_PACKED_SOLN init_soln, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_DAY_RANGE ddr;  int i, j, open_day_index;
  KHE_DRS_RESOURCE_ON_DAY drd;
  dr->open_resource_index = open_resource_index;
  open_day_index = 0;
  HaArrayForEach(drs->selected_day_ranges, ddr, i)
    for( j = ddr.first;  j <= ddr.last;  j++ )
    {
      /* unassign any task assigned on drd, if it lies entirely in ddr */
      drd = HaArray(dr->days, j);
      KheDrsResourceOnDayOpen(drd, open_resource_index, open_day_index,
        ddr, init_soln, drs);

      /* increase open_day_index */
      open_day_index++;
    }
}
```

The first step is to set `dr->open_resource_index`. After that, the two outer loops set `j` to the index in the cycle of each selected day, so each iteration of the inner loop opens one of `dr`'s resource on day objects.

The matching `KheDrsResourceClose` operation resets `dr->open_resource_index` to `-1` and calls `KheDrsResourceOnDayClose` repeatedly to clear the signer in each resource on day object. It does not make any task assignments, because `KheDrsTaskAssign` below does that, including setting the `closed_asst` fields in the affected resource on day objects.

After the `KHE_DRS_RESOURCE` submodule there is another submodule concerned with that part of the expansion operation that is concerned with resources. This submodule, and the `expand_assignments` and similar fields that we passed over briefly earlier, are presented in Appendix D.10.

### D.4.2. The resource on day type

Type `KHE_DRS_RESOURCE_ON_DAY` represents what one resource is doing on one day. Here is its type definition:

```
typedef struct khe_drs_resource_on_day_rec *KHE_DRS_RESOURCE_ON_DAY;
typedef HA_ARRAY(KHE_DRS_RESOURCE_ON_DAY) ARRAY_KHE_DRS_RESOURCE_ON_DAY;
```

```
struct khe_drs_resource_on_day_rec {
  KHE_DRS_RESOURCE              encl_dr;
  KHE_DRS_DAY                   day;
  bool                          open;
  KHE_DRS_TASK_ON_DAY           closed_dtd;
  KHE_DRS_TASK_ON_DAY           preasst_dtd;
  ARRAY_KHE_DRS_EXPR            external_today;
  KHE_DRS_SIGNER                signer;
};
```

Here `encl_dr` and `day` hold the DRS resource and day that this object is for; they are fixed. The `open` field is `true` when the resource is open on this day (when there is a solve underway, and this resource is open to reassignment on this day by the solve).

Suppose `dr` is an object of type `KHE_DRS_RESOURCE`, and suppose `drd` is one of its `KHE_DRS_RESOURCE_ON_DAY` objects. When `open` is `false`, `drd->closed_dtd` says what `dr` is doing on that day. It will be `NULL` if `dr` is free on that day. When `open` is `true`, `drd->closed_dtd` is unused and has value `NULL`.

The `preasst_dtd` field is always defined. If `dr` is preassigned to some task on this day, its value is the task on day that `dr` is preassigned to. Otherwise its value is `NULL`.

The `external_today` field is a fixed array of expressions representing parts of constraints (always resource constraints) that are affected by what `drd` is doing on this day. When what `drd` is doing changes, these expressions need to be informed. They are *external* expressions: they are leaves in their expression trees.

Similarly to day objects, the `signer` field contains a template for the signatures of solutions that end at this resource on day. These are solutions that are complete up to the day before this day, plus they hold one assignment, for this resource on this day.

Here is the operation for opening a resource on day object:

```
void KheDrsResourceOnDayOpen(KHE_DRS_RESOURCE_ON_DAY drd,
  int open_resource_index, int open_day_index, KHE_INTERVAL ddr,
  KHE_DRS_PACKED_SOLN init_soln, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_TASK dt;  KHE_DRS_EXPR e;  int i;  KHE_DRS_TASK_ON_DAY dtd;
  KHE_INTERVAL open_day_range;  KHE_RESOURCE r;

  /* unassign any affected task; possibly add the assts to init_soln */
  dtd = drd->closed_dtd;
  if( dtd != NULL )
  {
    dt = dtd->encl_dt;
    if( KheIntervalSubset(dt->encl_dmt->day_range, ddr) &&
        !KheTaskIsPreassigned(dt->task, &r) &&
        KheDrsTaskUnAssign(dt, true) )
    {
      /* dt has been successfully unassigned */
      drs->solve_start_cost -= dt->asst_cost;
      if( init_soln != NULL )
        KheDrsPackedSolnSetTaskOnDay(init_soln, open_day_index,
          open_resource_index, dtd);
    }
  }

  /* set make_correlated field of signer */
  if( drs->solve_correlated_exprs )
    KheDrsSignerMakeCorrelated(drd->signer);

  if( drd->closed_dtd == NULL )
  {
    /* open drd and gather its expressions for opening */
    drd->open = true;
    open_day_range = KheIntervalMake(open_day_index, open_day_index);
    HaArrayForEach(drd->external_today, e, i)
    {
      e->open_children_by_day.index_range = open_day_range;
      KheDrsExprGatherForOpening(e, drs);
    }
  }
}
```

The first paragraph unassigns any task assigned to `dr` on `drd`'s day, unless it is part of a multi-day task which extends beyond the current day range, or is preassigned, or will not unassign for any reason. A successful unassignment includes adding this assignment to packed solution `init_soln` (see Appendix D.9.9) so that it can be redone later if required.

The second step informs `drd->signer`, when appropriate, that it is to find correlations among expressions. This is a subject for later (Appendix D.6).

The third step gathers the expressions dependent on `drd` into a list. These expressions need to be opened, but that is delayed until that list is traversed later.

A potentially confusing point is that the calls to `KheDrsTaskUnAssign` unassign KHE tasks and so change the cost of the solution. Does this cause problems for the cost accounting? No, because the original solution cost is saved before these unassignments are made, and the costs stored in expressions are not affected by them: when those expressions are opened later, they subtract their costs from the total, and those costs do not take these unassignments into account.

The matching `KheDrsResourceOnDayClose` operation just clears the signer in the resource on day object and sets the `open` flag to `false`. It does not make any task assignments, because `KheDrsTaskAssign` below does that, including setting the `closed_asst` fields in the affected resource on day objects.

### D.4.3. Resource sets

A *resource set* is a set (actually a sequence) of resource objects:

```
typedef struct khe_drs_resource_set_rec *KHE_DRS_RESOURCE_SET;
typedef HA_ARRAY(KHE_DRS_RESOURCE_SET) ARRAY_KHE_DRS_RESOURCE_SET;

struct khe_drs_resource_set_rec {
  ARRAY_KHE_DRS_RESOURCE        resources;
};
```

There are operations for creating a new set, adding one resource on day to a set, iterating over the elements of a set (macro `KheDrsResourceSetForEach`), and so on.

## D.5. Events

This section presents the types related to events. All events have fixed times in this application, so actually it is the events' tasks that matter. The main DRS types are `KHE_DRS_TASK` representing one task, `KHE_DRS_MTASK` representing one mtask, and `KHE_DRS_SHIFT` representing one set of similar mtasks.

### D.5.1. Tasks

For each proper root task of the required resource type, there is a corresponding DRS task:

```
typedef struct khe_drs_task_rec *KHE_DRS_TASK;
typedef HA_ARRAY(KHE_DRS_TASK) ARRAY_KHE_DRS_TASK;

struct khe_drs_task_rec {
  KHE_DRS_MTASK                 encl_dmt;
  int                           index_in_encl_dmt;
  KHE_DRS_TASK_EXPAND_ROLE      expand_role;
  bool                          open;
  KHE_TASK                      task;
  KHE_DRS_RESOURCE              closed_dr;
  KHE_COST                      non_asst_cost;
  KHE_COST                      asst_cost;
  ARRAY_KHE_DRS_TASK_ON_DAY     days;
};
```

Each DRS task lies in one DRS mtask (Appendex D.5.2); `encl_dmt` is that mtask, and `index_in_encl_dmt` is the task's index in that mtask. The `expand_role` field is used by `KheDrsSolnExpand` and is explained later (Appendix D.10). The `open` field is `true` when this task is open (when there is a current solve and this task may be assigned or reassigned by it).

The `task` field is the corresponding KHE proper root task. When a DRS task is open, its `closed_dr` field is `NULL` and its KHE task is unassigned. When a DRS task is closed, its `closed_dr` field is set to the DRS resource corresponding to the KHE resource assigned to the KHE task, or to `NULL` when the KHE task is unassigned.

The `non_asst_cost` field is a constant lower bound on the cost of not assigning `task`, and the `asst_cost` field is a constant lower bound on the cost of assigning it. These values come from `KheMTaskTask` (Section 11.9.1).

The `days` field holds one `KHE_DRS_TASK_ON_DAY` object for each day the task is running. These are sorted chronologically, and the implementation uses this in one place to check whether a given task on day object is for the first day of its task. Here is `KHE_DRS_TASK_ON_DAY`:

```
typedef struct khe_drs_task_on_day_rec *KHE_DRS_TASK_ON_DAY;
typedef HA_ARRAY(KHE_DRS_TASK_ON_DAY) ARRAY_KHE_DRS_TASK_ON_DAY;

struct khe_drs_task_on_day_rec {
  KHE_DRS_TASK                  encl_dt;
  KHE_DRS_DAY                   day;
  KHE_TASK                      task;
  KHE_TIME                      time;
  KHE_DRS_RESOURCE_ON_DAY       closed_drd;
  ARRAY_KHE_DRS_EXPR            external_today;
};
```

Here `encl_dt` is the enclosing DRS task, `day` is the day concerned, and `task` is the KHE task running on this day: either the original KHE proper root task, or some other KHE task assigned, directly or indirectly, to that task. Also, `time` is the time within `day` that `task` is running. The `task` and `time` fields are always well-defined and non-`NULL`, because, as specified in Section 12.6, a multi-day task must run on every day of its busy day range, and it cannot run twice on one day.

These conditions always hold, because if they don't, a solver is not created.

The `closed_drd` field holds the resource on day object that this task on day is assigned to when the task is closed. It is non-`NULL` exactly when the adjacent `task` field and the `closed_dr` field of the enclosing DRS task are both non-`NULL`.

Finally, `external_today` holds a list of all external expressions (expressions with no child expressions) whose value depends on what this task is doing on this day. This is similar to the `external_today` field in resource on day objects, except that these leaves lie in expression trees representing event resource constraints (assign resource, prefer resources, and limit resources constraints) rather than in expression trees representing resource constraints. When the assignment of the task represented here changes, these expressions need to be informed.

This function makes a closed assignment of a DRS resource to a DRS task:

```
bool KheDrsTaskAssign(KHE_DRS_TASK dt, KHE_DRS_RESOURCE dr, bool task)
{
  KHE_DRS_TASK_ON_DAY dtd;  int i;  KHE_DRS_RESOURCE_ON_DAY drd;
  HnAssert(dt->closed_dr == NULL, "KheDrsTaskAssign internal error 1");
  HnAssert(dr != NULL, "KheDrsTaskAssign internal error 2");
  if( task && !KheTaskAssignResource(dt->task, dr->resource) )
    HnAbort("KheDrsTaskAssign internal error 3 (cannot assign %s to %s)",
      KheDrsResourceId(dr), KheTaskId(dt->task));
  dt->closed_dr = dr;
  HaArrayForEach(dt->days, dtd, i)
  {
    drd = KheDrsResourceOnDay(dr, dtd->day);
    HnAssert(dtd->closed_drd == NULL, "KheDrsTaskAssign internal error 4");
    if( drd->closed_dtd != NULL )
      return false;
    dtd->closed_drd = drd;
    drd->closed_dtd = dtd;
  }
  return true;
}
```

`KheDrsTaskAssign` only omits calling `KheTaskAssignResource` when the object is first built. `KheDrsResourceOnDay` returns the resource on day object representing what `dr` is doing on `dtd->day`. When `dtd->closed_drd` or `drd->closed_dtd` changes, the expressions in their `external_today` arrays must be informed. This is done separately from `KheDrsTaskAssign`.

At the time a DRS task is first made, if the corresponding KHE task is assigned a resource, then `KheDrsTaskAssign` is called to assign the DRS task correspondingly, which includes setting the `closed_drd` fields of the resource's resource on day objects, as shown above. At this point, `KheDrsTaskAssign` could discover that one of these fields is already set, meaning that in the initial state, the resource is assigned to two tasks on the same day. The solver cannot handle this, so if it occurs, `KheDrsTaskAssign` returns `false`, and `KheDynamicResourceSolverMake` takes this as a signal to discard the solver object it was initializing and return `NULL`.

It will become clear (Appendix D.5.2) that only unassigned tasks are ever opened, so all that needs to be done when opening a task is to set its `open` field to `true` and to gather for opening all the expressions in the `external_today` arrays of its task on day objects:

```
void KheDrsTaskOpen(KHE_DRS_TASK dt, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_TASK_ON_DAY dtd;  KHE_DRS_EXPR e;  int i, j, di;
  KHE_DRS_DAY_RANGE open_day_range;

  /* open dt */
  HnAssert(!dt->open, "KheDrsTaskOpen internal error 1");
  HnAssert(dt->closed_asst == NULL, "KheDrsTaskOpen internal error 2");
  dt->open = true;

  /* gather external expressions for opening */
  HaArrayForEach(dt->days, dtd, i)
  {
    di = dtd->day->open_day_index;
    open_day_range = KheDrsDayRangeMake(di, di);
    HaArrayForEach(dtd->external_today, e, j)
    {
      e->open_day_range = open_day_range;
      KheDrsExprGatherForOpening(e, drs);
    }
  }
}
```

Closing a DRS task sets the `open` field to `false`, and may also assign a DRS resource:

```
void KheDrsTaskClose(KHE_DRS_TASK dt, KHE_DRS_RESOURCE dr)
{
  if( dt->open )
  {
    dt->open = false;
    if( dr != NULL )
      KheDrsTaskAssign(dt, dr, true);
  }
}
```

`KheDrsTaskClose` may be called on the same task several times, but does the work only once.

## D.5.2. Multi-tasks

The basic idea of multi-tasks, or mtasks as we prefer to call them, is that in every instance there are often equivalent tasks. To be equivalent, two tasks must run at the same times, but they must also be subject to the same constraints, so that assigning a resource to one task of an mtask is really the same as assigning it to another. When trying alternative assignments we can save a lot of time by recognizing this and avoiding alternatives which formally are different but in reality are equivalent.

The dynamic resource solver calls on a multi-task finder from Section 11.9 to partition the set of all the proper root tasks of the required resource type into mtasks. Then for each of these `KHE_MTASK` objects it makes one `KHE_DRS_MTASK` object, and for each KHE task in each mtask it adds one DRS task to the DRS mtask. It calls `KheMTaskNoOverlap` on each mtask, and if any of the calls return `false`, no solver is created. So it is safe for the solver to assume that none of its tasks run twice at the same time or on the same day.

The type declarations for `KHE_DRS_MTASK` are:

```
typedef struct khe_drs_mtask_rec *KHE_DRS_MTASK;
typedef HA_ARRAY(KHE_DRS_MTASK) ARRAY_KHE_DRS_MTASK;

struct khe_drs_mtask_rec {
  KHE_MTASK                      orig_mtask;
  KHE_DRS_SHIFT                  encl_shift;
  KHE_DRS_DAY_RANGE              day_range;
  ARRAY_KHE_DRS_TASK             all_tasks;
  ARRAY_KHE_DRS_TASK             unassigned_tasks;
  int                            expand_must_assign_count;
  int                            expand_prev_unfixed;
};
```

Here `orig_mtask` is the `KHE_MTASK` that this DRS mtask is derived from, `encl_shift` is the shift (see below) that this mtask lies within, `day_range` says which days the tasks of this mtask are busy (they are all busy at the same times, hence on the same days), `all_tasks` contains the DRS tasks corresponding to the KHE tasks of `orig_mtask`, and `unassigned_tasks` contains those tasks from `all_tasks` which are open during the current solve. The last two fields, `expand_must_assign_count` and `expand_prev_unfixed`, are used by `KheDrsSolnExpand` and will be explained later (Appendix D.10).

During solving, we want `unassigned_tasks` to contain the open tasks of this mtask, that is, the tasks from `all_tasks` which are available for the open resources to be assigned to. By the time that solving starts, these tasks will all be unassigned. When we build `unassigned_tasks` at the start of each solve, there are two issues.

First, two tasks lying in the same mtask may differ in the cost incurred by assigning (or not assigning) them. Those which are least costly come first in the mtask, and should be chosen for assignment before later tasks in the mtask. This is explained fully in Section 11.9.4. So open tasks must appear within `unassigned_tasks` in the same order that they appear in `all_tasks`.

Second, we want the running time of opening and closing to be proportional to the number of objects opened, not the total number of objects. Accordingly, we cannot build `unassigned_tasks` by traversing `all_tasks` when opening, because `all_tasks` may contain many tasks which will not be opened, because they are assigned unselected resources.

So we proceed as follows. When the DRS mtask is created, `unassigned_tasks` is initialized to contain all unassigned DRS tasks from `all_tasks`. Whenever a DRS task from `all_tasks` is unassigned, its `encl_dts` field is followed to its enclosing DRS mtask and it is added to `unassigned_tasks`. But when it is assigned, it is not deleted from `unassigned_tasks`. So at any moment, `unassigned_tasks` must contain all the unassigned tasks from `all_tasks`, but it may contain some assigned tasks as well.

Each DRS mtask is stored in the `mtasks` field of one shift object, and that shift is stored in a day object which is the first day on which the mtask's tasks are busy. If that day is one of the selected days for solving, as part of opening it, each of its mtasks is visited and potentially opened by a call to `KheDrsMTaskOpen`:

```
bool KheDrsMTaskOpen(KHE_DRS_MTASK dmt, KHE_DRS_DAY_RANGE ddr,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_TASK dt;  int i;  bool res;
  if( KheDrsDayRangeSubset(dmt->day_range, ddr) &&
      !KheResourceSetDisjointGroup(drs->selected_resource_set,
        KheMTaskDomain(dmt->orig_mtask)) )
  {
    /* dmt can open; organize and open unassigned_tasks */
    KheDrsMTaskOrganizeUnassignedTasks(dmt);
    HaArrayForEach(dmt->unassigned_tasks, dt, i)
      KheDrsTaskOpen(dt, drs);
    dmt->expand_must_assign_count = 0;
    dmt->expand_prev_unfixed = -1;
    res = true;
  }
  else
  {
    /* dmt can't open; set dmt->expand_prev_unfixed to make that clear
*/
    dmt->expand_prev_unfixed = HaArrayCount(dmt->unassigned_tasks) - 1;
    res = false;
  }
  return res;
}
```

This function is slightly mis-named: it only opens `dmt` if its tasks lie entirely within open day range `ddr` and their shared domain is not disjoint from the set of open resources.

Opening an mtask begins by sorting `unassigned_tasks` so that the genuinely unassigned tasks come first, in their order in `all_tasks` (the `index_in_encl_dmt` field helps with this), and deleting any assigned tasks from the end. `KheDrsMTaskOrganizeUnassignedTasks` does these two steps. After that, the unassigned tasks are opened. This way, the issues identified above are handled correctly. This is done after resources are opened, by which time all tasks from the mtask that were assigned a selected resource are unassigned, and so lie in `unassigned_tasks`, justifying the statement made earlier that only unassigned tasks are ever opened.

The mtask submodule is followed by another submodule containing mtask code related to expansion. As usual, we'll return to this later (Appendix D.10).

### D.5.3. Shifts

To the solver, a *shift* is a set of mtasks whose tasks all have the same busy times and workloads. Usually this will be the tasks of one shift (defined informally), but not always; for example, not when some of the tasks are grouped. When a resource *r* is assigned to any of the tasks of the mtasks of a given shift, the effect on *r*'s resource constraints is the same.

Here is the type declaration:

```
typedef struct khe_drs_shift_rec *KHE_DRS_SHIFT;
typedef HA_ARRAY(KHE_DRS_SHIFT) ARRAY_KHE_DRS_SHIFT;

struct khe_drs_shift_rec {
  KHE_DRS_DAY                    encl_day;
  int                            open_shift_index;
  int                            expand_must_assign_count;
  int                            expand_max_included_free_resource_count;
  ARRAY_KHE_DRS_MTASK            mtasks;
  ARRAY_KHE_DRS_MTASK            open_mtasks;
  ARRAY_KHE_DRS_SHIFT_PAIR       shift_pairs;
  KHE_DRS_SIGNER                 signer;
  KHE_DRS_SHIFT_SOLN_TRIE        soln_trie;
};
```

Field `encl_day` is the day containing this shift (the day containing the first busy time of this shift's tasks). Field `open_shift_index` is `-1` when the shift is not open for solving, and has a unique non-negative value when the shift is open.

Fields `expand_must_assign_count` and `expand_max_included_free_resource_count` are defined during the expansion of a day solution from the previous day and will be explained later. Field `mtasks` holds the mtasks that contain the tasks of the shift, and `open_mtasks` holds the open ones when the shift is open for solving. Field `shift_pairs` contains objects representing all pairs of shifts from the same day whose first element is this shift.

Field `signer` holds a signer for evaluating signatures and dominance testing for the event resource monitors that monitor the tasks of this shift, and field `soln_trie` holds a set of shift solutions, that is, objects representing the assignment of sets of resources *R* to tasks of this shift. These will be explained later.

The operations on shifts include `KheDrsShiftMake` for making a new shift, initially holding one mtask, and `KheDrsShiftAcceptsMTask` for deciding whether to add a given mtask to a shift, because its busy times and workloads are the same as those of the mtasks already in the shift. There is also `KheDrsShiftOpen`, which opens a shift by assigning an open shift index to it and opening its mtasks:

```
void KheDrsShiftOpen(KHE_DRS_SHIFT ds, KHE_DRS_DAY_RANGE ddr,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_MTASK dmt;  int i;
  ds->open_shift_index = HaArrayCount(drs->open_shifts);
  HaArrayAddLast(drs->open_shifts, ds);
  HaArrayForEach(ds->mtasks, dmt, i)
    if( KheDrsMTaskOpen(dmt, ddr, drs) )
      HaArrayAddLast(ds->open_mtasks, dmt);
}
```

and `KheDrsShiftClose`, which closes its mtasks and clears its signer:

```
void KheDrsShiftClose(KHE_DRS_SHIFT ds, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_MTASK dmt;  int i;
  HaArrayForEach(ds->open_mtasks, dmt, i)
    KheDrsMTaskClose(dmt);
  HaArrayClear(ds->open_mtasks);
  ds->open_shift_index = -1;
  KheDrsSignerClear(ds->signer, drs);
}
```

After the `KHE_DRS_SHIFT` submodule there is another submodule which implements that part of the expansion operation concerned with shifts. That submodule is presented in Appendix D.10.

### D.5.4. Shift pairs

A *shift pair* is a pair of shifts:

```
typedef struct khe_drs_shift_pair_rec *KHE_DRS_SHIFT_PAIR;
typedef HA_ARRAY(KHE_DRS_SHIFT_PAIR) ARRAY_KHE_DRS_SHIFT_PAIR;

struct khe_drs_shift_pair_rec {
  KHE_DRS_SHIFT                shift[2];
  KHE_DRS_SIGNER               signer;
};
```

When the solver is created, one shift pair object is created for each pair of distinct shifts whose tasks' first times lie within the same day.

The main purpose of this type is to store the signer, which is used to control the signatures of shift pair solutions. Apart from `KheDrsShiftPairMake`, the only shift pair functions are functions related to the signer.

## D.6. Signatures

This section presents the types concerned with signatures and dominance testing.

### D.6.1. Signatures

The author is guilty of ambivalence in the use of the term *signature*. It can mean just an array of numbers (each an `int` or a `float`) recording the current states of some active monitors, but it can also mean a solution cost in addition to the array. This second meaning predominates in this section.

The type declaration for signatures is

```
typedef struct khe_drs_signature_rec *KHE_DRS_SIGNATURE;
typedef HA_ARRAY(KHE_DRS_SIGNATURE) ARRAY_KHE_DRS_SIGNATURE;

struct khe_drs_signature_rec {
  int                   reference_count;
  int                   asst_to_shift_index;
  KHE_COST              cost;
  ARRAY_KHE_DRS_VALUE   states;
};
```

`KHE_DRS_VALUE` is an untagged union of `int` and `float`:

```
typedef union {
  int                   i;
  float                 f;
} KHE_DRS_VALUE;
```

States derived from limit workload monitors are the only ones to use floating-point values. The context determines which field is currently in use.

Signatures have unpredictable lifetimes, but they are widely used and it is important to recycle them. Accordingly, a reference counting system is used. The `reference_count` field records the number of references to this object from other heap-allocated objects. Those other objects are required to register the addition or deletion of a reference to a signature, by calling

```
void KheDrsSignatureRefer(KHE_DRS_SIGNATURE sig)
{
  sig->reference_count++;
}

void KheDrsSignatureUnRefer(KHE_DRS_SIGNATURE sig,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  sig->reference_count--;
  if( sig->reference_count == 0 )
    HaArrayAddLast(drs->signature_free_list, sig);
}
```

`KheDrsSignatureUnRefer` adds `sig` to a free list in `drs` when its reference count drops to 0.

The `asst_to_shift_index` field is used to implement caching of dominance tests. It holds the index of the signature in an enclosing array.

The last two fields hold the value of the signature. Each signature depends on a particular set of monitors; its cost is the total cost of those monitors, and its states are state values for the monitors, as required. For each signature there is a signer (Appendix D.6.3) which knows which monitors these are. It would make a lot of sense to store a pointer to this signer in the signature. However, to save space (given that solutions contain signatures, and there can be many thousands of solutions), this pointer has been omitted. The functions that operate on signatures have to use context to find the signer.

The remaining operations on signatures are straightforward and won't be shown here. They include `KheDrsSignatureMake` to make a new signature object, `KheDrsSignatureAddCost` to add a cost to a signature, and `KheDrsSignatureAddState` to add a state.

### D.6.2. Signature sets

A *signature set* is a set of signatures. It is logically the same as a signature: it has a cost and an array of states. It is basically an optimization: one can build a signature set by adding pointers to signatures more quickly than by appending arrays of state values. Its type declaration is

```
typedef struct khe_drs_signature_set_rec *KHE_DRS_SIGNATURE_SET;
typedef HA_ARRAY(KHE_DRS_SIGNATURE_SET) ARRAY_KHE_DRS_SIGNATURE_SET;

struct khe_drs_signature_set_rec {
  KHE_COST                          cost;
  ARRAY_KHE_DRS_SIGNATURE           signatures;
};
```

The `cost` field always holds the sum of the `cost` fields of the individual signatures.

To save space, signature sets are stored in expanded form: fields representing them have type `struct khe_drs_signature_set_rec` rather than type `KHE_DRS_SIGNATURE_SET`. This has no drawbacks and saves one pointer, which is significant given that every day solution contains a signature set, and there may be many thousands of those.

Function `KheDrsSignatureSetInit` initializes a signature set. This name is used, rather than `KheDrsSignatureSetMake`, to indicate that the memory for the signature set is already allocated and just needs to be initialized. There are operations for clearing a signature set, hashing it, testing two signature sets for equality, and adding a signature to a signature set:

```
void KheDrsSignatureSetAddSignature(KHE_DRS_SIGNATURE_SET sig_set,
  KHE_DRS_SIGNATURE sig, bool with_cost)
{
  if( with_cost )
    sig_set->cost += sig->cost;
  HaArrayAddLast(sig_set->signatures, sig);
  KheDrsSignatureRefer(sig);
}
```

This implements the rule given earlier, that the cost holds the sum of the component signatures' costs. This is omitted (i.e. `false` is passed for `with_cost`) only in one rather special case where the component costs are already included. Note the call to `KheDrsSignatureRefer`.

### D.6.3. Signers

A *signer*, short for *signature controller*, is an object which acts as a controller for the construction of signatures and testing them for dominance.

Different signatures may have different formats (different numbers of states, or different meanings for their states). Signatures constructed using a given signer share the format defined by that signer, allowing them to be compared for dominance.

The type of signers is

```
typedef struct khe_drs_signer_rec *KHE_DRS_SIGNER;
typedef HA_ARRAY(KHE_DRS_SIGNER) ARRAY_KHE_DRS_SIGNER;

typedef enum {
  KHE_DRS_SIGNER_DAY,
  KHE_DRS_SIGNER_RESOURCE_ON_DAY,
  KHE_DRS_SIGNER_SHIFT,
  KHE_DRS_SIGNER_SHIFT_PAIR
} KHE_DRS_SIGNER_TYPE;

struct khe_drs_signer_rec {
  KHE_DYNAMIC_RESOURCE_SOLVER   solver;
  ARRAY_KHE_DRS_EXPR            internal_exprs;
  ARRAY_KHE_DRS_DOM_TEST        dom_tests;
  HA_ARRAY_INT                  eq_dom_test_indexes;
  int                           last_hard_cost_index;
  KHE_DRS_CORRELATOR            correlator;
  KHE_DRS_SIGNER_TYPE           type;
  union {
    KHE_DRS_DAY                 day;
    KHE_DRS_RESOURCE_ON_DAY     resource_on_day;
    KHE_DRS_SHIFT               shift;
    KHE_DRS_SHIFT_PAIR          shift_pair;
  } u;
};
```

The `solver` field holds the solver containing this signer. The `internal_exprs` field holds the internal (non-leaf) expressions that must be evaluated when creating a signature using this signer. All expressions that contribute some state or cost must enrol themselves onto this list, in postorder (children before parents) as evaluation requires.

The `dom_tests` field holds the dominance tests, one for each position in the signatures that this signer controls.

The `eq_dom_test_indexes` field contains a set of indexes into the `dom_tests` array. These point to those dominance tests whose test is equality. This is used by 'medium' dominance testing, which is obsolete but still supported.

The `last_hard_cost_index` field is another index into the `dom_tests` array. It points to the last dominance test whose cost is hard (at least `KheCost(1, 0)`), or is `-1` when there are no such tests. This makes it possible to carry out the dominance testing of the hard cost elements of a set of signatures before the soft cost ones. This often saves a lot of time, because when a hard cost element produces a cost, dominance testing can end immediately (with failure).

The `correlator` field points to a *correlator*, an internal element of the signer which, when present, allows it to detect and handle correlated expressions (Appendix D.6.5).

The `type` and `u` fields define a tagged union which records whether the signer is used for day solutions, task solutions, shift solutions, or shift pair solutions. The main use for this is that some `COUNTER` expressions are evaluated differently depending on the type of signer.

There are operations for creating a signer, clearing it back to the initial state (no internal expressions or dominance tests), adding an internal expression, and adding a dominance test. The latter adds an entry to `eq_dom_test_indexes` when an equality dominance test is added.

Function `KheDrsSignerAddExpr` decides whether expression `e` needs to be added to signer `dsg`, and if so whether a position needs to be reserved for its state in signatures or not:

```
bool KheDrsSignerAddExpr(KHE_DRS_SIGNER dsg, KHE_DRS_EXPR e,
  KHE_DYNAMIC_RESOURCE_SOLVER drs, int *index)
{
  KHE_DRS_DOM_TEST dom_test;
  switch( KheDrsExprEvalType(e, dsg, drs, &dom_test) )
  {
    case KHE_DRS_EXPR_EVAL_NO:

      /* do nothing */
      return *index = -1, false;

    case KHE_DRS_EXPR_EVAL_NOT_LAST:

      /* add expression and dom test */
      KheDrsSignerAddOpenExpr(dsg, e);
      return *index = KheDrsSignerAddDomTest(dsg, dom_test, drs), true;

    case KHE_DRS_EXPR_EVAL_LAST:

      /* add expression only */
      KheDrsSignerAddOpenExpr(dsg, e);
      return *index = -1, false;

    default:

      HnAbort("KheDrsSignerAddExpr internal error");
      return *index = -1, false;  /* keep compiler happy */
  }
}
```

Actually `KheDrsExprEvalType` makes this three-way decision; this code does the actual addition of the expression, and possibly of a dominance test as well, based on the decision.

Once all the necessary expressions and dominance tests have been added, the signer is able, first, to build a signature by visiting and evaluating the internal expressions:

```
KHE_DRS_SIGNATURE KheDrsSignerEvalSignature(KHE_DRS_SIGNER dsg,
  KHE_DRS_SIGNATURE prev_sig, KHE_DYNAMIC_RESOURCE_SOLVER drs,
  bool debug)
{
  KHE_DRS_EXPR e;  int i;  KHE_DRS_SIGNATURE res;
  res = KheDrsSignatureMake(drs);
  HaArrayForEach(dsg->internal_exprs, e, i)
    KheDrsExprEvalSignature(e, dsg, prev_sig, res, drs, debug);
  return res;
}
```

and second, to compare two signatures for dominance:

```
bool KheDrsSignerDominates(KHE_DRS_SIGNER dsg,
  KHE_DRS_SIGNATURE sig1, KHE_DRS_SIGNATURE sig2,
  KHE_COST *avail_cost)
{
  *avail_cost += (sig2->cost - sig1->cost);
  return KheDrsSignerDoDominates(dsg, sig1, sig2, KHE_DRS_SIGNER_ALL,
    0, true, avail_cost);
}
```

This function assumes that `*avail_cost` has already been initialized, allowing it to become part of a larger dominance test which has these signatures as just one part. It calls function `KheDrsSignerDoDominates` to do the actual work. We'll start by examining its header:

```
bool KheDrsSignerDoDominates(KHE_DRS_SIGNER dsg,
  KHE_DRS_SIGNATURE sig1, KHE_DRS_SIGNATURE sig2,
  KHE_DRS_SIGNER_TEST test, int trie_start_depth, bool stop_on_neg,
  KHE_COST *avail_cost);
```

This tests `sig1` and `sig2` for dominance, assuming that `*avail_cost` is already initialized to the available cost and includes `sig2->cost - sig1->cost`. Parameter `test` has type

```
typedef enum {
  KHE_DRS_SIGNER_HARD,
  KHE_DRS_SIGNER_SOFT,
  KHE_DRS_SIGNER_ALL
} KHE_DRS_SIGNER_TEST;
```

and specifies that only states up to position `last_hard_cost_index` are to be tested, or only states from `last_hard_cost_index + 1` onwards, or all states. As it turns out, when testing signature sets for dominance there are significant time savings to be made by testing all the hard constraints before testing all the soft ones.

Parameter `trie_start_depth` is non-zero only when this function is called from within the trie data structure; some dominance testing will have already occurred and we want to continue on from position `trie_start_depth`. In this case, `test` will be `KHE_DRS_SIGNER_ALL`. Actually the trie data structure for solutions is been withdrawn, so `trie_start_depth` will always be 0.

Parameter `stop_on_neg`, when `true`, says that if `*avail_cost` ever goes negative, we are to stop immediately and declare that there is no dominance. As discussed in Appendix C, this is best for efficiency, although there are rare cases where it declares that there is no dominance when in fact carrying on would show that there is dominance. It does not break anything to occasionally make this mistake; it just means that a few more solutions are kept than is absolutely necessary, because a slightly weaker dominance test is being applied than what could be.

Here now is `KheDrsSignerDoDominates`. It is more than one page long, so we'll start by presenting it in outline, then fill in each piece separately:

```
bool KheDrsSignerDoDominates(KHE_DRS_SIGNER dsg,
  KHE_DRS_SIGNATURE sig1, KHE_DRS_SIGNATURE sig2,
  KHE_DRS_SIGNER_TEST test, int trie_start_depth, bool stop_on_neg,
  KHE_COST *avail_cost, int verbosity, int indent, FILE *fp)
{
  KHE_DRS_DOM_TEST dt;  int start_index, stop_index, dt_index, sig_len;
  KHE_DRS_VALUE v1, v2;

  /* consistency checks */
  ... see first code excerpt below ...

  /* work out start_index and stop_index */
  ... see second code excerpt below ...

  /* quit immediately if no available cost */
  if( stop_on_neg && *avail_cost < 0 )
    return false;

  /* do the test from start_index inclusive to stop_index exclusive */
  for( dt_index = start_index;  dt_index < stop_index;  dt_index++ )
  {
    dt = HaArray(dsg->dom_tests, dt_index);
    v1 = HaArray(sig1->states, dt_index);
    v2 = HaArray(sig2->states, dt_index);
    switch( dt->type )
    {
      ... see third code excerpt below ...
    }

    /* quit early if *avail_cost is now negative */
    if( stop_on_neg && *avail_cost < 0 )
      return false;
  }
  return *avail_cost >= 0;
}
```

The two signatures are supposed to have been created by signer `dsg`. Because we have chosen not to store a signer within each signature, we cannot check this. But we can check that the number of dominance tests in the signer equals the number of states in each signature:

```
/* consistency checks */
sig_len = HaArrayCount(dsg->dom_tests);
HnAssert(HaArrayCount(sig1->states) == sig_len,
  "KheDrsSignerDoDominates internal error 1 (count %d != count %d)\n",
  HaArrayCount(sig1->states), sig_len);
HnAssert(HaArrayCount(sig2->states) == sig_len,
  "KheDrsSignerDoDominates internal error 2 (count %d != count %d)\n",
  HaArrayCount(sig2->states), sig_len);
```

Next we use `test` to work out `start_index`, the place along the states arrays to start testing, and `stop_index`, the index just past the stopping point:

```
/* work out start_index and stop_index */
switch( test )
{
  case KHE_DRS_SIGNER_HARD:

    HnAssert(trie_start_depth == 0,
      "KheDrsSignerDoDominates internal error 1");
    start_index = 0;
    stop_index = dsg->last_hard_cost_index + 1;
    break;

  case KHE_DRS_SIGNER_SOFT:

    HnAssert(trie_start_depth == 0,
      "KheDrsSignerDoDominates internal error 2");
    start_index = dsg->last_hard_cost_index + 1;
    stop_index = sig_len;
    break;

  case KHE_DRS_SIGNER_ALL:

    start_index = trie_start_depth;
    stop_index = sig_len;
    break;

  default:

    HnAbort("KheDrsSignerDoDominates internal error 3");
    start_index = 0, stop_index = 0;  /* keep compiler happy */
}
```

Now returning to the original function, we see that it iterates along the states arrays from `start_index` inclusive to `stop_index` exclusive, extracting the dominance test `dt` from the signer (this says how to test for dominance at this position), and state values `v1` and `v2` from the states arrays of the signature.

The switch on `dt->type` has 13 branches. Most of them are concerned with dominance testing involving correlated expressions, which we will omit for now. Here are the others:

```
switch( dt->type )
{
  case KHE_DRS_DOM_TEST_UNUSED:

    /* unused test, should never happen */
    HnAbort("internal error in KheDrsSignerDominates (UNUSED)");
    break;

  case KHE_DRS_DOM_TEST_STRONG:

    /* strong dominance */
    if( !KheDrsDomTestDominatesStrong(dt, v1, v2) )
      *avail_cost -= KheCost(1, 0);
    break;

  case KHE_DRS_DOM_TEST_SEPARATE_INT:

    /* separate dominance (int) */
    if( !KheDrsDomTestDominatesSeparateInt(dt, v1.i, v2.i) )
      *avail_cost -= KheCost(1, 0);
    break;

  case KHE_DRS_DOM_TEST_SEPARATE_FLOAT:

    /* separate dominance (float) */
    if( !KheDrsDomTestDominatesSeparateFloat(dt, v1.f, v2.f) )
      *avail_cost -= KheCost(1, 0);
    break;

  case KHE_DRS_DOM_TEST_TRADEOFF:

    /* tradeoff dominance */
    if( !KheDrsDomTestDominatesTradeoff(dt, v1.i, v2.i, avail_cost) )
      *avail_cost -= KheCost(1, 0);
    break;

  case KHE_DRS_DOM_TEST_TABULATED:

    /* tabulated dominance */
    KheDrsDomTestDominatesTabulated(dt, v1.i, v2.i, avail_cost);
    break;

  ... cases for correlated expressions omitted ...
}
```

The dominance test determines whether the values are interpreted as integers or floats, among other things.

### D.6.4. Signer sets

Just as a signature set is a set of signatures, so a signer set is a set of signers:

```
typedef struct khe_drs_signer_set_rec *KHE_DRS_SIGNER_SET;
typedef HA_ARRAY(KHE_DRS_SIGNER_SET) ARRAY_KHE_DRS_SIGNER_SET;

struct khe_drs_signer_set_rec {
  ARRAY_KHE_DRS_SIGNER          signers;
};
```

There are operations for creating and freeing a signer set, adding a signer to a signer set, and clearing a signer set back to empty. There is also this operation, called when a day is opened:

```
void KheDrsSignerSetDayOpen(KHE_DRS_SIGNER_SET signer_set,
  KHE_DRS_DAY day, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_RESOURCE dr;  int i;  KHE_DRS_RESOURCE_ON_DAY drd;
  KHE_DRS_SIGNER dsg;

  /* one signer for each open resource */
  HnAssert(HaArrayCount(signer_set->signers) == 0,
    "KheDrsSignerSetDayOpen internal error");
  KheDrsResourceSetForEach(drs->open_resources, dr, i)
  {
    drd = KheDrsResourceOnDay(dr, day);
    KheDrsSignerSetAddSigner(signer_set, drd->signer);
  }

  /* one additional signer for event resource expressions */
  dsg = KheDrsSignerMake(day, NULL, NULL, NULL, drs);
  KheDrsSignerSetAddSigner(signer_set, dsg);
}
```

This fills a signer set with one pre-existing signer for each open resource, and one newly created signer to hold the event resource constraints affected by the tasks of this day. When the day is closed, this function is called:

```
void KheDrsSignerSetDayClose(KHE_DRS_SIGNER_SET signer_set,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_SIGNER dsg;

  /* delete and free the last signer (the others were not made here) */
  HnAssert(HaArrayCount(signer_set->signers) > 0,
    "KheDrsSignerSetDayClose internal error");
  dsg = HaArrayLast(signer_set->signers);
  KheDrsSignerFree(dsg, drs);

  /* clear the signers */
  HaArrayClear(signer_set->signers);
}
```

This frees the last signer and clears the signer set.

After that come operations for hashing signatures and comparing them for equality, which we won't show because they are used only by superseded dominance tests. Finally we get to `KheDrsSignerSetDominates`, which performs dominance testing between two signature sets. It's a long one so we'll start with the header:

```
bool KheDrsSignerSetDominates(KHE_DRS_SIGNER_SET signer_set,
  KHE_DRS_SIGNATURE_SET sig_set1, KHE_DRS_SIGNATURE_SET sig_set2,
  KHE_COST trie_extra_cost, int trie_start_depth, bool use_caching,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
```

This returns `true` when `sig_set1` dominates `sig_set2`, using `signer_set` to determine what to do at each position. Parameters `trie_extra_cost` and `trie_start_depth` have non-zero values only when `KheDrsSignerSetDominates` is called from within the trie data structure, where some dominance testing (the first `trie_start_depth` positions) will have already been done. Parameter `use_caching` is `true` when we are using caching of the results of this function to speed up later calls to it. We'll see how that works shortly. Here is the function, in outline:

```
bool KheDrsSignerSetDominates(KHE_DRS_SIGNER_SET signer_set,
  KHE_DRS_SIGNATURE_SET sig_set1, KHE_DRS_SIGNATURE_SET sig_set2,
  KHE_COST trie_extra_cost, int trie_start_depth, bool use_caching,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  int i, count, last_cache;  KHE_DRS_SIGNER dsg;  KHE_COST avail_cost;
  KHE_DRS_SIGNATURE sig1, sig2;  KHE_DRS_RESOURCE dr;
  KHE_DRS_COST_TUPLE ct;
  count = HaArrayCount(signer_set->signers);
  HnAssert(HaArrayCount(sig_set1->signatures) == count,
    "KheDrsSignerSetDominates internal error 1");
  HnAssert(HaArrayCount(sig_set2->signatures) == count,
    "KheDrsSignerSetDominates internal error 2");
  avail_cost = sig_set2->cost - sig_set1->cost - trie_extra_cost;
  if( avail_cost < 0 )
    return false;
  if( drs->solve_dom_approx > 0 )
    avail_cost += (avail_cost * drs->solve_dom_approx) / 10;
  if( trie_start_depth > 0 )
  {
    /* won't happen anyway but do it the easy way if it does */
    ... see first code excerpt below ...
  }
  else if( USE_DOM_CACHING && use_caching )
  {
    /* use a cached value for all positions except the last */
    ... see second code excerpt below ...
  }
  else
  {
    /* visit hard constraints first; they often end the test quickly */
    ... see third code excerpt below ...
  }
  return true;
}
```

After checking that the number of signers in the signer set equals the number of signatures in each signature set, the function initializes `avail_cost` and returns `false` immediately if it is negative. This code:

```
if( drs->solve_dom_approx > 0 )
  avail_cost += (avail_cost * drs->solve_dom_approx) / 10;
```

implements the `dom_approx` feature, which arbitrarily enlarges `avail_cost`, increasing the chance of a successful test but giving up provable optimality.

If `trie_start_depth > 0`, we have to start `trie_start_depth` places along the signature, which is done like this:

```
/* won't happen anyway but do it the easy way if it does */
HaArrayForEach(signer_set->signers, dsg, i)
{
  if( trie_start_depth < HaArrayCount(dsg->dom_tests) )
  {
    sig1 = HaArray(sig_set1->signatures, i);
    sig2 = HaArray(sig_set2->signatures, i);
    if( !KheDrsSignerDoDominates(dsg, sig1, sig2, KHE_DRS_SIGNER_ALL,
          trie_start_depth, true, &avail_cost) )
      return false;
    trie_start_depth = 0;
  }
  else
    trie_start_depth -= HaArrayCount(dsg->dom_tests);
}
```

We saw `KheDrsSignerDoDominates` earlier. If we are using cached dominance test results, we execute this code:

```
/* use a cached value for all positions except the last */
HnAbort("KheDrsSignerSetDominates - dom caching unavailable");
last_cache = HaArrayCount(signer_set->signers) - 2;
for( i = 0;  i <= last_cache;  i++ )
{
  dr = KheDrsResourceSetResource(drs->open_resources, i);
  dsg = HaArray(signer_set->signers, i);
  sig1 = HaArray(sig_set1->signatures, i);
  HnAssert(sig1->asst_to_shift_index >= 0,
    "KheDrsSignerSetDominates internal error 3");
  sig2 = HaArray(sig_set2->signatures, i);
  HnAssert(sig2->asst_to_shift_index >= 0,
    "KheDrsSignerSetDominates internal error 4");
  ct = KheDrsDim2TableGet2(dr->expand_dom_test_cache,
    sig1->asst_to_shift_index, sig2->asst_to_shift_index);
  avail_cost += ct.unweighted_psi;
  if( avail_cost < 0 )
    return false;
}

/* regular test for the last position */
dsg = HaArrayLast(signer_set->signers);
sig1 = HaArrayLast(sig_set1->signatures);
sig2 = HaArrayLast(sig_set2->signatures);
if( !KheDrsSignerDoDominates(dsg, sig1, sig2, KHE_DRS_SIGNER_ALL,
      0, true, &avail_cost) )
  return false;
```

Caching is available only for resource signatures, and is only used within day solutions. When it is in use, the `asst_to_shift_index` fields of the two signatures must be set and are used to index a cache of results of calls to this function, stored in `dr->expand_dom_test_cache`. Looking up a table should be much faster than redoing the dominance test over and over, although the author's tests do not show any benefit.

Finally we come to the usual case, where we just run along each signature's state array in the usual way. But even here there is a wrinkle:

```
/* visit hard constraints first; they often end the test quickly */
HaArrayForEach(signer_set->signers, dsg, i)
{
  sig1 = HaArray(sig_set1->signatures, i);
  sig2 = HaArray(sig_set2->signatures, i);
  if( !KheDrsSignerDoDominates(dsg, sig1, sig2, KHE_DRS_SIGNER_HARD,
        0, true, &avail_cost) )
    return false;
}

/* visit soft constraints */
HaArrayForEach(signer_set->signers, dsg, i)
{
  sig1 = HaArray(sig_set1->signatures, i);
  sig2 = HaArray(sig_set2->signatures, i);
  if( !KheDrsSignerDoDominates(dsg, sig1, sig2, KHE_DRS_SIGNER_SOFT,
      0, true, &avail_cost) )
    return false;
}
```

The hard constraint states are visited first, because if they fail they end dominance testing immediately. This saves a lot of time, as the author's tests confirm.

### D.6.5. Correlators

*still to do*

### D.6.6. Types of signers and their signatures

At the risk of getting ahead of ourselves, we now give a detailed description of the four types of signers and their signatures. For each type of signer we will give: the type of solution it handles; which signers of this type there are and where they are kept; which internal expressions are added to each signer; which dominance tests are added to each signer; and which signatures are created by each signer of this type, how their costs and states are determined, and where they are kept.

But first, a few general points. When an expression *e* adds itself to a signer, it is requesting that the signer call it back as part of creating the signatures controlled by that signer. When it adds a dominance test to a signer, it is saying that there will be a position in the state array of the signatures controlled by that signer which holds the state of *e*, and that the supplied dominance test is to be used at that position during dominance testing. In that case, *e* is obliged to ensure that a state value is added to the signature during evaluation.

An expression which adds a dominance test to a signer must also add itself to the signer. However, it can add itself to the signer without adding a dominance test. This occurs when the expression needs to be evaluated but that evaluation will be the last one, the one that finalizes the expression's value, so that the value is stored somewhere other than in a signature's state array.

The code that adds internal expressions and dominance tests to signers is part of function `KheDrsExprOpen` (Appendix D.8.5). The code that adds costs and states to signatures is part of function `KheDrsExprEvalSignature` (Appendix D.8.7). Hopefully this unified and detailed presentation will make those functions easier to follow.

As we'll see, resource constraints are handled differently from event resource constraints. This is an optimization which exploits the fact that each resource constraint is affected by just one of the assignments on any one day (the one containing the resource that the constraint monitors). Event resource constraints may be affected by several of the assignments on any one day.

***Resource on day signers.*** When `type` is `KHE_DRS_SIGNER_RESOURCE_ON_DAY`, the signer is a *resource on day* signer. It handles signatures for task solutions. These consist of one $d_k$-complete solution plus one assignment of one resource $r$ on day $d_{k+1}$ to some task (or free day). Field `u.resource_on_day` holds the resource on day object representing $r$ on day $d_{k+1}$.

There is one of these signers for each open resource on day, held in the resource on day object `drd` and created when `drd` is opened (actually, it is created when `drd` is created and cleared out when `drd` is closed, which comes to the same thing). This same signer is used for all $d_k$-complete solutions and all tasks, which is fine: changing these things will lead to different signatures, but it does not change the signer (it has no effect on the signature format).

The expressions that enrol themselves with a resource on day signer are those derived from resource constraints for $r$ that are affected by what $r$ is doing on day $d_{k+1}$. When this is not the last day they are affected by, they also contribute dominance tests, reserving for themselves a place in the states array. Event resource constraints take no part in resource on day signatures.

For these signers, signatures exist only while expanding a given $d_k$-complete solution. Each signature is held in a `KHE_DRS_MTASK_SOLN` object representing the assignment of resource $r$ to an arbitrary task of an mtask $c$ whose first day is $d_{k+1}$. There is one of these mtask solution objects for each $(r, c)$ pair such that $r$ can be assigned to $c$. However, the implementation knows that all tasks from the same shift produce the same signature, and so two mtask solutions share a signature when they are from the same shift. All these mtask solution objects are held in the `KHE_DRS_RESOURCE` object representing $r$.

The cost of a resource on day signature is the sum, over all monitors enrolled in the signer, of their extra cost on day $d_{k+1}$ over their cost on day $d_k$. Extra costs are more convenient than full costs when these signatures go into signature sets, because they can be added to the cost of the $d_k$-solution with no risk of double counting.

Resource on day signers and signatures are essentially an optimization. Many $d_{k+1}$-complete solutions derived from a given $d_k$-complete solution contain an assignment of $r$ to a task of shift $s$. The effect of this assignment on $r$'s resource constraints is calculated only once, held in a resource on day signature, and added to the signature sets of $d_{k+1}$-solutions as required.

***Day signers.*** When `type` is `KHE_DRS_SIGNER_DAY`, the signer is a *day signer*. It handles signatures for the event resource constraints of $d_k$-solutions. Field `u.day` holds the day $d_k$ up to which these solutions are complete. There is one day signer for each open day, held in the `KHE_DRS_DAY` object representing that day and created when the day is opened.

A day signer's internal expressions are the internal expressions derived from event resource constraints which are affected by what happens on its day. As usual, there is one dominance test for each of these internal expressions for which this is not the last day.

This signer must not be confused with the signer set for a given day. The signer set contains one resource on day signer for each open resource $r$, each recording the cost and state of the resource constraints of $r$ up to day $d_k$ inclusive, plus the day signer for the event resource constraints as just explained. So the signer set covers the entire solution and defines signature sets whose cost is the full solution cost of one $d_k$-solution.

There is one solution object that none of the above applies to: the root solution. It is not on any day, so its signature can't be related to any day signer. But its signature contains just the initial solution cost (after the selected tasks are unassigned), has no states (initial states are stored in expressions, not in signatures), and never participates in dominance testing. So the absence of a signer does not matter in this case.

***Shift signers.*** When `type` is `KHE_DRS_SIGNER_SHIFT`, the signer is a *shift signer*. It handles shift solutions, made of one $d_k$-solution plus one assignment of each element of a set of resources $R$ to an unspecified task of a shift $s$ beginning on day $d_{k+1}$. These are assumed to be the only assignments to the tasks of $s$. Field `u.shift` holds $s$.

There is one shift signer for each open shift $s$, held in $s$, and one signature for each `KHE_DRS_SHIFT_SOLN` object associated with $s$. The internal expressions are those derived from event resource constraints which are affected by assignments to any of the tasks of $s$. As usual, there is one dominance test for each of these internal expressions $e$ for which the assignments to the $d_k$-solution, plus the assignments to $s$ (crucially, combined with the knowledge that there are no other assignments to $s$) leave some children of $e$ with undetermined values.

The same signer can be and is used for all sets $R$, although dominance tests are made only between `KHE_DRS_SHIFT_SOLN` objects with the same $R$. For objects with the same $R$, the effect on resource constraints is the same, because each resource of $R$ is assigned to some task of $s$, and those tasks have the same effect on resource constraints (by how shifts are defined). This is why resource constraints are omitted completely from these signatures: for dominance testing, which is all we are interested in here, they make no difference. The cost of each signature is the extra cost of its expressions on day $d_{k+1}$, beyond their cost on day $d_k$.

***Shift pair signers.*** When `type` is `KHE_DRS_SIGNER_SHIFT_PAIR`, the signer is a *shift pair signer*. It handles shift pair solutions, made of one $d_k$-solution, plus one assignment of a set of resources $R_1$ to tasks of a shift $s_1$ beginning on day $d_{k+1}$, plus one assignment of a set of resources $R_2$ to a tasks of a shift $s_2$ beginning on day $d_{k+1}$. We require $s_1 \neq s_2$ and $R_1 \cap R_2 = \emptyset$.

There is one shift pair signer for each pair of shifts that begin on the same day, held in the shift pair object for those two shifts. The expressions are all those derived from event resource monitors whose values are affected by assignments to the tasks of $s_1$ or $s_2$. As usual, if those assignments do not finish off the expression value, a dominance test is added as well.

A very similar result could be obtained by a signer set consisting of the two shift signers for $s_1$ and $s_2$. But this fails to work in the unlikely case where there is a monitor whose value is affected by assignments in both shifts. Such a monitor would have its cost counted twice.

### D.6.7. Dominance kinds and dominance test types

Type KHE_DRS_DOM_KIND is an enumerated type, defined publicly in khe_solvers.h, and used to specify the kind of dominance testing to employ:

```
typedef enum {
  KHE_DRS_DOM_LIST_NONE,
  KHE_DRS_DOM_LIST_SEPARATE,
  KHE_DRS_DOM_LIST_TRADEOFF,
  KHE_DRS_DOM_LIST_TABULATED,
  KHE_DRS_DOM_HASH_EQUALITY,
  KHE_DRS_DOM_HASH_MEDIUM,
  /* KHE_DRS_DOM_TRIE_SEPARATE, */
  /* KHE_DRS_DOM_TRIE_TRADEOFF, */
  KHE_DRS_DOM_INDEXED_TRADEOFF,
  KHE_DRS_DOM_INDEXED_TABULATED
} KHE_DRS_DOM_KIND;
```

Two ideas are mixed here: how to organize a set of solutions (simple list, hash table, the recently withdrawn trie, or indexed array), and how to test a pair of solutions for dominance (none, separate, tradeoff, tabulated, equality, or medium). They are mixed because different organizations support different tests.

These choices record the author's attempts to speed up dominance testing. Eventually it will become clear that one is better than the others, and the others can be forgotten, even deleted. At the time of writing the author inclines towards indexed tabulated testing; time will tell.

For the dominance test type aspect of KHE_DRS_DOM_KIND, the solver defines type

```
typedef enum {
  KHE_DRS_DOM_TEST_UNUSED,
  KHE_DRS_DOM_TEST_SEPARATE_GENERIC,
  KHE_DRS_DOM_TEST_SEPARATE_INT,
  KHE_DRS_DOM_TEST_SEPARATE_FLOAT,
  KHE_DRS_DOM_TEST_TRADEOFF,
  KHE_DRS_DOM_TEST_TABULATED,
  KHE_DRS_DOM_TEST_CORR1_PARENT,
  KHE_DRS_DOM_TEST_CORR1_CHILD,
  KHE_DRS_DOM_TEST_CORR2_CHILD,
  KHE_DRS_DOM_TEST_CORR3_FIRST,
  KHE_DRS_DOM_TEST_CORR3_MID,
  KHE_DRS_DOM_TEST_CORR3_LAST,
  KHE_DRS_DOM_TEST_CORR4_FIRST,
  KHE_DRS_DOM_TEST_CORR4_MID,
  KHE_DRS_DOM_TEST_CORR4_LAST
} KHE_DRS_DOM_TEST_TYPE;
```

The CORR1 through CORR4 values are for dominance testing of correlated expressions, and are not of concern to us here. The others request separate dominance, tradeoff dominance, or tabulated dominance. These values are extracted from a `KHE_DRS_DOM_KIND` value by function

```
KHE_DRS_DOM_TEST_TYPE KheDomKindToDomTestType(KHE_DRS_DOM_KIND dom_kind)
{
  switch( dom_kind )
  {
    case KHE_DRS_DOM_LIST_NONE:         return KHE_DRS_DOM_TEST_UNUSED;
    case KHE_DRS_DOM_LIST_SEPARATE: return KHE_DRS_DOM_TEST_SEPARATE_GENERIC;
    case KHE_DRS_DOM_LIST_TRADEOFF:     return KHE_DRS_DOM_TEST_TRADEOFF;
    case KHE_DRS_DOM_LIST_TABULATED:    return KHE_DRS_DOM_TEST_TABULATED;
    case KHE_DRS_DOM_HASH_EQUALITY:     return KHE_DRS_DOM_TEST_UNUSED;
    case KHE_DRS_DOM_HASH_MEDIUM:       return KHE_DRS_DOM_TEST_SEPARATE_INT;
    case KHE_DRS_DOM_INDEXED_TRADEOFF:  return KHE_DRS_DOM_TEST_TRADEOFF;
    case KHE_DRS_DOM_INDEXED_TABULATED: return KHE_DRS_DOM_TEST_TABULATED;

    default:

      HnAbort("KheDomKindToDomTestType: unknown dom_kind (%d)", dom_kind);
      return 0;  /* keep compiler happy */
  }
}
```

which carries out the obvious mapping. At this point, it is not clear whether any particular separate dominance test will be `int`-valued or `float`-valued, so this function reports that separate testing is wanted without specifying which; that will be filled in later.

As the description of `KheDynamicResourceSolverSolve` explains, when solutions are held in a cache as well as in a main table, a consistency issue arises: the main table's dominance kind and the cache's dominance kind do not have to agree, but their dominance test types do. This is implemented by the following little function:

```
KHE_DRS_DOM_TEST_TYPE KheDomKindCheckConsistency(
  KHE_DRS_DOM_KIND main_dom_kind, bool cache,
  KHE_DRS_DOM_KIND cache_dom_kind)
{
  KHE_DRS_DOM_TEST_TYPE main_dom_test_type, cache_dom_test_type;
  main_dom_test_type = KheDomKindToDomTestType(main_dom_kind);
  if( cache )
  {
    cache_dom_test_type = KheDomKindToDomTestType(cache_dom_kind);
    HnAssert(main_dom_test_type == KHE_DRS_DOM_TEST_UNUSED ||
      cache_dom_test_type == KHE_DRS_DOM_TEST_UNUSED ||
      main_dom_test_type == cache_dom_test_type,
      "KheDomKindCheckConsistency: inconsistent main_dom_kind "
      "and cache_dom_kind arguments");
  }
  return main_dom_test_type;
}
```

The only other function on type KHE_DRS_DOM_KIND is KheDomKindShow, used to display a value of type KHE_DRS_DOM_KIND during debugging. Values of this type are mainly used to control switch statements which implement the different kinds of dominance testing.

### D.6.8. Dominance tables

A *dominance table* is a table used to cache available costs during uniform dominance testing, as explained in Appendix C. The type of one entry in such a table is a triple of three costs:

```
typedef struct khe_drs_cost_tuple_rec {
  short                         unweighted_psi;
  short                         unweighted_psi0;
  short                         unweighted_psi_plus;
} KHE_DRS_COST_TUPLE;

typedef HA_ARRAY(KHE_DRS_COST_TUPLE) ARRAY_KHE_DRS_COST_TUPLE;
```

To save memory it stores 16-bit *unweighted costs* rather than 64-bit costs. These unweighted costs need to be multiplied by a constraint weight, stored elsewhere, to produce costs.

Next comes type

```
typedef struct khe_drs_dim1_table_rec {
  ARRAY_KHE_DRS_COST_TUPLE      children;
  int                           offset;
} *KHE_DRS_DIM1_TABLE;

typedef HA_ARRAY(KHE_DRS_DIM1_TABLE) ARRAY_KHE_DRS_DIM1_TABLE;
```

representing a one-dimensional table of arbitrary length whose elements have type KHE_DRS_COST_TUPLE. As far as the caller is concerned, the first element has index offset. This offset field is maintained automatically as elements are added, as we'll see. Next comes

```
struct khe_drs_dim2_table_rec {
  ARRAY_KHE_DRS_DIM1_TABLE      children;
  int                           offset;
};

typedef HA_ARRAY(KHE_DRS_DIM2_TABLE) ARRAY_KHE_DRS_DIM2_TABLE;
```

which is a one-dimensional table of arbitrary length whose children are one-dimensional arrays of cost tuples. This same pattern is continued up to type

```
typedef struct khe_drs_dim5_table_rec {
  ARRAY_KHE_DRS_DIM4_TABLE      children;
  int                           offset;
} *KHE_DRS_DIM5_TABLE;

typedef HA_ARRAY(KHE_DRS_DIM5_TABLE) ARRAY_KHE_DRS_DIM5_TABLE;
```

which offers five-dimensional tables whose individual elements are cost tuples, and which are extensible with an adjustable starting index (stored in `offset`) in each sub-array.

There is a lot of repetitive code in these types, but the author wanted the strong typing. Here is a typical function, which adds a new cost tuple `ct` to five-dimensional array `d5`:

```
void KheDrsDim5TablePut(KHE_DRS_DIM5_TABLE d5, int index5, int index4,
  int index3, int index2, int index1, KHE_DRS_COST_TUPLE ct,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_DIM4_TABLE d4;  int pos;  HA_ARENA a;

  /* set offset if this is the first insertion */
  if( HaArrayCount(d5->children) == 0 )
    d5->offset = index5;

  /* make sure index5 is within, or just beyond, the current range */
  pos = index5 - d5->offset;
  HnAssert(0 <= pos && pos <= HaArrayCount(d5->children),
    "KheDrsDim5TablePut: index5 %d is out of range %d .. %d", index5,
    d5->offset, d5->offset + HaArrayCount(d5->children));

  /* find or make-and-add d4, the sub-array to insert into */
  if( pos < HaArrayCount(d5->children) )
    d4 = HaArray(d5->children, pos);
  else
  {
    a = HaArrayArena(d5->children);
    d4 = KheDrsDim4TableMake(a);
    HaArrayAddLast(d5->children, d4);
  }

  /* do the insertion */
  KheDrsDim4TablePut(d4, index4, index3, index2, index1, ct, drs);
}
```

If this is the first insertion into this array, its `offset` field is set to `index5`, the new entry's position in this array, as far as the caller is concerned. This assumes that the smallest index is passed first. That assumption could be avoided but there is no need for that in this application.

The next step is to find `pos`, the internal index corresponding to the external index `index5`; this is just `index5 - d5->offset`. There must already be a four-dimensional array at position `pos`, in which case `d4` is set to that array, or else `pos` must be just off the end, in which case `d4` is set to a new four-dimensional array and added to the end. Again, this 'just off the end' assumption could be avoided, but there is no need here.

Finally, `ct` is inserted into `d4` by a call to `KheDrsDim4TablePut`, the four-dimensional version of `KheDrsDim5TablePut`.

There is a `KheDrsDim5TableGet` operation which retrieves one four-dimensional table from a five-dimensional table:

```
KHE_DRS_DIM4_TABLE KheDrsDim5TableGet(KHE_DRS_DIM5_TABLE d5, int index5)
{
  int pos;
  pos = index5 - d5->offset;
  HnAssert(0 <= pos && pos < HaArrayCount(d5->children),
    "KheDrsDim5TableGet: index %d out of range %d .. %d", index5,
    d5->offset, HaArrayCount(d5->children) + d5->offset - 1);
  return HaArray(d5->children, pos);
}
```

Applying this idea three times produces this function, which uses three indexes to retrieve a two-dimensional table from a five-dimensional table:

```
KHE_DRS_DIM2_TABLE KheDrsDim5TableGet3(KHE_DRS_DIM5_TABLE d5,
  int index5, int index4, int index3)
{
  return KheDrsDim3TableGet(KheDrsDim4TableGet(
    KheDrsDim5TableGet(d5, index5), index4), index3);
}
```

This will be used when constructing tabulated dominance tests: we extract the appropriate two-dimensional table for a given test from a five-dimensional table and store it in the test.

## D.6.9. Dominance tests

A value of type `KHE_DRS_DOM_TEST` represents the dominance test at one position along some signature. It knows which constraint owns that position, and when dominance testing reaches that position it supplies the information needed to carry out the correct test for that constraint.

This tyoe should really be a union, storing just the values needed by each type of dominance test. However, at present alll fields needed by all types are lumped in together:

```
typedef struct khe_drs_dom_test_rec *KHE_DRS_DOM_TEST;
typedef HA_ARRAY(KHE_DRS_DOM_TEST) ARRAY_KHE_DRS_DOM_TEST;

struct khe_drs_dom_test_rec {
  KHE_DRS_DOM_TEST_TYPE type;
  int                   correlated_delta;
  KHE_DRS_EXPR          expr;
  bool                  allow_zero;
  int                   min_limit;
  int                   max_limit;
  int                   a;
  int                   b;
  KHE_COST              combined_weight;
  KHE_DRS_DIM2_TABLE    main_dom_table2;
  KHE_DRS_DIM4_TABLE    corr_dom_table4;
  KHE_MONITOR           monitor;
};
```

Most of these fields have self-explanatory names. When `type` is `KHE_DRS_DOM_TEST_TABULATED`, `main_dom_table2` contains the two-dimensional table consulted by tabulated dominance. When `type` denotes a correlated expression, `correlated_delta` may hold a value to add to the index of this dominance test to obtain the index of a correlated one, and `corr_dom_table4` may hold a four-dimensional table which is consulted to obtain a correlated available cost.

The operations on dominance tests include `KheDrsDomTestMake` for creating one, as well as `KheDrsDomTestDominatesSeparateInt`, `KheDrsDomTestDominatesSeparateFloat`, `KheDrsDomTestDominatesTradeoff`, and `KheDrsDomTestDominatesTabulated` for carrying out one test. Here is the last of these:

```
void KheDrsDomTestDominatesTabulated(KHE_DRS_DOM_TEST dt,
  int val1, int val2, KHE_COST *avail_cost)
{
  KHE_DRS_COST_TUPLE ct;
  ct = KheDrsDim2TableGet2(dt->main_dom_table2, val1, val2);
  *avail_cost += ct.unweighted_psi * dt->combined_weight;
}
```

It uses the two values (taken from two signatures) to index into `dt->main_dom_table2`, and uses the `unweighted_psi` value from the resulting cost tuple, multiplied by a weight, as the change in available cost.

### D.6.10. Dominance test caching

Function `KheDrsSignerDoDominates` is a good candidate for caching. Assuming that tries are not in use and that we want to do a dominance test of the whole signature (not the hard parts and soft parts separately), there are essentially just two parameters—the two signatures to be compared—and the result is just the cost, usually zero or negative, to add to the available cost.

So the cache can be very simple: a two-dimensional table whose elements are costs. Retrieving from it should be a lot faster than slogging through the two signatures.

All this applies equally well to signature sets. However, caching them would not be useful, because a given ordered pair of signature sets is almost never tested for dominance twice. In the same way, we should not cache all pairs of signatures that ever get tested for dominance (although logically we could), because many of those would never recur. We need to choose pairs of signatures that are easy to cache and likely to be tested for dominance repeatedly.

At the start of expanding each $d_k$-complete solution $S$, for each open resource $r$, and for each shift $s$ beginning on day $d_{k+1}$ that $r$ is qualified for (including the special shift denoting a free day), the solver builds one signature holding the state of $r$'s resource monitors when $r$ has its assignments from $S$, and is also assigned to $s$. These signatures are stored in an array within the `KHE_DRS_RESOURCE` object representing $r$.

Dominance testing caches are built immediately after these signatures are created. Each cache is a two-dimensional array containing one entry for each ordered pair of signatures for the same $S$ and $r$, whose value is the cost to add to the available cost when these two signatures are compared for dominance. The cache is stored in $r$ alongside $r$'s signatures.

At position $(i,j)$ of the cache, what is stored is the cost associated with comparing the signature of $r$'s $i$th signature with its $j$th signature. The cache is constructed by calling `KheDrsSignerDoDominates` once for each of these pairs of signatures and storing the result. Each resource signature contains its own index, so given two signatures it is easy to use these indexes to access the cache and avoid calling `KheDrsSignerDoDominates`.

The cache must only be consulted when a value for the two signatures being compared is in it. To ensure this, whenever two solutions are tested for dominance, we first check whether both have the same previous solution $S$. If they do, then the test must be part of the expansion of $S$ (because all dominance tests between pairs of solutions are part of some expansion, and at least one of the solutions in each test must have the solution being expanded as its previous solution), and at each position along the two solutions' signature sets except the last, the signatures at that position form a pair that must be in the cache (because the cache is present for the entire expansion of $S$). So the whole signature set test for dominance can and does use cached values at each position except the last, where `KheDrsSignerDoDominates` is called. The last position usually has empty signatures anyway.

If two solutions have different previous solutions, the test is part of the expansion of one of those but not the other. Caching such cases would be much more expensive in time and memory than what we are doing. Our method might do a lot of good and costs very little.

## D.7. Constraints

This section documents the solver's types that parallel KHE's constraint and monitor types. These are only a minor part of the system, needed for technical reasons that will be explained.

### D.7.1. Constraints

The solver has a constraint type defined by

```
typedef struct khe_drs_constraint_rec {
  KHE_CONSTRAINT                constraint;
  int                           min_history;
  int                           max_history;
  int                           max_child_count;
  bool                          needs_corr_table;
  KHE_DRS_EXPR                  sample_expr;
  KHE_DRS_DIM3_TABLE            counter_main_dom_table3;
  KHE_DRS_DIM5_TABLE            counter_corr_dom_table5;
  KHE_DRS_DIM5_TABLE            sequence_main_dom_table5;
} *KHE_DRS_CONSTRAINT;

typedef HA_ARRAY(KHE_DRS_CONSTRAINT) ARRAY_KHE_DRS_CONSTRAINT;
```

There is one of these objects for each KHE constraint whose resource type agrees with the solver's resource type. There is no absolute requirement to have such a type; it has been included because the tables needed for uniform dominance testing are the same for all monitors derived from the same constraint. Having this type allows them to be built only once per constraint rather than once per monitor. This is a worthwhile saving because these tables can be very large.

There is the usual `KheDrsConstraintMake` operation, which however just leaves `NULL` values in the tables. Then there is

```
void KheDrsExprCostSetConstraint(KHE_DRS_EXPR_COST ec,
  int history, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_CONSTRAINT c;  int index;  KHE_DRS_CONSTRAINT dc;
  c = KheMonitorConstraint(ec->monitor->monitor);
  index = KheConstraintIndex(c);
  HaArrayFill(drs->all_constraints, index + 1, NULL);
  dc = HaArray(drs->all_constraints, index);
  if( dc == NULL )
  {
    /* new, so build and add a new object */
    dc = KheDrsConstraintMake(c, history, (KHE_DRS_EXPR) ec, drs);
    HaArrayPut(drs->all_constraints, index, dc);
  }
  else
  {
    /* already built, so just update the fields */
    if( history < dc->min_history )
      dc->min_history = history;
    if( history > dc->max_history )
      dc->max_history = history;
    if( HaArrayCount(ec->children) > dc->max_child_count )
      dc->max_child_count = HaArrayCount(ec->children);
    dc->needs_corr_table = dc->needs_corr_table ||
      KheDrsExprNeedsCorrTable((KHE_DRS_EXPR) ec, drs->days_frame);
  }
}
```

It uses the expression's constraint's index to look up the `all_constraints` array in the solver. If there is no constraint there, it calls `KheDrsConstraintMake` and adds one. If there is already one there, it updates its history and maximum number of children. Once all expressions are created, all constraints will be too.

The other main function builds the three tables stored in the constraint object:

```
void KheDrsConstraintSetTables(KHE_DRS_CONSTRAINT dc,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  switch( dc->sample_expr->tag )
  {
    case KHE_DRS_EXPR_COUNTER_TAG:

      /* set counter dom tables */
      KheDrsConstraintSetCounterDomTables(dc,
        (KHE_DRS_EXPR_COUNTER) dc->sample_expr, drs);
      break;

    case KHE_DRS_EXPR_SEQUENCE_TAG:

      /* set sequence dom tables */
      KheDrsConstraintSetSequenceDomTables(dc,
        (KHE_DRS_EXPR_SEQUENCE) dc->sample_expr, drs);
      break;

    default:

      HnAbort("KheDrsConstraintSetTables internal error (tag %d)",
        dc->sample_expr->tag);
      break;
  }
}
```

When the constraint supports `KHE_DRS_EXPR_COUNTER` expressions, the `counter_main_dom_table3` and `counter_corr_dom_table5` tables are set. When the constraint supports `KHE_DRS_EXPR_SEQUENCE` expressions, the `sequence_main_dom_table5` table is set. The code for this, beginning with `KheDrsConstrainSetCounterDomTables` and `KheDrsConstrainSetSequenceDomTables`, may be found in this submodule, but we won't show it here. It is a direct transcription of the tabulated dominance formulas of Appendix C.

## D.7.2. Monitors

The solver has a `KHE_DRS_MONITOR` type:

```
typedef struct khe_drs_monitor_rec {
  KHE_MONITOR          monitor;
  KHE_COST             rerun_open_and_search_cost;
  KHE_COST             rerun_open_and_close_cost;
  KHE_DRS_EXPR         sample_expr;
} *KHE_DRS_MONITOR;

typedef HA_ARRAY(KHE_DRS_MONITOR) ARRAY_KHE_DRS_MONITOR;
```

There is one of these monitor objects for each KHE monitor whose resource type agrees with the resource type of the solver.

As for constraints, this type is not absolutely needed. Here, the motive is not to save space, but rather to test the solver. It is not practicable to debug a full run, because there is too much data. But one can debug a single path through the search tree, which the solver calls a rerun, as explained in detail in Appendix D.12.4. While doing this, the `rerun_open_and_search_cost` and `rerun_open_and_close_cost` fields are kept up to date, and at the end one can check for disagreements with the authoritative costs produced by the KHE platform.

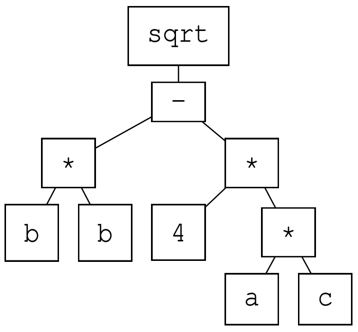
This work needs to be done per monitor, not per expression, because the costs obtainable from KHE are per monitor. So all expressions derived from a given KHE monitor contain a pointer to a shared DRS monitor object. (The solver often converts one monitor into several expressions. This often allows these expressions to avoid having to contribute state information to solution signatures, a significant saving.)

The operations on DRS monitors include `KheDrsMonitorMakeAndAdd`, which makes a monitor and adds it to the solver's `all_monitors` array; `KheDrsMonitorUpdateRerunCost`, which is called during reruns to update the two cost fields, and optionally to produce debug output saying what was done; and `KheDrsMonitorCheckRerunCost`, which checks at the end of the run that `rerun_open_and_search_cost` and `rerun_open_and_close_cost` both agree with KHE's value for the monitor's cost.

## D.8. Expressions

### D.8.1. Introduction

The reader is assumed to be familiar with *expression trees*, which are tree structures representing algebraic expressions. For example, $\sqrt{b^2 - 4ac}$ may be represented by the expression tree



If variables have values, each node has a value, dependent on its type and its children's values.

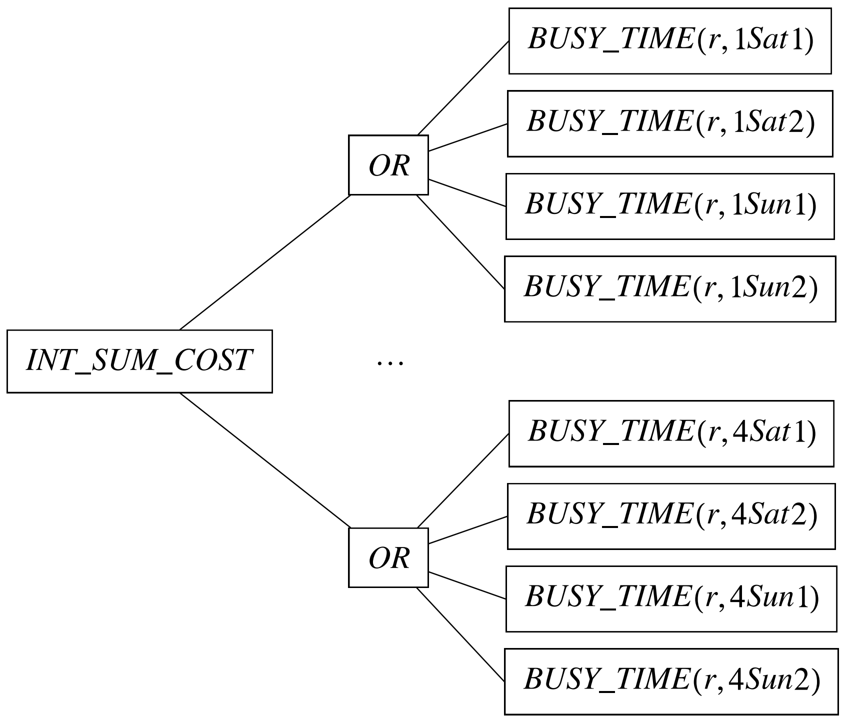
Each node is similar to the other nodes in some ways (they are all expression tree nodes), but different in others (for example, in the operations they perform). This situation calls for inheritance, with an abstract base class representing expression tree nodes in general, inherited by several concrete child classes representing particular kinds of expressions.

In our application, each constraint (strictly speaking, each point of application of each constraint, represented in KHE by a monitor) is represented by an expression tree which, given a particular solution, can be evaluated to yield the cost of the monitor. The abstract base class is `KHE_DRS_EXPR`. There are 15 concrete subclasses representing particular types of expressions.

Here we are concerned with introducing expressions generally, so although we will use some concrete subtypes in examples, we leave the full list for later (Appendix D.8.8).

Although the term ‘expression’ most naturally means ‘expression tree’, we usually use it to mean ‘expression tree node’. As explained earlier, this is done to avoid the term ‘node’, which is ambiguous here because it could also mean ‘search tree node’.

Here is an expression tree for constraining the number of busy weekends for resource *r*:



To fit it onto the page, it is drawn sideways with the subtrees for two weekends omitted. We assume that the instance has 28 days, starting on a Monday, with two shifts per day.

A *BUSY_TIME*$(r,t)$ expression has value 1 when *r* is busy at time *t*. An *OR* expression has value 1 when at least one of its children has value 1. An *INT_SUM_COST* expression sums the values of its children, compares the result with the limits (stored in the expression, but not shown here), and calculates a cost, using a cost function and weight stored in the expression.

Although every expression has a value, different types of expressions have different types of values. Most have values of type `int`; *INT_SUM_COST* expressions have values of type `KHE_COST`; and there are also expressions whose values have type `float`.

An *external expression* is an expression with no children. Its value depends on the state of the solution. For example, the *BUSY_TIME* expressions above are external expressions. An *internal expression* is an expression with one or more children. Its value depends on its children’s values. The *INT_SUM_COST* and *OR* expressions above are internal expressions.

External and internal expressions are sometimes handled differently. For example, although the implementation allows arbitrary common sub-expressions (that is, it allows any tree to be a subtree of any number of larger trees), only external expressions utilize this option.

The KHE platform does not use expression trees; it implements each kind of constraint with its own data structure. Expression trees allow more code sharing than special data structures do: *INT_SUM_COST*, for example, is used by several constraints. Another reason for using expression trees will be given when we come to consider signatures in detail (Appendix D.8.3).

Here is the base class, `KHE_DRS_EXPR`:

```
typedef struct khe_drs_expr_rec *KHE_DRS_EXPR;
typedef HA_ARRAY(KHE_DRS_EXPR) ARRAY_KHE_DRS_EXPR;

#define INHERIT_KHE_DRS_EXPR                                         \
  KHE_DRS_EXPR_TAG                      tag;                         \
  bool                                  gathered;                    \
  bool                                  open;                        \
  int                                   postorder_index;             \
  KHE_DRS_RESOURCE                      resource;                    \
  KHE_DRS_VALUE                         value;                       \
  KHE_DRS_VALUE                         value_ub;                    \
  ARRAY_KHE_DRS_PARENT                  parents;                     \
  ARRAY_KHE_DRS_EXPR                    children;                    \
  struct khe_drs_open_children_rec      open_children_by_day;        \
  HA_ARRAY_INT                          sig_indexes;

struct khe_drs_expr_rec {
  INHERIT_KHE_DRS_EXPR
};
```

The fields lie in a macro to facilitate inheritance, as we'll see. The `tag` field has enumerated type and says which concrete type of expression this is. The `gathered` field is `true` when the expression has been gathered for opening (explained later) but not actually opened yet. The `open` field is `true` when the expression is open.

Each expression has a unique value of the `postorder_index` field. Children have smaller values than their parents, so that if the expressions are sorted by increasing `postorder_index`, they appear in postorder. These fields are set as expressions are created, and remain fixed.

The `resource` field is set in expressions that represent resource constraints, to the resource that the constraint applies to. It is `NULL` in expressions that represent event resource constraints.

The `value` field contains the value of the expression when a value is defined. It was stated earlier that an expression's value could have type `int`, `float`, or `KHE_COST`. However, values of type `KHE_COST` are not stored in expressions (instead, as we will see later, costs are reported immediately to solutions), so (as we saw earlier) type `KHE_DRS_VALUE` is

```
typedef union {
  int                   i;
  float                 f;
} KHE_DRS_VALUE;
```

In an expression `e` of type *OR*, say, which has an integer value, the value is `e->value.i`.

The `value` field has a defined value in two contexts. First, when `e` is closed, `e`'s value is fixed and its `value` field holds that value. Indeed, the `value` field is assumed to continue to hold the closed value even after `e` opens, but only until its parents open. Second, when `e` is open, and a solution is being evaluated which happens to be for `e`'s last open day, `value` holds `e`'s value temporarily, from when the value is calculated until its parents have retrieved it. Otherwise the `value` field is undefined.

The `value_ub` field is a constant upper bound for `value`, set immediately after the expression is created, and never changed.

The `parents` field contains pointers to the expression's parents. Most expressions have one parent, but external expressions may have several. `KHE_DRS_PARENT` is

```
typedef struct khe_drs_parent_rec {
  KHE_DRS_EXPR          expr;
  int                   index;
} KHE_DRS_PARENT;

typedef HA_ARRAY(KHE_DRS_PARENT) ARRAY_KHE_DRS_PARENT;
```

and holds the parent plus the child expression's index in the list of children of the parent.

Field `children` holds the children of this expression. Fields `open_children_by_day` and `sig_indexes` are used only when the expression is open. We'll discuss them later.

As an example of inheritance, here is type `KHE_DRS_EXPR_OR`, the type of *OR* expressions:

```
typedef struct khe_drs_expr_or_rec {
  INHERIT_KHE_DRS_EXPR
  int                           closed_state;
} *KHE_DRS_EXPR_OR;
```

It inherits all the fields of `KHE_DRS_EXPR`, making a C typecast from `KHE_DRS_EXPR_OR` to `KHE_DRS_EXPR` safe. Its tag field has the enumerated value `KHE_DRS_EXPR_OR_TAG`.

When present in an expression $x$, the `closed_state` field holds a summary of the values of $x$'s closed children. It is always defined, even when $x$ is open. (If an expression is open, its parents must also be open, but not all of its children need be open.) In *OR* expressions, the closed state is the number of closed children with value 1. Consulting this rather than the closed children themselves avoids visiting closed expressions during the solve, needed to fulfil the promise of running in time proportional to the number of open objects, not the number of objects.

## D.8.2. Construction

Constructing expression trees is basically a simple matter of creating the right objects and linking them together correctly. There are however a couple of things that deserve some attention.

The solver uses three private functions, `KheDrsExprInitBegin`, `KheDrsExprInitEnd`, and `KheDrsExprAddChild`, for constructing expression trees. For example, suppose we want to construct an *OR* expression with some children. We do this as follows:

```
KHE_DRS_EXPR_OR res;
HaMake(res, drs->arena);
KheDrsExprInitBegin((KHE_DRS_EXPR) res, KHE_DRS_EXPR_OR_TAG, dr, drs);
... initialize fields specific to OR expressions ...
... make child expressions and call KheDrsExprAddChild on each ...
KheDrsExprInitEnd((KHE_DRS_EXPR) res, drs);
res->value_ub.i = 1;
```

`HaMake` obtains memory for the new object, `res`, as usual. `KheDrsExprInitBegin` initializes its fields that are common to all expressions, including `tag` and `resource`, which vary from one expression to another. Next, fields specific to the type of expression being constructed must be initialized. For *OR* expressions this is just the `closed_state` field. Then the children of the new expression must be created, which involves, for each child, carrying out this same sequence, from `KheDrsExprInitBegin` to `KheDrsExprInitEnd`, followed by a call to `KheDrsExprAddChild` to link parent and child.

Correct construction requires that `KheDrsExprInitEnd` be called immediately after the children have been constructed and linked in, but not before. We can see why by studying the functions. `KheDrsExprInitBegin` is quite trivial, although we will have to look carefully at `KheDrsOpenChildrenInit` later:

```
void KheDrsExprInitBegin(KHE_DRS_EXPR e, KHE_DRS_EXPR_TAG tag,
  KHE_DRS_RESOURCE dr, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_OPEN_CHILDREN_INDEX_TYPE index_type;  float float_ub;
  e->tag = tag;
  e->gathered = false;
  e->open = false;
  e->postorder_index = -1;
  e->resource = dr;
  /* e->value and e->value_ub are not initialized here */
  HaArrayInit(e->parents, drs->arena);
  HaArrayInit(e->children, drs->arena);
  index_type = (tag == KHE_DRS_EXPR_SEQUENCE_TAG ?
    KHE_DRS_OPEN_CHILDREN_INDEX_DAY_ADJUSTED :
    KHE_DRS_OPEN_CHILDREN_INDEX_DAY);
  float_ub = (tag == KHE_DRS_EXPR_SUM_FLOAT_TAG);
  KheDrsOpenChildrenInit(&e->open_children_by_day, index_type,
    float_ub, drs);
  HaArrayInit(e->sig_indexes, drs->arena);
}
```

`KheDrsExprInitEnd` is more interesting:

```
void KheDrsExprInitEnd(KHE_DRS_EXPR e, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  /* postorder index */
  e->postorder_index = drs->postorder_count++;

  /* closed value; e->value is initialized here */
  KheDrsExprSetClosedValue(e, drs);
}
```

There are two points here. First, `KheDrsExprInitEnd` assigns `e->postorder_index` using a value from the solver. Clearly, this will only work as intended when `KheDrsExprInitEnd` is called on `e` after it has been called on each of `e`'s children.

Second, `KheDrsExprInitEnd` calls `KheDrsExprSetClosedValue` (Appendix D.8.5) to initialize `e->value`. Although we mainly use `KheDrsExprSetClosedValue` to find the closed value of `e` at the end of a solve, what it does is just right here: it sets the closed value of `e`, assuming that the children of `e` have their correct closed values, and that any closed state in `e` is correct. The closed value is wanted here, because initially all expressions are closed.

Now here is `KheDrsExprAddChild`:

```
void KheDrsExprAddChild(KHE_DRS_EXPR parent, KHE_DRS_EXPR child,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_PARENT prnt;

  /* link parent and child */
  HnAssert(HaArrayCount(parent->parents) == 0,
    "KheDrsExprAddChild internal error:  too late to add child (1)");
  HnAssert(parent->postorder_index == -1,
    "KheDrsExprAddChild internal error:  too late to add child (2)");
  prnt.expr = parent;
  prnt.index = HaArrayCount(parent->children);
  HaArrayAddLast(child->parents, prnt);
  HaArrayAddLast(parent->children, child);

  /* update state in each parent */
  switch( parent->tag )
  {
    case KHE_DRS_EXPR_OR_TAG:

      KheDrsExprOrAddChild((KHE_DRS_EXPR_OR) parent, child, drs);
      break;

    ...
  }
}
```

The first part is common to all expressions: it adds `child` to `parent`'s list of children, and it adds `parent` to `child`'s list of parents. The second part updates the state of the parent to include the child, and is specific to each type of expression, hence the large switch. Here is one branch:

```
void KheDrsExprOrAddChild(KHE_DRS_EXPR_OR eo, KHE_DRS_EXPR child_e,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  if( child_e->value.i == 1 )
    eo->closed_state += 1;
}
```

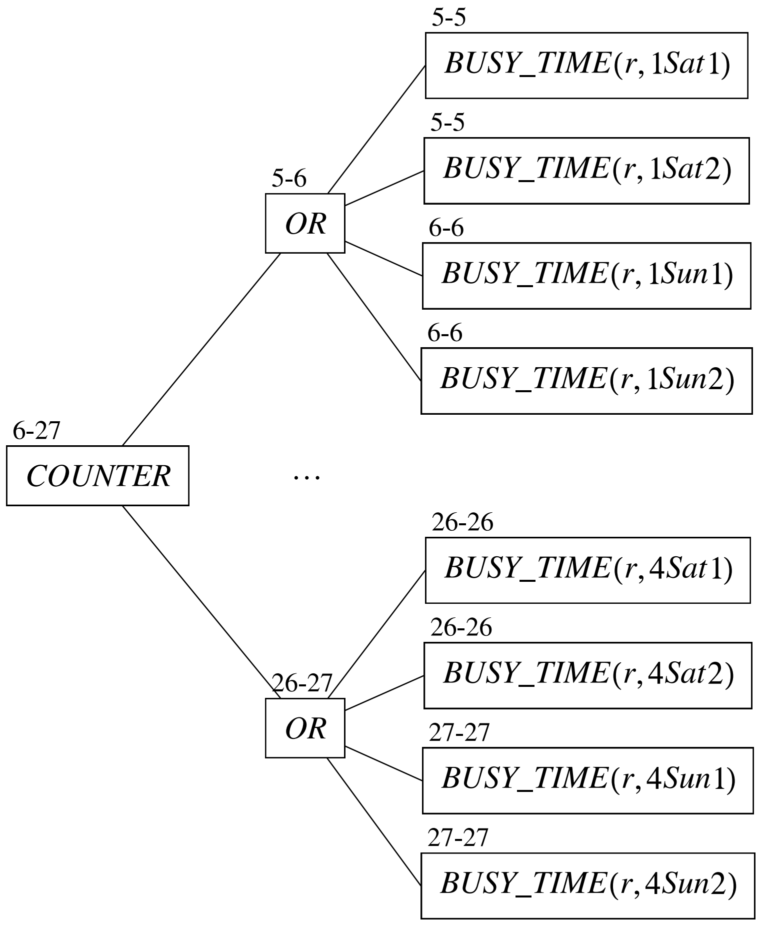
The closed state of `eo` is the number of closed children with value 1. All children are closed initially, so this adds 1 to `eo->closed_state` if `child_e`'s value is 1. The value is well-defined, because `KheDrsExprInitEnd` is called on `child_e` before this call is made.

`KheDrsExprAddChild` is declared in the expression construction submodule, but not defined until after `KHE_DRS_EXPR`'s subtypes, to avoid having to give forward declarations of its subtype versions. This is done for each function which switches on the expression's tag field: `KheDrsExprAddChild`, `KheDrsExprChildHasOpened`, `KheDrsExprChildDomTest`, `KheDrsExprDayDomTest`, `KheDrsExprChildHasClosed`, `KheDrsExprSetClosedValue`, `KheDrsExprLeafSet`, `KheDrsExprLeafClear`, and `KheDrsExprEvalSignature`.

### D.8.3. Open day ranges and signatures

This section explains in detail the values that expressions add to signatures. The implementation is part of expression opening and will be given later, in Appendix D.8.5.

We may have used the term *open day range* previously, more or less synonymously with *selected day range*. But now we define the open day range of an expression precisely. As an example, we'll use the constraint that limits the number of busy weekends for resource *r* in a four-week timetable beginning on a Monday. The weekend days are 5, 6, 12, 13, 19, 20, 26, and 27. Here is the expression tree, showing open day ranges:



A review of the external expression types (like *BUSY_TIME*) given in Appendix D.8.8 will show that each is affected by what happens on exactly one day. The open day range of an external expression contains exactly this one day. The open day range of an internal expression (like *COUNTER* and *OR*) is the smallest range of days which includes the last day of each of its children's open day ranges. For example, the last days of the open day ranges of the children of the *COUNTER* expression above are 6, 13, 20, and 27, so its open day range is 6-27. (The reader who expected it to be 5-27 needs to disabuse himself of that idea now.)

The numbers used in open day ranges are open day indexes, not frame indexes. The example assumes that all days are open. If some are closed, the open day indexes will be different.

When we speak of an expression's open days, we mean the days of its open day range. For example, we can say that every open expression has at least one open day, meaning that every open expression has a non-empty open day range.

A solve takes a whole set of solutions for some open day $d_i$, and from each of them it makes solutions for day $d_{i+1}$. It needs to be able to pick up a solution for day $d_i$, build a new solution for day $d_{i+1}$ consisting of the solution for $d_i$ plus one day's worth of new assignments, and then set the new solution aside. But it can't ignore the constraints and their costs, because it needs to prune solutions whose cost so far is not less than the cost of the initial solution, and it needs to implement dominance testing, as described in Appendix C.1. Information about a solution's constraints and costs is stored in its `cost` and `signature` fields.

So then, what does a solution for up to day $d_i$ need to store about our example constraint?

Clearly, the number of open busy weekends up to $d_i$. This way, as we proceed along any path in the search tree from the root solution of the tree to a final complete solution, each solution will record the number of open busy weekends so far. It will be easy, at each solution, to combine the number of open busy weekends up to the previous day with the task assignment for $r$ on the new day to find the number of busy weekends up to the new day. Then, on the constraint's last open day, the number of open busy weekends can be added to the number of closed busy weekends, and the sum compared with the limits to find a cost. (Actually, we calculate a cost on every day, since it may be useful in pruning solutions. See Appendix D.8.10.)

But suppose $d_i$ is a Saturday. Then the solution must also remember whether that Saturday is busy. It is not enough to store just a number of busy weekends, because then it is not possible to say whether a busy next day (Sunday) makes one more busy weekend or not.

The reader who ponders this will find that an expression contributes a value to store on each day of its open day range except its last. Before its first open day, there is nothing to remember (not counting closed state, which is the same for each solution so is stored in the expression). On days during the open day range other than the last, there is information from that day and previous days to store. For example, a day 7 solution needs to store, for the *COUNTER* expression, the number of open busy weekends so far. This is true even though none of that expression's children are open on day 7, which explains why we use open day ranges rather than open day sets. On the last open day, the expression's value is found and reported to its parents. It becomes the parents' responsibility, so there is nothing to store on that day, or afterwards.

Applying this rule to the tree above, information needs to be stored for the *COUNTER* expression on days 6-26, information for the first *OR* expression needs to be stored on day 5, and so on. Nothing ever needs to be stored for the *BUSY_TIME* expressions, because their open day ranges contain no days that are not last. Nothing is stored for the *COUNTER* expression on day 5, because none of its children will have reported anything then.

It is clear now why we use expression trees to represent constraints. A solution stores one item of information (or nothing) per expression, not per constraint. The item stored need not be the expression's value. For example, the *COUNTER* expression's value is a cost, but what is stored for it is an integer number of open busy weekends.

The signature of a solution for a given day $d_i$ consists of one item of information for each open expression for which $d_i$ is one of its open days other than the last. The items' types are not clear at this point, but, looking ahead, it will turn out that the root of each expression tree will have type `KHE_COST`, and the sum of these costs will be stored in the solution's `cost` field; while the non-root expressions will contribute one `int` or `float` each, held in the `signature` field.

When two signatures are compared during dominance testing, each position along the signature has its own test. While this keeps things simple, it does cause some cases of dominance to be missed. For example, suppose that the current day is `2Sat`, and some resource is busy on that day in solution $S_1$ but not in solution $S_2$. If there is a maximum limit on the number of busy weekends, $S_1$ cannot dominate $S_2$, because the '$\leq$' test at the *OR* expression affected by `2Sat` fails. But suppose the '$\leq$' test at the enclosing *COUNTER* expression succeeds with one weekend to spare. Then $S_1$ does in fact dominate $S_2$. This problem has in fact been fixed, by *correlated expressions*. But we won't delve into them now.

We'll see in Appendix D.9 that the parts of each signature made by expressions representing resource constraints are calculated separately, before `KheDrsSolnExpand` generates any new solutions, while the parts representing event resource constraints are calculated as new solutions are generated. This distinction is irrelevant to the actual process of calculating the signature, and expression objects are unaware of it.

Here are two points that the author is inclined to view as rather profound. The reader can make up his own mind. First, each expression only ever needs to store a small constant amount of information in a signature. It never stores anything complicated, such as a set of values. However, if we were supporting the avoid split assignments constraint we would need to store a set: the set of distinct resources assigned so far. So this first point may be just luck.

Second, although signatures were created to ensure that costs can be calculated efficiently as solving proceeds, it turns out that they are just what is needed for dominance testing too. This has something to do with the fact that a signature contains complete information about the state of the constraints in its solution, but still it seems somewhat miraculous that the form in which this information is held for cost calculating should also suit dominance testing.

### D.8.4. Open children

Our next job is to explain how expressions are opened, but before that we need to consider type `KHE_DRS_OPEN_CHILDREN`, whose functions do most of the actual work of opening expressions. Each expression has a field of this type, holding its open children:

```
typedef struct khe_drs_open_children_rec {
  ARRAY_KHE_DRS_OPEN_CHILD              open_children;
  KHE_INTERVAL                          index_range;
  KHE_DRS_OPEN_CHILDREN_INDEX_TYPE      index_type;
  bool                                  float_ub;
  HA_ARRAY_INT                          child_indexes;
} *KHE_DRS_OPEN_CHILDREN;
```

It is a pointer type as usual, but (as we saw above) the field within `KHE_DRS_EXPR` is expanded, that is, it is a struct rather than a pointer to a struct:

```
#define INHERIT_KHE_DRS_EXPR                                           \
  ...                                                                  \
  struct khe_drs_open_children_rec      open_children_by_day;          \
  ...
```

It is done this way purely to save memory.

The `open_children` field holds its expression's open children. `KHE_DRS_OPEN_CHILD` is

```
typedef struct khe_drs_open_child_rec {
  KHE_DRS_EXPR          child_e;
  int                   open_index;
  KHE_DRS_VALUE         rev_cum_value_ub;
} KHE_DRS_OPEN_CHILD;
```

This is a non-pointer type, again to save space. The `child_e` field is the child itself; `open_index` is the child's *open index*, of which more in a moment; and `rev_cum_value_ub` is the sum, over all open children x from here to the end of the open_children array, of x->value_ub.

Returning to type KHE_DRS_OPEN_CHILDREN, the index_range field contains the minimum and maximum, over the open children, of their open indexes. If the open_children array is empty, then index_range.first > index_range.last, except that in a leaf expression, index_range.first and index_range.last are both set to some value whose provenance does not concern us here.

The open index of a child node is usually its last open day, the day on which it is expected to report its final value to its parent. However, other kinds of open index are occasionally used, specified by the index_type field:

```
typedef enum {
  KHE_DRS_OPEN_CHILDREN_INDEX_DAY,
  KHE_DRS_OPEN_CHILDREN_INDEX_DAY_ADJUSTED,
  KHE_DRS_OPEN_CHILDREN_INDEX_SHIFT
} KHE_DRS_OPEN_CHILDREN_INDEX_TYPE;
```

Value KHE_DRS_OPEN_CHILDREN_INDEX_DAY indicates that each child's open index is its last open day, the usual value. KHE_DRS_OPEN_CHILDREN_INDEX_DAY_ADJUSTED is the same except that some of the values are adjusted, as we'll see later. KHE_DRS_OPEN_CHILDREN_INDEX_SHIFT is quite different; it says that the open indexes are shift indexes.

The float_ub field is true when the children's value_ub fields (and indeed their value fields) all contain values of type float.

The child_indexes array is used to speed up access to the elements of children with a given open index, assuming that open indexes are monotone increasing as one proceeds along the sequence of open children (which is always the case). For each i from index_range.first to index_range.last + 1 inclusive,

```
HaArray(oc->child_indexes, i - oc->index_range.first)
```

is the number of elements of oc->children whose open index is less than i. For example, if i == oc->index_range.first the result is 0, and if i == oc->index_range.last + 1 the result is HaArrayCount(oc->children). Later we'll see an iterator, defined by a macro, that uses child_indexes to efficiently visit all children with a given open index.

The functions begin with KheDrsOpenChildrenInit, which initializes an open children object (giving it no children and an empty range), KheDrsOpenChildrenClear, which clears an open children object back to that state, and KheDrsOpenChildrenCount, which returns the length of the children array. After that come several functions used to keep a sequence of open children up to date as children are added. The first of these is

```
void KheDrsOpenChildrenUpdateIndexRange(KHE_DRS_OPEN_CHILDREN oc)
{
  int first_index, last_index;
  if( HaArrayCount(oc->open_children) == 0 )
    oc->index_range = KheIntervalMake(1, 0);
  else
  {
    first_index = HaArrayFirst(oc->open_children).open_index;
    last_index = HaArrayLast(oc->open_children).open_index;
    oc->index_range = KheIntervalMake(first_index, last_index);
  }
}
```

This brings `oc->index_range` up to date with any change in the open children. It assumes that the open children's open indexes are set and are monotone non-decreasing as we proceed along `oc->open_children`. Next is a function which brings the `open_children` array up to date, assuming that `oc->index_range` is up to date:

```
void KheDrsOpenChildrenUpdateChildIndexes(KHE_DRS_OPEN_CHILDREN oc)
{
  int index, i;  KHE_DRS_OPEN_CHILD open_child;
  HaArrayClear(oc->child_indexes);
  index = oc->index_range.first - 1;
  HaArrayForEach(oc->open_children, open_child, i)
  {
    while( open_child.open_index > index )
    {
      index++;
      HaArrayAddLast(oc->child_indexes, i);
    }
  }
  HaArrayAddLast(oc->child_indexes, i);
}
```

We leave the reader to verify that this sets `oc->child_indexes` to the value described above. Next comes a function to bring the `rev_cum_value_ub` field in each open child up to date:

```
void KheDrsOpenChildrenUpdateUpperBounds(KHE_DRS_OPEN_CHILDREN oc)
{
  int i, sum_i;  KHE_DRS_OPEN_CHILD open_child;  float sum_f;
  if( oc->float_ub )
  {
    /* float version */
    sum_f = 0.0;
    HaArrayForEachReverse(oc->open_children, open_child, i)
    {
      sum_f += open_child.child_e->value_ub.f;
      HaArray(oc->open_children, i).rev_cum_value_ub.f = sum_f;
    }
  }
  else
  {
    /* int version */
    sum_i = 0;
    HaArrayForEachReverse(oc->open_children, open_child, i)
    {
      sum_i += open_child.child_e->value_ub.i;
      HaArray(oc->open_children, i).rev_cum_value_ub.i = sum_i;
    }
  }
}
```

It uses `oc->float_ub` to decide whether `int` or `float` values are in use. Next comes a small function for finding the open shift index of the shift that a given expression monitors:

```
int KheDrsExprOpenShiftIndex(KHE_DRS_EXPR e)
{
  KHE_DRS_EXPR_ASSIGNED_TASK eat;  int res;
  eat = (KHE_DRS_EXPR_ASSIGNED_TASK) e;
  res = eat->task_on_day->encl_dt->encl_dmt->encl_shift->open_shift_index;
  HnAssert(res >= 0, "KheDrsExprOpenShiftIndex internal error 2");
  return res;
}
```

This function only applies to assigned task expressions.

After all these preparations we are ready for the function that adds a child to the sequence of open children:

```
void KheDrsOpenChildrenAddChild(KHE_DRS_OPEN_CHILDREN oc,
  KHE_DRS_EXPR child_e)
{
  int i, open_index, prev_open_index;  KHE_DRS_OPEN_CHILD tmp, open_child;

  /* make the child's open index */
  switch( oc->index_type )
  {
    case KHE_DRS_OPEN_CHILDREN_INDEX_DAY:

      /* open index is child_e's last day */
      open_index = child_e->open_children_by_day.index_range.last;
      break;

    case KHE_DRS_OPEN_CHILDREN_INDEX_DAY_ADJUSTED:

      /* open index is child_e's last day after adjustment */
      if( HaArrayCount(oc->open_children) > 0 )
      {
        prev_open_index = HaArrayLast(oc->open_children).open_index;
        if( child_e->open_children_by_day.index_range.last < prev_open_index )
          child_e->open_children_by_day.index_range.last = prev_open_index;
      }
      open_index = child_e->open_children_by_day.index_range.last;
      break;

    case KHE_DRS_OPEN_CHILDREN_INDEX_SHIFT:

      /* open index is child_e's shift index */
      open_index = KheDrsExprOpenShiftIndex(child_e);
      break;

    default:
      HnAbort("KheDrsOpenChildrenAddChild internal error");
      open_index = 0;  /* keep compiler happy */
  }

  /* add child_e to oc->children in sorted position */
  ... see below ...

  /* update oc->index_range, oc->child_indexes, and rev_cum_value_ub fields */
  ... see below ...
}
```

The first paragraph sets `open_index` to the open index of `child_e`. If the index type is `KHE_DRS_OPEN_CHILDREN_INDEX_DAY`, his is value `index_range.last` from `child_e`'s open children. If the index type is `KHE_DRS_OPEN_CHILDREN_INDEX_DAY_ADJUSTED` it is this same value, except that if its value is out of order it is increased until it isn't. Finally, if the index type is `KHE_DRS_OPEN_CHILDREN_INDEX_SHIFT`, the open index is the open shift index.

The second paragraph inserts `child_e`, or rather a new open child object containing `child_e`, into the sequence of open children, ensuring that the childrens' open indexes are sorted into non-decreasing order as required:

```
/* add child_e to oc->children in sorted position */
tmp = KheDrsOpenChildMake(child_e, open_index);
HaArrayAddLast(oc->open_children, tmp);  /* not really, just to make space */
for( i = HaArrayCount(oc->open_children) - 2;  i >= 0;  i-- )
{
  open_child = HaArray(oc->open_children, i);
  if( open_child.open_index <= open_index )
    break;
  HaArrayPut(oc->open_children, i + 1, open_child);
}
HaArrayPut(oc->open_children, i + 1, tmp);  /* for real this time */
```

The last paragraph updates the index range, child indexes, and upper bounds:

```
/* update oc->index_range, oc->child_indexes, and rev_cum_value_ub fields */
KheDrsOpenChildrenUpdateIndexRange(oc);
KheDrsOpenChildrenUpdateChildIndexes(oc);
KheDrsOpenChildrenUpdateUpperBounds(oc);
```

These three functions are given above. All of this is arguably slower than it needs to be, but since it is only done during opening that hardly matters.

A similar but much simpler function deletes a child from the sequence of open children, keeping everything up to date:

```
void KheDrsOpenChildrenDeleteChild(KHE_DRS_OPEN_CHILDREN oc,
  KHE_DRS_EXPR child_e)
{
  KHE_DRS_OPEN_CHILD open_child;  int i;

  HaArrayForEach(oc->open_children, open_child, i)
    if( open_child.child_e == child_e )
    {
      /* delete and shift here */
      HaArrayDeleteAndShift(oc->open_children, i);

      /* update oc->index_range, oc->child_indexes, and rev_cum_value_ub */
      KheDrsOpenChildrenUpdateIndexRange(oc);
      KheDrsOpenChildrenUpdateChildIndexes(oc);
      KheDrsOpenChildrenUpdateUpperBounds(oc);

      /* all done */
      return;
    }

  /* should never get here */
  HnAbort("KheDrsOpenChildrenDeleteChild internal error");
}
```

Again, this is slower than it needs to be, but it is done only during closing.

With the open children in good order, several operations are available. First we have

```
int KheDrsOpenChildrenBefore(KHE_DRS_OPEN_CHILDREN oc, int index)
{
  if( index < oc->range.first )
    return 0;
  else if( index > oc->range.last )
    return HaArrayCount(oc->children);
  else
    return HaArray(oc->child_indexes, index - oc->range.first);
}
```

This returns the number of open children whose open index is less than `index`. At the time of writing, the author is unsure whether the first two cases, which return correct values for out-of-range indexes, are needed. And

```
int KheDrsOpenChildrenAtOrAfter(KHE_DRS_OPEN_CHILDREN oc, int index)
{
  return HaArrayCount(oc->children) - KheDrsOpenChildrenBefore(oc, index);
}
```

returns the number of open children whose open index is greater than or equal to `index`.

There is a function for finding a reverse cumulative upper bound from a certain point on:

```
int KheDrsOpenChildrenUpperBoundInt(KHE_DRS_OPEN_CHILDREN oc,
  int open_index)
{
  int i;
  HnAssert(!oc->float_ub,
    "KheDrsOpenChildrenUpperBoundInt internal error 1");
  if( open_index < oc->index_range.first )
    HnAbort("KheDrsOpenChildrenUpperBoundInt internal error 2");
  if( open_index > oc->index_range.last )
    return 0;
  else
  {
    i = HaArray(oc->child_indexes, open_index - oc->index_range.first);
    return HaArray(oc->open_children, i).rev_cum_value_ub.i;
  }
}
```

This one finds the total reverse cumulative upper bound:

```
int KheDrsOpenChildrenUpperBoundIntAll(KHE_DRS_OPEN_CHILDREN oc)
{
  HnAssert(!oc->float_ub,
    "KheDrsOpenChildrenUpperBoundIntAll internal error 1");
  if( HaArrayCount(oc->child_indexes) == 0 )
    return 0;
  else
    return HaArrayFirst(oc->open_children).rev_cum_value_ub.i;
}
```

`KheDrsOpenChildrenUpperBoundFloat` and `KheDrsOpenChildrenUpperBoundFloatAll` are the same, except that they assume that values are floating-point.

Next we have two iterators, implemented by macros that expand to C `for` statements. The first iterates over each open index, as stored in the index range:

```
#define KheDrsOpenChildrenForEachIndex(oc, i)                  \
  for( i = (oc)->index_range.first;  i <= (oc)->index_range.last;  i++ )
```

The second iterates over all open children `x` with a given open index `index`:

```
#define KheDrsOpenChildrenForEach(oc, index, x, i)                    \
  i1 = KheDrsOpenChildrenBefore((oc), (index));                       \
  i2 = KheDrsOpenChildrenBefore((oc), (index) + 1);                   \
  for( (i) = i1;                                                      \
   (i) < i2 ? ((x) = HaArray((oc)->open_children, (i)).child_e, true) : \
   false; (i)++ )
```

This relies on the sentinel value at the end of `oc->child_indexes`.

Next come four simple functions (we won't show them) for comparing an index with the open range index: `KheDrsOpenChildrenIndexInRange`, `KheDrsOpenChildrenIndexIsFirst`, `KheDrsOpenChildrenIndexIsFirstOrLess`, and `KheDrsOpenChildrenIndexIsLast`. Also,

```
int KheDrsOpenChildrenWithIndex(KHE_DRS_OPEN_CHILDREN oc, int index)
{
  return KheDrsOpenChildrenBefore(oc, index + 1) -
    KheDrsOpenChildrenBefore(oc, index);
}
```

which returns the number of open children with a given open index.

### D.8.5. Opening

This section explains how expressions are opened. An expression needs to be opened when its value may be affected by an assignment to some open task.

The first step in opening expressions is to build a complete list of all expressions that need to be opened, in field `open_exprs` of the solver. This is done by calls to this function:

```
void KheDrsExprGatherForOpening(KHE_DRS_EXPR e,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_PARENT prnt;  int i;
  if( !e->gathered )
  {
    e->gathered = true;
    HaArrayAddLast(drs->open_exprs, e);
    HaArrayForEach(e->parents, prnt, i)
      KheDrsExprGatherForOpening(prnt.expr, drs);
  }
}
```

Whenever `e` should open, `KheDrsExprGatherForOpening` is called. If `e->gathered` is `true`, meaning that `e` has already been gathered, this does nothing. Otherwise it sets `e->gathered` to `true` to ensure that `e` will not be gathered again on this solve, adds `e` to `drs->open_exprs`, and gathers its parents (an open expression's parents must also be open).

We have already seen the calls to `KheDrsExprGatherForOpening` which start the gathering process, in `KheDrsResourceOnDayOpen`:

```
open_day_range = KheDrsDayRangeMake(open_day_index, open_day_index);
HaArrayForEach(drd->external_today, e, i)
{
  e->open_children_by_day.range = open_day_range;
  KheDrsExprGatherForOpening(e, drs);
}
```

and in `KheDrsTaskOpen`, which we won't show again. These gather all external expressions that need to be opened, because they depend on what an open resource on day or task on day is doing.

They also set the open day range in each external expression to the single day that the expression is affected by. Then `KheDrsExprGatherForOpening` gathers their ancestors, which accounts for all expressions that need to be opened.

After all expressions have been gathered, they are sorted by increasing postorder index and opened. The code for this is far ahead of where we are now, in `KheDrsSolveOpen`:

```
HaArraySort(drs->open_exprs, &KheDrsExprPostorderCmp);
HaArrayForEach(drs->open_exprs, e, i)
  KheDrsExprOpen(e, drs);
HaArrayForEach(drs->open_exprs, e, i)
  KheDrsExprNotifySigners(e, drs);
```

Sorting ensures that parents are opened after their children. `KheDrsExprNotifySigners` informs various signers that `e` has opened, as required. We need to do this in a separate pass over the expressions, but at the time of writing the author has forgotten why.

At the moment each expression opens, it calls `KheDrsExprChildHasOpened` once for each parent to inform it that one of its children has opened:

```
void KheDrsExprChildHasOpened(KHE_DRS_EXPR e, KHE_DRS_EXPR child_e,
  int child_index, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  switch( e->tag )
  {
    case KHE_DRS_EXPR_OR_TAG:

      KheDrsExprOrChildHasOpened((KHE_DRS_EXPR_OR) e,
        child_e, child_index, drs);
      break;

    ...
  }
}
```

Here `child_e` is the child that has just opened, and `e` is the parent, not yet opened. Clearly `e` must be an internal expression, since it has a child. This function is just a type switch; each case updates `e` to take account of the fact that `child_e` has opened. Here is one branch of the switch:

```
void KheDrsExprOrChildHasOpened(KHE_DRS_EXPR_OR eo,
  KHE_DRS_EXPR child_e, int child_index, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KheDrsOpenChildrenAddChild(&eo->open_children_by_day, child_e);
  if( child_e->value.i == 1 )
    eo->closed_state -= 1;
}
```

The first step is to add `child_e` to the list of open children of `eo`. Then, since within *OR* expressions the `closed_state` field holds the number of closed children whose value is 1, it has to be reduced by 1 if `child_e`'s value is 1, since `child_e` is no longer closed.

There are two important points here. First, while an expression is closed, its value is up to date, and does not change during the current solve. When a closed expression is opened, as `child_e` is opened here, it retains its closed value for some time, at least until its parents are opened. So it is safe here to access `child_e->value.i`. Second, this code only touches open expressions. It avoids closed children, as it must if we are to meet our efficiency goals.

Here now is `KheDrsExprOpen`:

```
void KheDrsExprOpen(KHE_DRS_EXPR e, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_PARENT prnt;  int i;

  /* inform e's parents that e is now open */
  e->gathered = false;
  e->open = true;
  HaArrayForEach(e->parents, prnt, i)
    KheDrsExprChildHasOpened(prnt.expr, e, prnt.index, drs);

  /* if e is external, clear its value */
  if( e->tag <= KHE_DRS_EXPR_WORK_DAY_TAG )
    KheDrsExprLeafClear(e, drs);
}
```

When `KheDrsExprOpen(e, drs)` begins, `e` is considered to open. So the first step is to set `e->gathered` to `false` ('gathered' means 'gathered but not opened'), set `e->open` to `true`, and inform `e`'s parents that `e` has opened, by calling `KheDrsExprChildHasOpened` on each of them. After that, if `e` is external, searching assumes that `e`'s initial value is correct for when there are no assignments of open tasks to open resources. So `KheDrsExprOpen` calls `KheDrsExprLeafClear` to give this value to `e`.

After all the expressions are opened, as shown above we re-traverse them and call `KheDrsExprNotifySigners`:

```
void KheDrsExprNotifySigners(KHE_DRS_EXPR e,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  if( e->tag <= KHE_DRS_EXPR_WORK_DAY_TAG )
  {
    /* external expression; nothing to do */
  }
  else if( e->resource != NULL )
  {
    /* add e to its resource on day signers */
    KheDrsExprNotifyResourceSigners(e, drs);
  }
  else
  {
    /* add e to its cover signers */
    HnAssert(e->tag == KHE_DRS_EXPR_COUNTER_TAG,
      "KheDrsExprNotifySigners: internal error");
    KheDrsExprCounterNotifyCoverSigners((KHE_DRS_EXPR_COUNTER) e, drs);
  }
}
```

We previously gave a detailed description of each kind of signer, including the expressions and dominance tests each kind requires (Appendix D.6.3). We're now about to make that happen, but organized for each expression rather than for each signer.

If `e` is derived from a resource constraint, the code for enrolling it into its signers is

```
void KheDrsExprNotifyResourceSigners(KHE_DRS_EXPR e,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_DAY day;  KHE_DRS_RESOURCE_ON_DAY drd;  int di, sig_index;
  KHE_DRS_SIGNER dsg;  KHE_DRS_EXPR_COUNTER ec;

  HaArrayClear(e->sig_indexes);
  KheDrsOpenChildrenForEachIndex(&e->open_children_by_day, di)
  {
    day = HaArray(drs->open_days, di);
    drd = KheDrsResourceOnDay(e->resource, day);
    dsg = KheDrsResourceOnDaySigner(drd);
    if( KheDrsSignerAddExpr(dsg, e, drs, &sig_index) )
      HaArrayAddLast(e->sig_indexes, sig_index);
  }
}
```

First, we clear the `sig_indexes` array. Then we call `KheDrsSignerAddExpr` for each open day, to add `e` to the signer for resource on day `drd`. This ensures that `e` is called back for evaluation when we are constructing the signature for that resource on day. If it returns `true`, that means that this is not `e`'s last day, so `e` needs to reserve a position in signatures controlled by the signer to store its state. This position is returned in value `sig_index` and appended to `e->sig_indexes`.

If `e` is an internal expression derived from an event resource constraint, as we stated above it must be a *COUNTER* expression. The code for notifying its signers is

```
void KheDrsExprCounterNotifyCoverSigners(KHE_DRS_EXPR_COUNTER ec,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  int di, si, i, j, sig_index;  KHE_DRS_SHIFT ds;
  KHE_DRS_DAY day;  KHE_DRS_SHIFT_PAIR dsp;  KHE_DRS_SIGNER dsg;

  /* notify day signers */
  HaArrayClear(ec->sig_indexes);
  KheDrsOpenChildrenForEachIndex(&ec->open_children_by_day, di)
  {
    day = HaArray(drs->open_days, di);
    dsg = KheDrsDaySigner(day);
    if( KheDrsSignerAddExpr(dsg, (KHE_DRS_EXPR) ec, drs, &sig_index) )
      HaArrayAddLast(ec->sig_indexes, sig_index);
  }

  /* notify shift signers */
  KheDrsOpenChildrenForEachIndex(&ec->open_children_by_shift, si)
  {
    ds = HaArray(drs->open_shifts, si);
    dsg = KheDrsShiftSigner(ds);
    KheDrsSignerAddExpr(dsg, (KHE_DRS_EXPR) ec, drs, &sig_index);
  }

  /* notify shift pair signers */
  KheDrsOpenChildrenForEachIndex(&ec->open_children_by_day, di)
  {
    day = HaArray(drs->open_days, di);
    HaArrayForEach(day->shifts, ds, i)
      HaArrayForEach(ds->shift_pairs, dsp, j)
      {
        dsg = KheDrsShiftPairSigner(dsp);
        KheDrsSignerAddExpr(dsg, (KHE_DRS_EXPR) ec, drs, &sig_index);
      }
  }
}
```

Each paragraph follows the pattern set by `KheDrsExprNotifyResourceSigners`: it retrieves some relevant signers and enrols `e` into each of them. In this case they are not resource on day signers; rather, they are day signers, shift signers, and shift pair signers. But `sig_index` values are only needed, and only stored, for day signers. The iterator

```
KheDrsOpenChildrenForEachIndex(&eisc->open_children_by_shift, si)
```

visits all shifts that `ec` is connected with.

We saw `KheDrsSignerAddExpr` previously (Appendix D.6.3). A key part of it was a function called `KheDrsExprEvalType` that decided whether expression `e` needs to be added to signer `dsg`, and if so whether a dominance test is needed, because `dsg`'s day is not `e`'s last day. Here is `KheDrsExprEvalType`:

```
KHE_DRS_EXPR_EVAL_TYPE KheDrsExprEvalType(KHE_DRS_EXPR e, KHE_DRS_SIGNER dsg,
  KHE_DYNAMIC_RESOURCE_SOLVER drs, KHE_DRS_DOM_TEST *dom_test)
{
  int di, si, si1, si2, scount;  KHE_DRS_EXPR_COUNTER ec;
  di = dsg->encl_day->open_day_index;
  if( dom_test != NULL )  *dom_test = NULL;
  switch( dsg->type )
  {
    case KHE_DRS_SIGNER_DAY:
    case KHE_DRS_SIGNER_RESOURCE_ON_DAY:

      if( !KheDrsOpenChildrenIndexInRange(&e->open_children_by_day, di) )
        return KHE_DRS_EXPR_EVAL_NO;
      else if( KheDrsOpenChildrenIndexIsLast(&e->open_children_by_day, di) )
        return KHE_DRS_EXPR_EVAL_LAST;
      else
      {
        if( dom_test != NULL )  *dom_test = KheDrsExprDomTest(e, di, drs);
        return KHE_DRS_EXPR_EVAL_NOT_LAST;
      }

    case KHE_DRS_SIGNER_SHIFT:

      ec = (KHE_DRS_EXPR_COUNTER) e;
      si = dsg->u.shift->open_shift_index;
      scount = KheDrsOpenChildrenWithIndex(&ec->open_children_by_shift, si);
      return KheDrsExprEvalTypeShift(ec, di, scount, drs, dom_test);

    case KHE_DRS_SIGNER_SHIFT_PAIR:

      ec = (KHE_DRS_EXPR_COUNTER) e;
      si1 = dsg->u.shift_pair->shift[0]->open_shift_index;
      si2 = dsg->u.shift_pair->shift[1]->open_shift_index;
      scount = KheDrsOpenChildrenWithIndex(&ec->open_children_by_shift, si1)
        + KheDrsOpenChildrenWithIndex(&ec->open_children_by_shift, si2);
      return KheDrsExprEvalTypeShift(ec, di, scount, drs, dom_test);

    default:

      HnAbort("KheDrsExprEvalType internal error (%d)\n", dsg->type);
      return KHE_DRS_EXPR_EVAL_NO;    /* keep compiler happy */
  }
}
```

For example, for a resource on day signer the expression needs to be added if it is active on the day of the signer; then if it is the expression's last day, it is needed but no dominance test is wanted; otherwise a dominance test is needed.

When a dominance test is added to a resource on day or day signer, its index in that signer's signatures is stored in `e->sig_indexes`, as we saw above. So when `e` wants to retrieve its state from a signature, it can consult its own `sig_indexes` array to work out where to look. (These retrievals are only needed from signatures controlled by resource on day and day signers, not signatures controlled by the other two types of signers.) This function performs that retrieval:

```
int KheDrsExprDaySigVal(KHE_DRS_EXPR e, int open_day_index,
  KHE_DRS_SIGNATURE sig)
{
  int pos;
  pos = HaArray(e->sig_indexes, open_day_index -
    e->open_children_by_day.range.first);
  return HaArray(sig->states, pos);
}
```

It returns the state of `e` stored in the signature of `soln`, using `e->sig_indexes` to find its index.

### D.8.6. Closing

After solving, the open expressions need to be closed. The `open_exprs` array is used to visit each open expression and close it:

```
HaArrayForEach(drs->open_exprs, e, i)
  KheDrsExprClose(e, drs);
```

Again, this closes children before parents. To close one expression, the code is

```
void KheDrsExprClose(KHE_DRS_EXPR e, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_PARENT prnt;  int i;  KHE_DRS_EXPR_COUNTER ec;

  /* set e's closed value */
  KheDrsExprSetClosedValue(e, drs);

  /* clear fields that are used only when e is open */
  KheDrsOpenChildrenClear(&e->open_children_by_day);
  HaArrayClear(e->sig_indexes);
  if( e->tag == KHE_DRS_EXPR_COUNTER_TAG )
  {
    ec = (KHE_DRS_EXPR_COUNTER) e;
    KheDrsOpenChildrenClear(&ec->open_children_by_shift);
  }

  /* close e and inform e's parents that e has closed */
  e->open = false;
  HaArrayForEach(e->parents, prnt, i)
    KheDrsExprChildHasClosed(prnt.expr, e, prnt.index, drs);
}
```

The first step is to set `e`'s value to whatever it is to be in the closed state, assuming for external expressions that all assignments are expressed in the `closed_asst` fields of tasks and resources (as we can do because expressions are closed after all assignments are made), and for internal expressions that `e`'s children are now all closed (as we can do because of the expression sorting). `KheDrsExprSetClosedValue` is the usual large switch:

```
void KheDrsExprSetClosedValue(KHE_DRS_EXPR e,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  switch( e->tag )
  {
    case KHE_DRS_EXPR_OR_TAG:

      KheDrsExprOrSetClosedValue((KHE_DRS_EXPR_OR) e, drs);
      break;

    ...
  }
}
```

This is different for each concrete expression type; here is one example:

```
void KheDrsExprOrSetClosedValue(KHE_DRS_EXPR_OR eo,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  eo->value.i = (eo->closed_state > 0 ? 1 : 0);
}
```

In *OR* expressions, the value is 1 if there is at least one child with value 1, and, since all the children are now closed, the `closed_state` field can tell us how many such children there are.

After `KheDrsExprClose` calls `KheDrsExprSetClosedValue`, it clears `e`'s fields and then ends with this code that we saw above:

```
/* inform e's parents that e has closed */
HaArrayForEach(e->parents, prnt, i)
  KheDrsExprChildHasClosed(prnt.expr, e, prnt.index, drs);
```

This informs `e`'s parents that `e` has closed, by calling this function on each parent:

```
void KheDrsExprChildHasClosed(KHE_DRS_EXPR e,
  KHE_DRS_EXPR child_e, int child_index, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  switch( e->tag )
  {
    case KHE_DRS_EXPR_OR_TAG:

      KheDrsExprOrChildHasClosed((KHE_DRS_EXPR_OR) e,
        child_e, child_index, drs);
      break;

    ...
  }
}
```

Even though `KheDrsExprChildHasOpened` adds `child_e` to `e`'s list of open children, `KheDrsExprChildHasClosed` does not remove `child_e` from `e`'s list of open children. Instead, when `e` is closed later its open children are cleared out, as we have seen in function `KheDrsExprClose`. Once again the details of `KheDrsExprChildHasClosed` depend on the expression type. Here they are for *OR* expressions:

```
void KheDrsExprOrChildHasClosed(KHE_DRS_EXPR_OR eo,
  KHE_DRS_EXPR child_e, int child_index)
{
  if( child_e->value.i == 1 )
    eo->closed_state += 1;
}
```

If the child's value is 1, that makes one more closed child with value 1.

### D.8.7. Searching

For expressions, searching is basically about evaluating an expression in the context of some solution. External expressions are evaluated by these functions:

```
void KheDrsExprLeafSet(KHE_DRS_EXPR e, KHE_DRS_TASK_ON_DAY dtd,
  KHE_DRS_RESOURCE dr);

void KheDrsExprLeafClear(KHE_DRS_EXPR e);
```

`KheDrsExprLeafSet` is called when `drd` is assigned to `dtd`, and `KheDrsExprLeafClear` is called when that assignment is removed. Both functions contain a switch with one branch for each external expression type. Here is an example of one of the branches:

```
void KheDrsExprBusyTimeLeafSet(KHE_DRS_EXPR_BUSY_TIME ebt,
  KHE_DRS_TASK_ON_DAY dtd, KHE_DRS_RESOURCE dr)
{
  ebt->value.i = (dtd->time == ebt->time ? 1 : 0);
}
```

If `dr` is assigned to `dtd`, then `ebt` has value 1 if `dtd`'s time is `ebt`'s time, and 0 otherwise (no resource is busy twice on one day). `KheDrsExprBusyTimeLeafClear` sets the value to 0.

For internal nodes evaluation is more complicated. It is done by calls on this function:

```
void KheDrsExprEvalSignature(KHE_DRS_EXPR e, KHE_DRS_SIGNER dsg,
  KHE_DRS_SIGNATURE prev_sig, KHE_DRS_SIGNATURE next_sig,
  KHE_DYNAMIC_RESOURCE_SOLVER drs);
```

This evaluates `e` on the day covered by `next_sig` (the open day after `prev_sig`'s day), and updates `next_sig`, which is controlled by `dsg`, by adding a value to the end of its signature, or changing its cost, or both. Its body is the usual large switch, this time with one branch for each internal expression type. Here is an example of one of the branches:

```
void KheDrsExprOrEvalSignature(KHE_DRS_EXPR_OR eo, KHE_DRS_SIGNER dsg,
  KHE_DRS_SIGNATURE prev_sig, KHE_DRS_SIGNATURE next_sig,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  int i, i1, i2, next_di;  KHE_DRS_EXPR child_e;  KHE_DRS_VALUE val;

  next_di = KheDrsSignerOpenDayIndex(dsg);
  if( KheDrsOpenChildrenIndexIsFirst(&eo->open_children_by_day, next_di) )
  {
    /*  no previous day, so we have a 0 (false) value here */
    val.i = 0;
  }
  else
  {
    /* not first day, so retrieve a previous value */
    val = KheDrsExprDaySigVal((KHE_DRS_EXPR) eo, next_di - 1, prev_sig);
  }

  /* accumulate the values of the children of eo that finalized today */
  KheDrsOpenChildrenForEach(&eo->open_children_by_day, next_di, child_e, i)
    if( child_e->value.i == 1 )
      val.i = 1;

  if( KheDrsOpenChildrenIndexIsLast(&eo->open_children_by_day, next_di) )
  {
    /* last day; incorporate closed state and set value */
    if( eo->closed_state > 0 )
      val.i = 1;
    eo->value = val;
  }
  else
  {
    /* not last day; store val in next_soln */
    KheDrsSignatureAddState(next_sig, val, dsg, (KHE_DRS_EXPR) eo);
  }
}
```

The details depend on the particular expression type, but the structure is common to all types.

First, find the expression's value before this day. This will be an initial value (here 0) if this is the expression's first open day, and will come from the signature of `prev_soln` otherwise.

Second, use iterator macro `KheDrsOpenChildrenForEach` (Appendix D.8.4) to visit the children for which `next_di` is the last open day, retrieve their values, and incorporate those values into the value of this expression. Here, to implement the *OR* function, any child whose value is 1 causes `val.i` to be set to 1. The children have their final values, because the postorder sorting ensures that `KheDrsExprEvalSignature` is called on the children before the parent.

Third, save the value. If this is the expression's last open day, the value simply remains in the expression (here, in `eo->value`) where it will be picked up by the expression's parents during their `KheDrsExprEvalSignature` calls. If this is not the expression's last open day, the value (or whatever state needs to be stored) is added to `next_sig`.

When the value is a cost, things are slightly different. No value is kept in the expression; instead, an extra cost (Appendix C.3) is added to `next_sig` each day.

This function does not really need to use parameter `dsg`. However, in some cases (*COUNTER* expressions derived from event resource constraints) evaluation needs to know the type of the signer, hence the presence of `dsg`.

### D.8.8. Types of expressions

In this section we present the types of expressions needed for the XESTT constraints.

First we have the types of external expressions in expression trees for event resource constraints. There is just one of these:

*ASSIGNED_TASK*$(t, g)$
An expression whose value is 1 when $t$, a task on day object, is assigned a resource from resource group $g$, and 0 otherwise.

Next we have the types of external expressions in expression trees for resource constraints:

*BUSY_TIME*$(r, t)$
An expression whose value is 1 when resource $r$ is busy at time $t$, otherwise 0.

*FREE_TIME*$(r, t)$
An expression whose value is 1 when resource $r$ is free at time $t$, otherwise 0.

*WORK_TIME*$(r, t)$
An expression whose value is the workload of resource $r$ at time $t$ (a `float` value). This will be 0.0 when $r$ is free at time $t$.

*BUSY_DAY*$(r, d)$
An expression whose value is 1 when resource $r$ is busy on day $d$, otherwise 0.

*FREE_DAY*$(r, d)$
An expression whose value is 1 when resource $r$ is free on day $d$, otherwise 0.

*WORK_DAY*$(r, d)$
An expression whose value is the workload of resource $r$ on day $d$ (a `float` value). This will be 0.0 when $r$ is free on day $d$.

The last three could be omitted, but they speed up some common cases, and they produce better value upper bounds. And here are the types of internal expressions:

*OR*
An expression whose value is 1 when at least one of its children has value 1, else 0. All its children must have value 0 or 1.

*AND*
An expression whose value is 1 when all of its children have value 1, else 0. All its children must have value 0 or 1.

*COUNTER*
An expression whose value is the deviation from given limits of the number of its children whose value is 1. All its children must have value 0 or 1. This is the cluster busy times monitor, essentially,

*SUM_INT*
An expression whose value is the deviation from given limits of the sum of the values of its children. All values are non-negative integers. This ought to subsume *COUNTER*, but the two types handle dominance testing differently, so they remain separate.

*SUM_FLOAT*
Identical to *SUM_INT* except that the children have non-negative `float` values, and this expression produces a `float` sum before converting it into an integer deviation.

*SEQUENCE*
Like *COUNTER*, except that its value is the set of deviations of sequences of children with value 1. This is the limit active intervals monitor, essentially. It is easily the most complex expression type to implement. A full description is given in Appendix D.8.11.

The last four types may also report a cost based on their deviation (or a set of costs in the case of *SEQUENCE*). They also handle history; *SUM_FLOAT* is ahead of XESTT in that respect.

These expression types are implemented as subtypes of `KHE_DRS_EXPR`. All subtypes have the same operations. For example, the operations on `KHE_DRS_EXPR_OR` are `KheDrsExprOrMake`, `KheDrsExprOrAddChild`, `KheDrsExprOrChildHasOpened`, `KheDrsExprOrChildHasClosed`, `KheDrsExprOrSetClosedValue`, `KheDrsExprOrEvalSignature`, and `KheDrsExprOrDoDebug`. We have seen most of these functions already, in the preceding sections.

With two exceptions, *COUNTER* and *SEQUENCE*, we won't present the implementations of these subtypes, because they are quite straightforward. The two exceptions are much more complicated, and our next task is to study them in detail.

### D.8.9. The *COST* expression type

Type `KHE_DRS_EXPR_COST` is the abstract supertype of the solver's four expression types, `KHE_DRS_EXPR_COUNTER`, `KHE_DRS_EXPR_SUM_INT`, `KHE_DRS_EXPR_SUM_FLOAT`, and `KHE_DRS_EXPR_SEQUENCE`, which could constribute a cost:

```
#define INHERIT_KHE_DRS_EXPR_COST                              \
  INHERIT_KHE_DRS_EXPR                                         \
  KHE_COST_FUNCTION      cost_fn;                              \
  KHE_COST               combined_weight;                      \
  KHE_DRS_MONITOR        monitor;
```

```
typedef struct khe_drs_expr_cost_rec {
  INHERIT_KHE_DRS_EXPR_COST
} *KHE_DRS_EXPR_COST;
```

The type has only a few operations. They include `KheDrsExprCostUnweightedCost`, which returns the cost of the expression, for a given deviation, before multiplication by the weight:

```
int KheDrsExprCostUnweightedCost(KHE_DRS_EXPR_COST ec, int dev)
{
  switch( ec->cost_fn )
  {
    case KHE_STEP_COST_FUNCTION:

      return dev > 0 ? 1 : 0;

    case KHE_LINEAR_COST_FUNCTION:

      return dev;

    case KHE_QUADRATIC_COST_FUNCTION:

      return dev * dev;

    default:

      HnAbort("KheDrsExprCostUnweightedCost internal error");
      return 0;  /* keep compiler happy */
  }
}
```

For the actual cost there is

```
KHE_COST KheDrsExprCostCost(KHE_DRS_EXPR_COST ec, int dev)
{
  return ec->combined_weight * KheDrsExprCostUnweightedCost(ec, dev);
}
```

These two macros allow the child types to access these two functions without a visible upcast, and with a much briefer function name:

```
#define uf(e, d) KheDrsExprCostUnweightedCost((KHE_DRS_EXPR_COST) (e), (d))
#define f(e, d) KheDrsExprCostCost((KHE_DRS_EXPR_COST) (e), (d))
```

There are also two operations which associate a cost expression with a DRS constraint object: `KheDrsExprCostSetConstraint` (Appendix D.7.1) and `KheDrsExprCostConstraint`.

### D.8.10. The *COUNTER* expression type

This section presents the implementation of the *COUNTER* expression type. It is all based on the formulas from Appendix C.5, where these expressions were called *counter monitors*, and we refer freely to that Appendix and its terminology.

The children of a counter expression have value 0 or 1. The counter expression counts the number of children with value 1, compares this with given limits, and finds a cost. Its type is

```
typedef struct khe_drs_expr_counter_rec {
  INHERIT_KHE_DRS_EXPR_COST
  int                          min_limit;
  int                          max_limit;
  bool                         allow_zero;
  int                          history_before;
  int                          history_after;
  int                          history;
  KHE_DRS_ADJUST_TYPE          adjust_type;
  int                          closed_state;
  struct khe_drs_open_children_rec  open_children_by_shift;
} *KHE_DRS_EXPR_COUNTER;
```

A counter expression always represents a monitor *m*, and the first six uninherited fields are directly defined by *m* or its constraint. For the rest, `adjust_type` is an enumerated value saying what type of signature value adjustment to use; `closed_state` holds the number of closed children whose value is 1; and `open_children_by_shift`, which is used only when *m* is an event resource monitor (when the inherited `resource` field is `NULL`), holds the same open children as `open_children_by_day`, only sorted by open shift index rather than open day index.

Before the main *COUNTER* submodule there is a submodule which is concerned with notifying signers of the existence of this expression. We have presented most of that already (function `KheDrsExprCounterNotifyCoverSigners` in Appendix D.8.5).

The main *COUNTER* submodule begins with two functions for handling deviations:

```
int KheDrsExprCounterDelta(KHE_DRS_EXPR_COUNTER ec,
  int lower_det, int upper_det)
{
  if( ec->allow_zero && lower_det == 0 )
    return 0;
  else if( lower_det > ec->max_limit )
    return lower_det - ec->max_limit;
  else if( upper_det < ec->min_limit )
    return ec->min_limit - upper_det;
  else
    return 0;
}
```

This is $\delta(l, u)$ from Appendix C. Next comes

```
int KheDrsExprCounterDev(KHE_DRS_EXPR_COUNTER ec,
  int lower_det, int upper_det_minus_lower_det)
{
  return KheDrsExprCounterDelta(ec, lower_det,
    lower_det + upper_det_minus_lower_det);
}
```

which is a more convenient way to call δ sometimes.

Next we have a function for working out the kind of signature value adjustment that is appropriate for `ec`. Its result is a value of type

```
typedef enum {
  KHE_DRS_ADJUST_ORDINARY,
  KHE_DRS_ADJUST_NO_MAX,
  KHE_DRS_ADJUST_LINEAR,
  KHE_DRS_ADJUST_STEP
} KHE_DRS_ADJUST_TYPE;
```

and the function itself is

```
KHE_DRS_ADJUST_TYPE KheDrsAdjustType(KHE_COST_FUNCTION cost_fn,
  int max_limit)
{
  if( max_limit == INT_MAX )
    return KHE_DRS_ADJUST_NO_MAX;
  else if( cost_fn == KHE_LINEAR_COST_FUNCTION )
    return KHE_DRS_ADJUST_LINEAR;
  else if( cost_fn == KHE_STEP_COST_FUNCTION )
    return KHE_DRS_ADJUST_STEP;
  else
    return KHE_DRS_ADJUST_ORDINARY;
}
```

After that come two functions, for beginning the creation of a new *COUNTER* object, and for beginning the creation of a new *COUNTER* object with maximum limit 0 (a common special case). When a child is added during the initial construction of the expression, we do this:

```
void KheDrsExprCounterAddChild(KHE_DRS_EXPR_COUNTER ec,
  KHE_DRS_EXPR child_e, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  ec->closed_state += child_e->value.i;
}
```

to make `closed_state` hold the number of children with value 1. After all children are added,

```
void KheDrsExprCounterMakeEnd(KHE_DRS_EXPR_COUNTER ec,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  int ub;

  KheDrsExprInitEnd((KHE_DRS_EXPR) ec, drs);

  /* get the total value upper bound */
  ub = HaArrayCount(ec->children) + ec->history + ec->history_after;

  /* set the value upper bound (not actually used, but anyway) */
  ec->value_ub.i = KheDrsExprCounterValueUpperBound(ec, ub);

  /* set constraint */
  KheDrsExprCostSetConstraint((KHE_DRS_EXPR_COST) ec, ec->history, drs);
}
```

is called to end the initialization of `ec`, sort out its upper bound (which is not used, so this is only for completeness), and set its constraint field. When a child is opened we do this:

```
void KheDrsExprCounterChildHasOpened(KHE_DRS_EXPR_COUNTER ec,
  KHE_DRS_EXPR child_e, int child_index, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  int old_dev, new_dev, ub;  KHE_COST old_cost, new_cost;

  /* find the deviation before the change */
  ub = KheDrsOpenChildrenCount(&ec->open_children_by_day);
  old_dev = KheDrsExprCounterDev(ec,
    ec->history + ec->closed_state, ub + ec->history_after);

  /* update ec to reflect the new open child */
  KheDrsOpenChildrenAddChild(&ec->open_children_by_day, child_e);
  ec->closed_state -= child_e->value.i;

  /* find the deviation after the change */
  ub = KheDrsOpenChildrenCount(&ec->open_children_by_day);  /* one more */
  new_dev = KheDrsExprCounterDev(ec,
    ec->history + ec->closed_state, ub + ec->history_after);

  /* report the change in cost, if any, to drs->solve_start_cost */
  if( old_dev != new_dev )
  {
    old_cost = f(ec, old_dev);
    new_cost = f(ec, new_dev);
    drs->solve_start_cost += (new_cost - old_cost);
    KheDrsMonitorUpdateRerunCost(ec->monitor, (KHE_DRS_EXPR) ec, drs,
      NULL, KHE_DRS_OPEN, "open", child_index, "+-", new_cost, old_cost);
  }

  /* update open_children_by_shift, if required */
  if( ec->resource == NULL )
    KheDrsOpenChildrenAddChild(&ec->open_children_by_shift, child_e);
}
```

The old deviation is based on `eisc->history + eisc->closed_state` active children and `KheDrsOpenChildrenCount(&ec->open_children_by_day) + eisc->history_after` unassigned children. The new deviation follows the same formula, after updating to reflect the new open child. Any change in cost is added to `drs->solve_start_cost`. And if required, we add `child_e` to `eisc->open_children_by_shift`.

When a child is closed we do the reverse:

```
void KheDrsExprCounterChildHasClosed(KHE_DRS_EXPR_COUNTER ec,
  KHE_DRS_EXPR child_e, int child_index, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  int old_dev, new_dev, ub;  KHE_COST old_cost, new_cost;

  /* find the deviation before the change */
  ub = KheDrsOpenChildrenCount(&ec->open_children_by_day);
  old_dev = KheDrsExprCounterDev(ec,
    ec->history + ec->closed_state, ub + ec->history_after);

  /* update ec to reflect one less open child */
  KheDrsOpenChildrenDeleteChild(&ec->open_children_by_day, child_e);
  ec->closed_state += child_e->value.i;

  /* find the deviation after the change */
  ub = KheDrsOpenChildrenCount(&ec->open_children_by_day);  /* one less */
  new_dev = KheDrsExprCounterDev(ec,
    ec->history + ec->closed_state, ub + ec->history_after);

  /* report the change in cost, if any, to drs->solve_start_cost */
  if( old_dev != new_dev )
  {
    new_cost = f(ec, new_dev);
    old_cost = f(ec, old_dev);
    drs->solve_start_cost += (new_cost - old_cost);
    KheDrsMonitorUpdateRerunCost(ec->monitor, (KHE_DRS_EXPR) ec, drs,
      NULL, KHE_DRS_CLOSE, "close", child_index, "+-", new_cost, old_cost);
  }

  /* update open_children_by_shift, if required */
  if( ec->resource == NULL )
    KheDrsOpenChildrenDeleteChild(&ec->open_children_by_shift, child_e);
}
```

There is a `KheDrsExprCounterSetClosedValue` function, but it has nothing to do here, because counter expressions do not store a closed value.

Finally comes the function for adding this expression's extra cost and signature value to a new signature:

```
void KheDrsExprCounterEvalSignature(KHE_DRS_EXPR_COUNTER ec,
  KHE_DRS_SIGNER dsg, KHE_DRS_SIGNATURE prev_sig,
  KHE_DRS_SIGNATURE next_sig, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  int ld1, ld2, ud1, ud2, i, i1, i2, dev1, dev2, si, count, next_di;
  KHE_DRS_EXPR child_e;  KHE_DRS_VALUE val;  KHE_DRS_EXPR_EVAL_TYPE eval_type;

  /* get ld1, ud1, and dev1 (for all days before next_di) */
  next_di = KheDrsSignerOpenDayIndex(dsg);
  if( KheDrsOpenChildrenIndexIsFirstOrLess(&ec->open_children_by_day,next_di))
    ld1 = KheDrsExprCounterInitialValue(ec);
  else
    ld1 = KheDrsExprDaySigVal((KHE_DRS_EXPR) ec, next_di - 1, prev_sig).i;
  ud1 = ld1 + ec->history_after +
    KheDrsOpenChildrenAtOrAfter(&ec->open_children_by_day, next_di);
  dev1 = KheDrsExprCounterDelta(ec, ld1, ud1);

  /* get ld2, ud2, and dev2 for one more day, shift, or shift pair */
  ld2 = ld1, ud2 = ud1;
  switch( dsg->type )
  {
    case KHE_DRS_SIGNER_DAY:
    case KHE_DRS_SIGNER_RESOURCE_ON_DAY:

      /* signer is for day or mtask solutions */
      KheDrsOpenChildrenForEach(&ec->open_children_by_day, next_di,child_e,i)
        KheDrsExprCounterUpdateLU(child_e, &ld2, &ud2, i, debug);
      break;

    ... see below for other cases ...
  }
  dev2 = KheDrsExprCounterDelta(ec, ld2, ud2);

  /* if not the last evaluation, store ld2 (adjusted) in next_sig */
  eval_type = KheDrsExprEvalType((KHE_DRS_EXPR) ec, dsg, drs, NULL);
  if( eval_type == KHE_DRS_EXPR_EVAL_NOT_LAST )
  {
    val.i = KheDrsAdjustedSigVal(ld2, ec->adjust_type,
      ec->min_limit, ec->max_limit, ec->history_after);
    KheDrsSignatureAddState(next_sig, val, dsg, (KHE_DRS_EXPR) ec);
  }

  /* report the extra cost, if any */
  if( dev2 != dev1 )
  {
    KheDrsSignatureAddCost(next_sig, f(ec, dev2) - f(ec, dev1));
    KheDrsMonitorUpdateRerunCost(ec->monitor, (KHE_DRS_EXPR) ec, drs,
      dsg, KHE_DRS_SEARCH, "search", -1, "+-", f(ec, dev2), f(ec, dev1));
  }
}
```

The structure here is the same as that of `KheDrsExprOrEvalSignature` that we saw earlier, except that here the value (an extra cost) is added to the signature rather than stored in `ec`. We'll go through it step by step now.

Here `ld1` and `ud1` are lower and upper determinants, the $l$ and $u$ of Appendix C, for the solution up to and including the day before the open day with open day index `next_di`. Now $l$ is the number of active children (children whose value is known to be 1), and $u$ is the number of active or unassigned children (children whose value is not known). We find `ld1` by calling

```
int KheDrsExprCounterInitialValue(KHE_DRS_EXPR_COUNTER ec)
{
  return ec->history + ec->closed_state;
}
```

if there is no previous day, and by retrieving it from `prev_sig` if there is. Then `ud1` is `ld1` plus the children lying on or after the day with open day index `next_di`, or in the history after range.

When the signer is a day or resource on day signer, we can derive `ld2` and `ud2`, the lower and upper determinants for the solution including the open day with open day index `next_di`, by starting from `ld1` and `ud1` and updating them to take account of the children whose values become finalized on the additional day:

```
KheDrsOpenChildrenForEach(&ec->open_children_by_day, next_di, child_e, i)
  KheDrsExprCounterUpdateLU(child_e, &ld2, &ud2, i);
```

where the updating for one child `child_e` is done by a call to

```
void KheDrsExprCounterUpdateLU(KHE_DRS_EXPR child_e,
  int *ld2, int *ud2, int i, bool debug)
{
  *ld2 += child_e->value.i;
  *ud2 += (child_e->value.i - child_e->value_ub.i);
}
```

The child has changed from unassigned to either active or inactive. This function either adds 1 to `*ld2` and leaves `*ud2` unchanged, or it adds 1 to `*ud2` and leaves `*ld2` unchanged. It has been written this way to show that the effect of giving a value to `child_e` is to add its value to `*ld2` and replace its upper bound in `*ud2` (always 1 here) by its value.

With the old and new values of $l$, $u$, and $\delta$ in hand, we are ready to report the results. If there are more unassigned children, we add `ld2`, possibly adjusted, to `next_sig`. Separately, if the cost has changed, we add the extra cost `f(eisc, dev2) - f(eisc, dev1)` to `next_sig`.

It remains to present the two switch cases omitted above. Of all the expression types, `KHE_DSR_EXPR_COUNTER` is the only one whose evaluation depends on the type of the signer, and even then, only when the expression is derived from an event resource constraint. The idea is the same but there is a different set of newly evaluated children to traverse:

```
case KHE_DRS_SIGNER_SHIFT:

  /* signer is for shift solutions */
  si = dsg->u.shift->open_shift_index;
  KheDrsOpenChildrenForEach(&ec->open_children_by_shift, si, child_e, i)
    KheDrsExprCounterUpdateLU(child_e, &ld2, &ud2, i, debug);
  break;

case KHE_DRS_SIGNER_SHIFT_PAIR:

  /* signer is for shift pair solutions */
  for( count = 0;  count < 2;  count++ )
  {
    si = dsg->u.shift_pair->shift[count]->open_shift_index;
    KheDrsOpenChildrenForEach(&ec->open_children_by_shift, si, child_e, i)
      KheDrsExprCounterUpdateLU(child_e, &ld2, &ud2, i, debug);
  }
  break;
```

Instead of traversing all the children on the next day, this traverses all the children affected by a particular shift, or pair of shifts, on that day. No child is affected by two or more shifts, because the children are always *ASSIGNED_TASK* expressions, each representing one task at one time.

### D.8.11. The *SEQUENCE* expression type

This section presents the implementation of the *SEQUENCE* expression type, based on the formulas from Appendix C.7, where these expressions were called *sequence monitors*. We refer freely to that Appendix and its terminology.

***Child order.*** In a *SEQUENCE* expression, the order of the children matters. Although it never happens in practice, the last open days of the open children (taken in order) could be out of chronological order. For other kinds of expressions, where the children's order does not matter, the open children are sorted by function `KheDrsExprChildHasOpened` (Appendix D.8.5) so that their last open days are in chronological order. But doing that to a *SEQUENCE* expression would change its meaning.

Instead of sorting the children, we change their open day ranges. For each open child $y_i$ after the first, if the last open day of $y_i$ precedes the last open day of $y_{i-1}$, then the last open day of $y_i$ is increased to the last open day of $y_{i-1}$. This does not break anything, it merely causes $y_i$ to contribute a value to the signature on more days than it otherwise would have done. It is done as each child is opened, so the last open day of $y_{i+1}$ is affected by any previous adjustment to the last open day of $y_i$, and so on. Thankfully, after doing this we can forget about it.

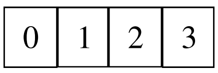
We do not allow a child of a *SEQUENCE* expression to be a leaf (external expression), and this need to change open day ranges is one of the two reasons why. A leaf may be shared with other expressions, and increasing its open day range might well disrupt them. But non-leaf expressions are not shared, so their open day ranges can be increased safely.

The other reason is that the *SEQUENCE* expression type is much easier to implement if it can be assumed that the children of a *SEQUENCE* expression are opened in increasing child index order. This will happen if the postorder indexes of the children are increasing, which is easily ensured if the children are all newly created, simply by visiting the time groups of the monitor in the natural order during construction. But it cannot be guaranteed for shared expressions, since they may be created at arbitrary points during the initialization.

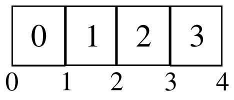
These redundant expressions slow down the evaluation of some constraints slightly, for example constraints on consecutive night shifts. But, importantly, they do not make signatures any longer (except when open day ranges are extended), as a moment's thought will show.

In practice, the number of children with a given last open day is always at most 1. However, to cover all cases we allow any number of children to have the same last open day.

***Sequences.*** In this section, a *sequence* means a sequence of adjacent children of a *SEQUENCE* expression. The implementation makes use of three kinds of sequences: closed sequences (defined below), a-intervals, and au-intervals. We represent a sequence by a pair of indexes $[a, b]$ such that $a \leq b$. Consider this sequence of four children:



We've shown their indexes inside, starting from 0 as is usual in C. But actually, the indexes that define a sequence are indexes into the sequence of gaps that precede and follow the children:

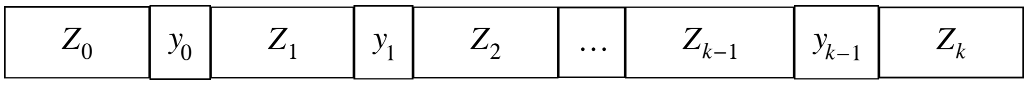


So the pair of indexes $[1, 3]$ for example specifies the children with indexes 1 and 2, because gap 1 precedes child 1 and gap 3 follows child 2. We call the first index the *start index*; as well as being the index of a gap, it also happens to be the index of the first specified child. We call the second index the *stop index*; it is one greater than the index of the last specified child.

These details are important because they make an empty sequence be more than an empty sequence of children; it has a definite location in the enclosing sequence. For example, $[1, 1]$ is the empty sequence starting at index 1. It is different from, say, $[4, 4]$. We do it this way with good reason. For example, there is an operation which extends a sequence $k$ places to the right. Applied to $[a, b]$, the result is $[a, b + k]$. This makes sense even when $[a, b]$ is empty.

***Closed sequences.*** A *closed sequence*, denoted $Z_i$, is the sequence of closed children lying between two open children, or between an open child and one end of the sequence of children. Each *SEQUENCE* object contains a sequence of closed sequences. They summarise the closed children, allowing them to be skipped over quickly while solving.

Consider the sequence $y_0, \ldots, y_{k-1}$ of all open children of $C$. We index them starting from 0 to agree with the C implementation. They appear in $C$'s list of open children in the same order that they appear in $C$'s list of all children, thanks to the work done above on the order that the children are opened. This order is the one used when naming them $y_0, \ldots, y_{k-1}$. Now consider the list of all children. Within this list, assuming $k > 0$, let $Z_0$ be the closed sequence of zero or more closed children preceding $y_0$; for $i$ in the range $0 < i < k$ let $Z_i$ be the closed sequence of zero or more closed children following $y_{i-1}$ and preceding $y_i$; and let $Z_k$ be the closed sequence of zero or more closed children following $y_{k-1}$. The full sequence of all children thus looks like this:



If $k = 0$ the whole sequence is a closed sequence; let $Z_0$ be that sequence. Although we prefer to think of history values as sequences of children, they are not included here, because there is no efficient way here to represent $c_i$, which could be very large.

This way of defining the $Z_i$ can be confusing, because it has little connection with open days. The open day ranges of the open children may be adjusted, as explained above, and the closed children have no open day ranges at all. Instead, the definition relies on the order of the children, which is after all what matters, and on the fact that the open children are not reordered.

Each $Z_i$ is represented in the implementation by an object of type `KHE_DRS_CLOSED_SEQ`:

```
typedef struct {
  int           start_index;
  int           stop_index;
  int           active_at_left;
  int           active_at_right;
} *KHE_DRS_CLOSED_SEQ;
```

Fields `start_index` and `stop_index` are the start index and stop index of the closed sequence. Field `active_at_left` is the number of active children within $Z_i$ adjacent to its left end, and `active_at_right` is the number of active children within $Z_i$ adjacent to its right end. If every child in $Z_i$ is active, `active_at_left` and `active_at_right` are equal to each other and to the length of $Z_i$. This will be the case, for example, when $Z_i$ is empty.

Before a solve, the $y_i$ are opened in increasing order, as we know. Initially only $Z_0$ is present, representing all the children. As each $y_i$ is opened, it is appended to the list of open children, and the last closed sequence, $Z_i$, is split into two, a shortened $Z_i$ and a new $Z_{i+1}$. The reverse procedure is followed as open children are closed at the end of the solve. Splitting a closed sequence into two and merging two closed sequences into one are the only non-trivial operations on this type.

***A-intervals.*** Here is the type representing an a-interval:

```
typedef struct {
  int           start_index;
  int           stop_index;
  bool          unassigned_precedes;
} KHE_DRS_A_INTERVAL;
```

It is a non-pointer type, to avoid memory allocation. In addition to the start index and stop index, it contains `unassigned_precedes`, which is `true` when an unassigned child immediately precedes this interval. This is needed when calculating deviations:

```
int KheDrsAIntervalDev(KHE_DRS_A_INTERVAL ai,
  KHE_DRS_EXPR_INT_SEQ_COST eisc)
{
  int len;
  if( ai.unassigned_precedes && eisc->cost_fn == KHE_STEP_COST_FUNCTION )
    return 0;
  len = ai.stop_index - ai.start_index;
  return len > eisc->max_limit ? len - eisc->max_limit : 0;
}
```

If an unassigned child immediately precedes this interval and the cost function is `Step`, the deviation is 0. Otherwise the deviation is the amount by which the interval's length exceeds $U$. All this follows Appendix C.7 exactly.

There are also straightforward functions for creating a-intervals, finding the a-interval adjacent to a given point, merging two a-intervals, and so on. An example appears below. They optimize by not searching the children directly; instead they assume that the closed sequences are up to date and search those, where most of the work has already been done.

Unlike a closed sequence, an a-interval at the extreme left includes the `eisc->history` active children from history. It does this by setting its start index to `-eisc->history`. Because of this, `KheDrsAIntervalDev` does not need to pay any special attention to history.

An example of this treatment of history occurs in the following function, which finds the (possibly empty) a-interval just to the left of the open child with a given `open_index`:

```
KHE_DRS_A_INTERVAL KheDrsAIntervalFindLeft(
  KHE_DRS_EXPR_INT_SEQ_COST eisc, int open_index)
{
  KHE_DRS_CLOSED_SEQ dcs;
  dcs = HaArray(eisc->closed_seqs, open_index);
  if( !KheDrsClosedSeqAllActive(dcs) )
  {
    /* an inactive child precedes the active_at_right active children */
    return KheDrsAIntervalMake(dcs->stop_index - dcs->active_at_right,
      dcs->stop_index, false);
  }
  else if( open_index > 0 )
  {
    /* an unassigned child precedes the active_at_right active children */
    return KheDrsAIntervalMake(dcs->stop_index - dcs->active_at_right,
      dcs->stop_index, true);
  }
  else
  {
    /* nothing but history precedes the active_at_right active children */
    return KheDrsAIntervalMake(dcs->stop_index - dcs->active_at_right
      - eisc->history, dcs->stop_index, false);
  }
}
```

The stop index of this a-interval is the stop index of the closed sequence just to the left. Its start index is `dcs->active_at_right` places left of there, plus `eisc->history` more places to the left if we are at the start. The function also finds a suitable value for `unassigned_precedes`, the third parameter of `KheDrsAIntervalMake`.

In Appendix C.7, a-intervals were said to be maximal and non-empty. There is nothing about the `KHE_DRS_A_INTERVAL` type which guarantees these conditions. The functions that use a-intervals never create non-maximal ones, but they may create empty ones. This is done to reduce the number of cases. For example, if one of the children of an a-interval becomes unassigned or inactive, the a-interval splits into two pieces, one on each side of the changed child. Either or both could be empty, but by allowing a-intervals to be empty the implementation has just one case to handle. Empty a-intervals have deviation 0, so they cause no problems.

***AU-intervals.*** Here is the type representing an au-interval:

```
typedef struct {
  int           start_index;
  int           stop_index;
  bool          has_active_child;
} KHE_DRS_AU_INTERVAL;
```

Once again it is a non-pointer type. In addition to the start index and stop index, it contains `has_active_child`, which is `true` when the interval contains at least one active child. This is needed when calculating deviations:

```
int KheDrsAUIntervalDev(KHE_DRS_AU_INTERVAL aui,
  KHE_DRS_EXPR_INT_SEQ_COST eisc)
{
  int len;
  if( !aui.has_active_child )
    return 0;
  len = aui.stop_index - aui.start_index;
  return len < eisc->min_limit ? eisc->min_limit - len : 0;
}
```

If the interval contains no active children, the deviation is 0. Otherwise the deviation is the amount by which the interval's length falls short of *L*. As for a-intervals, there are functions for creating, finding, merging, and splitting au-intervals, which assume that closed sequences are up to date and search them rather than the children. Examples of these functions appear below. Once again, the code that creates au-intervals never creates non-maximal ones, and although it does create empty ones, those have deviation 0, because `has_active_child` is necessarily 0.

An au-interval at the extreme left includes the active children from history, by setting its start index to `-eisc->history`. An au-interval at the extreme right includes the unassigned children from history, by increasing its stop index by `eisc->history_after`. When there are no inactive children (unlikely, but possible), both of these adjustments apply to the same au-interval. Because of this, `KheDrsAUIntervalDev` does not need to pay any special attention to history.

Here is an example of an au-interval function. It finds the (possibly empty) au-interval just to the left of the open child with the given `open_index`:

```
KHE_DRS_AU_INTERVAL KheDrsAUIntervalFindLeft(
  KHE_DRS_EXPR_INT_SEQ_COST eisc, int open_index)
{
  KHE_DRS_CLOSED_SEQ dcs;  KHE_DRS_AU_INTERVAL res;  int i;

  /* initialize res to the active children at the right of dcs */
  dcs = HaArray(eisc->closed_seqs, open_index);
  res = KheDrsAUIntervalMake(dcs->stop_index - dcs->active_at_right,
    dcs->stop_index, true);
  if( !KheDrsClosedSeqAllActive(dcs) )
    return res;

  /* now keep looking to the left of there */
  for( i = open_index - 1;  i >= 0;  i-- )
  {
    /* return early if eisc->min_limit reached */
    if( KheDrsAUIntervalLength(res) >= eisc->min_limit )
      return res;

    /* res includes the open unassigned child before the previous dcs */
    KheDrsAUIntervalExtendToLeft(&res, 1, false);

    /* res includes the active children at the right of the next dcs */
    dcs = HaArray(eisc->closed_seqs, i);
    KheDrsAUIntervalExtendToLeft(&res, dcs->active_at_right, true);
    if( !KheDrsClosedSeqAllActive(dcs) )
      return res;
  }

  /* at the start, so res includes history */
  KheDrsAUIntervalExtendToLeft(&res, eisc->history, true);
  return res;
}
```

It starts with the closed sequence object `dcs` immediately to the left of the open child. The `active_at_right` active children at the right of `dcs` are part of the au-interval, but if they are preceded by an inactive child (if `dcs` is not entirely active) it's time to stop. Otherwise the open child preceding `dcs` is included, as are the `active_at_right` active children of the preceding closed sequence, and so on.

The loop in this function could cause it to run for longer than a constant amount of time. However, it returns early if the interval length reaches `eisc->min_limit`. This is safe because the cost at that point is 0, so there is no need to make the interval any longer. It keeps the running time constant, assuming (as is true in practice) that the minimum limit is a small constant.

`KheDrsAUIntervalExtendToLeft` extends an au-interval to the left:

```
void KheDrsAUIntervalExtendToLeft(KHE_DRS_AU_INTERVAL *aui,
  int extra_len, bool has_active_child)
{
  if( extra_len > 0 )
  {
    aui->start_index -= extra_len;
    if( has_active_child )
      aui->has_active_child = true;
  }
}
```

This is done by reducing its start index by `extra_len`, and updating its `has_active_child` if new children are actually added.

***Opening and closing.*** Each of the four changes to the state of a child (inactive or active to unassigned when opening, and unassigned to inactive or active when closing) takes away old intervals (both a-intervals and au-intervals) and adds in new ones. We treat any change to any interval as taking away one interval and adding another. We need to find the old intervals and subtract their costs, and find the new intervals and add their costs.

This is straightforward in principle, although to explain all the code in detail would be tedious. As an example, here is what happens when the child whose index in the sequence of open children is `open_index` is opened and changes its state from inactive to unassigned. First, it is added to the list of open children and its $Z_i$ is split into $Z_i$ and $Z_{i+1}$. Then comes this:

```
/* the au-intervals on each side merge */
aui_left = KheDrsAUIntervalFindLeft(eisc, open_index);
aui_right = KheDrsAUIntervalFindRight(eisc, open_index);
aui_merged = KheDrsAUIntervalMerge(aui_left, aui_right, false);
drs->solve_start_cost += KheDrsAUIntervalCost(aui_merged, eisc)
  - KheDrsAUIntervalCost(aui_left, eisc)
  - KheDrsAUIntervalCost(aui_right, eisc);

/* the a-interval to the right changes its unassigned_precedes */
ai_before = KheDrsAIntervalFindRight(eisc, open_index, false);
ai_after  = KheDrsAIntervalFindRight(eisc, open_index, true);
drs->solve_start_cost += KheDrsAIntervalCost(ai_after, eisc)
  - KheDrsAIntervalCost(ai_before, eisc);
```

The au-intervals on each side of the changed child become merged, so we add in the cost of the new merged interval and subtract away the costs of the two old unmerged intervals (possibly empty). And the a-interval to the right changes its `unassigned_precedes` from `false` to `true`, which could change its cost, so again we add the new and subtract the old.

No au-intervals or a-intervals are preserved in any data structure. As in the example above, they are all calculated on the fly as required.

***Searching.*** Searching is basically function `KheDrsExprSequenceEvalSignature`:

```
void KheDrsExprSequenceEvalSignature(KHE_DRS_EXPR_SEQUENCE es,
  KHE_DRS_SIGNER dsg, KHE_DRS_SIGNATURE prev_sig,
  KHE_DRS_SIGNATURE next_sig, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  int index, i1, i2, active_len, next_di;  KHE_DRS_EXPR child_e;
  KHE_DRS_AU_INTERVAL aui_left, aui_right, aui_merged;
  KHE_DRS_AU_INTERVAL aui_before, aui_after;  KHE_DRS_CLOSED_SEQ dcs;
  KHE_DRS_A_INTERVAL ai_right_before, ai_right_after;
  KHE_DRS_A_INTERVAL ai_left, ai_right, ai_merged;  KHE_DRS_VALUE val;

  /* initialize active_len, depending on first day or not */
  next_di = KheDrsSignerOpenDayIndex(dsg);
  if( KheDrsOpenChildrenIndexIsFirst(&es->open_children_by_day, next_di) )
  {
    dcs = HaArrayFirst(es->closed_seqs);
    active_len = dcs->active_at_right;
    if( KheDrsClosedSeqAllActive(dcs) )  active_len += es->history;
  }
  else
    active_len = KheDrsExprDaySigVal((KHE_DRS_EXPR) es, next_di-1,prev_sig).i;

  /* handle each child_e whose last open day is next_di */
  KheDrsOpenChildrenForEach(&es->open_children_by_day, next_di, child_e,index)
  {
    if( child_e->value.i == 0 )
    {
      /* child_e moves from unassigned to inactive: update cost */
      ... see below ...

      /* set active_len for next iteration (child_e is now inactive) */
      dcs = HaArray(es->closed_seqs, index + 1);
      active_len = dcs->active_at_right;
    }
    else
    {
      /* child_e moves from unassigned to active: update cost */
      ... see below ...

      /* set active_len for next iteration (child_e is now active) */
      dcs = HaArray(es->closed_seqs, index + 1);
      if( KheDrsClosedSeqAllActive(dcs) )
        active_len += 1 + dcs->active_at_right;
      else
        active_len = dcs->active_at_right;
    }
  }

  /* if not last day, store active_len (adjusted) in sig */
  ... see below ...
}
```

It iterates over the open children whose value is being finalized on some day, and over the adjacent closed sequences, and makes the same cost changes as closing a child makes, only adding the changes to `next_sig->cost`, rather than to `drs->solve_start_cost`. We've omitted for the moment the parts that update `next_sig->cost`.

The signature value is the number of active children immediately to the left of the start point of the iteration, called `active_len` in the code. Any unassigned children there were given values earlier in the search path leading to the current solution, so this is the length of both the a-interval and the au-interval immediately to the left. There is no need to search for these intervals.

The main focus of what we've shown here is to initialize `active_len` and keep it up to date as the children are processed. If this is the first day, there is no signature to retrieve `active_len` from. Instead, it is equal to the `active_at_right` field of the (only) closed sequence just to the left of the current day, increased by `es->history` if all the children to the left are active. On other days, `active_len` is stored in the signature and retrieved from there.

The code then visits each open child `child_e` whose last open day is the current day, and examines its value. If it has changed from unassigned to inactive, the cost is updated as explained below, then `active_len` is updated to the correct value for the following child. Because `child_e` is now inactive, that value is the `active_at_right` field of the next closed sequence.

If `child_e` has changed from unassigned to active, the new `active_len` will still be the `active_at_right` value if there is an inactive child within the next closed sequence. But if the next closed sequence consists entirely of active children, `active_len` will have its previous value plus 1 for `child_e` plus the `active_at_right` value.

After the last child has been handled, the remaining `active_len` value has to be stored in the signature of `next_soln` for retrieval on the next day. Here is the code omitted above:

```
/* if not last day, store adjusted active_len in next_sig */
if( !KheDrsOpenChildrenIndexIsLast(&es->open_children_by_day, next_di) )
{
  val.i = KheDrsAdjustedSigVal(active_len,
    es->adjust_type, es->min_limit, es->max_limit, 0);
  KheDrsSignatureAddState(next_sig, val, dsg, (KHE_DRS_EXPR) es);
}
```

As usual an adjusted value is stored.

We turn now to the two other parts of the function that were omitted, that update solution cost. When `child_e` changes from unassigned to inactive, the enclosing au-interval splits, and the a-interval to the right changes its `unassigned_precedes` flag from `false` to `true`:

```
/* child_e moves from unassigned to inactive: update cost */
/* the enclosing au-interval splits */
aui_left = KheDrsAUIntervalMakeLeft(es, index, active_len);
aui_right = KheDrsAUIntervalFindRight(es, index, drs);
aui_merged = KheDrsAUIntervalMerge(aui_left, aui_right, false);
KheDrsSignatureAddCost(next_sig, KheDrsAUIntervalCost(aui_left, es)
  + KheDrsAUIntervalCost(aui_right, es)
  - KheDrsAUIntervalCost(aui_merged, es));

/* the a-interval to the right changes its unassigned_precedes */
ai_right_before = KheDrsAIntervalFindRight(es, index, true);
ai_right_after  = KheDrsAIntervalFindRight(es, index, false);
KheDrsSignatureAddCost(next_sig, KheDrsAIntervalCost(ai_right_after, es)
  - KheDrsAIntervalCost(ai_right_before, es));
```

Function `KheDrsAUIntervalMakeLeft` makes an au-interval ending just before `open_index` with length `active_len`; no searching is required for this.

When `child_e` changes from unassigned to active, the enclosing au-interval is unchanged, but it may gain an active child for the first time, which could change its cost; and the a-intervals on each side merge:

```
/* child_e moves from unassigned to active: update cost */
/* the enclosing au-interval is unchanged, but its cost may change */
aui_left = KheDrsAUIntervalMakeLeft(es, index, active_len);
aui_right = KheDrsAUIntervalFindRight(es, index, drs);
aui_before = KheDrsAUIntervalMerge(aui_left, aui_right, false);
aui_after = KheDrsAUIntervalMerge(aui_left, aui_right, true);
KheDrsSignatureAddCost(next_sig, KheDrsAUIntervalCost(aui_after, es)
  - KheDrsAUIntervalCost(aui_before, es));

/* the a-intervals on each side merge */
ai_left = KheDrsAIntervalMakeLeft(es, index, active_len);
ai_right = KheDrsAIntervalFindRight(es, index, true);
ai_merged = KheDrsAIntervalMerge(ai_left, ai_right);
KheDrsSignatureAddCost(next_sig, KheDrsAIntervalCost(ai_merged, es)
  - KheDrsAIntervalCost(ai_left, es)
  - KheDrsAIntervalCost(ai_right, es));
```

Function `KheDrsAIntervalMakeLeft` makes an a-interval ending just before `open_index` with length `active_len`; no searching is required for this.

This ends our presentation of the `KHE_DRS_EXPR_INT_SEQ_COST` type. Including code for the various kinds of sequences, this type occupies about 1900 lines of the source file.

## D.9. Solutions

A solution is a set of assignments of resources to tasks. Curiously enough, when we come to implement solutions and assignments we find that the two ideas seem to merge: an assignment could represent just itself, but it could also represent the solution created by adding that assignment to some other solution. Experience has shown that it is best to have no assignment objects, strictly speaking, in the implementation, only solution objects.

There are several solution types, representing variants of the idea. It would be wonderful if they formed a neat inheritance hierarchy, with their shared fields in an abstract parent type. Sadly, efficiency demands prevent that. One of the types is not even a pointer type, as we'll see. But we can say that a solution object `S` of any type contains two main kinds of fields.

First, there are fields which define the assignments. Some of them may be pointers to other solution objects. This almost always means that the assignments of those other solutions are included in the assignments of `S`. Some of them may be (resource, task) pairs, meaning that those basic assignments are included in `S`.

Many solutions are created in the context of expanding one day solution. They logically include that solution, but the pointer to it is often omitted, since it is known from the context.

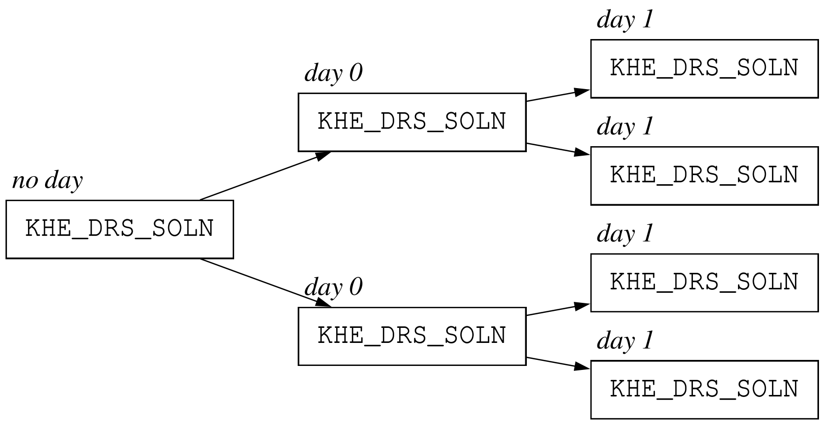
Second, there is one field, of type `KHE_DRS_SIGNATURE` or `KHE_DRS_SIGNATURE_SET`, which holds the signature of `S`, including its cost. It is often convenient to include in the signature only things that differ from the signature of some other solution that `S` is based on. We take care below to define precisely what goes into each signature.

When another solution's assignments are included in `S`, that other solution's signature will be relevant to `S`. But different kinds of solutions have different ways of incorporating the signatures of other solutions into their own signatures. This could be as simple as adding a signature to a signature set, or as complicated as creating a new signature by evaluating expressions.

### D.9.1. Day solutions

The *day solution* is the most important type of solution. It implicitly contains all the time assignments from the initial solution, and all the assignments of closed tasks from the initial solution. It explicitly contains all the assignments that there are going to be of open tasks on days up to and including one particular known open day, called the solution's day, and no asssignments for days after that. We often write '$d_k$-solution' for a day solution whose day is $d_k$.

Day solution objects should probably have type `KHE_DRS_DAY_SOLN`, but at present their type is `KHE_DRS_SOLN`. The nodes of the dynamic programming search tree are day solutions:



The day indexes in this diagram are open day indexes, not frame indexes. The search tree has one level of solutions for each open day, plus the extra level holding the root solution. The root solution is special in that, despite being a day solution, it has no day. There may be closed days, obviously, but they are not visible in the search tree.

Type `KHE_DRS_SOLN` is defined by

```
typedef struct khe_drs_soln_rec *KHE_DRS_SOLN;
typedef HA_ARRAY(KHE_DRS_SOLN) ARRAY_KHE_DRS_SOLN;
typedef HP_TABLE(KHE_DRS_SOLN) TABLE_KHE_DRS_SOLN;

struct khe_drs_soln_rec {
  struct khe_drs_signature_set_rec sig_set;
  KHE_DRS_SOLN                     prev_soln;
  ARRAY_KHE_DRS_TASK_ON_DAY        prev_tasks;
  int                              priqueue_index;
#if TESTING
  int                              sorted_rank;
#endif
};
```

The `sig_set` field is the solution's signature. There may be many thousands of day solution objects, so to save one pointer, the signature has type `struct khe_drs_signature_set_rec` rather than the pointer type `KHE_DRS_SIGNATURE_SET`.

The `prev_soln` field points to this solution's predecessor (its parent in the search tree). The root solution has value `NULL` for this field. No valid `KHE_DRS_SOLN` has value `NULL`.

The `prev_tasks` field really belongs to the incoming edge, but we are saving memory by not having edge objects. In the root solution it is empty, since there is no incoming edge. In other solutions, its length equals the number of open resources, and the `i`th value is the task on day object assigned the `i`th open resource on this solution's day, or `NULL` if that resource is free.

It would arguably be more consistent for these task fields to have type `KHE_DRS_TASK_SOLN` (Appendix D.9.4), the type of a solution containing one assignment of a resource to a task. But objects of that type have short lifetimes, whereas objects of type `KHE_DRS_TASK_ON_DAY` have lifetime equal to the lifetime of the solver. So the use of `KHE_DRS_TASK_ON_DAY` objects can be understood as another memory optimization.

The `priqueue_index` field holds the index of the solution in the priority queue, if there is one. This 'back index' allows the solution to be deleted efficiently from the priority queue when it is found to be dominated by some other solution, and so needs to be deleted and freed. If the solution is not in the priority queue, either because it has been deleted from it or because the priority queue is not in use, `priqueue_index` holds `-1`.

The `sorted_rank` field holds the rank of this solution in the sequence of all undominated solutions for its day, when those solutions are sorted into non-decreasing cost order. It is used only for gathering statistics, which is why it is optional.

The operations on solutions begin with `KheDrsSolnMake`, which makes a new solution object, and `KheDrsSolnFree`, which frees a solution. Then come `KheDrsSolnMarkExpanded` and `KheDrsSolnNotExpanded`, which set and test the special `-1` value of the `priqueue_index` field. Then comes this rather ugly operation to work out which day the solution is for:

```
KHE_DRS_DAY KheDrsSolnDay(KHE_DRS_SOLN soln,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_TASK_ON_DAY dtd;  int i;  KHE_DRS_DAY prev_day;

  /* return NULL if this is the root solution, not on any day */
  if( soln->prev_soln == NULL )
    return NULL;

  /* if prev_tasks has a non-NULL task on day, return its day */
  HaArrayForEach(soln->prev_tasks, dtd, i)
    if( dtd != NULL )
      return dtd->day;

  /* else have to recurse back */
  prev_day = KheDrsSolnDay(soln->prev_soln, drs);
  if( prev_day == NULL )
    return HaArray(drs->open_days, 0);
  else
    return HaArray(drs->open_days, prev_day->open_day_index + 1);
}
```

The day comes from any non-`NULL` task, or else from the parent solution. Previously, solutions stored their day as an attribute, but the author removed it to save memory.

After `KheDrsSolnDay` there are functions used when hashing a solution's signature, which just delegate their work to the `sig_set` attribute: `KheDrsSolnSignatureSetFullHash` and so on. For dominance testing there are two functions:

```
bool KheDrsSolnDoDominates(KHE_DRS_SOLN soln1, KHE_DRS_SOLN soln2,
  KHE_DRS_SIGNER_SET signer_set, KHE_COST trie_extra_cost,
  int trie_start_depth, int *dom_test_count,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  *dom_test_count += 1;
  return KheDrsSignerSetDominates(signer_set, &soln1->sig_set,
    &soln2->sig_set, trie_extra_cost, trie_start_depth,
    soln1->prev_soln == soln2->prev_soln, drs);
}

bool KheDrsSolnDominates(KHE_DRS_SOLN soln1, KHE_DRS_SOLN soln2,
  KHE_DRS_SIGNER_SET signer_set, int *dom_test_count,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  return KheDrsSolnDoDominates(soln1, soln2, signer_set, 0, 0,
    dom_test_count, drs);
}
```

We've omitted some debugging and testing code. `KheDrsSolnDoDominates` is called directly only when solutions are stored in a trie data structure; it avoids visiting parts of the signatures that the trie has already handled. For the most part, `KheDrsSolnDominates` is called.

Next comes a function for overwriting one solution by another, used when solutions are held in hash tables, and then this little helper function:

```
bool KheDrsSolnResourceIsAssigned(KHE_DRS_SOLN soln,
  KHE_DRS_RESOURCE dr, KHE_DRS_TASK_ON_DAY *dtd)
{
  if( soln->prev_soln == NULL )
  {
    /* this is the root solution, so there can be no assignment */
    return *dtd = NULL, false;
  }
  else
  {
    /* non-root solution, get assignment from soln->prev_tasks */
    *dtd = HaArray(soln->prev_tasks, dr->open_resource_index);
    return *dtd != NULL;
  }
}
```

If `dr` is assigned a task in `soln`, this sets `*dtd` to the appropriate task on day object and returns `true`. Otherwise (if `soln` is the root solution or `dr` is not assigned in `soln`), it returns `false`. This function is called by `KheDrsResourceOnDayIsFixed` (Appendix D.10.3).

Following this come some debug functions, including one that prints a neat table showing the timetable of a given resource in a given solution.

### D.9.2. Mtask solutions

An *mtask solution* is an object representing a day solution *S* plus the assignment on the following day of a resource *r* to an unspecified task from a given mtask *c*. The mtask may be `NULL`, meaning that the resource has a free day. We often write '$c_i$-solution' for an mtask solution whose mtask is $c_i$. Here is the type:

```
typedef struct Khe_drs_mtask_soln_rec *KHE_DRS_MTASK_SOLN;
typedef HA_ARRAY(KHE_DRS_MTASK_SOLN) ARRAY_KHE_DRS_MTASK_SOLN;

struct Khe_drs_mtask_soln_rec {
  KHE_DRS_SIGNATURE                   sig;
  KHE_DRS_RESOURCE_ON_DAY             resource_on_day;
  KHE_DRS_MTASK                       mtask;
  KHE_DRS_TASK_ON_DAY                 fixed_task_on_day;
  ARRAY_KHE_DRS_MTASK_SOLN            skip_assts;
  int                                 skip_count;
};
```

The day solution *S* is known from the context and is not stored explicitly.

The `sig` field contains a signature holding the states of the resource monitors of *r* after the assignment. Its `cost` field holds the extra cost of those monitors, beyond their cost in *S*. It is this field that makes this object best interpreted as a solution, rather than as an assignment.

Even though the exact task to which the resource is assigned is not specified, the signature is fully specified. This is because the tasks of one mtask have the same busy times and the same workloads, and so they have the same effect on resource monitors.

The `resource_on_day` field, which is always non-`NULL`, holds the resource on day object representing *r* on the day of the assignment—the day following *S*'s day. The `mtask` and `fixed_task_on_day` fields determine what the resource on day is assigned to, as follows.

If `mtask != NULL`, the assignment may be to any task of that mtask. Someone will have to decide which of `mtask`'s tasks to use before the assignment can actually be made. In this case `fixed_task_on_day` is not used. Its value will be `NULL`.

Otherwise, `mtask == NULL`. The decision about which task to use has already been made, and `fixed_task_on_day` holds that decision. It could be `NULL`, in which case the decision is to assign a free day. Otherwise its day is the day of `resource_on_day`.

The last two fields support the implementation of mtask pair dominance. As explained in Appendix C.8.3, this involves storing a list of mtask solutions in each mtask solution object, and incrementing a counter in those mtask solutions when this one is used. The `skip_assts` field holds the mtask solutions, and the `skip_count` field holds the counter.

The `KHE_DRS_MTASK_SOLN` submodule holds a few simple operations on mtask solution objects, including `KheDrsMTaskSolnMake` for creating them, and `KheDrsMTaskSolnFree` for freeing them. After that there is another submodule holding the code for dominance testing between mtask solutions, which implements the method given in the theory appendix:

```
bool KheDrsMTaskSolnDominates(KHE_DRS_MTASK_SOLN dms_r_c1,
  KHE_DRS_MTASK_SOLN dms_r_c2, KHE_DYNAMIC_RESOURCE_SOLVER drs,
  int verbosity, int indent, FILE *fp)
{
  KHE_COST avail_cost;  int m;  bool res;
  KHE_DRS_RESOURCE dr;  KHE_DRS_MTASK dmt_r_c1, dmt_r_c2;
  dr = dms_r_c1->resource_on_day->encl_dr;
  m = KheDrsResourceSetCount(drs->open_resources);
  avail_cost = 0;
  dmt_r_c1 = dms_r_c1->mtask;
  dmt_r_c2 = dms_r_c2->mtask;
  res = KheDrsMTaskOneExtraAvailable(dmt_r_c1, m) &&
    KheDrsMTaskMinCost(dmt_r_c1, KHE_DRS_ASST_OP_UNASSIGN, dr,
      NULL, m, &avail_cost, verbosity, indent, fp) &&
    KheDrsMTaskMinCost(dmt_r_c2, KHE_DRS_ASST_OP_ASSIGN, dr,
      NULL, m, &avail_cost, verbosity, indent, fp) &&
    KheDrsSignerDominates(dms_r_c1->resource_on_day->signer,
      KheDrsMTaskSolnSignature(dms_r_c1),
      KheDrsMTaskSolnSignature(dms_r_c2),
      &avail_cost, verbosity, indent, fp);
  return res;
}
```

We've omitted some debugging code here. There is also a function (arguably out of place) for finding all pairs of mtask solutions that could be tested for dominance, making the tests, and removing any dominated mtask solutions:

```
void KheDrsMTaskSolnDominanceInit(KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  int ri, i, j;  KHE_DRS_RESOURCE dr;
  KHE_DRS_MTASK_SOLN dms_r_c1, dms_r_c2;
  KheDrsResourceSetForEach(drs->open_resources, dr, ri)
    HaArrayForEach(dr->expand_mtask_solns, dms_r_c1, i)
      for( j = i + 1;  j < HaArrayCount(dr->expand_mtask_solns);  j++ )
      {
        /* for each distinct pair of assignments (dms_r_c1, dms_r_c2) */
        dms_r_c2 = HaArray(dr->expand_mtask_solns, j);
        if( KheDrsMTaskSolnDominates(dms_r_c1, dms_r_c2, drs) )
        {
          /* dms_r_c1 dominates dms_r_c2, so delete dms_r_c2 */
          HaArrayDeleteAndShift(dr->expand_mtask_solns, j);
          if( dr->expand_free_mtask_soln == dms_r_c2 )
            dr->expand_free_mtask_soln = NULL;
          KheDrsMTaskSolnFree(dms_r_c2, drs);
          j--;  /* and try the next dms_r_c2 */
        }
        else if( KheDrsMTaskSolnDominates(dms_r_c2, dms_r_c1, drs) )
        {
          /* dms_r_c2 dominates dms_r_c1, so delete dms_r_c1 */
          HaArrayDeleteAndShift(dr->expand_mtask_solns, i);
          if( dr->expand_free_mtask_soln == dms_r_c1 )
            dr->expand_free_mtask_soln = NULL;
          KheDrsMTaskSolnFree(dms_r_c1, drs);
          i--;
          break;  /* and try the next dms_r_c1 */
        }
      }
}
```

Again we've omitted some debugging code. By the time this function is called, all mtask solution objects for a given resource `dr` on the current day are stored in array `dr->expand_mtask_solns`. This function finds all unordered pairs of those, tests each pair both ways for dominance, and deletes any dominated ones. Care is needed to continue iterating correctly when a mtask solution is deleted.

Finally comes another submodule, holding the code for the part of the expansion operation which is concerned with mtask solutions. This code is presented in Appendix D.10.

### D.9.3. Mtask pair solutions

An *mtask pair solution* is like an mtask solution except that it adds two assignments of resources to mtasks on the day after *S*, rather than one. We might use the notation '$c_i c_j$-solution' for an mtask pair solution involving mtasks $c_i$ and $c_j$.

There is no `KHE_DRS_MTASK_PAIR_SOLN` object type. Instead, two mtask solutions are passed side by side that together make up one mtask pair solution.

Here is the code for deciding whether mtask pair solution `{dcs_r1_c1, dcs_r2_c2}` dominates mtask pair solution `{dcs_r1_c2, dcs_r2_c1}`. The variable names indicate which resource is involved (`r1` or `r2`) and which mtask (`c1` or `c2`):

```
bool KheDrsMTaskPairSolnDominates(KHE_DRS_MTASK_SOLN dms_r1_c1,
  KHE_DRS_MTASK_SOLN dms_r2_c2, KHE_DRS_MTASK_SOLN dms_r1_c2,
  KHE_DRS_MTASK_SOLN dms_r2_c1, KHE_DRS_RESOURCE dr1,
  KHE_DRS_RESOURCE dr2, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_COST avail_cost;  int m;  bool res;
  KHE_DRS_MTASK dmt_r1_c1, dmt_r2_c2;

  m = KheDrsResourceSetCount(drs->open_resources);
  avail_cost = 0;
  dmt_r1_c1 = dms_r1_c1->mtask;
  dmt_r2_c2 = dms_r2_c2->mtask;
  res =
    KheDrsMTaskMinCost(dmt_r1_c1, KHE_DRS_ASST_OP_REPLACE,
      dr1, dr2, m, &avail_cost) &&
    KheDrsMTaskMinCost(dmt_r2_c2, KHE_DRS_ASST_OP_REPLACE,
      dr2, dr1, m, &avail_cost) &&
    KheDrsSignerDominates(dms_r1_c1->resource_on_day->signer,
      KheDrsMTaskSolnSignature(dms_r1_c1),
      KheDrsMTaskSolnSignature(dms_r1_c2), &avail_cost) &&
    KheDrsSignerDominates(dms_r2_c1->resource_on_day->signer,
      KheDrsMTaskSolnSignature(dms_r2_c2),
      KheDrsMTaskSolnSignature(dms_r2_c1), &avail_cost);
  return res;
}
```

Some debug code has been omitted. The algorithm is the one presented in the theory appendix.

To help with testing all pairs of mtask pair solutions for dominance, we need this function, which determines whether resource `dr` contains a mtask solution object corresponding to `dms`:

```
bool KheDrsResourceHasMTaskSoln(KHE_DRS_RESOURCE dr,
  KHE_DRS_MTASK_SOLN dms, KHE_DRS_MTASK_SOLN *res)
{
  KHE_DRS_MTASK_SOLN dms2;  int i;

  if( dms->mtask != NULL )
  {
    /* Case 1: mtask != NULL */
    HaArrayForEach(dr->expand_mtask_solns, dms2, i)
      if( dms2->mtask == dms->mtask )
        return *res = dms2, true;
    return *res = NULL, false;
  }
  else if( dms->fixed_task_on_day != NULL )
  {
    /* Case 2: mtask == NULL && fixed_task_on_day != NULL */
    return *res = NULL, false;
  }
  else
  {
    /* Case 3: mtask == NULL && fixed_task_on_day == NULL */
    HaArrayForEach(dr->expand_mtask_solns, dms2, i)
      if( dms2->mtask == NULL && dms2->fixed_task_on_day == NULL )
        return *res = dms2, true;
    return *res = NULL, false;
  }
}
```

If `dcs` has a non-`NULL` `mtask` field, we search `dr`’s mtask solutions for one for the same mtask. Else, we do a similar search if the assignment is for a free day; else we give up.

Now we are ready to find all pairs of mtask pair solutions and record cases of dominance in the `skip_assts` arrays of the dominated mtask pair solutions:

```
void KheDrsMTaskPairSolnDominanceInit(KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  int i1, i2, i, j, verbosity, indent;  KHE_DRS_RESOURCE dr1, dr2;  FILE *fp;
  KHE_DRS_MTASK_SOLN dms_r1_c1, dms_r1_c2, dms_r2_c1, dms_r2_c2;
  KheDrsResourceSetForEach(drs->open_resources, dr1, i1)
    if( HaArrayCount(dr1->expand_mtask_solns) >= 2 )
      for( i2 = i1 + 1; i2 < KheDrsResourceSetCount(drs->open_resources); i2++ )
      {
        dr2 = KheDrsResourceSetResource(drs->open_resources, i2);
        if( HaArrayCount(dr2->expand_mtask_solns) >= 2 )
          HaArrayForEach(dr1->expand_mtask_solns, dms_r1_c1, i)
          {
            if( KheDrsResourceHasMTaskSoln(dr2, dms_r1_c1, &dms_r2_c1) )
            {
              for( j = i + 1;  j < HaArrayCount(dr1->expand_mtask_solns);  j++ )
              {
                dms_r1_c2 = HaArray(dr1->expand_mtask_solns, j);
                if( KheDrsResourceHasMTaskSoln(dr2, dms_r1_c2, &dms_r2_c2) )
                {
                  if( KheDrsMTaskPairSolnDominates(dms_r1_c1, dms_r2_c2,
                      dms_r1_c2, dms_r2_c1, dr1, dr2, drs,verbosity,indent,fp) )
                  {
                    /* S + dms_r1_c2 + dms_r2_c1 is dominated */
                    HaArrayAddLast(dms_r1_c2->skip_assts, dms_r2_c1);
                    HaArrayAddLast(dms_r2_c1->skip_assts, dms_r1_c2);
                  }
                  else if( KheDrsMTaskPairSolnDominates(dms_r1_c2, dms_r2_c1,
                      dms_r1_c1, dms_r2_c2, dr1, dr2, drs,verbosity,indent,fp) )
                  {
                    /* S + dms_r1_c1 + dms_r2_c2 is dominated */
                    HaArrayAddLast(dms_r1_c1->skip_assts, dms_r2_c2);
                    HaArrayAddLast(dms_r2_c2->skip_assts, dms_r1_c1);
                  }
                }
              }
            }
          }
      }
}
```

The two outer loops iterate over all unordered pairs of distinct open resources `{dr1, dr2}` such that both resources have two or more mtask solutions. The two inner loops iterate over all unordered pairs of distinct mtask solutions for `dr1`, called `dcs_r1_c1` and `dcs_r1_c2`, for which there are corresponding mtask solutions for `dr2`, called `dcs_r2_c1` and `dcs_r2_c2`. These four mtask solutions make two mtask pair solutions, which are then tested for dominance both ways. Dominated solutions are marked by adding entries to their `skip_assts` arrays.

This algorithm could mark a given mtask pair solution as dominated more than once. This does not affect its correctness, so it has seemed simplest not to prevent it.

### D.9.4. Task solutions

A *task solution* represents one day solution *S* plus one assignment of a resource *r* to a specific task *t* on the day after *S*'s day. We might write '*t*-solution' for a task solution whose task is *t*.

Type `KHE_DRS_TASK_SOLN` is defined by

```
typedef struct khe_drs_task_soln_rec KHE_DRS_TASK_SOLN;
typedef HA_ARRAY(KHE_DRS_TASK_SOLN) ARRAY_KHE_DRS_TASK_SOLN;

struct khe_drs_task_soln_rec {
  KHE_DRS_MTASK_SOLN            mtask_soln;
  KHE_DRS_TASK_ON_DAY           fixed_dtd;
};
```

As shown, this is implemented by taking an mtask solution `mtask_soln`, which assigns *r* to an unspecified task of some mtask, and adding `fixed_dtd` to it, which specifies the task within the mtask. Here `fixed_dtd` may be `NULL`, meaning that the assignment is to a free day, but `mtask_soln` is never `NULL`.

Task solution objects come and go quickly during expansion, so it has seemed best to implement them as record types, to avoid allocating and deallocating objects:

```
KHE_DRS_TASK_SOLN KheDrsTaskSolnMake(KHE_DRS_MTASK_SOLN dms,
  KHE_DRS_TASK_ON_DAY fixed_dtd)
{
  KHE_DRS_TASK_SOLN res;
  res.mtask_soln = dms;
  res.fixed_dtd = fixed_dtd;
  return res;
}
```

After this come a few simple functions for accessing the attributes of a task solution, such as

```
KHE_DRS_RESOURCE KheDrsTaskSolnResource(KHE_DRS_TASK_SOLN dts)
{
  return dts.mtask_soln->resource_on_day->encl_dr;
}
```

The last two functions are for requesting the task on day of a given task solution to update the affected external expressions to reflect the assignment expressed by the task solution:

```
void KheDrsTaskSolnLeafSet(KHE_DRS_TASK_SOLN dts, bool whole_task)
{
  KHE_DRS_RESOURCE dr;  KHE_DRS_TASK dt;  int i;
  KHE_DRS_TASK_ON_DAY dtd;
  if( dts.fixed_dtd != NULL )
  {
    dr = KheDrsTaskSolnResource(dts);
    if( whole_task )
    {
      dt = dts.fixed_dtd->encl_dt;
      HaArrayForEach(dt->days, dtd, i)
        KheDrsTaskOnDayLeafSet(dtd, dr);
    }
    else
      KheDrsTaskOnDayLeafSet(dts.fixed_dtd, dr);
  }
}
```

If the assignment is for a free day, there is nothing to do. Otherwise this code offers the choice of using `dts` as a template for assigning the task on all of its days (this will be significant if it is a multi-day task), or just assigning it on `dts`'s day. Either way, `KheDrsTaskOnDayLeafSet` is called to inform the expressions affected by `dtd` that `dr` is being assigned to it. Then

```
void KheDrsTaskSolnLeafClear(KHE_DRS_TASK_SOLN dts, bool whole_task)
{
  KHE_DRS_TASK dt;  int i;  KHE_DRS_TASK_ON_DAY dtd;
  if( dts.fixed_dtd != NULL )
  {
    if( whole_task )
    {
      dt = dts.fixed_dtd->encl_dt;
      HaArrayForEach(dt->days, dtd, i)
        KheDrsTaskOnDayLeafClear(dtd);
    }
    else
      KheDrsTaskOnDayLeafClear(dts.fixed_dtd);
  }
}
```

may be called to undo the effect of `KheDrsTaskSolnLeafSet`.

After these functions there is a submodule holding the task solution functions related to expansion, which will be documented later.

### D.9.5. Task solution sets

A *task solution set* is a set of task solutions. It has type `KHE_DRS_TASK_SOLN_SET`:

```
typedef struct khe_drs_task_soln_set *KHE_DRS_TASK_SOLN_SET;
typedef HA_ARRAY(KHE_DRS_TASK_SOLN_SET) ARRAY_KHE_DRS_TASK_SOLN_SET;

struct khe_drs_task_soln_set {
  ARRAY_KHE_DRS_TASK_SOLN       task_solns;
};
```

Its operations include the straightforward `KheDrsTaskSolnSetMake`, `KheDrsTaskSolnSetFree`, `KheDrsTaskSolnSetClear`, `KheDrsTaskSolnSetCount`, `KheDrsTaskSolnSetAddLast`, and

```
void KheDrsTaskSolnSetLeafSet(KHE_DRS_TASK_SOLN_SET dtss,
  bool whole_task)
{
  KHE_DRS_TASK_SOLN dts;  int i;
  KheDrsTaskSolnSetForEach(dtss, dts, i)
    KheDrsTaskSolnLeafSet(dts, whole_task);
}
```

with its corresponding

```
void KheDrsTaskSolnSetLeafClear(KHE_DRS_TASK_SOLN_SET dtss,
  bool whole_task)
{
  KHE_DRS_TASK_SOLN dts;  int i;
  KheDrsTaskSolnSetForEach(dtss, dts, i)
    KheDrsTaskSolnLeafClear(dts, whole_task);
}
```

These last two functions assign and unassign a whole set of task solutions at once.

### D.9.6. Shift solutions

A *shift solution* is a solution whose assignments consist of the assignments of a day solution $S$, plus one assignment for each resource in a subset $R$ of the open resources to an open task of a shift $s$, whose tasks begin on the day following $S$'s day. These assigments are the only ones that will be made to the open tasks of $s$; all others will remain unassigned. We may write '$s_i$-solution' for a shift solution whose shift is $s_i$.

A shift solution is represented by type `KHE_DRS_SHIFT_SOLN`:

```
typedef struct khe_drs_shift_soln_rec *KHE_DRS_SHIFT_SOLN;
typedef HA_ARRAY(KHE_DRS_SHIFT_SOLN) ARRAY_KHE_DRS_SHIFT_SOLN;

struct khe_drs_shift_soln_rec {
  KHE_DRS_SIGNATURE                    sig;
  KHE_DRS_TASK_SOLN_SET                task_solns;
  ARRAY_KHE_DRS_SHIFT_SOLN             skip_assts;
  int                                  skip_count;
};
```

*S* and *s* are known from the context, so this just stores one `KHE_DRS_TASK_SOLN_SET` object, representing the assignment of one resource to one task for each member of *R*.

The `sig` field holds a signature, used for dominance testing between shift solution objects with the same *S*, *s*, and *R*. It is created by evaluating each expression derived from an event resource monitor which is affected by at least one task from *s*. These are the only expressions relevant to dominance testing between solutions for the same *S*, *s*, and *R*: event resource monitors not affected by the tasks of *s* are clearly irrelevant, and resource monitors for each resource *r* in *R* have the same values whichever task *r* is assigned to, because the tasks of *s* have the same busy times and workloads, by definition.

The `skip_assts` and `skip_count` fields hold the results of shift pair dominance testing (Appendix D.9.8). They work just like the corresponding fields in mtask solution objects, to ensure that pairs of shift solutions known to be uncompetitive are never used together.

The operations on shift solutions include `KheDrsShiftSolnMake`, for making a new shift solution object, and `KheDrsShiftSolnFree`, for freeing it. There is also a function for dominance testing between shift solution objects, assuming the signatures are set:

```
bool KheDrsShiftSolnDominates(KHE_DRS_SHIFT_SOLN dss1,
  KHE_DRS_SHIFT_SOLN dss2, KHE_DRS_SIGNER dsg,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_COST avail_cost;
  avail_cost = 0;
  return KheDrsSignerDominates(dsg, dss1->sig, dss2->sig, &avail_cost);
}
```

After this comes a submodule which implements the part of expansion that is concerned with shift solutions. For that, see Appendix D.10.

## D.9.7. Shift solution tries

For many objects the solver offers a type representing a set of those objects. For example, `KHE_DRS_TASK_SOLN_SET` represents a set of `KHE_DRS_TASK_SOLN` objects.

Type `KHE_DRS_SHIFT_SOLN_TRIE` represents a set of `KHE_DRS_SHIFT_SOLN` objects. For efficient retrieval, a trie data structure is used rather than the usual array. As we saw above, the main attributes that define a shift solution are a day solution *S*, a shift *s* for the day following *S*'s day, and a set of open resources *R*. There is one trie for each combination of *S* and *s*, organized so that *R* can be used as an index to efficiently retrieve the shift solutions for *S*, *s*, and any given *R*. This trie is stored in *s*'s `soln_trie` field during the expansion of *S*.

The indexing is straightforward. Given a set of open resources *R*, sorted by increasing open resource index, the open resource index of the first resource is used to index into the root of the trie, producing a child trie which is indexed using the open resource index of the second resource of *R*, and so on. When all resources have been used up, the node we are at contains a simple array of shift solutions (in fact, all undominated ones, as we will see) for the given *S*, *s*, and *R*. Note that *R* may be empty, in which case retrieval ends at the root of the trie.

Here is the type declaration for `KHE_DRS_SHIFT_SOLN_TRIE`:

```
typedef struct khe_drs_shift_soln_trie_rec *KHE_DRS_SHIFT_SOLN_TRIE;
typedef HA_ARRAY(KHE_DRS_SHIFT_SOLN_TRIE) ARRAY_KHE_DRS_SHIFT_SOLN_TRIE;

struct khe_drs_shift_soln_trie_rec {
  ARRAY_KHE_DRS_SHIFT_SOLN             shift_solns;
  ARRAY_KHE_DRS_SHIFT_SOLN_TRIE        children;
};
```

At any level, we may come to the end of $R$; then the `shift_solns` field holds the undominated shift solutions for $S$, $s$, and $R$. Or if we have not exhausted $R$, the open resource index of the next resource is used to index into the `children` field to take the search to the next level down. `NULL` is a legal shift solution trie and represents a trie containing no shift solution objects.

Some open resources are forced, for various reasons, to be assigned to particular known tasks. We call them *fixed resources*, and we call the remaining open resources *free resources*. If a resource is fixed to a task not in $s$, then it may not appear in an $R$ associated with $s$. If it is fixed to a task in $s$, then it must appear in every $R$ associated with $s$. We ensure this by storing fixed resources separately; that is, we store $R$ in the form $R = R_{fixed} \cup R_{free}$. $R_{fixed}$ is represented by a set of assignments (task solutions). The indexing uses $R_{free}$ only.

An arguably more natural data structure would be a binary tree in which the left subtree of the root handles all subsets $R$ that do not contain the first open resource, while the right subtree of the root handles all subsets $R$ that do contain the first open resource, and so on recursively. We prefer the trie because searching the binary tree takes time proportional to the number of open resources, whereas searching the trie takes time proportional to the cardinality of $R$.

Operations on shift solution tries include `KheDrsShiftSolnTrieMake` for making a new trie node, and `KheDrsShiftSolnTrieFree` for freeing a trie node along with its shift solutions and proper descendants:

```
void KheDrsShiftSolnTrieFree(KHE_DRS_SHIFT_SOLN_TRIE dsst,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_SHIFT_SOLN dss;  int i;  KHE_DRS_SHIFT_SOLN_TRIE child_dsst;

  if( dsst != NULL )
  {
    /* free the shift solution objects */
    HaArrayForEach(dsst->shift_solns, dss, i)
      KheDrsShiftSolnFree(dss, drs);

    /* free the proper descendant trie objects */
    HaArrayForEach(dsst->children, child_dsst, i)
      KheDrsShiftSolnTrieFree(child_dsst, drs);

    /* free dsst itself */
    HaArrayAddLast(drs->shift_soln_trie_free_list, dsst);
  }
}
```

For dominance testing there is

```
bool KheDrsShiftSolnTrieDominates(KHE_DRS_SHIFT_SOLN_TRIE dsst,
  KHE_DRS_SHIFT_SOLN dss, KHE_DRS_SIGNER dsg,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_SHIFT_SOLN other_dss;  int i;
  HaArrayForEach(dsst->shift_solns, other_dss, i)
    if( KheDrsShiftSolnDominates(other_dss, dss, dsg, drs) )
      return true;
  return false;
}
```

which returns `true` if any of the shift solutions within node `dsst` dominates new shift solution `dss`. For `KheDrsShiftSolnDominates`, see Appendix D.9.6. The dominance test does not recurse, because dominance testing is only between shift solutions for the same *S*, *s*, and *R*, and these are all held in one node of the trie. There are similar functions for removing dominated shift solutions from `dsst` and adding a new shift solution to it, and these combine to make

```
void KheDrsShiftSolnTrieMeldShiftSoln(KHE_DRS_SHIFT_SOLN_TRIE dsst,
  KHE_DRS_SHIFT_SOLN dss, KHE_DRS_SIGNER dsg,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  if( KheDrsShiftSolnTrieDominates(dsst, dss, dsg, drs) )
  {
    /* dss is dominated, so free dss */
    KheDrsShiftSolnFree(dss, drs);
  }
  else
  {
    /* remove other solns that dss dominates, then add dss to dsst */
    KheDrsShiftSolnTrieRemoveDominated(dsst, dss, dsg, drs);
    KheDrsShiftSolnTrieAddShiftSoln(dsst, dss);
  }
}
```

which follows the usual algorithm: if any of the existing shift solutions dominates the new one, free the new one, otherwise remove and free dominated ones and add the new one.

Each trie node will probably contain at most one solution. This is because the signatures of shift solutions are concerned only with the event resource constraints of the tasks of the shift. These constraints can in principle also be affected by tasks outside the shift as well, in which case they will need to add a value to the signature's states array. But in practice they are not, so all they contribute to the signature is a cost, and so dominance testing is just cost comparison, and every dominance test has a winner. But our code supports arbitrary XESTT constraints, so it allows for any number of shift solutions in each trie node. Nevertheless a major part of the motivation for expansion by shifts is this expectation of at most one shift solution per trie node.

Given a shift solution trie it is straightforward to index into it. The main challenge is to build it in the first place, and the remainder of this section is devoted to that challenge. Here is the function that does it:

```
void KheDrsShiftBuildShiftSolnTrie(KHE_DRS_SHIFT ds,
  KHE_DRS_SOLN prev_soln, KHE_DRS_DAY prev_day, KHE_DRS_DAY next_day,
  KHE_DRS_RESOURCE_SET all_free_resources,
  KHE_DRS_TASK_SOLN_SET all_fixed_assts,
  KHE_DRS_EXPANDER de, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_RESOURCE_SET included_free_resources;
  HnAssert(ds->soln_trie == NULL,
    "KheDrsShiftBuildShiftSolnTrie internal error");
  included_free_resources = KheDrsResourceSetMake(drs);
  ds->soln_trie = KheDrsShiftSolnTrieBuild(ds, prev_soln, prev_day,
    next_day, all_free_resources, 0, included_free_resources,
    all_fixed_assts, de);
  KheDrsResourceSetFree(included_free_resources, drs);
}
```

Here `ds` is *s*, `prev_soln` is *S*, `prev_day` is *S*'s day, `next_day` is *s*'s day (the day after `prev_day`), `all_free_resources` contains all free open resources, and `all_fixed_assts` contains all fixed open resources, in the form of assignments to the tasks they are fixed to (task solutions). `KheDrsShiftBuildShiftSolnTrie` creates and frees `included_free_resources`, which will hold the resource sets $R_{free}$ as the operation proceeds, and calls

```
KHE_DRS_SHIFT_SOLN_TRIE KheDrsShiftSolnTrieBuild(KHE_DRS_SHIFT ds,
  KHE_DRS_SOLN prev_soln, KHE_DRS_DAY prev_day, KHE_DRS_DAY next_day,
  KHE_DRS_RESOURCE_SET all_free_resources, int all_free_resources_index,
  KHE_DRS_RESOURCE_SET included_free_resources,
  KHE_DRS_TASK_SOLN_SET all_fixed_assts, KHE_DRS_EXPANDER de)
{
  KHE_DRS_SHIFT_SOLN_TRIE res, child_dsst;  bool no_non_null_children;
  int i, count, included_free_resource_count;  KHE_DRS_RESOURCE dr;

  /* return NULL immediately if too many included free resources */
  included_free_resource_count =
    KheDrsResourceSetCount(included_free_resources);
  if( included_free_resource_count >
      ds->expand_max_included_free_resource_count)
    return NULL;

  /* make res and add shift solutions for included_free_resources to res */
  res = KheDrsShiftSolnTrieMake(de->solver);
  ... code omitted here, see below ...

  /* add a NULL child for every open resource */
  count = KheDrsResourceSetCount(de->solver->open_resources);
  HaArrayFill(res->children, count, NULL);

  /* add a potentially non-NULL child for each possible next resource */
  no_non_null_children = true;
  count = KheDrsResourceSetCount(all_free_resources);
  for( i = all_free_resources_index;  i < count;  i++ )
  {
    dr = KheDrsResourceSetResource(all_free_resources, i);
    KheDrsResourceSetAddLast(included_free_resources, dr);
    child_dsst = KheDrsShiftSolnTrieBuild(ds, prev_soln, prev_day,
      next_day, all_free_resources, i + 1, included_free_resources,
      all_fixed_assts, de);
    if( child_dsst != NULL )
    {
      HaArrayPut(res->children, dr->open_resource_index, child_dsst);
      no_non_null_children = false;
    }
    KheDrsResourceSetDeleteLast(included_free_resources);
  }

  /* replace by NULL if the tree contains no solutions */
  if( no_non_null_children && HaArrayCount(res->shift_solns) == 0 )
  {
    KheDrsShiftSolnTrieFree(res, de->solver);
    return NULL;
  }
  else
    return res;
}
```

Most parameters are as for `KheDrsShiftBuildShiftSolnTrie`; `all_free_resources_index` is an index into `all_free_resources` used to generate all subsets of the free resources, and `included_free_resources` is the current value of $R_{free}$.

Now `ds->expand_max_included_free_resource_count` is the maximum number of free resources that can be assigned to `ds` without leaving too few other free resources for the other shifts. So the first step is to return immediately if the number of included free resources exceeds this number. The `NULL` result represents a shift solution trie containing no shift solutions.

The next step is to create a shift solution trie node, `res`, and fill its `shift_solns` array with the undominated shift solution objects that assign the resources of `included_free_resources`, plus any fixed resources which are assigned to tasks from this shift. Most of this code is omitted above; we return to it below.

The next step is to build the children of the new node `res`. We want one child for each open resource, because we intend to index these children using an open resource index. So we start by filling `res->children` with one `NULL` value for each open resource. Then for each free resource `dr` whose open resource index we have not used in higher levels of the trie (for each free resource whose index in `all_free_resources` is `all_free_resources_index` or greater), we add `dr` to `included_free_resources`, call this same function recursively to build the child node, add that child to the `children` array of `dsst`, and delete `dr` from `included_free_resources`.

Finally, we check whether `res` contains no shift solutions at all: no non-`NULL` children, and an empty `res->shift_solns`. In that case, we free `res` and return `NULL` instead.

Here is the code (omitted above) to build the shift solutions for a given *S*, *s*, and *R*:

```
if( included_free_resource_count >= ds->expand_must_assign_count )
{
  KheDrsExpanderReset(de, true, KheDrsSolnCost(prev_soln),
    de->solver->solve_init_cost, included_free_resource_count,
    ds->expand_must_assign_count);
  KheDrsExpanderMarkBegin(de);
  KheDrsExpanderAddTaskSolnSet(de, all_fixed_assts, ds);
  if( KheDrsExpanderIsOpen(de) )
    KheDrsShiftSolnTrieBuildShiftSolns(res, ds, included_free_resources,
      0, prev_soln, prev_day, next_day, de);
  KheDrsExpanderMarkEnd(de);
}
```

Here `ds->expand_must_assign_count` is the number of must-assign tasks within the mtasks of `ds`. If the number of included free resources is less than this, then there is no point in building any shift assignments at this node, because there are too few free resources to cover the must-assign tasks. Otherwise, we create (in fact, reset) an expander, add the relevant fixed assignments to it (those fixed assignments from `all_fixed_assts` whose tasks lie in `ds`), and call `KheDrsShiftSolnTrieBuildShiftSolns`, which builds this node's shift assignments:

```
void KheDrsShiftSolnTrieBuildShiftSolns(KHE_DRS_SHIFT_SOLN_TRIE dsst,
  KHE_DRS_SHIFT ds, KHE_DRS_RESOURCE_SET included_free_resources,
  int included_free_resources_index, KHE_DRS_SOLN prev_soln,
  KHE_DRS_DAY prev_day, KHE_DRS_DAY next_day, KHE_DRS_EXPANDER de)
{
  KHE_DRS_RESOURCE dr;  int i, indent;  KHE_DRS_MTASK_SOLN dms;

  if( included_free_resources_index >=
      KheDrsResourceSetCount(included_free_resources) )
  {
    /* all included resources assigned, so build soln and add it now */
    KheDrsExpanderMakeAndMeldShiftSoln(de, dsst, ds, prev_soln, prev_day,
      next_day);
  }
  else
  {
    /* try all assignments of the next included resource */
    dr = KheDrsResourceSetResource(included_free_resources,
      included_free_resources_index);
    HaArrayForEach(dr->expand_mtask_solns, dms, i)
      if( KheDrsMTaskSolnShift(dms) == ds )
        KheDrsMTaskSolnShiftSolnTrieBuildShiftSolns(dms, dsst, ds,
          included_free_resources, included_free_resources_index + 1,
          prev_soln, prev_day, next_day, de);
  }
}
```

This is just expansion by resources, only for *R* instead of all the open resources, for the mtasks of *s* instead of for all mtasks, and with the solutions kept in `dsat->shift_solns` rather than in `next_day`'s solution set. If choices have been made for all the resources, it is time to call `KheDrsExpanderMakeAndMeldShiftSoln` (Appendix D.10.1). Otherwise, for each assignment of the next resource within `ds` we call `KheDrsMTaskSolnShiftSolnTrieBuildShiftSolns`:

```
void KheDrsMTaskSolnShiftSolnTrieBuildShiftSolns(KHE_DRS_MTASK_SOLN dms,
  KHE_DRS_SHIFT_SOLN_TRIE dsst, KHE_DRS_SHIFT ds,
  KHE_DRS_RESOURCE_SET included_free_resources,
  int included_free_resources_index, KHE_DRS_SOLN prev_soln,
  KHE_DRS_DAY prev_day, KHE_DRS_DAY next_day, KHE_DRS_EXPANDER de)
{
  KHE_DRS_TASK_ON_DAY dtd;  KHE_DRS_MTASK dmt;
  KHE_DRS_TASK_SOLN dts;  int indent;
  dmt = dms->mtask;
  if( dmt != NULL )
  {
    /* select a task from dmt and assign it */
    if( KheDrsMTaskAcceptResourceBegin(dmt, dms->resource_on_day, &dtd) )
    {
      dts = KheDrsTaskSolnMake(dms, dtd);
      KheDrsTaskSolnShiftSolnTrieBuildShiftSolns(dts, dsst, ds,
        included_free_resources, included_free_resources_index,
        prev_soln, prev_day, next_day, de);
      KheDrsMTaskAcceptResourceEnd(dmt, dtd);
    }
  }
  else
  {
    /* use dms->fixed_task_on_day, possibly NULL meaning a free day */
    dts = KheDrsTaskSolnMake(dms, dms->fixed_task_on_day);
    KheDrsTaskSolnShiftSolnTrieBuildShiftSolns(dts, dsst, ds,
      included_free_resources, included_free_resources_index,
      prev_soln, prev_day, next_day, de);
  }
}
```

This finds a specific task to assign, by calling `KheDrsMTaskAcceptResourceBegin` and `KheDrsMTaskAcceptResourceEnd` in the usual way if `dcs` has an mtask, or directly if the resource is already fixed to a specific task. Actually this second case cannot occur here, because the resources are not fixed and a free day is not an option. The resulting task solution object `dts` is then passed to `KheDrsTaskSolnShiftSolnTrieBuildShiftSolns`:

```
void KheDrsTaskSolnShiftSolnTrieBuildShiftSolns(KHE_DRS_TASK_SOLN dts,
  KHE_DRS_SHIFT_SOLN_TRIE dsst, KHE_DRS_SHIFT ds,
  KHE_DRS_RESOURCE_SET included_free_resources,
  int included_free_resources_index, KHE_DRS_SOLN prev_soln,
  KHE_DRS_DAY prev_day, KHE_DRS_DAY next_day, KHE_DRS_EXPANDER de)
{
  /* save the expander so it can be restored later */
  KheDrsExpanderMarkBegin(de);

  /* add dts to the expander */
  KheDrsExpanderAddTaskSoln(de, dts);

  /* if the expander is still open, recurse */
  if( KheDrsExpanderIsOpen(de) )
    KheDrsShiftSolnTrieBuildShiftSolns(dsst, ds, included_free_resources,
      included_free_resources_index, prev_soln, prev_day, next_day, de);

  /* restore the expander */
  KheDrsExpanderMarkEnd(de);
}
```

This assigns `dts` and recurses on the next resource if the expander is still open.

### D.9.8. Shift pair solutions

A shift pair solution is a pair of shift solutions. We may write ‘$s_i s_j$-solution’ for a shift pair solution whose shifts are $s_i$ and $s_j$.

The shift pair solution resembles the mtask pair solution (Appendix D.9.3), which represents a pair of mtask solutions. However this time there is a declared type:

```
typedef struct khe_drs_shift_pair_soln_rec *KHE_DRS_SHIFT_PAIR_SOLN;
typedef HA_ARRAY(KHE_DRS_SHIFT_PAIR_SOLN) ARRAY_KHE_DRS_SHIFT_PAIR_SOLN;

struct khe_drs_shift_pair_soln_rec {
  struct khe_drs_signature_set_rec      sig_set;
  KHE_DRS_SHIFT_SOLN                    dss1;
  KHE_DRS_SHIFT_SOLN                    dss2;
};
```

Also, the dominance testing is more conventional here, because the assignments are to specific tasks rather than to mtasks.

After the usual `KheDrsShiftPairSolnMake` and `KheDrsShiftPairSolnFree` functions for creating and freeing a shift pair solution, we have two functions that take us to the heart of things. First is `KheDrsShiftPairSolnSignerSetBuild`, which builds a signer set suited to comparing for dominance two shift pair solutions made from two given shift solutions:

```
KHE_DRS_SIGNER_SET KheDrsShiftPairSolnSignerSetBuild(
  KHE_DRS_SHIFT_SOLN dss1, KHE_DRS_SHIFT_SOLN dss2,
  KHE_DRS_SHIFT_PAIR dsp, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_SIGNER_SET res;  KHE_DRS_TASK_SOLN dts;  int i;
  KHE_DRS_RESOURCE_ON_DAY drd;

  /* make a signer set object */
  res = KheDrsSignerSetMake(drs);

  /* add resource signers of the resources of dss1 */
  KheDrsTaskSolnSetForEach(dss1->task_solns, dts, i)
  {
    drd = dts.mtask_soln->resource_on_day;
    KheDrsSignerSetAddSigner(res, drd->signer);
  }

  /* add resource signers of the resources of dss2 */
  KheDrsTaskSolnSetForEach(dss2->task_solns, dts, i)
  {
    drd = dts.mtask_soln->resource_on_day;
    KheDrsSignerSetAddSigner(res, drd->signer);
  }

  /* add dsp's shift pair signer */
  KheDrsSignerSetAddSigner(res, dsp->signer);
  return res;
}
```

The signer set contains one signer for each resource assigned by `dss1`, one signer for each resource assigned by `dss2`, and one signer, taken from shift pair object `dsp`, for the event resource monitors that monitor the tasks of the two shifts. These are all pre-existing signers; only their packaging into a single signer set is new.

Next we have `KheDrsShiftPairSolnBuild`, which creates a shift pair solution object and builds its signature set:

```
KHE_DRS_SHIFT_PAIR_SOLN KheDrsShiftPairSolnBuild(
  KHE_DRS_SHIFT_SOLN dss1, KHE_DRS_SHIFT_SOLN dss2,
  KHE_DRS_SHIFT_PAIR dsp, KHE_DRS_SOLN prev_soln,
  KHE_DRS_DAY next_day, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_SHIFT_PAIR_SOLN res;  int i;  KHE_DRS_TASK_SOLN dts;
  KHE_DRS_SIGNATURE sig;

  /* make a shift soln object */
  res = KheDrsShiftPairSolnMake(dss1, dss2, drs);

  /* add resource signatures of the resources of dss1 */
  KheDrsTaskSolnSetForEach(dss1->task_solns, dts, i)
    KheDrsSignatureSetAddSignature(&res->sig_set,
      KheDrsMTaskSolnSignature(dts.mtask_soln), true);

  /* add resource signatures of the resources of dss2 */
  KheDrsTaskSolnSetForEach(dss2->task_solns, dts, i)
    KheDrsSignatureSetAddSignature(&res->sig_set,
      KheDrsMTaskSolnSignature(dts.mtask_soln), true);

  /* evaluate the shift pair solution signature and add it */
  /* sig = KheDrsSignatureMake(drs); */
  KheDrsTaskSolnSetLeafSet(dss1->task_solns, true);
  KheDrsTaskSolnSetLeafSet(dss2->task_solns, true);
  sig = KheDrsSignerEvalSignature(dsp->signer, false,
    KheDrsSolnEventResourceSignature(prev_soln),
    next_day->open_day_index, /* sig, */ drs, false);
  KheDrsSignatureSetAddSignature(&res->sig_set, sig, true);
  KheDrsTaskSolnSetLeafClear(dss1->task_solns, false);
  KheDrsTaskSolnSetLeafClear(dss2->task_solns, false);
  return res;
}
```

The resource signatures already exist, as usual, but the signature of the event resource monitors that monitor the two shifts has to be built by evaluating expressions.

Now for the function we really want, for testing dominance between shift pair solutions:

```
bool KheDrsShiftPairSolnDominates(KHE_DRS_SHIFT_PAIR_SOLN dsps1,
  KHE_DRS_SHIFT_PAIR_SOLN dsps2, KHE_DRS_SIGNER_SET signer_set,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  return KheDrsSignerSetDominates(signer_set, &dsps1->sig_set,
    &dsps2->sig_set, 0, 0, false, drs);
}
```

It’s trivial given the work already done to build the signer set and signature sets.

We now switch to the submodule of type `KHE_DRS_SHIFT_SOLN_TRIE` which constructs shift pair solutions, calls `KheDrsShiftPairSolnDominates` to test them for dominance, and marks any dominated shift pairs. We'll work backwards through the submodule, starting with `KheDrsShiftSolnTrieFindDominatedShiftPairs`, which is called directly from the function for expanding solution `prev_soln` and does the job for all pairs of shift pair solutions:

```
void KheDrsShiftSolnTrieFindDominatedShiftPairs(KHE_DRS_SOLN prev_soln,
  KHE_DRS_DAY next_day, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_RESOURCE_SET rs1, rs2;  int i, j;  KHE_DRS_SHIFT ds1;
  bool first;  KHE_DRS_SHIFT_PAIR dsp;
  rs1 = KheDrsResourceSetMake(drs);
  rs2 = KheDrsResourceSetMake(drs);
  first = true;
  HaArrayForEach(next_day->shifts, ds1, i)
    if( ds1->soln_trie != NULL )
      HaArrayForEach(ds1->shift_pairs, dsp, j)
        if( dsp->shift[1]->soln_trie != NULL )
        {
          KheDrsShiftSolnTrieFindDominatedShiftPairs1(ds1->soln_trie,
            dsp, rs1, rs2, prev_soln, next_day, drs);
          first = false;
        }
  KheDrsResourceSetFree(rs1, drs);
  KheDrsResourceSetFree(rs2, drs);
}
```

By iterating over the shifts of `next_day` and then over the shift pairs for each shift, this code visits all shift pairs `dsp`. It skips shift pairs where either shift has no shift solutions. Then

```
void KheDrsShiftSolnTrieFindDominatedShiftPairs1(
  KHE_DRS_SHIFT_SOLN_TRIE dsst_rs1_ds1, KHE_DRS_SHIFT_PAIR dsp,
  KHE_DRS_RESOURCE_SET rs1, KHE_DRS_RESOURCE_SET rs2,
  KHE_DRS_SOLN prev_soln, KHE_DRS_DAY next_day,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  int i;  KHE_DRS_RESOURCE dr;
  KHE_DRS_SHIFT_SOLN_TRIE child_dsst, dsst_rs1_ds2;

  /* pairs for each of dsst_rs1_ds1's assignments */
  if( HaArrayCount(dsst_rs1_ds1->shift_solns) > 0 &&
      KheDrsShiftSolnTrieContains(dsp->shift[1]->soln_trie, rs1, 0,
        &dsst_rs1_ds2)
        && HaArrayCount(dsst_rs1_ds2->shift_solns) > 0 )
  {
    HnAssert(KheDrsResourceSetCount(rs2) == 0,
      "KheDrsShiftSolnTrieFindDominatedShiftPairs1 internal error");
    KheDrsShiftSolnTrieFindDominatedShiftPairs2(dsp->shift[0]->soln_trie,
      dsst_rs1_ds1, dsst_rs1_ds2,
      dsp, rs1, rs2, prev_soln, next_day, drs);
  }

  /* pairs for children */
  HaArrayForEach(dsst_rs1_ds1->children, child_dsst, i)
    if( child_dsst != NULL )
    {
      dr = KheDrsResourceSetResource(drs->open_resources, i);
      KheDrsResourceSetAddLast(rs1, dr);
      KheDrsShiftSolnTrieFindDominatedShiftPairs1(child_dsst, dsp,
        rs1, rs2, prev_soln, next_day, drs, debug);
      KheDrsResourceSetDeleteLast(rs1);
    }
}
```

traverses `ds1`'s shift solution trie. The variable name '`dsst_rs1_ds1`' means 'a shift solution trie node for free resources `rs1` and shift `ds1`'.

The second paragraph calls `KheDrsShiftSolnTrieFindDominatedShiftPairs1` for each non-`NULL` child recursively, updating `rs1` before each recursive call. So this proves that `KheDrsShiftSolnTrieFindDominatedShiftPairs1` visits every node of `ds1`'s shift solution trie, setting `rs1` correctly for each node.

The first paragraph calls

```
bool KheDrsShiftSolnTrieContains(KHE_DRS_SHIFT_SOLN_TRIE dsst,
  KHE_DRS_RESOURCE_SET rs, int rs_index, KHE_DRS_SHIFT_SOLN_TRIE *res)
{
  KHE_DRS_RESOURCE dr;  KHE_DRS_SHIFT_SOLN_TRIE child_dsst;
  if( dsst == NULL )
    return *res = NULL, false;
  else if( rs_index >= KheDrsResourceSetCount(rs) )
    return *res = dsst, true;
  else
  {
    dr = KheDrsResourceSetResource(rs, rs_index);
    child_dsst = HaArray(dsst->children, dr->open_resource_index);
    return KheDrsShiftSolnTrieContains(child_dsst, rs, rs_index + 1, res);
  }
}
```

to work out whether `dsp->shift[1]->soln_trie`, the shift solution trie of the second shift, has a node for solutions for resource set `rs1`, setting `dsst_rs1_ds2` to that node if so. Then if both nodes, `dsst_rs1_ds1` and `dsst_rs1_ds2`, have at least one shift solution, we proceed to the next step by calling `KheDrsShiftSolnTrieFindDominatedShiftPairs2`:

```
void KheDrsShiftSolnTrieFindDominatedShiftPairs2(
  KHE_DRS_SHIFT_SOLN_TRIE dsst_rs2_ds1,
  KHE_DRS_SHIFT_SOLN_TRIE dsst_rs1_ds1,
  KHE_DRS_SHIFT_SOLN_TRIE dsst_rs1_ds2, KHE_DRS_SHIFT_PAIR dsp,
  KHE_DRS_RESOURCE_SET rs1, KHE_DRS_RESOURCE_SET rs2,
  KHE_DRS_SOLN prev_soln, KHE_DRS_DAY next_day,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  int i;  KHE_DRS_RESOURCE dr;
  KHE_DRS_SHIFT_SOLN_TRIE dsst_rs2_ds2, child_dsst;

  /* pairs for each of dsst_rs2_ds1's assignments */
  if( KheDrsResourceSetCount(rs1) + KheDrsResourceSetCount(rs2) > 0 &&
      HaArrayCount(dsst_rs2_ds1->shift_solns) > 0 &&
      KheDrsShiftSolnTrieContains(dsp->shift[1]->soln_trie, rs2, 0,
        &dsst_rs2_ds2)
        && HaArrayCount(dsst_rs2_ds2->shift_solns) > 0 )
    KheDrsShiftSolnTrieTestShiftPairs(dsst_rs1_ds1, dsst_rs2_ds2,
      dsst_rs1_ds2, dsst_rs2_ds1, dsp, rs1, rs2, prev_soln, next_day, drs);

  /* pairs for children */
  HaArrayForEach(dsst_rs2_ds1->children, child_dsst, i)
    if( child_dsst != NULL )
    {
      dr = KheDrsResourceSetResource(drs->open_resources, i);
      if( !KheDrsResourceSetContains(rs1, dr) )
      {
        KheDrsResourceSetAddLast(rs2, dr);
        KheDrsShiftSolnTrieFindDominatedShiftPairs2(child_dsst, dsst_rs1_ds1,
          dsst_rs1_ds2, dsp, rs1, rs2, prev_soln, next_day, drs);
        KheDrsResourceSetDeleteLast(rs2);
      }
    }
}
```

This is like `KheDrsShiftSolnTrieFindDominatedShiftPairs1`, except that it traverses all nodes `dsst_rs2_ds1`, building `rs2` as it goes, and finding the corresponding `dsst_rs2_ds2`. However, it only accepts what it finds when at least one of `rs1` and `rs2` is non-empty, and by the test `!KheDrsResourceSetContains(rs1, dr)` it ensures that `rs1` and `rs2` are disjoint.

The ultimate result here is a call to `KheDrsShiftSolnTrieTestShiftPairs`, passing it four nodes, `dsst_rs1_ds1`, `dsst_rs2_ds2`, `dsst_rs1_ds2`, and `dsst_rs2_ds1`, whose shift solutions are suited to constructing two shift pair solutions:

```
void KheDrsShiftSolnTrieTestShiftPairs(
  KHE_DRS_SHIFT_SOLN_TRIE dsst_rs1_ds1,
  KHE_DRS_SHIFT_SOLN_TRIE dsst_rs2_ds2,
  KHE_DRS_SHIFT_SOLN_TRIE dsst_rs1_ds2,
  KHE_DRS_SHIFT_SOLN_TRIE dsst_rs2_ds1,
  KHE_DRS_SHIFT_PAIR dsp, KHE_DRS_RESOURCE_SET rs1,
  KHE_DRS_RESOURCE_SET rs2, KHE_DRS_SOLN prev_soln,
  KHE_DRS_DAY next_day, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_SHIFT_SOLN dss_rs1_ds1, dss_rs2_ds2, dss_rs1_ds2, dss_rs2_ds1;
  int i1, i2, i3, i4;  KHE_DRS_SHIFT_PAIR_SOLN dsps1, dsps2;
  KHE_DRS_SIGNER_SET signer_set;
  signer_set = NULL;
  HaArrayForEach(dsst_rs1_ds1->shift_solns, dss_rs1_ds1, i1)
    HaArrayForEach(dsst_rs2_ds2->shift_solns, dss_rs2_ds2, i2)
    {
      dsps1 = KheDrsShiftPairSolnBuild(dss_rs1_ds1, dss_rs2_ds2,
        dsp, prev_soln, next_day, drs);
      HaArrayForEach(dsst_rs1_ds2->shift_solns, dss_rs1_ds2, i3)
        HaArrayForEach(dsst_rs2_ds1->shift_solns, dss_rs2_ds1, i4)
        {
          dsps2 = KheDrsShiftPairSolnBuild(dss_rs1_ds2, dss_rs2_ds1,
            dsp, prev_soln, next_day, drs);
          if( signer_set == NULL )
            signer_set = KheDrsShiftPairSolnSignerSetBuild(dss_rs1_ds1,
              dss_rs2_ds2, dsp, drs);
          if( KheDrsShiftPairSolnDominates(dsps1, dsps2, signer_set, drs) )
          {
            HaArrayAddLast(dss_rs1_ds2->skip_assts, dss_rs2_ds1);
            HaArrayAddLast(dss_rs2_ds1->skip_assts, dss_rs1_ds2);
          }
          else if( KheDrsShiftPairSolnDominates(dsps2, dsps1, signer_set, drs) )
          {
            HaArrayAddLast(dss_rs1_ds1->skip_assts, dss_rs2_ds2);
            HaArrayAddLast(dss_rs2_ds2->skip_assts, dss_rs1_ds1);
          }
          KheDrsShiftPairSolnFree(dsps2, drs);
        }
      KheDrsShiftPairSolnFree(dsps1, drs);
    }
  if( signer_set != NULL )
    KheDrsSignerSetFree(signer_set, drs);
}
```

Most shift solution trie nodes contain just one shift solution (or none, but nodes with none do not make it this far). So the four loops, although formally correct, serve in practice to retrieve the one shift solution of the node. Then `KheDrsShiftPairSolnBuild` is called twice to make two shift pair solution objects out of the four shift solutions, `KheDrsShiftPairSolnDominates` is called both ways to test for dominance, and if dominance is found, the `skip_assts` fields of the dominated shift solutions are updated appropriately, so that those shift pair solutions will not be generated during expansion by shifts.

This code is capable of discovering that a particular shift pair solution is dominated, and recording that fact, more than once. This does not matter: it does not make the algorithm incorrect, and dominance is uncommon, so there is little wasted time.

### D.9.9. Packed solutions

The solver has a solution type, quite separate from the types we have just seen, called the *packed solution.* Despite its name, its purpose is not to save space. Rather it is designed to provide easy access to the solution's assignment of open resource *i* on open day *j*. Packed solutions only represent complete solutions, and they are never tested for dominance or inserted into tables.

Packed solutions are used in two ways. First, the initial solution, the one that we want to improve, is stored in a packed solution, so that if we fail to improve on it we can return to it. This is like using a mark, except that it returns the whole solver data structure to its initial state, not just the KHE solution. Function `KheDrsResourceOpen` (Appendix D.4) builds this solution.

Second, the solver offers the option of rerunning a new best solution as an aid to debugging (Appendix D.12.4). A packed solution holds the new best solution while the rerun is going on.

Type `KHE_DRS_PACKED_SOLN_DAY` represents one day of a packed solution:

```
typedef struct khe_drs_packed_soln_day_rec {
  KHE_DRS_DAY                  day;
  ARRAY_KHE_DRS_TASK_ON_DAY    prev_tasks;
} *KHE_DRS_PACKED_SOLN_DAY;

typedef HA_ARRAY(KHE_DRS_PACKED_SOLN_DAY) ARRAY_KHE_DRS_PACKED_SOLN_DAY;
```

The `prev_tasks` field is exactly as in the corresponding `KHE_DRS_SOLN` object for this `day`. Type `KHE_DRS_PACKED_SOLN` represents a complete packed solution:

```
typedef struct khe_drs_packed_soln_rec {
  KHE_COST                     cost;
  ARRAY_KHE_DRS_PACKED_SOLN_DAY days;
} *KHE_DRS_PACKED_SOLN;

typedef HA_ARRAY(KHE_DRS_PACKED_SOLN) ARRAY_KHE_DRS_PACKED_SOLN;
```

It holds the cost of the solution, and an array with one element for each open day.

Packed solutions operations include `KheDrsPackedSolnBuildFromSoln`, which converts a complete `KHE_DRS_SOLN` solution into a packed solution; `KheDrsPackedSolnDelete`, which deletes a packed solution using free lists in the usual way; and `KheDrsPackedSolnTaskOnDay` and `KheDrsPackedSolnSetTaskOnDay`, which get and set the assignment of open resource *i* on open day *j*. Their implementations are all straightforward, so are not given here.

## D.10. Expansion

This section presents the implementation of the key operation on solutions: *expansion.* Starting with a given $d_k$-solution $S$, expansion creates all the $d_{k+1}$-solution extensions of $S$ and adds them to $P_{k+1}$, the set of all undominated $d_{k+1}$-solutions. The actual addition to $P_{k+1}$, including dominance testing between the existing solutions and each new solution, is a separate subject; here we concentrate on creating the new solutions.

Expansion is implemented by function

```
void KheDrsSolnExpand(KHE_DRS_SOLN prev_soln, KHE_DRS_DAY prev_day,
  KHE_DRS_DAY next_day, KHE_DYNAMIC_RESOURCE_SOLVER drs);
```

Given $d_k$-solution `prev_soln` whose day $d_k$ is `prev_day`, and day $d_{k+1}$ `next_day`, this finds all $d_{k+1}$-solution extensions of `prev_soln`, and adds them, with dominance testing, to `next_day`'s solution set. Here `prev_soln` could be the root solution, which is not on any day, and in that case `prev_day` is `NULL`. However, `prev_soln` will never be a solution for the last open day, so `next_day` is always a well-defined open day.

The implementation is spread over several types. Several of these types `X` have a submodule called '`X - expansion`' following their main submodule, containing `X`'s part of the expansion code. However, our presentation here often works top-down, ranging across these submodules as required. This works better than presenting each type's expansion code separately.

### D.10.1. Expanders

Expansion is always about trying certain assignments, then undoing those and trying others. These steps are supported by an *expander* object, of type `KHE_DRS_EXPANDER`:

```
typedef struct khe_drs_expander_rec *KHE_DRS_EXPANDER;
typedef HA_ARRAY(KHE_DRS_EXPANDER) ARRAY_KHE_DRS_EXPANDER;

struct khe_drs_expander_rec {
  KHE_DYNAMIC_RESOURCE_SOLVER   solver;
  ARRAY_KHE_DRS_TASK_SOLN       task_solns;
  ARRAY_KHE_DRS_TASK_SOLN       tmp_task_solns;
  bool                          whole_tasks;
  bool                          open;
  KHE_COST                      cost;
  KHE_COST                      cost_limit;
  int                           free_resource_count;
  int                           must_assign_count;
  HA_ARRAY_INT                  marks;
};
```

Each expansion begins by creating an expander, and ends by freeing it.

Field `solver` is the enclosing solver. It is not often used, and when it is used it is usually for something fairly trivial, like access to a free list.

Field `task_solns` holds the *current assignments*: task solution objects that the expansion wants to include in the next solution it creates. Field `tmp_task_solns` is a scratch variable used by function `KheDrsExpanderMakeAndMeldSoln` below.

Field `whole_tasks` changes the meaning given to one task solution when there are multi-day tasks. When it is `false`, the task solution represents the assignment of one resource on day to one task on day. When it is `true`, although the task solution object itself is the same, it is interpreted to mean that the task on day's task is assigned the resource on day's resource on every day that the task is running.

Field `open` is `true` when the expander can see nothing wrong with the current assignments: it is 'open' to adding zero or more additional assignments to them until a complete solution's worth of assignments has been made. Otherwise, there is some problem and expansion should back out of the point it has reached and try something else: the expander is *closed* (not open).

Field `cost` is a lower bound on the cost of any solution containing the current assignments. We'll see later how the expander keeps this up to date. Field `cost_limit` is an upper limit on how much a solution is allowed to cost. If the assignments chosen by the expansion cause `cost` to equal or exceed `cost_limit`, the expander will close.

As defined in Appendix D.10.3, a *free resource* is an open resource which is not part of a fixed assignment. Field `free_resource_count` holds the number of free resources available to this expansion and not assigned (not even to a free day) by the current assignments. The expander handles fixed resources as well, they just don't affect `free_resource_count`.

As defined in Appendix D.10.4, a *must-assign task* is an open task, not part of a fixed assignment, that the current expansion must assign a resource to, otherwise the cost will be too great. Field `must_assign_count` holds the number of must-assign tasks that are part of the current expansion but are not assigned by the current assignments. The expander handles all kinds of open tasks, but only must-assign tasks affect `must_assign_count`.

If `free_resource_count < must_assign_count`, then there are too few unused free resources to cover the must-assign tasks that are not currently assigned. The expander will close. This assumes that all the tasks that are part of the expansion are running on the same day, so that no resource can be assigned to two of them.

Finally, `marks` is an array of indexes into the `task_solns` array. It allows the expander to mark the point that it has reached and to return to that point, as implemented by functions `KheDrsExpanderMarkBegin` and `KheDrsExpanderMarkEnd` below.

At the start of an expansion, `KheDrsExpanderMake` is called to make a new expander. But we'll start with `KheDrsExpanderReset`, which resets an expander using fresh attributes:

```
void KheDrsExpanderReset(KHE_DRS_EXPANDER de, bool whole_tasks,
  KHE_COST cost, KHE_COST cost_limit, int free_resource_count,
  int must_assign_count)
{
  de->whole_tasks = whole_tasks;
  de->cost = cost;
  de->cost_limit = cost_limit;
  de->free_resource_count = free_resource_count;
  de->must_assign_count = must_assign_count;
  KheDrsExpanderSetOpen(de);
}
```

`KheDrsExpanderSetOpen` sets the `open` field:

```
void KheDrsExpanderSetOpen(KHE_DRS_EXPANDER de)
{
  de->open = de->cost < de->cost_limit &&
    de->free_resource_count >= de->must_assign_count;
}
```

The expander is open if its cost has not reached the limit, and there are enough as-yet-unassigned free resources to cover the as-yet-unassigned must-assign tasks.

Here now is `KheDrsExpanderMake`:

```
KHE_DRS_EXPANDER KheDrsExpanderMake(bool whole_tasks, KHE_COST cost,
  KHE_COST cost_limit, int free_resource_count, int must_assign_count,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_EXPANDER res;

  /* get an expander from scratch or from the free list */
  if( HaArrayCount(drs->expander_free_list) > 0 )
  {
    res = HaArrayLastAndDelete(drs->expander_free_list);
    HaArrayClear(res->task_solns);
    HaArrayClear(res->tmp_task_solns);
    HaArrayClear(res->marks);
  }
  else
  {
    HaMake(res, drs->arena);
    HaArrayInit(res->task_solns, drs->arena);
    HaArrayInit(res->tmp_task_solns, drs->arena);
    HaArrayInit(res->marks, drs->arena);
  }

  /* initialize its fields and return it */
  res->solver = drs;
  KheDrsExpanderReset(res, whole_tasks, cost, cost_limit,
    free_resource_count, must_assign_count);
  return res;
}
```

It takes the object from a free list, or makes it from scratch, as usual. `KheDrsExpanderReset` is also called directly, to make a fresh start with an existing expander.

At the end of expansion, the expander is freed by a call to `KheDrsExpanderFree`:

```
void KheDrsExpanderFree(KHE_DRS_EXPANDER de)
{
  HnAssert(HaArrayCount(de->task_solns) == 0,
    "KheDrsExpanderFree internal error 1");
  HnAssert(HaArrayCount(de->marks) == 0,
    "KheDrsExpanderFree internal error 2");
  HaArrayAddLast(de->solver->expander_free_list, de);
}
```

It checks that the expansion ended cleanly, then adds the expander to a free list in the solver.

It is not really safe to access the fields of expander objects outside the expander, other than `solver`. Instead there are these small and self-explanatory functions:

```
void KheDrsExpanderAddCost(KHE_DRS_EXPANDER de, KHE_COST cost)
{
  de->cost += cost;
  KheDrsExpanderSetOpen(de);
}

void KheDrsExpanderReduceCostLimit(KHE_DRS_EXPANDER de,
  KHE_COST cost_limit)
{
  if( cost_limit < de->cost_limit )
  {
    de->cost_limit = cost_limit;
    KheDrsExpanderSetOpen(de);
  }
}

bool KheDrsExpanderOpenToExtraCost(KHE_DRS_EXPANDER de,
  KHE_COST extra_cost)
{
  return de->cost + extra_cost < de->cost_limit;
}

void KheDrsExpanderAddMustAssign(KHE_DRS_EXPANDER de)
{
  de->must_assign_count++;
  KheDrsExpanderSetOpen(de);
}

void KheDrsExpanderDeleteFreeResource(KHE_DRS_EXPANDER de)
{
  de->free_resource_count--;
  KheDrsExpanderSetOpen(de);
}

int KheDrsExpanderExcessResourceCount(KHE_DRS_EXPANDER de)
{
  return de->free_resource_count - de->must_assign_count;
}
```

To add a task solution object to an expander, the call is

```
void KheDrsExpanderAddTaskSoln(KHE_DRS_EXPANDER de,
  KHE_DRS_TASK_SOLN dts)
{
  KHE_DRS_SIGNATURE sig;  KHE_DRS_TASK dt;  KHE_DRS_MTASK_SOLN asst;
  int i;  KHE_COST cost;  int must_assign_count, free_resource_count;
  KHE_DRS_RESOURCE dr;

  /* if not open, do nothing */
  if( !de->open )
    return;

  /* if there is a skip count problem, close and do nothing */
  if( dts.mtask_soln->skip_count > 0 )
  {
    de->open = false;
    return;
  }

  /* find new must_assign_count, free_resource_count, and cost values */
  must_assign_count = de->must_assign_count;
  free_resource_count = de->free_resource_count;
  cost = de->cost;
  if( dts.fixed_dtd != NULL )
  {
    dt = dts.fixed_dtd->encl_dt;
    if( dts.fixed_dtd == HaArrayFirst(dt->days) )
      cost += dt->asst_cost;
    if( dt->expand_role == KHE_DRS_TASK_EXPAND_MUST )
      must_assign_count--;
  }
  sig = KheDrsMTaskSolnSignature(dts.mtask_soln);
  cost += sig->cost;
  dr = KheDrsTaskSolnResource(dts);
  if( dr->expand_role == KHE_DRS_RESOURCE_EXPAND_FREE )
    free_resource_count--;

  /* if there is a problem with the values just found, close and return */
  if( cost >= de->cost_limit || free_resource_count < must_assign_count )
  {
    de->open = false;
    return;
  }

  ... code omitted here, see below ...
}
```

First, if the expander is already closed, it returns immediately. If the new assignment should not be used, because it has a non-zero `skip_count` field, the expander closes and returns. Otherwise, it finds the effect of the new assignment on `must_assign_count`, `free_resource_count`, and `cost`. If the assignment is to a task (i.e. not to a free day), the cost increases by the cost of that assignment unless this is not the task's first day, and if the task is a must-assign task, `must_assign_count` decreases by one. Whatever the resource is assigned to, cost increases by the cost of the assignment's resource monitors signature, and if the assignment involves a free resource, then the free resource count decreases by one. If these new values lead to problems, the expander closes and returns.

If we get past all that, the expander can accept the new assignment and remain open:

```
/* no problems with the addition; change the state of de */
de->must_assign_count = must_assign_count;
de->free_resource_count = free_resource_count;
de->cost = cost;
HaArrayAddLast(de->task_solns, dts);

/* increment the skip counts of the skip_assts */
HaArrayForEach(dts.mtask_soln->skip_assts, asst, i)
  asst->skip_count++;

/* update dtd's leaf expressions */
KheDrsTaskSolnLeafSet(dts, de->whole_tasks, de->solver);
```

It assigns the new values to the `must_assign_count`, `free_resource_count`, and `cost` fields, and adds `dts` to `de->task_solns`. It then increments the skip count fields of `dts`'s skip list, as required, and ends by informing the task on day objects affected by `dts` that `dts` is now in force, by a call to `KheDrsTaskSolnLeafSet` (Appendix D.9.4).

There is also function `KheDrsExpanderAddTaskSolnSet`, which adds a whole set of task solutions to the expander by calling `KheDrsExpanderAddTaskSoln` on each:

```
void KheDrsExpanderAddTaskSolnSet(KHE_DRS_EXPANDER de,
  KHE_DRS_TASK_SOLN_SET dtss, KHE_DRS_SHIFT ds)
{
  KHE_DRS_TASK_SOLN dts;  int i;
  if( ds != NULL )
  {
    KheDrsTaskSolnSetForEach(dtss, dts, i)
      if( KheDrsTaskSolnShift(dts) == ds )
        KheDrsExpanderAddTaskSoln(de, dts);
  }
  else
    KheDrsTaskSolnSetForEach(dtss, dts, i)
      KheDrsExpanderAddTaskSoln(de, dts);
}
```

If `ds` is non-`NULL`, only those elements of `dtss` which assign tasks from shift `ds` are added.

When a task solution is no longer required, the next function removes it, assuming that it has already been removed from `de->assts_to_tasks`:

```
void KheDrsExpanderDoDeleteTaskSoln(KHE_DRS_EXPANDER de,
  KHE_DRS_TASK_SOLN dts)
{
  KHE_DRS_SIGNATURE sig;  KHE_DRS_TASK dt;  KHE_DRS_RESOURCE dr;
  int i;  KHE_DRS_MTASK_SOLN asst;

  /* update dtd's leaf expressions */
  KheDrsTaskSolnLeafClear(dts, de->whole_tasks, de->solver);

  /* decrement the skip counts of the skip_assts */
  HaArrayForEach(dts.mtask_soln->skip_assts, asst, i)
    asst->skip_count--;

  /* update cost, must_assign_count, and free_resource_count fields */
  if( dts.fixed_dtd != NULL )
  {
    dt = dts.fixed_dtd->encl_dt;
    if( dts.fixed_dtd == HaArrayFirst(dt->days) )
      de->cost -= dt->asst_cost;
    if( dt->expand_role == KHE_DRS_TASK_EXPAND_MUST )
      de->must_assign_count++;
  }
  sig = KheDrsMTaskSolnSignature(dts.mtask_soln);
  de->cost -= sig->cost;
  dr = KheDrsTaskSolnResource(dts);
  if( dr->expand_role == KHE_DRS_RESOURCE_EXPAND_FREE )
    de->free_resource_count++;
}
```

This reverses the state changes made by `KheDrsExpanderAddTaskSoln`. It is called only by `KheDrsExpanderMarkEnd` (see below), never directly by any expansion.

Expansion algorithms need to say ‘remember the assignments we have now; we’ll return to them later’. For this we have `KheDrsExpanderMarkBegin` and `KheDrsExpanderMarkEnd`:

```
void KheDrsExpanderMarkBegin(KHE_DRS_EXPANDER de)
{
  HaArrayAddLast(de->marks, HaArrayCount(de->assts_to_tasks));
}
```

```
void KheDrsExpanderMarkEnd(KHE_DRS_EXPANDER de)
{
  int prev_count;  KHE_DRS_TASK_SOLN dts;
  HnAssert(HaArrayCount(de->marks) > 0,
    "KheDrsExpanderMarkEnd internal error");
  prev_count = HaArrayLastAndDelete(de->marks);
  while( HaArrayCount(de->task_solns) > prev_count )
  {
    dts = HaArrayLastAndDelete(de->task_solns);
    KheDrsExpanderDoDeleteTaskSoln(de, dts);
  }
  KheDrsExpanderSetOpen(de);
}
```

`KheDrsExpanderMarkBegin` remembers the number of assignments; `KheDrsExpanderMarkEnd` returns the expander to them by popping assignments off the `task_solns` array and removing them. The expander does not have to be open when `KheDrsExpanderMarkBegin` is called, so `KheDrsExpanderSetOpen` is called to set the correct value of `open`.

Expansions can test whether the expander is open:

```
bool KheDrsExpanderIsOpen(KHE_DRS_EXPANDER de)
{
  return de->open;
}
```

to find out whether they should continue down the current path.

The expander also offers a function which makes a new day solution object and melds it into a solution set:

```
void KheDrsExpanderMakeAndMeldSoln(KHE_DRS_EXPANDER de,
  KHE_DRS_SOLN prev_soln, KHE_DRS_DAY next_day)
{
  KHE_DRS_SOLN next_soln;  KHE_DRS_TASK_SOLN dts, junk;  int i, ri;
  KHE_DRS_SIGNATURE sig, prev_sig;  KHE_DRS_SIGNER dsg;

  /* make a soln object */
  next_day->soln_made_count++;
  next_soln = KheDrsSolnMake(prev_soln, de->cost, de->solver);

  /* make sure de->tmp_task_solns has the right length */
  if( HaArrayCount(de->tmp_task_solns) != HaArrayCount(de->task_solns) )
  {
    HaArrayClear(de->tmp_task_solns);
    junk = KheDrsTaskSolnMake(NULL, NULL);
    HaArrayFill(de->tmp_task_solns, HaArrayCount(de->task_solns), junk);
  }

  /* reorder de->task_solns into de->tmp_task_solns */
  HaArrayForEach(de->task_solns, dts, i)
  {
    ri = KheDrsResourceOnDayIndex(dts.mtask_soln->resource_on_day);
    HaArrayPut(de->tmp_task_solns, ri, dts);
  }

  /* add each dts's task and signature (but not its cost) to soln */
  HaArrayForEach(de->tmp_task_solns, dts, i)
  {
    HaArrayAddLast(next_soln->prev_tasks, dts.fixed_dtd);
    KheDrsSignatureSetAddSignature(&next_soln->sig_set, dts.mtask_soln->sig,
      false);
  }

  /* set the event resource monitor part of next_soln's signature set */
  /* this last signature will be freed when next_soln is freed */
  dsg = HaArrayLast(next_day->signer_set->signers);
  if( HaArrayCount(prev_soln->sig_set.signatures) > 0 )
    prev_sig = HaArrayLast(prev_soln->sig_set.signatures);
  else
    prev_sig = NULL;  /* prev_soln is root, so prev_sig won't be accessed */
  sig = KheDrsSignerEvalSignature(dsg, prev_sig, de->solver, false);
  KheDrsSignatureSetAddSignature(&next_soln->sig_set, sig, true);

  /* depending on cost, either add next_soln to next_day or free it */
  if( KheDrsSolnCost(next_soln) < de->cost_limit )
    KheDrsSolnSetMeldSoln(next_day->soln_set, next_soln, next_day, de,
      de->solver);
  else
    KheDrsSolnFree(next_soln, de->solver);
}
```

The first step here is to make a new `KHE_DRS_SOLN` object, `next_soln`. Then, after ensuring that `de->tmp_task_solns` has the same length as `de->task_solns`, the assignment to task objects are copied into `de->tmp_task_solns`, reordering them into open resource index order. It is then easy to copy them into `next_soln->prev_tasks`, and also to add in their signatures' states. After that, the signatures of the solution's event resource monitors are added in, and finally the new solution is either melded into `next_day`'s solution set (if its cost is competitive) or freed.

A logically similar but much shorter function makes a new `KHE_DRS_SHIFT_SOLN` object and melds it into the set of shift assignment objects held in shift solution trie node `dsst`:

```
void KheDrsExpanderMakeAndMeldShiftSoln(KHE_DRS_EXPANDER de,
  KHE_DRS_SHIFT_SOLN_TRIE dsst, KHE_DRS_SHIFT ds,
  KHE_DRS_SOLN prev_soln, KHE_DRS_DAY prev_day, KHE_DRS_DAY next_day)
{
  KHE_DRS_SHIFT_SOLN dss;  int i;  KHE_DRS_TASK_SOLN dts;
  KHE_DRS_SIGNATURE prev_sig;

  /* make dss and add de's non-fixed assignments to tasks to it */
  dss = KheDrsShiftSolnMake(de->solver);
  HaArrayForEach(de->task_solns, dts, i)
    if( !KheDrsTaskSolnIsFixed(dts) )
      KheDrsTaskSolnSetAddLast(dss->task_solns, dts);

  /* set dss's signature and cost */
  if( prev_soln != NULL )
    prev_sig = HaArrayLast(prev_soln->sig_set.signatures);
  else
    prev_sig = NULL;  /* prev_soln is root, so prev_sig won't be accessed */
  dss->sig = KheDrsSignerEvalSignature(ds->signer, prev_sig, de->solver));
  KheDrsSignatureRefer(dss->sig);

  /* depending on cost, either add dss to dsst or free it */
  if( KheDrsSolnCost(prev_soln) + dss->sig->cost < de->cost_limit )
    KheDrsShiftSolnTrieMeldShiftSoln(dsst, dss, ds->signer, de->solver);
  else
    KheDrsShiftSolnFree(dss, de->solver);
}
```

The function begins by making a new shift solution object. It just copies the task solutions into the new object; their order there does not matter. Only non-fixed tasks are stored in shift solution objects; we'll defer the reason for this until we come to study expansion by shifts. Then a signature is calculated for the event resource monitors only, as required in shift solution objects, and finally the new object is either melded into `dsst`'s list of shift solution objects (if its cost is competitive) by `KheDrsShiftSolnTrieMeldShiftSoln` (Appendix D.9.7). or else it is freed.

### D.10.2. The main solution expansion function

We begin our top-down presentation at the top, with function `KheDrsSolnExpand`. We'll be examining the functions it calls later; for now the idea is to get an overview.

`KheDrsSolnExpand` is too big for one page, so we've broken it into chunks:

```
void KheDrsSolnExpand(KHE_DRS_SOLN prev_soln, KHE_DRS_DAY prev_day,
  KHE_DRS_DAY next_day, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  int i, j, k;  KHE_DRS_RESOURCE dr;  KHE_DRS_EXPANDER de, shift_de;
  KHE_DRS_SHIFT ds, ds2;  KHE_DRS_RESOURCE_SET free_resources;
  KHE_DRS_TASK_SOLN_SET fixed_assts;  KHE_DRS_TASK_SOLN asst;

  ... see the five chunks of code below ...
}
```

As described previously, given $d_k$-solution `prev_soln` whose day $d_k$ is `prev_day`, and day $d_{k+1}$ `next_day`, this finds all $d_{k+1}$-solution extensions of `prev_soln`, and adds them, with dominance testing, to `next_day`'s solution set. Here `prev_soln` could be the root solution, which is not on any day, and in that case `prev_day` is `NULL`. However, `prev_soln` will never be a solution for the last open day, so `next_day` is always a well-defined open day.

Here is the first chunk of code:

```
/* check on and update the number of expansions from prev_day */
if( prev_day != NULL )
{
  if( drs->solve_daily_expand_limit > 0 &&
      prev_day->solve_expand_count >= drs->solve_daily_expand_limit )
    return;
  prev_day->solve_expand_count += 1;
}

/* check the time limit and return early if it has been reached */
if( KheOptionsTimeLimitReached(drs->options) )
{
  if( DEBUG22 )
    fprintf(stderr, "  KheDrsSolnExpand returning early (time
limit)\n");
  return;
}

/* mark prev_soln as expanded */
KheDrsSolnMarkExpanded(prev_soln);
```

The solver offers an option to limit the number of expansions carried out on each day. If this option is in effect (if `drs->solve_daily_expand_limit > 0`) then `KheDrsSolnExpand` returns immediately if the limit has been reached. It then checks the time limit and returns immediately if it has been reached. Then KheDrsSolnMarkExpanded is called to set the priority queue back index of prev_soln to -1, to indicate that prev_soln has been expanded.

The second chunk of code does some setting up for expansion:

```
/* make the main expander */
de = KheDrsExpanderMake(false, KheDrsSolnCost(prev_soln),
  drs->solve_init_cost, KheDrsResourceSetCount(drs->open_resources),
0, drs);

/* begin expansion in each open resource */
free_resources = KheDrsResourceSetMake(drs);
fixed_assts = KheDrsTaskSolnSetMake(drs);
KheDrsResourceSetForEach(drs->open_resources, dr, i)
  KheDrsResourceExpandBegin(dr, prev_soln, next_day, free_resources,
    fixed_assts, de);
KheDrsResourceSetForEach(free_resources, dr, i)
  KheDrsResourceExpandBeginFree(dr, prev_soln, next_day, de);

/* begin expansion in next_day and its shifts */
KheDrsDayExpandBegin(next_day, prev_soln, prev_day, de);

/* set up for mtask soln and mtask pair soln dominance, if requested */
if( drs->solve_extra_selection )
{
  KheDrsMTaskSolnDominanceInit(drs);
  KheDrsMTaskPairSolnDominanceInit(drs);
}
```

The first step here is to create an expander. Its initial cost is KheDrsSolnCost(prev_soln). Its initial cost_limit is drs->solve_init_cost, the cost of the solution that we are trying to improve on. Its initial free_resource_count is the number of open resources, but the following calls to KheDrsResourceExpandBegin will reduce that to the number of free resources (open resources not subject to fixed assignments). And its initial must_assign_count is 0, but KheDrsDayExpandBegin will increase that as it discovers must-assign tasks.

The next step is to create two sets: free_resources, which will grow from its initial empty value to hold the set of all open resources not subject to a fixed assignment on next_day, and fixed_assts, which will grow from its initial empty value to hold the set of all fixed assignments of open resources on next_day.

After that, for each open resource dr we call KheDrsResourceExpandBegin to inform dr that an expansion is beginning. We'll see this function later. Among other jobs it either adds dr to free_resources or else it adds dr's fixed assignment to fixed_assts.

Next we call KheDrsResourceExpandBeginFree for each free resource. We'll see why we visit the free resources a second time like this, rather than doing all the work just once in KheDrsResourceExpandBegin, when we study these two functions in detail.

Next we call KheDrsDayExpandBegin to inform next_day that an expansion is beginning.

Then mtask solution dominance and mtask pair solution dominance are initialized if requested.

The third chunk of code is concerned with setting up for expansion by shifts:

```
if( drs->solve_expand_by_shifts )
{
  /* initialize shift solution tries */
  shift_de = KheDrsExpanderMake(true, 0, 0, 0, 0, drs);
  HaArrayForEach(next_day->shifts, ds, i)
    KheDrsShiftBuildShiftSolnTrie(ds, prev_soln, prev_day, next_day,
      free_resources, fixed_assts, shift_de, drs);
  KheDrsExpanderFree(shift_de);

  /* find forced assignments, prune shift solutions invalidated by them */
  HaArrayForEach(next_day->shifts, ds, i)
    KheDrsResourceSetForEach(drs->open_resources, dr, j)
      if( KheDrsShiftSolnTrieResourceIsForced(ds->soln_trie, dr) )
      {
        /* dr is forced in ds, so prune it from the others */
        HaArrayForEach(next_day->shifts, ds2, k)
          if( ds2 != ds )
            KheDrsShiftSolnTriePruneForced(ds2->soln_trie, dr, drs);
      }

  /* find pairs of dominated shifts on next_day */
  if( drs->solve_shift_pairs )
    KheDrsShiftSolnTrieFindDominatedShiftPairs(prev_soln, next_day, drs);
}
```

This begins by calling `KheDrsShiftBuildShiftSolnTrie` for each shift on `next_day`, to build the shift solution trie for that shft. Here `shift_de` is a scratch expander which is reset each time an expander is needed.

If all assignments to a shift *s* demand some resource *r*, then *r* is not available for assignment to any other shift. `KheDrsShiftSolnTrieResourceIsForced` checks this condition for one shift and one resource, and if it is true, `KheDrsShiftSolnTriePruneForced` removes all assignments containing that resource from other shift solution tries.

Then `KheDrsShiftSolnTrieFindDominatedShiftPairs` is called to initialize for shift pair dominance. This will prevent pairs of shift solutions being chosen that have previously been shown to be uncompetitive.

The fourth chunk of code carries out the expansion proper:

```
/* carry out the main part of the expansion */
KheDrsExpanderMarkBegin(de);
KheDrsExpanderAddTaskSolnSet(de, fixed_assts, NULL);
if( KheDrsExpanderIsOpen(de) )
{
  if( drs->solve_expand_by_shifts )
    KheDrsSolnExpandByShifts(prev_soln, next_day, de, free_resources, 0);
  else
    KheDrsSolnExpandByResources(prev_soln, prev_day, next_day, de,
      free_resources, 0);
}
KheDrsExpanderMarkEnd(de);
```

The fixed assignments are added to the expander, and then the free ones are assigned by calling `KheDrsSolnExpandByShifts` or `KheDrsSolnExpandByResources`, if the expander is open.

Finally, the fifth and last chunk of code finishes the expansion by ending expansion in `next_day` and in the open resources, and freeing the two sets and the expander:

```
/* end expansion in next_day and in the open resources */
KheDrsDayExpandEnd(next_day, de);
KheDrsResourceSetForEach(drs->open_resources, dr, i)
  KheDrsResourceExpandEnd(dr, de);
KheDrsResourceSetFree(free_resources, drs);

/* free fixed_assts and the expander and return */
KheDrsTaskSolnSetForEach(fixed_assts, asst, i)
  if( asst.fixed_dtd != NULL )
    asst.fixed_dtd->encl_dt->expand_role = KHE_DRS_TASK_EXPAND_NO_VALUE;
KheDrsTaskSolnSetFree(fixed_assts, drs);
KheDrsExpanderFree(de);
```

The assignments to `expand_role` should be encapsulated in some logically defined function.

### D.10.3. Initializing resources for expansion

In this section we present the code that sets up each open resource for expansion and clears it back again at the end. We start with the fields of `KHE_DRS_RESOURCE` concerned with expansion:

```
struct khe_drs_resource_rec {
  ...
  KHE_DRS_RESOURCE_EXPAND_ROLE  expand_role;
  ARRAY_KHE_DRS_SIGNATURE       expand_signatures;
  ARRAY_KHE_DRS_MTASK_SOLN      expand_mtask_solns;
  KHE_DRS_MTASK_SOLN            expand_free_mtask_soln;
  KHE_DRS_DIM2_TABLE            expand_dom_test_cache;
};
```

These fields appear in type `KHE_DRS_RESOURCE` rather than, say, `KHE_DRS_RESOURCE_ON_DAY` because the algorithm is single-threaded and there is only one expansion going on at any given moment, so only one value of these fields is needed for a given resource at any given moment.

Field `expand_role` has type

```
typedef enum {
  KHE_DRS_RESOURCE_EXPAND_NO,
  KHE_DRS_RESOURCE_EXPAND_FIXED,
  KHE_DRS_RESOURCE_EXPAND_FREE
} KHE_DRS_RESOURCE_EXPAND_ROLE;
```

and records the role that this resource (call it *r*) takes in the current expansion, as follows.

`KHE_DRS_RESOURCE_EXPAND_NO`: there is no expansion in progress, or there is one but *r* is not an open resource, so it does not participate in it.

`KHE_DRS_RESOURCE_EXPAND_FIXED`: there is an expansion in progress, *r* is open so it participates in it, and *r* must be assigned by it to a specific task. Usually this will be because that task is a multi-day task and *r* was assigned to it on its first day. `KheDrsResourceOnDayIsFixed` (see below) has the full story. We say that the resource is *fixed*.

`KHE_DRS_RESOURCE_EXPAND_FREE`: neither of the other cases applies: an expansion is in progress, *r* is open so it participates in it, but *r* is not fixed to any specific task, and indeed need not be assigned at all. We say that the resource is *free*.

By definition, the tasks of one shift have the same busy times and workload. Assigning *r* to any one of them has the same effect on *r*'s resource monitors. For each open resource and each shift on the `next_day` of the expansion, there is a single `KHE_DRS_SIGNATURE` object holding this common resource signature. There is also one signature for a free day. All these signatures are kept in the `expand_signatures` field of the resource, in arbitrary order.

Field `expand_mtask_solns` contains the mtask solution objects open to `dr` on `next_day`. Typically this will be one per mtask plus one denoting a free day. If `dr` is fixed there will be just the one mtask solution. Again, these mtask solutions appear in arbitrary order. As we'll see, they get sorted into non-decreasing cost order. This simple heuristic helps expand by resources to find better solutions earlier, which somewhat reduces running time.

Field `expand_free_mtask_soln` contains the free day element of `expand_mtask_solns`. If a free day is not possible for any reason, `expand_free_mtask_soln` is `NULL`.

Finally, `expand_dom_test_cache` is an optional cache containing the results of dominance tests between pairs of elements of `expand_mtask_solns`. The intention is to speed up dominance testing in these cases, but the author has not observed any significant speedup.

Our first function is `KheDrsResourceOnDayIsFixed`, a helper function, called only by `KheDrsResourceExpandBegin`, which finds whether a resource is fixed and if so to what:

```
bool KheDrsResourceOnDayIsFixed(KHE_DRS_RESOURCE_ON_DAY drd,
  KHE_DRS_SOLN soln, KHE_DYNAMIC_RESOURCE_SOLVER drs,
  KHE_DRS_TASK_ON_DAY *dtd)
{
  KHE_DRS_TASK_ON_DAY dtd1, dtd2;

  /* (1) if this is a rerun, that fixes the assignment */
  if( drs->rerun_soln != NULL )
    return *dtd = KheDrsPackedSolnTaskOnDay(drs->rerun_soln,
      drd->day, drd->encl_dr), true;

  /* (2) if drd is preassigned to a task on this day, it's fixed to that */
  if( drd->preasst_dtd != NULL )
    return *dtd = drd->preasst_dtd, true;

  /* (3) if drd has a closed assignment, it's fixed to that */
  if( drd->closed_dtd != NULL )
    return *dtd = drd->closed_dtd, true;

  /* (4) if drd's resource is assigned to a task in soln which is still */
  /* running, then drd is fixed to that */
  if( KheDrsSolnResourceIsAssigned(soln, drd->encl_dr, &dtd1) &&
      KheDrsTaskRunningOnDay(dtd1->encl_dt, drd->day, &dtd2) )
    return *dtd = dtd2, true;

  /* otherwise drd has no fixed assignment */
  return *dtd = NULL, false;
}
```

First, some runs are *reruns* and for them the resource on day is fixed to a task on day that may be retrieved from the `drs->rerun` packed solution object (Appendix D.12.4). Second, there could be a preassignment of this resource to a task running on this day, fixing the resource to that task. Third, even if the resource and day are open, there could still be a closed assignment, usually arising from a multi-day task, which is only opened if all the days it is running are open. Fourth, the resource could also have been assigned to a multi-day task yesterday (in `soln`); if so it must continue with that task today. `KheDrsSolnResourceIsAssigned` returns `true` if the resource is busy in `soln`, and `KheDrsTaskBusyOnDay` returns `true` if the task it is busy with then is still running. This is where the solver fails to handle tasks which run on multiple days but with gaps in the days: it looks for assignments to multi-day tasks only on the previous day.

`KheDrsResourceExpandBegin` begins the job of initializing the fields we saw earlier:

```
void KheDrsResourceExpandBegin(KHE_DRS_RESOURCE dr,
  KHE_DRS_SOLN prev_soln, KHE_DRS_DAY next_day,
  KHE_DRS_RESOURCE_SET free_resources,
  KHE_DRS_TASK_SOLN_SET fixed_assts, KHE_DRS_EXPANDER de)
{
  KHE_DRS_TASK_ON_DAY fixed_dtd;  KHE_DRS_RESOURCE_ON_DAY drd;
  KHE_DRS_TASK_SOLN dts;  KHE_DRS_MTASK_SOLN dms;
  KHE_DRS_TASK dt;  KHE_DRS_SIGNATURE prev_sig, sig;

  /* add signatures, and assignments to mtasks */
  drd = KheDrsResourceOnDay(dr, next_day);
  if( KheDrsResourceOnDayIsFixed(drd, prev_soln, de->solver, &fixed_dtd) )
  {
    /* mark the resource as fixed, and inform the expander */
    dr->expand_role = KHE_DRS_RESOURCE_EXPAND_FIXED;
    KheDrsExpanderDeleteFreeResource(de);

    /* fixed assignment to fixed_dtd; make and add sasst and dms */
    /* NB the optional addition is forced here, so sig is always referred to */
    prev_sig = HaArray(prev_soln->sig_set.signatures, dr->open_resource_index);
    sig = KheDrsResourceSignatureMake(drd, fixed_dtd, prev_sig, de->solver);
    dms = KheDrsResourceOptionallyAddMTaskSoln(sig, drd, NULL, fixed_dtd,
      true, de);
    KheDrsResourceAddExpandSignature(dr, sig);

    /* mark the task as fixed */
    if( fixed_dtd != NULL )
    {
      dt = fixed_dtd->encl_dt;
      HnAssert(dt->expand_role == KHE_DRS_TASK_EXPAND_NO_VALUE,
        "KheDrsResourceExpandBegin internal error 4 (role %s)",
        KheDrsTaskExpandRoleShow(dt->expand_role));
      dt->expand_role = KHE_DRS_TASK_EXPAND_FIXED;
    }

    /* make dts and add to fixed_assts */
    dts = KheDrsTaskSolnMake(dms, fixed_dtd);
    KheDrsTaskSolnSetAddLast(fixed_assts, dts);

    /* and build dominance cache */
    KheDrsResourceBuildDominanceTestCache(dr, drd, de->solver);
  }
  else
  {
    /* mark the resource as free */
    dr->expand_role = KHE_DRS_RESOURCE_EXPAND_FREE;

    /* add dr to free_resources */
    KheDrsResourceSetAddLast(free_resources, dr);
  }
}
```

`KheDrsResourceOnDayIsFixed` says whether `dr` is subject to a fixed assignment or not. If it is, then the full initialization of `dr` is done here; we'll explain the details in a moment. If it isn't, then just the minimum is done here, marking `dr` as free and adding it to `free_resources`, the set of all free resources, leaving the rest to a later call to `KheDrsResourceExpandBeginFree`.

When `dr` has a fixed assignment, `dr`'s role is set to fixed, and the expander is told that there is one fewer free resource than previously thought. A signature and mtask solution object are made for the fixed assignment and added to `dr`; we'll see the functions that do this in a moment. Then the enclosing task is marked as fixed, a task solution object is made and added to `fixed_assts`, and the dominance test cache (field `expand_dom_test_cache`) is initialized.

The code goes to some trouble to store only signatures and mtask solution objects that are actually useful. For a fixed assignment this is not needed, but it is done anyway, as follows. The call to `KheDrsResourceSignatureMake` makes a new signature object. Then `KheDrsResourceOptionallyAddMTaskSoln` makes and adds an mtask solution to `dr->expand_mtask_solns`, and returns it:

```
KHE_DRS_MTASK_SOLN KheDrsResourceOptionallyAddMTaskSoln(
  KHE_DRS_SIGNATURE sig, KHE_DRS_RESOURCE_ON_DAY drd, KHE_DRS_MTASK dmt,
  KHE_DRS_TASK_ON_DAY fixed_dtd, bool force, KHE_DRS_EXPANDER de)
{
  KHE_DRS_MTASK_SOLN res;
  if( force || KheDrsExpanderOpenToExtraCost(de, sig->cost) )
  {
    res = KheDrsMTaskSolnMake(sig, drd, dmt, fixed_dtd, de->solver);
    HaArrayAddLast(drd->encl_dr->expand_mtask_solns, res);
    return res;
  }
  else
    return NULL;
}
```

If `KheDrsExpanderOpenToExtraCost(de, sig->cost)` is false, a solution that uses this assignment has cost larger than the cost we are trying to improve on, so there is no point in creating the assignment object. This happens, for example, when the assignment violates a hard resource constraint. So nothing is created. Otherwise, the assignment is created and added to `dr->expand_mtask_solns`.

As an exception, the assignment object is created anyway when `force` is `true`. This is for fixed assignments, which our algorithm needs an assignment for, even if it is not competitive.

For adding the new signature to the resource, the call is

```
void KheDrsResourceAddExpandSignature(KHE_DRS_RESOURCE dr,
  KHE_DRS_SIGNATURE sig)
{
  HaArrayAddLast(dr->expand_signatures, sig);
  KheDrsSignatureRefer(sig);
}
```

This keeps sig's reference count up to date, as required.

After KheDrsResourceExpandBegin has been called for each of the open resources, KheDrsSolnExpand calls this function on each free resource:

```
void KheDrsResourceExpandBeginFree(KHE_DRS_RESOURCE dr,
  KHE_DRS_SOLN prev_soln, KHE_DRS_DAY next_day, KHE_DRS_EXPANDER de)
{
  int i, j;  KHE_DRS_RESOURCE_ON_DAY drd;  KHE_DRS_TASK_ON_DAY dtd;
  KHE_DRS_MTASK dmt;  KHE_DRS_SHIFT ds;  KHE_DRS_SIGNATURE prev_sig, sig;

  /* unfixed assignment; make signatures and mtask solns as required */
  HnAssert(dr->expand_role == KHE_DRS_RESOURCE_EXPAND_FREE,
    "KheDrsResourceExpandBeginFree internal error");
  drd = KheDrsResourceOnDay(dr, next_day);
  prev_sig = HaArray(prev_soln->sig_set.signatures, dr->open_resource_index);
  HaArrayForEach(next_day->shifts, ds, i)
  {
    sig = NULL;
    HaArrayForEach(ds->open_mtasks, dmt, j)
      if( KheDrsMTaskAcceptResourceBegin(dmt, drd, &dtd) )
      {
        if( sig == NULL )
          sig = KheDrsResourceSignatureMake(drd, dtd, prev_sig, de->solver);
        KheDrsResourceOptionallyAddMTaskSoln(sig, drd, dmt, NULL, false, de);
        KheDrsMTaskAcceptResourceEnd(dmt, dtd);
      }
    if( sig != NULL && !KheDrsSignatureOptionallyFree(sig, de->solver) )
      KheDrsResourceAddExpandSignature(dr, sig);
  }

  /* make one signature and mtask soln for a free day */
  sig = KheDrsResourceSignatureMake(drd, NULL, prev_sig, de->solver);
  dr->expand_free_mtask_soln =
    KheDrsResourceOptionallyAddMTaskSoln(sig, drd, NULL, NULL, false, de);
  if( !KheDrsSignatureOptionallyFree(sig, de->solver) )  /* yes, we need this */
    KheDrsResourceAddExpandSignature(dr, sig);

  /* sort expand_mtask_solns by increasing cost and keep only the best */
  KheDrsSortAndReduceMTaskSolns(dr, de->solver);

  /* move any unavoidable cost into the expander */
  KheDrsResourceAdjustSignatureCosts(dr, de);

  /* add a dominance test cache */
  KheDrsResourceBuildDominanceTestCache(dr, drd, de->solver);
}
```

Since `dr` is free, any mtask from any shift is a possible assignment, as is a free day. So this code iterates over all shifts and their mtasks, adding mtask solution objects where they are feasible, including for a free day, and signature objects where they are used.

At the end come two adjustments to the mtask solution objects. First is

```
void KheDrsSortAndReduceMTaskSolns(KHE_DRS_RESOURCE dr,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_MTASK_SOLN dms, dms2;  KHE_DRS_SIGNATURE sig, sig2;

  /* sort the mtask solutions by increasing cost */
  HaArraySort(dr->expand_mtask_solns, &KheDrsMTaskSolnCmp);

  /* if there is a resource expand limit and it will make a difference here */
  if( drs->solve_resource_expand_limit > 0 &&
      drs->solve_resource_expand_limit < HaArrayCount(dr->expand_mtask_solns) )
  {
    /* find dms, a signature with the largest cost that we want to keep */
    dms = HaArray(dr->expand_mtask_solns, drs->solve_resource_expand_limit-1);
    sig = KheDrsMTaskSolnSignature(dms);

    /* delete and free all signatures whose cost exceeds dms's */
    dms2 = HaArrayLast(dr->expand_mtask_solns);
    sig2 = KheDrsMTaskSolnSignature(dms2);
    while( sig2->cost > sig->cost )
    {
      HaArrayDeleteLast(dr->expand_mtask_solns);
      if( dms2 == dr->expand_free_mtask_soln )
        dr->expand_free_mtask_soln = NULL;
      KheDrsMTaskSolnFree(dms2, drs);
      dms2 = HaArrayLast(dr->expand_mtask_solns);
      sig2 = KheDrsMTaskSolnSignature(dms2);
    }
  }
}
```

This sorts the newly created mtask solution objects by increasing cost (their order does not matter). This is useful because it means that lower cost solutions are tried first. After that, if the `solve_resource_expand_limit` user option is in use (if its value is positive), it removes some of the assignments, the most costly ones, until the number remaining is approximately equal to `solve_resource_expand_limit`. Of course, this destroys the optimality guarantee.

The other function called at the end of `KheDrsResourceExpandBeginFree` is

```
void KheDrsResourceAdjustSignatureCosts(KHE_DRS_RESOURCE dr,
  KHE_DRS_EXPANDER de)
{
  KHE_COST movable_cost;  int i;
  KHE_DRS_MTASK_SOLN dms;  KHE_DRS_SIGNATURE sig;
  if( HaArrayCount(dr->expand_mtask_solns) > 0 )
  {
    dms = HaArrayFirst(dr->expand_mtask_solns);
    movable_cost = dms->sig->cost;
    KheDrsExpanderAddCost(de, movable_cost);
    HaArrayForEach(dr->expand_signatures, sig, i)
    {
      sig->cost -= movable_cost;
      HnAssert(sig->cost >= 0,
        "KheDrsResourceAdjustSignatureCosts internal error");
    }
  }
}
```

At least one of `dr`'s assignments must be used, even when `dr` is assigned a free day, and so, since the assignment to mtask objects are now sorted by non-decreasing cost, a cost at least equal to the cost of the first of them must be incurred. The code finds this cost and moves it out of the signatures and into the expander. This has zero net effect on cost, but a higher cost in the expander may lead to more pruning.

So much for starting off an expansion. Here is the function, called at the end of expansion, for clearing away the fields used by the expansion:

```
void KheDrsResourceExpandEnd(KHE_DRS_RESOURCE dr, KHE_DRS_EXPANDER de)
{
  KHE_DRS_MTASK_SOLN dms;

  /* mark the resource as not involved in any expansion */
  dr->expand_role = KHE_DRS_RESOURCE_EXPAND_NO;

  /* clear out the dominance test cache */
  if( USE_DOM_CACHING )
    KheDrsDim2TableClear(dr->expand_dom_test_cache, de->solver);

  /* clear signatures */
  KheDrsResourceClearExpandSignatures(dr, de->solver);

  /* clear assignments to mtasks */
  while( HaArrayCount(dr->expand_mtask_solns) > 0 )
  {
    dms = HaArrayLastAndDelete(dr->expand_mtask_solns);
    KheDrsMTaskSolnFree(dms, de->solver);
  }
  dr->expand_free_mtask_soln = NULL;
}
```

This puts these fields into the state assumed at the start of the next expansion.

## D.10.4. Initializing days, shifts, mtasks, and tasks for expansion

This section presents the code for setting up days, shifts, mtasks, and tasks for expansion.

Type `KHE_DRS_TASK` has one field relevant to expansion:

```
struct khe_drs_task_rec {
  ...
  KHE_DRS_TASK_EXPAND_ROLE              expand_role;
};
```

Its type is

```
typedef enum {
  KHE_DRS_TASK_EXPAND_NO_VALUE,
  KHE_DRS_TASK_EXPAND_FIXED,
  KHE_DRS_TASK_EXPAND_MUST,
  KHE_DRS_TASK_EXPAND_FREE
} KHE_DRS_TASK_EXPAND_ROLE;
```

This defines the role that this task (call it *t*) has in the current expansion, as follows.

`KHE_DRS_TASK_EXPAND_NO_VALUE`: there is no expansion in progress, or there is one but *t* does not participate in it.

KHE_DRS_TASK_EXPAND_FIXED: there is an expansion in progress, *t* participates in it, and *t* is a *fixed task*: it must be assigned a specific resource. Usually this will be because *t* is a multi-day task and it was assigned that resource on its first day. For the full story, consult KheDrsResourceOnDayIsFixed (Appendix D.10.3).

KHE_DRS_TASK_EXPAND_MUST: there is an expansion in progress, *t* participates in it, and *t* is a *must-assign task*: although it is not fixed to any particular resource, it must be assigned some resource, since the cost of leaving it unassigned is too great. For example, it is common for some tasks to be subject to hard constraints that require them to be assigned, and those would always become must-assign tasks in practice, except when they are fixed.

KHE_DRS_TASK_EXPAND_FREE: none of the other cases applies; an expansion is in progress, *t* participates in it, but *t* need not be assigned a resource, specific or otherwise.

Type KHE_DRS_MTASK has two relevant fields:

```
struct khe_drs_mtask_rec {
  ...
  int                            expand_must_assign_count;
  int                            expand_prev_unfixed;
};
```

Here expand_must_assign_count is the mtask's number of must-assign tasks: the number of tasks whose expand_role is KHE_DRS_TASK_EXPAND_MUST. The other field works as follows.

The mtask can be requested to give away one of its unfixed tasks (one whose expand_role is KHE_DRS_TASK_EXPAND_MUST or KHE_DRS_TASK_EXPAND_FREE) for assignment. This is what a call to KheDrsMTaskAcceptResourceBegin does:

```
bool KheDrsMTaskAcceptResourceBegin(KHE_DRS_MTASK dmt,
  KHE_DRS_RESOURCE_ON_DAY drd, KHE_DRS_TASK_ON_DAY *dtd)
{
  KHE_DRS_TASK dt;  int i, count;

  count = HaArrayCount(dmt->unassigned_tasks);
  for( i = dmt->expand_prev_unfixed + 1;  i < count;  i++ )
  {
    dt = HaArray(dmt->unassigned_tasks, i);
    if( dt->expand_role != KHE_DRS_TASK_EXPAND_FIXED )
    {
      if( !KheDrsTaskRunningOnDay(dt, drd->day, dtd) )
        HnAbort("KheDrsMTaskAcceptResourceBegin internal error");
      dmt->expand_prev_unfixed = i;
      return true;
    }
  }

  /* if we get here we've failed to identify a suitable task */
  return *dtd = NULL, false;
}
```

It searches forwards along its unassigned_tasks array for the next unfixed task. If it finds such a task, it increases dmt->expand_prev_unfixed to its index, sets *dtd to the relevant task on day, and returns true. Otherwise it sets *dtd to NULL and returns false.

When expansion no longer needs a task that it previously successfully requested using KheDrsMTaskAcceptResourceBegin, it calls KheDrsMTaskAcceptResourceEnd to return it:

```
void KheDrsMTaskAcceptResourceEnd(KHE_DRS_MTASK dmt,
  KHE_DRS_TASK_ON_DAY dtd)
{
  KHE_DRS_TASK dt;  int i;
  HnAssert(dmt->expand_prev_unfixed < HaArrayCount(dmt->unassigned_tasks) &&
    dtd->encl_dt == HaArray(dmt->unassigned_tasks, dmt->expand_prev_unfixed),
    "KheDrsMTaskAcceptResourceEnd internal error");

  /* search backwards for next unfixed task, or start of array */
  for( i = dmt->expand_prev_unfixed - 1;  i >= 0;  i-- )
  {
    dt = HaArray(dmt->unassigned_tasks, dmt->expand_prev_unfixed);
    if( dt->expand_role != KHE_DRS_TASK_EXPAND_FIXED )
      break;
  }
  dmt->expand_prev_unfixed = i;
}
```

This reduces dmt->expand_prev_unfixed to the index of the previous unfixed task, or to -1 (also the initial value of dmt->expand_prev_unfixed) when there is no previous unfixed task. The mtask does not need to record which tasks it has given away, because it gives them away in the order they appear in unassigned_tasks, and receives them back in reverse order.

In KHE_DRS_SHIFT there are two relevant fields:

```
struct khe_drs_shift_rec {
  ...
  int                     expand_must_assign_count;
  int                     expand_max_included_free_resource_count;
};
```

Field expand_must_assign_count holds the total number of must-assign tasks lying within the shift's mtasks. Field expand_max_included_free_resource_count holds the maximum number of free resources (open resources not fixed to a particular task during this expansion) that can be assigned to tasks of this shift without leaving too few free resources available to cover the must-assign tasks of other shifts. Importantly, expand_must_assign_count excludes fixed tasks, and expand_max_included_free_resource_count excludes fixed resources.

All these fields are initialized at the start of expansion by the call to

```
void KheDrsDayExpandBegin(KHE_DRS_DAY next_day, KHE_DRS_SOLN prev_soln,
  KHE_DRS_DAY prev_day, KHE_DRS_EXPANDER de)
{
  int i, excess;  KHE_DRS_SHIFT ds;

  /* begin caching expansions */
  KheDrsSolnSetBeginCacheSegment(next_day->soln_set, de->solver);

  /* begin expansion in each shift */
  HaArrayForEach(next_day->shifts, ds, i)
    KheDrsShiftExpandBegin(ds, prev_soln, prev_day, de);

  /* set expand_max_included_free_resource_count in each shift */
  excess = KheDrsExpanderExcessFreeResourceCount(de);
  HaArrayForEach(next_day->shifts, ds, i)
    ds->expand_max_included_free_resource_count =
      ds->expand_must_assign_count + excess;
}
```

This tells `next_day->soln_set` that an expansion is beginning, so that it can initialize its cache if desired. Then it calls `KheDrsShiftExpandBegin` (see below) for each shift of `next_day`. The last part sets `expand_max_included_free_resource_count` in each shift `ds` to hold the maximum number of free resources that can be assigned to tasks of `ds` without leaving too few free resources available for the must-assign tasks of other shifts. The reader can confirm this.

At the end of the expansion the opposite function is called:

```
void KheDrsDayExpandEnd(KHE_DRS_DAY next_day, KHE_DRS_EXPANDER de)
{
  int i;  KHE_DRS_SHIFT ds;

  /* end expansion in each shift */
  HaArrayForEach(next_day->shifts, ds, i)
    KheDrsShiftExpandEnd(ds, de);

  /* end caching expansions */
  KheDrsSolnSetEndCacheSegment(next_day->soln_set, next_day, de, de-
>solver);
}
```

This tells `next_day`'s shifts and solution set that the current expansion is ending.

Here is how one shift is initialized for expansion:

```
void KheDrsShiftExpandBegin(KHE_DRS_SHIFT ds, KHE_DRS_SOLN prev_soln,
  KHE_DRS_DAY prev_day, KHE_DRS_EXPANDER de)
{
  KHE_DRS_MTASK dmt;  int i;
  ds->expand_must_assign_count = 0;
  ds->expand_max_included_free_resource_count = 0;
  HaArrayForEach(ds->open_mtasks, dmt, i)
    KheDrsMTaskExpandBegin(dmt, prev_soln, prev_day, de);
}
```

This begins by initializing the two expansion fields to placeholder values. We'll see shortly how `ds->expand_must_assign_count` receives its true value, and we have already seen, just above, how `ds->expand_max_included_free_resource_count` receives its true value. The function then tells each mtask that an expansion is beginning. The function for ending expansion is

```
void KheDrsShiftExpandEnd(KHE_DRS_SHIFT ds, KHE_DRS_EXPANDER de)
{
  KHE_DRS_MTASK dmt;  int i;
  HaArrayForEach(ds->open_mtasks, dmt, i)
    KheDrsMTaskExpandEnd(dmt, de);
  KheDrsShiftSolnTrieFree(ds->soln_trie, de->solver);
  ds->soln_trie = NULL;
}
```

This tells each mtask that the expansion is ending. Elsewhere we present a second function for initializing shifts for expansion, one which initializes the shift solution tries used by expand by shifts. `KheDrsShiftSolnTrieFree` (Appendix D.9.7) removes these tries.

For informing an mtask that expansion is beginning, the code is

```
void KheDrsMTaskExpandBegin(KHE_DRS_MTASK dmt,
  KHE_DRS_SOLN prev_soln, KHE_DRS_DAY prev_day, KHE_DRS_EXPANDER de)
{
  KHE_DRS_TASK dt;  int i;
  dmt->expand_must_assign_count = 0;
  HaArrayForEach(dmt->unassigned_tasks, dt, i)
    if( dt->expand_role != KHE_DRS_TASK_EXPAND_FIXED )
    {
      if( KheDrsExpanderOpenToExtraCost(de, dt->non_asst_cost) )
        dt->expand_role = KHE_DRS_TASK_EXPAND_FREE;
      else
      {
        /* dt must be assigned, otherwise cost will be too high */
        dt->expand_role = KHE_DRS_TASK_EXPAND_MUST;
        dmt->expand_must_assign_count++;
        dmt->encl_shift->expand_must_assign_count++;
        KheDrsExpanderAddMustAssign(de);
      }
    }
}
```

This visits each unassigned unfixed task of `dmt`, setting its `expand_role` field, and ensuring that the `expand_must_assign_count` fields of the enclosing mtask and shift hold the correct total number of must-assign tasks, and that `de` holds the number of must-assign tasks in all shifts.

This code assumes that `KheDrsResourceExpandBegin` has already been called for each open resource, so that fixed tasks have already been discovered and had their `expand_role` fields set. It is done this way because the solver's data structures are much better at deciding whether a given open resource is fixed, and if so which task it is fixed to, than they are at deciding whether a given task is fixed, and if so which resource it is fixed to.

After expansion is complete, the opposite function is called:

```
void KheDrsMTaskExpandEnd(KHE_DRS_MTASK dmt, KHE_DRS_EXPANDER de)
{
  KHE_DRS_TASK dt;  int i;
  HaArrayForEach(dmt->unassigned_tasks, dt, i)
    dt->expand_role = KHE_DRS_TASK_EXPAND_NO_VALUE;
}
```

There is no expansion now so all roles are `KHE_DRS_TASK_EXPAND_NO_VALUE`.

### D.10.5. Expansion by resources

Expansion by resources is carried out by function `KheDrsSolnExpandByResources`:

```
void KheDrsSolnExpandByResources(KHE_DRS_SOLN prev_soln,
  KHE_DRS_DAY prev_day, KHE_DRS_DAY next_day, KHE_DRS_EXPANDER de,
  KHE_DRS_RESOURCE_SET free_resources, int free_resources_index)
{
  KHE_DRS_RESOURCE dr;  int i;  KHE_DRS_MTASK_SOLN dms;

  if( free_resources_index >= KheDrsResourceSetCount(free_resources) )
  {
    /* de has enough mtask solns to make into a soln and evaluate */
    KheDrsExpanderMakeAndMeldSoln(de, prev_soln, next_day);
  }
  else
  {
    dr = KheDrsResourceSetResource(free_resources, free_resources_index);
    HaArrayForEach(dr->expand_mtask_solns, dms, i)
      KheDrsMTaskSolnExpandByResources(dms, prev_soln, prev_day,
        next_day, de, free_resources, free_resources_index);
  }
}
```

This expands `prev_soln` into `next_day` in all possible ways, assuming that all possibilities have been explored for the free resources of `free_resources` whose indexes in `free_resources` are less than `free_resource_index`.

If `free_resources_index >= KheDrsResourceSetCount(free_resources)`, the expander has a current assignment for each open resource, so `KheDrsExpanderMakeAndMeldSoln` (Appendix D.10.1) is called to make a day solution from those assignments and possibly add it to `next_day`'s solution set. Otherwise, the code finds the next unassigned free resource `dr`, and for each of `dr`'s mtask solution objects that `KheDrsResourceExpandBegin` created previously, it continues the recursion using that assignment:

```
void KheDrsMTaskSolnExpandByResources(KHE_DRS_MTASK_SOLN dms,
  KHE_DRS_SOLN prev_soln, KHE_DRS_DAY prev_day, KHE_DRS_DAY next_day,
  KHE_DRS_EXPANDER de, KHE_DRS_RESOURCE_SET free_resources,
  int free_resources_index)
{
  KHE_DRS_TASK_ON_DAY dtd;  KHE_DRS_MTASK dmt;  KHE_DRS_TASK_SOLN dts;
  dmt = dms->mtask;
  if( dmt != NULL )
  {
    /* select a task from dms->mtask and assign it */
    if( KheDrsMTaskAcceptResourceBegin(dmt, dms->resource_on_day, &dtd) )
    {
      dts = KheDrsTaskSolnMake(dms, dtd);
      KheDrsTaskSolnExpandByResources(dts, prev_soln, prev_day, next_day,
        de, free_resources, free_resources_index);
      KheDrsMTaskAcceptResourceEnd(dmt, dtd);
    }
  }
  else
  {
    /* use dms->fixed_task_on_day, possibly NULL meaning a free day */
    dts = KheDrsTaskSolnMake(dms, dms->fixed_task_on_day);
    KheDrsTaskSolnExpandByResources(dts, prev_soln, prev_day, next_day,
      de, free_resources, free_resources_index);
  }
}
```

If `dmt != NULL`, the assignment is to an unspecified task of mtask `dmt`. It is now time to choose a specific task, which is done by the calls to `KheDrsMTaskAcceptResourceBegin` and `KheDrsMTaskAcceptResourceEnd`. The call to `KheDrsTaskSolnExpandByResources` carries on the recursion using the task solution built from `dms` and the task on day object returned by a successful call to `KheDrsMTaskAcceptResourceBegin`.

If `dmt == NULL`, the assignment is to `dms->fixed_task_on_day`, a specific task on day. `KheDrsTaskSolnExpandByResources` is called with a task solution for this task on day.

Either way, `KheDrsTaskSolnExpandByResources` has a specific task solution to add to the expander:

```
void KheDrsTaskSolnExpandByResources(KHE_DRS_TASK_SOLN dts,
  KHE_DRS_SOLN prev_soln, KHE_DRS_DAY prev_day, KHE_DRS_DAY next_day,
  KHE_DRS_EXPANDER de, KHE_DRS_RESOURCE_SET free_resources,
  int free_resources_index)
{
  /* save the expander so it can be restored later */
  KheDrsExpanderMarkBegin(de);

  /* add dts to the expander */
  KheDrsExpanderAddTaskSoln(de, dts);

  /* if the expander is still open, recurse */
  if( KheDrsExpanderIsOpen(de) )
    KheDrsSolnExpandByResources(prev_soln, prev_day, next_day, de,
      free_resources, free_resources_index + 1);

  /* restore the expander */
  KheDrsExpanderMarkEnd(de);
}
```

It adds `dts` to the expander, then, if the expander is still open, it makes a recursive call to `KheDrsSolnExpandByResources`, moving on to the next free resource. After that the change is undone by the call to `KheDrsExpanderMarkEnd`. All very simple, thanks to the expander.

### D.10.6. Expansion by shifts

`KheDrsSolnExpand` carries out expansion by shifts by calling `KheDrsSolnExpandByShifts`:

```
void KheDrsSolnExpandByShifts(KHE_DRS_SOLN prev_soln,
  KHE_DRS_DAY next_day, KHE_DRS_EXPANDER de,
  KHE_DRS_RESOURCE_SET free_resources, int shift_index)
{
  KHE_DRS_SHIFT ds;  int i;  KHE_DRS_RESOURCE dr;  KHE_DRS_TASK_SOLN dts;
  if( shift_index >= HaArrayCount(next_day->shifts) )
  {
    /* assign a free day to each remaining free resource; first, */
    /* abandon this path if any free resource has no free day asst */
    KheDrsResourceSetForEach(free_resources, dr, i)
      if( dr->expand_free_mtask_soln == NULL )
        return;

    /* save the expander */
    KheDrsExpanderMarkBegin(de);

    /* add free day assignments to the expander */
    KheDrsResourceSetForEach(free_resources, dr, i)
    {
      dts = KheDrsTaskSolnMake(dr->expand_free_mtask_soln, NULL);
      KheDrsExpanderAddTaskSoln(de, dts);
    }

    /* if the expander is still open then make the solution */
    if( KheDrsExpanderIsOpen(de) )
      KheDrsExpanderMakeAndMeldSoln(de, prev_soln, next_day);

    /* restore the expander */
    KheDrsExpanderMarkEnd(de);
  }
  else
  {
    ds = HaArray(next_day->shifts, shift_index);
    KheDrsShiftExpandByShifts(ds, shift_index, prev_soln, next_day,
      de, free_resources);
  }
}
```

Only the free (non-fixed) resources, held in `free_resources`, need to be assigned; assignments to all fixed resources have already been added to `de` by `KheDrsSolnExpand`.

When `shift_index >= HaArrayCount(next_day->shifts)`, by now each shift has been assigned a set of resources, and the remaining free resources have to be assigned a free day.

The first step of this case is to check that these resources have free day assignments, held in their `expand_free_mtask_soln` fields. If not, one or more of them has to be assigned some task and this has not happened, so we've reached a dead end and the code returns early. If this were to occur often then a lot of running time would be wasted, but in fact it happens only rarely.

The next step is to mark the expander, then update it with the free day assignments. Then, if the expander is still open, `KheDrsExpanderMakeAndMeldSoln` (Appendix D.10.1) is called to make a solution from the current set of assignments and add it, with dominance testing, to `next_day`'s solution set. Then the assignments are removed and the expander state is restored.

When `shift_index < HaArrayCount(next_day->shifts)`, we still have shifts to assign sets of free resources to, beginning with the shift whose index in `next_day` is `shift_index`. We set `ds` to this shift and call `KheDrsShiftExpandByShifts`:

```
void KheDrsShiftExpandByShifts(KHE_DRS_SHIFT ds, int shift_index,
  KHE_DRS_SOLN prev_soln, KHE_DRS_DAY next_day, KHE_DRS_EXPANDER de,
  KHE_DRS_RESOURCE_SET free_resources)
{
  KHE_DRS_RESOURCE_SET omitted_resources;
  if( ds->soln_trie != NULL )
  {
    omitted_resources = KheDrsResourceSetMake(de->solver);
    KheDrsShiftSolnTrieExpandByShifts(ds->soln_trie, ds, shift_index,
      prev_soln, next_day, de, free_resources, 0, 0, omitted_resources);
    KheDrsResourceSetFree(omitted_resources, de->solver);
  }
}
```

This is a wrapper for `KheDrsShiftSolnTrieExpandByShifts`. It creates a set of resource objects, `omitted_resources`, for passing to `KheDrsShiftSolnTrieExpandByShifts`.

Before we present `KheDrsShiftSolnTrieExpandByShifts` we'll examine its header:

```
void KheDrsShiftSolnTrieExpandByShifts(KHE_DRS_SHIFT_SOLN_TRIE dsst,
  KHE_DRS_SHIFT ds, int shift_index, KHE_DRS_SOLN prev_soln,
  KHE_DRS_DAY next_day, KHE_DRS_EXPANDER de,
  KHE_DRS_RESOURCE_SET free_resources, int free_index,
  int selected_resource_count, KHE_DRS_RESOURCE_SET omitted_resources);
```

We'll refer to the *available resources*, meaning the elements of `free_resources`, but only those whose index in `free_resources` is `free_index` or larger.

Three conditions restrict the values of the parameters. First, `ds` is the shift beginning on `next_day` whose index in `next_day` is `shift_index`. Second, `dsst` is the subtrie that is reached using the indexes of the *selected resources*, that is, the resources selected so far for this shift. We don't keep track of those resources explicitly (although we could), because `dsst` itself does that sufficiently for our purposes. Third, the available resources as just defined, the selected resources as just defined, and the omitted resources are pairwise disjoint and their union contains every free resource. The reader can verify that these conditions hold for the initial call to `KheDrsShiftSolnTrieExpandByShifts`, the one from within `KheDrsShiftExpandByShifts`.

`KheDrsShiftSolnTrieExpandByShifts` must assign the selected resources to `ds`, but it is free to also assign any subset of the available resources to `ds` as well. It tries assigning to `ds` each subset of the set of free resources that satisfies these conditions. For each of these subsets *R*, for each of the undominated shift solutions held in the trie node for *R*, it tries each undominated shift solution in turn, recursing to assign the remaining shifts:

```
void KheDrsShiftSolnTrieExpandByShifts(KHE_DRS_SHIFT_SOLN_TRIE dsst,
  KHE_DRS_SHIFT ds, int shift_index, KHE_DRS_SOLN prev_soln,
  KHE_DRS_DAY next_day, KHE_DRS_EXPANDER de,
  KHE_DRS_RESOURCE_SET free_resources, int free_index,
  int selected_resource_count, KHE_DRS_RESOURCE_SET omitted_resources)
{
  int i, avail_resource_count;  KHE_DRS_SHIFT_SOLN dss;
  KHE_DRS_SHIFT_SOLN_TRIE child_dsst;  KHE_DRS_RESOURCE dr;
  avail_resource_count =
    KheDrsResourceSetCount(free_resources) - free_index;
  if( avail_resource_count <= 0 )
  {
    /* resources all done, so try each solution in dsst */
    HaArrayForEach(dsst->shift_solns, dss, i)
      KheDrsShiftSolnExpandByShifts(dss, ds, shift_index, prev_soln,
        next_day, de, omitted_resources);
  }
  else
  {
    /* try solutions that select dr, the next available resource */
    dr = KheDrsResourceSetResource(free_resources, free_index);
    child_dsst = HaArray(dsst->children, dr->open_resource_index);
    if( child_dsst != NULL )
      KheDrsShiftSolnTrieExpandByShifts(child_dsst, ds, shift_index,
        prev_soln, next_day, de, free_resources, free_index + 1,
        selected_resource_count + 1, omitted_resources);

    /* try solutions that do not select dr, staying in dsst */
    if( selected_resource_count + avail_resource_count >
          ds->expand_must_assign_count )
    {
      KheDrsResourceSetAddLast(omitted_resources, dr);
      KheDrsShiftSolnTrieExpandByShifts(dsst, ds, shift_index,
        prev_soln, next_day, de, free_resources, free_index + 1,
        selected_resource_count, omitted_resources);
      KheDrsResourceSetDeleteLast(omitted_resources);
    }
  }
}
```

If `avail_resource_count <= 0`, then then all available resources are selected or omitted, and it is time to try the shift assignments of `dsst->shift_solns`. So for each of them we call `KheDrsShiftSolnExpandByShifts`. We'll return to that shortly.

Otherwise, we need to try subsets *R* that include the next available resource `dr`, and also subsets *R* that omit it. For the first we have

```
/* try solutions that select dr, the next available resource */
dr = KheDrsResourceSetResource(free_resources, free_index);
child_dsst = HaArray(dsst->children, dr->open_resource_index);
if( child_dsst != NULL )
  KheDrsShiftSolnTrieExpandByShifts(child_dsst, ds, shift_index,
    prev_soln, next_day, de, free_resources, free_index + 1,
    selected_resource_count + 1, omitted_resources);
```

This accesses the child `child_dsst` whose index is the next available resource's open resource index. If `child_dsst != NULL`, we recurse, adding `dr` implicitly to the set of selected resources by utilizing the child node representing it, and removing `dr` explicitly from the available resources by increasing `free_index`. For omitting `dr` the code is

```
/* try solutions that do not select dr, staying in dsst */
if( selected_resource_count + avail_resource_count >
      ds->expand_must_assign_count )
{
  KheDrsResourceSetAddLast(omitted_resources, dr);
  KheDrsShiftSolnTrieExpandByShifts(dsst, ds, shift_index,
    prev_soln, next_day, de, free_resources, free_index + 1,
    selected_resource_count, omitted_resources);
  KheDrsResourceSetDeleteLast(omitted_resources);
}
```

This adds `dr` to the set of omitted resources, and again recurses with `free_index` incremented to remove `dr` from the set of available resources. It recurses on the same trie node, `dsst`.

Next comes `KheDrsShiftSolnExpandByShifts`, called when all available resources have either been selected or omitted, and we have reached a specific shift solution `dss`:

```
void KheDrsShiftSolnExpandByShifts(KHE_DRS_SHIFT_SOLN dss,
  KHE_DRS_SHIFT ds, int shift_index, KHE_DRS_SOLN prev_soln,
  KHE_DRS_DAY next_day, KHE_DRS_EXPANDER de,
  KHE_DRS_RESOURCE_SET omitted_resources)
{
  KHE_DRS_SHIFT_SOLN dss2;  int i;
  if( dss->skip_count == 0 )
  {
    /* save the expander */
    KheDrsExpanderMarkBegin(de);

    /* add the assignments stored in dss to the expander */
    KheDrsExpanderAddTaskSolnSet(de, dss->task_solns, NULL);

    /* if the expander is still open, recurse */
    if( KheDrsExpanderIsOpen(de) )
    {
      HaArrayForEach(dss->skip_assts, dss2, i)
        dss2->skip_count++;
      KheDrsSolnExpandByShifts(prev_soln, next_day, de, omitted_resources,
        shift_index + 1);
      HaArrayForEach(dss->skip_assts, dss2, i)
        dss2->skip_count--;
    }

    /* restore the expander */
    KheDrsExpanderMarkEnd(de);
  }
}
```

We mark the expander, add `dss`'s task solutions, recurse on the next shift (with index `shift_index + 1`) if the expander is still open, then restore the expander. The free resources for the next shift are the omitted resources for this shift. We made sure previously that `dss` contains no fixed assignments, which is just as well because all fixed assignments have already been added to the expander, before expand by shifts begins.

This code also implements shift pair dominance by not expanding using shifts for which `skip_count > 0`, and by incrementing the approppriate `skip_count` fields as appropriate when `dss` is in use. I should look into whether the expander could take over this work, by offering an operation to add a shift solution, not just a shift solution's task solution set.

## D.11. Sets of solutions

A *solution set* is a set of undominated solutions $P_k$ for some day $d_k$. Type `KHE_DRS_SOLN_SET` represents a solution set in the implementation.

As mere collections of solutions, solution sets should be very simple. However, there are three complications. First, the operation for adding a new solution $x$ to a solution set has to check for dominance relationships between $x$ and the other solutions. This involves three steps:

1. Check whether $P_k$ contains a solution $y$ that dominates $x$. If so, delete $x$ and stop;
2. Remove from $P_k$ and delete all solutions $y$ such that $x$ dominates $y$;
3. Add $x$ to $P_k$.

Since this is not a normal add-to-collection operation we call it a *meld*. Second, because melding is potentially slow, the solver offers alternative kinds of dominance testing, including alternative collection data structures. And third, there is the option of *caching*, which involves having two collections, the *main solution set* holding most of the solutions, and a *cache solution set* holding a smaller number of recently inserted solutions.

In this section we work bottom-up through a variety of collection data structures that combine to implement all these variations of the basic idea within type `KHE_DRS_SOLN_SET`.

## D.11.1. Solution lists

Type `KHE_DRS_SOLN_LIST` defines a simple list of solutions, stored in an array:

```
typedef struct khe_drs_soln_list_rec *KHE_DRS_SOLN_LIST;
typedef HA_ARRAY(KHE_DRS_SOLN_LIST) ARRAY_KHE_DRS_SOLN_LIST;
typedef HP_TABLE(KHE_DRS_SOLN_LIST) TABLE_KHE_DRS_SOLN_LIST;

struct khe_drs_soln_list_rec {
  KHE_DYNAMIC_RESOURCE_SOLVER   solver;
  ARRAY_KHE_DRS_SOLN            solns;
};
```

This is basically just a simple array of day solution objects.

One type of dominance testing, called medium dominance, requires a hash table whose elements are solution lists, and that is what `TABLE_KHE_DRS_SOLN_LIST` provides. It uses the `HP_TABLE` type definition macro from Appendix A. This hash table also needs access to a solver object, or something similar, since its hash function uses a signer to determine which elements of the signature to hash. If these hash tables were removed from the implementation (as could easily be done, since they are obsolete now) the `solver` field could be deleted.

Function `KheDrsSolnListMake` makes a new, empty solution list; `KheDrsSolnListFree` frees a solution list without freeing its solutions; and `KheDrsSolnListFreeSolns` frees the solutions of a given solution list, without freeing the solution list object.

A more interesting operation is

```
void KheDrsSolnListGather(KHE_DRS_SOLN_LIST soln_list,
  KHE_DRS_SOLN_LIST res)
{
  int i;
  HaArrayAppend(res->solns, soln_list->solns, i);
}
```

Every data structure holding a collection of solutions has such a ‘gather’ operation. It adds the collection's solutions (here `soln_list`) to a given solution list (here `res`). This is how we extract the solutions from complex data structures (tries, etc.): we gather them into a solution list.

Here is another operation found in all collection types. It decides whether solution list `soln_list` dominates `soln`, by which we mean contains a solution which dominates `soln`:

```
bool KheDrsSolnListDominates(KHE_DRS_SOLN_LIST soln_list,
  KHE_DRS_SOLN soln, KHE_DRS_SIGNER_SET signer_set,
  int *dom_test_count, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_SOLN other_soln;  int i;
  HaArrayForEach(soln_list->solns, other_soln, i)
    if( KheDrsSolnDominates(other_soln, soln, signer_set,
        dom_test_count, drs, 0, 0, NULL) )
      return true;
  return false;
}
```

This implements Step 1 of the meld operation when the collection is a solution list, except for deleting and freeing `soln` if it returns `true`.

For Step 2 of the meld operation we have

```
void KheDrsSolnListRemoveDominated(KHE_DRS_SOLN_LIST soln_list,
  KHE_DRS_SOLN soln, KHE_DRS_SIGNER_SET signer_set,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_SOLN other_soln;  int i, dom_test_count;
  dom_test_count = 0;
  HaArrayForEach(soln_list->solns, other_soln, i)
    if( KheDrsSolnNotExpanded(other_soln) && KheDrsSolnDominates(soln,
          other_soln, signer_set, &dom_test_count, drs, 0, 0, NULL) )
    {
      KheDrsPriQueueDeleteSoln(drs, other_soln);
      KheDrsSolnFree(other_soln, drs);
      HaArrayDeleteAndPlug(soln_list->solns, i);
      i--;
    }
}
```

This deletes and frees all elements of `soln_list` that are dominated by `soln`. The calls to `KheDrsSolnNotExpanded` and `KheDrsPriQueueDeleteSoln` will be explained later; they are needed when the priority queue is in use, and do nothing when it isn't.

For Step 3 of the meld operation, simply adding a solution to a solution list, we have

```
void KheDrsSolnListAddSoln(KHE_DRS_SOLN_LIST soln_list,
  KHE_DRS_SOLN soln, KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  HaArrayAddLast(soln_list->solns, soln);
  KheDrsPriQueueAddSoln(drs, soln);
}
```

Again, `KheDrsPriQueueAddSoln` does nothing if there is no priority queue.

The next operation sorts a solution list by increasing solution cost, and optionally deletes and frees the most expensive solutions, depending on an option:

```
int KheDrsSolnCmp(const void *t1, const void *t2)
{
  KHE_DRS_SOLN soln1 = * (KHE_DRS_SOLN *) t1;
  KHE_DRS_SOLN soln2 = * (KHE_DRS_SOLN *) t2;
  return KheCostCmp(KheDrsSolnCost(soln1), KheDrsSolnCost(soln2));
}

void KheDrsSolnListSortAndReduce(KHE_DRS_SOLN_LIST soln_list,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_SOLN soln;  int i;
  HaArraySort(soln_list->solns, &KheDrsSolnCmp);
  if( drs->solve_daily_expand_limit > 0 )
    while( HaArrayCount(soln_list->solns) > drs->solve_daily_expand_limit )
    {
      soln = HaArrayLastAndDelete(soln_list->solns);
      KheDrsPriQueueDeleteSoln(drs, soln);
      KheDrsSolnFree(soln, drs);
    }
}
```

As usual, dropping solutions is a heuristic that gives up any optimality guarantee.

The remaining solution list operations are mostly hash code calculations for the hash table, and debug functions. There is one exception, however:

```
void KheDrsSolnListExpand(KHE_DRS_SOLN_LIST soln_list,
  KHE_DRS_DAY prev_day, KHE_DRS_DAY next_day,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_SOLN prev_soln;  int i;
  HaArrayForEach(soln_list->solns, prev_soln, i)
    KheDrsSolnExpand(prev_soln, prev_day, next_day, drs);
}
```

To build $P_{k+1}$ from $P_k$, we traverse $P_k$ and expand each of its solutions. `KheDrsSolnListExpand` does this when the collection is held in a solution list.

### D.11.2. Other collection data structures

In attempting to speed up dominance testing, the author has tried two data structures other than a simple list for holding solutions. This section is a brief tour through these two data structures. The interest-to-code ratio is rather low, so we show only one code sample.

The first data structure is a *trie*. A traditional trie is a kind of tree data structure for holding objects retrieved by a key which is a string of characters. In the root of the tree is an array of subtrees indexed by the first character of the key. For example, all objects whose key begins with `'a'` might be in the first subtree, all whose key begins with `'b'` might be in the second subtree, and so on. The root of each subtree also has an array of subtrees, this time indexed by the second character of the key, and so on. To retrieve an object by key, use the first character of the key to find a subtree, then use the second character to find a sub-subtree, and so on.

A solution's signature is an array of (usually) small integers, ideal for tries. If we store the solutions in a trie we can easily retrieve by signature. We don't need to do that, but when testing for dominance using strong dominance, we may need all the solutions whose first element is no larger than a given element, or no smaller. Tries are excellent for this.

The implementation has the usual collection operations, for making and freeing tries, gathering a trie's solutions into a solution list, and so on. It all works, but tends not to be used, because tries do not combine well with tabulated dominance.

The other non-trivial collection type is the *indexed solution set*, or just *indexed set*. It is an array of solution lists indexed by solution cost. All solutions with a given cost appear in the one list, and that list is accessed by using their common cost as an index in the array.

It would not be efficient to index using the cost as is. Instead, there is a formula for converting a cost to an index:

```
    index = (cost - base) / increment;
```

Here `base` is the smallest cost that occurs in the set, and `increment` is the least common divisor of all the constraint weights. For example, if all constraint weights are multiples of 5, then `increment` is 5. Here `base` is updated whenever a solution whose cost is a new minimum is added to the set, while `increment` is calculated once and for all when the solver is created.

Again there are the usual operations for indexed set creating, freeing, gathering, and so on. Indexed sets work well with tradeoff dominance and tabulated dominance, because those tests need access to all solutions whose cost is at most a given cost, or all solutions whose cost is at least a given cost. These can be found by traversing the array positions equal to and to the left of the index of the given cost, or equal to and to the right. As an example, here is the operation for deciding whether an indexed set `iss` contains a solution which dominates `soln`:

```
bool KheDrsSoftIndexedSolnSetDominates(
  KHE_DRS_SOFT_INDEXED_SOLN_SET siss, KHE_DRS_SOLN soln,
  KHE_DRS_DAY soln_day, int *dom_test_count,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  int i, pos;  KHE_DRS_SOLN_LIST soln_list;  int soft_cost;
  KheDrsSoftIndexedSolnSetCheck(siss);
  *dom_test_count = 0;
  if( HaArrayCount(siss->soln_lists) == 0 )
    return false;
  else
  {
    soft_cost = KheSoftCost(KheDrsSolnCost(soln));
    pos = (soft_cost - siss->base) / siss->increment;
    if( pos >= HaArrayCount(siss->soln_lists) )
      pos = HaArrayCount(siss->soln_lists) - 1;
    for( i = 0;  i <= pos;  i++ )
    {
      soln_list = HaArray(siss->soln_lists, i);
      if( soln_list != NULL && KheDrsSolnListDominates(soln_list, soln,
          soln_day->signer_set, dom_test_count, drs) )
        return true;
    }
  }
  return false;
}
```

It only traverses the array positions up to the index of `soln`'s cost in the array, because a solution which dominates `soln` must have a cost which is no larger than `soln->cost`.

`KheDrsSoftIndexedSolnSetDominates` assumes that all costs are soft costs. The indexed set data structure is actually a two-level structure. At the higher level is an indexed set which deals only with hard costs (type `KHE_DRS_HARD_INDEXED_SOLN_SET`); its elements are indexed sets which deal only with soft costs (type `KHE_DRS_SOFT_INDEXED_SOLN_SET`).

### D.11.3. Solution set parts

A *solution set part* is yet another collection of solutions, represented by type `KHE_DRS_SOLN_SET_PART`. Why this is not the same as `KHE_DRS_SOLN_SET` is a fair question that we will have to answer later, given than we are working bottom-up.

Type `KHE_DRS_SOLN_SET_PART` is an abstract supertype with a rather large number of concrete subtypes. The type tag that distinguishes these subtypes is no other than public type `KHE_DRS_DOM_KIND`, that we have seen before:

```
typedef enum {
  KHE_DRS_DOM_LIST_NONE,
  KHE_DRS_DOM_LIST_SEPARATE,
  KHE_DRS_DOM_LIST_TRADEOFF,
  KHE_DRS_DOM_LIST_TABULATED,
  KHE_DRS_DOM_HASH_EQUALITY,
  KHE_DRS_DOM_HASH_MEDIUM,
  /* KHE_DRS_DOM_TRIE_SEPARATE, */
  /* KHE_DRS_DOM_TRIE_TRADEOFF, */
  KHE_DRS_DOM_INDEXED_TRADEOFF,
  KHE_DRS_DOM_INDEXED_TABULATED
} KHE_DRS_DOM_KIND;
```

Each subtype knows which kind of dominance testing to use, if any (none, strong, tradeoff, or uniform), and which kind of data structure to use (a simple list, a hash table, a trie, or an indexed set). The two are mixed together in type `KHE_DRS_DOM_KIND` because some data structures are not compatible with some kinds of dominance testing.

Type `KHE_DRS_SOLN_SET_PART` contains just the type tag:

```
#define INHERIT_KHE_DRS_SOLN_SET_PART                           \
  KHE_DRS_DOM_KIND              dom_kind;

typedef struct khe_drs_soln_set_part_rec {
  INHERIT_KHE_DRS_SOLN_SET_PART
} *KHE_DRS_SOLN_SET_PART;
```

Here are its ten concrete subtypes, one for each value of type `KHE_DRS_DOM_KIND`:

```
typedef struct khe_drs_soln_set_part_dom_none_rec {
  INHERIT_KHE_DRS_SOLN_SET_PART
  KHE_DRS_SOLN_LIST             soln_list;
} *KHE_DRS_SOLN_SET_PART_DOM_NONE;

typedef struct khe_drs_soln_set_part_dom_weak_rec {
  INHERIT_KHE_DRS_SOLN_SET_PART
  TABLE_KHE_DRS_SOLN            soln_table;
} *KHE_DRS_SOLN_SET_PART_DOM_WEAK;

typedef struct khe_drs_soln_set_part_dom_medium_rec {
  INHERIT_KHE_DRS_SOLN_SET_PART
  TABLE_KHE_DRS_SOLN_LIST       soln_list_table;
} *KHE_DRS_SOLN_SET_PART_DOM_MEDIUM;

typedef struct khe_drs_soln_set_part_dom_separate_rec {
  INHERIT_KHE_DRS_SOLN_SET_PART
  KHE_DRS_SOLN_LIST             soln_list;
} *KHE_DRS_SOLN_SET_PART_DOM_SEPARATE;
```

```
/* ***
typedef struct khe_drs_soln_set_part_dom_trie_rec {
  INHERIT_KHE_DRS_SOLN_SET_PART
  KHE_DRS_SOLN_TRIE             soln_trie;
} *KHE_DRS_SOLN_SET_PART_DOM_TRIE;
*** */

typedef struct khe_drs_soln_set_part_dom_indexed_rec {
  INHERIT_KHE_DRS_SOLN_SET_PART
  KHE_DRS_INDEXED_SOLN_SET      indexed_solns;
} *KHE_DRS_SOLN_SET_PART_DOM_INDEXED;

typedef struct khe_drs_soln_set_part_dom_tabulated_rec {
  INHERIT_KHE_DRS_SOLN_SET_PART
  KHE_DRS_SOLN_LIST             soln_list;
} *KHE_DRS_SOLN_SET_PART_DOM_TABULATED;

typedef struct khe_drs_soln_set_part_dom_indexed_tabulated_rec {
  INHERIT_KHE_DRS_SOLN_SET_PART
  KHE_DRS_HARD_INDEXED_SOLN_SET indexed_solns;
} *KHE_DRS_SOLN_SET_PART_DOM_INDEXED_TABULATED;
```

Each contains one field holding the data structure appropriate to its type: a solution list, a hash table of solutions, a hash table of solution lists, or an indexed array of solution lists. Type `KHE_DRS_SOLN_SET_PART` offers the usual operations on collections, implemented by large switches on the `dom_kind` field.

### D.11.4. Solution sets

Finally, we reach type `KHE_DRS_SOLN_SET`, used to hold each set of undominated solutions $P_k$:

```
typedef struct khe_drs_soln_set_rec *KHE_DRS_SOLN_SET;

struct khe_drs_soln_set_rec {
  KHE_DRS_SOLN_SET_PART         cache;
  KHE_DRS_SOLN_SET_PART         main;
};
```

This holds an optional `cache` part, which when non-`NULL` holds a collection of recently inserted solutions; and a `main` part, a non-optional collection holding most of the solutions. The idea of the cache is that solutions derived from the same predecessor solution are likely to exhibit dominance relations, so keeping them together might save time.

When caching is used, insertions go into the cache rather than into the main table:

```
void KheDrsSolnSetMeldSoln(KHE_DRS_SOLN_SET soln_set, KHE_DRS_SOLN soln,
  KHE_DRS_DAY soln_day, KHE_DRS_EXPANDER de,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  if( soln_set->cache != NULL )
    KheDrsSolnSetPartMeldSoln(soln_set->cache, soln, soln_day, de, drs);
  else
    KheDrsSolnSetPartMeldSoln(soln_set->main, soln, soln_day, de, drs);
}
```

`KheDrsSolnSetMeldSoln` is called by `KheDrsMakeEvaluateAndMeldSoln` (Appendix D.9.1) to add a new solution to `soln_set`. Functions `KheDrsSolnSetBeginCacheSegment` and `KheDrsSolnSetEndCacheSegment` instruct the solution set to begin and end caching:

```
void KheDrsSolnSetBeginCacheSegment(KHE_DRS_SOLN_SET soln_set,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  /* actually there is nothing to do here */
}

void KheDrsSolnSetEndCacheSegment(KHE_DRS_SOLN_SET soln_set,
  KHE_DYNAMIC_RESOURCE_SOLVER drs)
{
  KHE_DRS_SOLN_LIST soln_list;  KHE_DRS_SOLN soln;  int i;
  KHE_DRS_DOM_KIND cache_dom_kind;
  if( soln_set->cache != NULL )
  {
    cache_dom_kind = soln_set->cache->dom_kind;
    soln_list = KheDrsSolnListMake(drs);
    KheDrsSolnSetPartGather(soln_set->cache, soln_list);
    KheDrsSolnSetPartFree(soln_set->cache, drs);
    soln_set->cache = NULL;
    HaArrayForEach(soln_list->solns, soln, i)
      KheDrsSolnSetMeldSoln(soln_set, soln, drs);
    soln_set->cache = KheDrsSolnSetPartMake(cache_dom_kind, drs);
    KheDrsSolnListFree(soln_list, drs);
  }
}
```

There is nothing to do to begin caching, but to end it we have to move every element from the cache (if there is one) to the main table. This is rather messy. We make a simple list of solutions, `soln_list`, and call `KheDrsSolnSetPartGather` to gather all the solutions from the cache into this list, then `KheDrsSolnSetPartFree` to free the cache. Then without a cache we call `KheDrsSolnSetMeldSoln` on each element of `soln_list` to meld every solution from the cache into the main part. Finally, we create a new, empty cache and free the solution list.

## D.12. Solving

A solver is represented by an object of public type `KHE_DYNAMIC_RESOURCE_SOLVER`. It would be too tedious to show all the fields, but here is a selection:

```
struct khe_dynamic_resource_solver_rec {

  /* fields constant throughout the lifetime of the solver */
  HA_ARENA                        arena;
  KHE_SOLN                        soln;
  KHE_RESOURCE_TYPE               resource_type;
  KHE_OPTIONS                     options;
  KHE_FRAME                       days_frame;
  KHE_MTASK_FINDER                mtask_finder;
  ARRAY_KHE_DRS_RESOURCE          all_resources;
  ARRAY_KHE_DRS_DAY               all_days;
  ARRAY_KHE_DRS_TASK              all_root_tasks;
  ...

  /* free list fields */
  ...

  /* priority queue (always initialized, but use is optional) */
  KHE_PRIQUEUE                    priqueue;

  /* fields that vary with the solve */
  ARRAY_KHE_DRS_DAY_RANGE         selected_day_ranges;
  KHE_RESOURCE_SET                selected_resource_set;
  ...
  ARRAY_KHE_DRS_DAY               open_days;
  ARRAY_KHE_DRS_SHIFT             open_shifts;
  ARRAY_KHE_DRS_RESOURCE          open_resources;
  ARRAY_KHE_DRS_EXPR              open_exprs;
  ...
};
```

The first group of fields hold values that remain constant after the solver object is constructed. The most interesting ones are the last three, holding one DRS resource for each resource of type `rt`, one DRS day for each time group of the common frame, and one DRS task for each proper root task which accepts a resource of type `rt`.

After that come fields holding free lists for recycling objects between solves, and a field holding a priority queue of solutions for when that form of solving is requested. Finally, there are fields that vary with the solve. We've shown `selected_day_ranges`, which holds the day ranges selected by calls to public function `KheDynamicResourceSolverAddDayRange`, and `selected_resource_set`, which holds the resources selected by calls to public function `KheDynamicResourceSolverAddResource`. Then come arrays holding the days, shifts, resources, and expressions that have been opened for a particular solve.

### D.12.1. Construction

Here is the public function for creating a new solver, drastically abbreviated:

```
KHE_DYNAMIC_RESOURCE_SOLVER KheDynamicResourceSolverMake(KHE_SOLN soln,
  KHE_RESOURCE_TYPE rt, KHE_OPTIONS options)
{
  KHE_DYNAMIC_RESOURCE_SOLVER res;

  /* make the basic object */
  a = KheSolnArenaBegin(soln, false);
  HaMake(res, a);

  /* straightforward initializations (omitted) */
  /* construct the days (see below) */
  /* construct the resources (see below) */
  /* construct the tasks, mtasks, and shifts (see below) */
  /* construct the expressions (see below) */

  return res;
}
```

It obtains an arena from `soln` and creates a solver object `res`. Then it initializes every field of `res`, a tedious process that we won't show. After that it constructs the days, using this code:

```
/* drs day objects for the days of the frame */
for( i = 0;  i < KheFrameTimeGroupCount(days_frame);  i++ )
{
  day = KheDrsDayMake(i, res);
  HaArrayAddLast(res->all_days, day);
}
```

Next come the resources:

```
/* resources of rt but not their monitors (assumes days done) */
HaArrayInit(res->all_resources, a);
for( i = 0;  i < KheResourceTypeResourceCount(rt);  i++ )
{
  r = KheResourceTypeResource(rt, i);
  dr = KheDrsResourceMake(r, res);
  HaArrayAddLast(res->all_resources, dr);
}
```

Next come the tasks, mtasks, and shifts:

```
/* tasks, mtasks, and shifts (assumes days and resources) */
res->mtask_finder = KheMTaskFinderMake(soln, rt, days_frame, etm,
  true, a);
incomplete_times = false;  /* still to do */
if( incomplete_times )
{
  KheDynamicResourceSolverDelete(res);
  return NULL;  /* second dot point in doc */
}
for( i = 0;  i < KheMTaskSolverMTaskCount(res->mtask_solver);  i++ )
{
  mt = KheMTaskSolverMTask(res->mtask_solver, i);
  if( !KheMTaskNoGaps(mt) ||          /* third dot point in doc */
      !KheMTaskNoOverlap(mt) ||       /* fourth dot point in doc */
      !KheDrsMTaskMake(mt, res) )     /* fourth dot point in doc */
  {
    KheDynamicResourceSolverDelete(res);
    return NULL;
  }
}
```

Next comes code for working out the maximum possible workload that a resource could incur at each time:

```
/* initialize drs->max_workload_per_time */
KheDrsMaxWorkloadPerTimeInit(res);
```

After that comes code for initializing shift pair objects:

```
/* shift pairs */
HaArrayForEach(res->all_days, day, i)
  HaArrayForEach(day->shifts, ds1, j)
    for( k = j + 1;  k < HaArrayCount(day->shifts);  k++ )
    {
      ds2 = HaArray(day->shifts, k);
      dsp = KheDrsShiftPairMake(ds1, ds2, res);
      HaArrayAddLast(ds1->shift_pairs, dsp);
    }
```

After that it constructs the expressions corresponding to `soln`'s monitors:

```
/* resource monitors */
HaArrayForEach(res->all_resources, dr, i)
  if( !KheDrsResourceAddMonitors(dr, res) )
  {
    /* can't run this instance */
    KheDynamicResourceSolverDelete(res);
    return NULL;
  }

/* event resource monitors of type rt (assumes tasks done) */
HaArrayInit(er_monitors, a);
for( i = 0;  i < KheInstanceEventResourceCount(ins);  i++ )
{
  er = KheInstanceEventResource(ins, i);
  if( KheEventResourceResourceType(er) == rt )
    for( j = 0;  j < KheSolnEventResourceMonitorCount(soln, er);  j++ )
    {
      m = KheSolnEventResourceMonitor(soln, er, j);
      HaArrayAddLast(er_monitors, m);
    }
}
HaArraySortUnique(er_monitors, &KheMonitorCmp);
HaArrayForEach(er_monitors, m, i)
  if( !KheDrsAddEventResourceMonitor(res, m) )
  {
    /* can't run this instance */
    KheDynamicResourceSolverDelete(res);
    return NULL;
  }
HaArrayFree(er_monitors);
```

The KHE platform offers no simple way to visit each event resource monitor once, so this code puts them all into temporary array `er_monitors` and uniqueifies that array before making the corresponding expressions. Expression construction is carried out by `KheDrsResourceAddMonitors`, which adds expressions for all the monitors of DRS resource `dr`, and `KheDrsAddEventResourceMonitor`, which adds an expression for event resource monitor `m`. The following section explains which expressions are constructed for each kind of monitor. Limit idle times monitors and avoid split assignments monitors are not supported; if any of those are present, `KheDynamicResourceSolverMake` deletes the solver object and returns `NULL`.

Next comes a quick check that at least one monitor has cost greater than its lower bound:

```
if( !KheDrsSolnCanBeImproved(res) )
{
  /* no point running this instance; soln can't be improved on */
  KheDynamicResourceSolverDelete(res);
  return NULL;
}
```

Next comes some code to ensure that the increments used by hard and soft indexed solution sets have non-zero values:

```
/* make sure that the increments are non-zero */
if( res->hard_increment <= 0 )
  res->hard_increment = 1;
if( res->soft_increment <= 0 )
  res->soft_increment = 1;
```

Finally we construct the uniform dominance tables stored within constraint objects:

```
/* table objects within constraints */
HaArrayForEach(res->all_constraints, dc, i)
  if( dc != NULL )
    KheDrsConstraintSetTables(dc, res);
```

The constraints themselves were constructed earlier, while traversing the monitors.

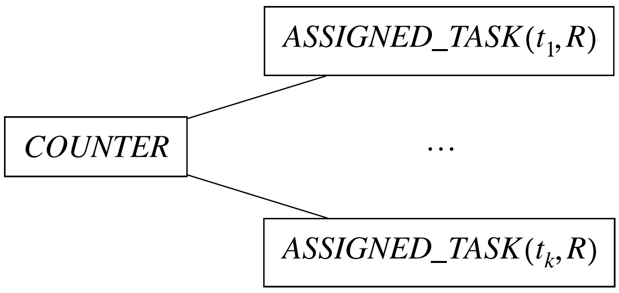
## D.12.2. Construction of expression trees

In this section we present expression trees for the XESTT event resource and resource monitors. These trees are built when the solver is created, by the calls to `KheDrsResourceAddMonitors` and `KheDrsAddEventResourceMonitor` shown above. Only parts of these expressions are open on any particular solve. We omit the actual construction code, since it can be derived from the diagrams presented here, using calls to the various functions for creating expression objects.

All non-root expressions have a value (either an `int` or a `float`) which is used by their parent. All root expressions report a cost which is added to the solution cost. However, we do not consider this cost to be the value of the root expression, because a value becomes available only on the expression's last active day, whereas a cost (strictly speaking, an extra cost) is added to the solution cost on each active day of the expression.

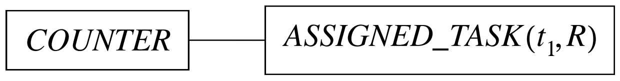
We start with event resource monitors.

**Assign resource monitors**. Let the atomic tasks monitored be $t_1, \dots, t_k$ after breaking them into single-day pieces of duration 1; their total duration is $k$. The expression tree is



where the *COUNTER* expression has minimum limit $k$ and $R$ is the resource type. Each leaf contributes 1 when its task is assigned, and the deviation is the amount by which the sum of these values falls short of the total duration, $k$.

When the cost function is linear, this tree may be divided into one tree per task on day:



and so on, where the *COUNTER* expression has minimum limit 1. An *ASSIGNED_TASK* expression has only a single open day, the day that its task on day is running, so these smaller trees never contribute to signature state arrays, which is why we prefer them.

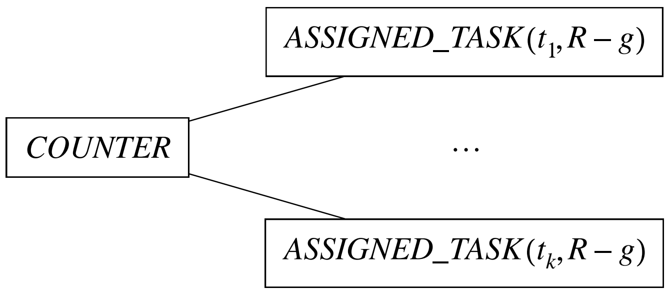
Assign resource monitors contribute to the `non_asst_cost` attributes of mtasks. So there is some danger of double counting their costs. However, a review of the solver's code shows that although `non_asst_cost` attributes are used to rule out some non-assignments, `non_asst_cost` itself is never added to any sum of costs. So there is in fact no double counting.

**Prefer resources monitors**. We use the same terminology as for assign resource monitors, plus we let $g$ be the set of preferred resources.

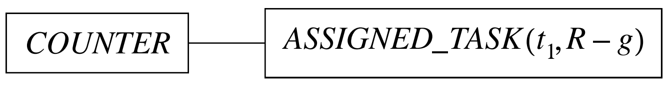
If $g$ includes every resource of the resource type concerned, the cost must always be zero and the monitor is ignored.

If $g$ is empty, meaning that it is preferable to not assign the monitored tasks, the monitor may contribute to the `asst_cost` attributes of those tasks' mtasks. These attributes are added to the current solution cost when expanding a solution, so again the monitor should be ignored, but this time to avoid double counting. For the `asst_cost` attribute to be affected, the monitor has to either monitor a single task or else have a linear cost function, as an examination of file `khe_sr_mtask_finder.c` will show. So we ignore the monitor when $g$ is empty and it either monitors a single task or has a linear cost function.

If we do not ignore the monitor, in general its expression tree is



where the *COUNTER* expression has maximum limit 0. Each $t_i$ assigned a resource not in $g$ contributes 1 to the determinant. (An unassigned $t_i$ contributes 0, as required.) When the cost function is linear this can be divided into one tree per task on day, again with maximum limit 0:

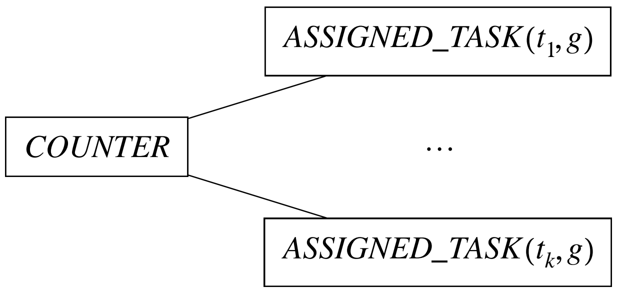


and so on. Again, these smaller trees never contribute to signature states, so are preferred.

**Avoid split assignments monitors**. These do not occur in nurse rostering, and a solver is not made when they are present. What needs to be remembered on any day is the set of distinct resources assigned to the monitored tasks. Without this, one cannot tell whether a later assignment increases the number of distinct resources or not. There are various ways to encode this into the signature, although none seem to be ideal. A bit set packed into integers leads to very large arrays in a trie structure. An unpacked bit set leads to a large number of signature entries.

A set of resource indexes varies in length, although there is an upper limit: the number of tasks monitored that are running at or before the current day. Perhaps a sequence of resource indexes, sorted and uniqueified, and padded out to the upper limit on length, would be best. One signature dominates another when its resources are a subset of the other's.

**Limit resources monitors**. Let $g$ be the set of resources of interest. The tree is



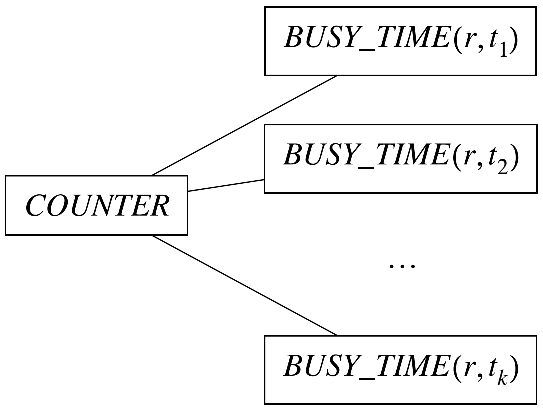
where the *COUNTER* expression's limits are taken from the monitor. This only divides into separate trees when the cost function is linear and both limits are 0.

A limit resources monitor can mimic a prefer resources monitor. In that case, the mtask solver treats the limit resources monitor like a prefer resources monitor, including adding a cost to the `asst_cost` attribute when appropriate. An examination of file `khe_sr_mtask_finder.c` shows that `asst_cost` is affected when the limit resource monitor's maximum limit is 0, the set $g$ (whose complement becomes the set of preferred resources of the corresponding prefer resources monitor) contains every resource of the given resource type, and the number of tasks is 1 or the cost function is linear. In this case we ignore the limit resources monitor.

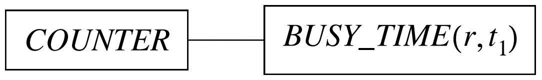
We turn now to the resource monitors for resource $r$.

**Avoid clashes monitors**. No clashes can occur, because tasks with clashes are excluded, and each resource is assigned to at most one task on each day. So these monitors are ignored.

**Avoid unavailable times monitors**. If the unavailable times are $t_1, t_2, \dots, t_k$, the tree is



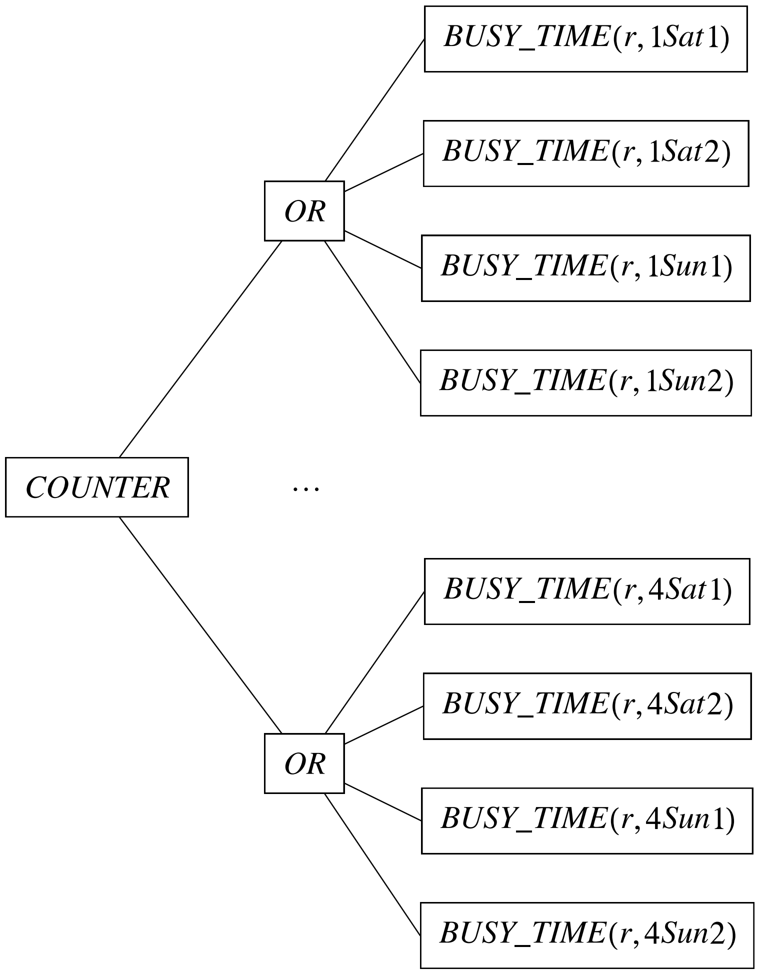
The *COUNTER* expression has maximum limit 0. If the cost function is linear, each time contributes an independent value to the total cost, and we use multiple trees instead:



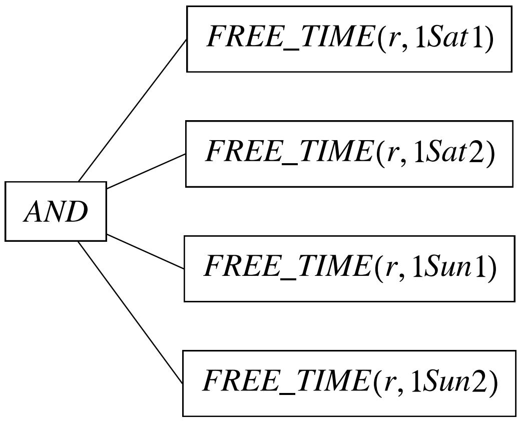
and so on. We prefer this because these expressions do not store a value in the signature.

**Limit idle times monitors**. These do not occur in nurse rostering, and a solver is not made when they are present. Handling them is future work (feasible, but low priority).

**Cluster busy times monitors**. We have already seen a cluster busy times tree, for limiting busy weekends, assuming a four-week instance with two shifts per day. Here it is again:

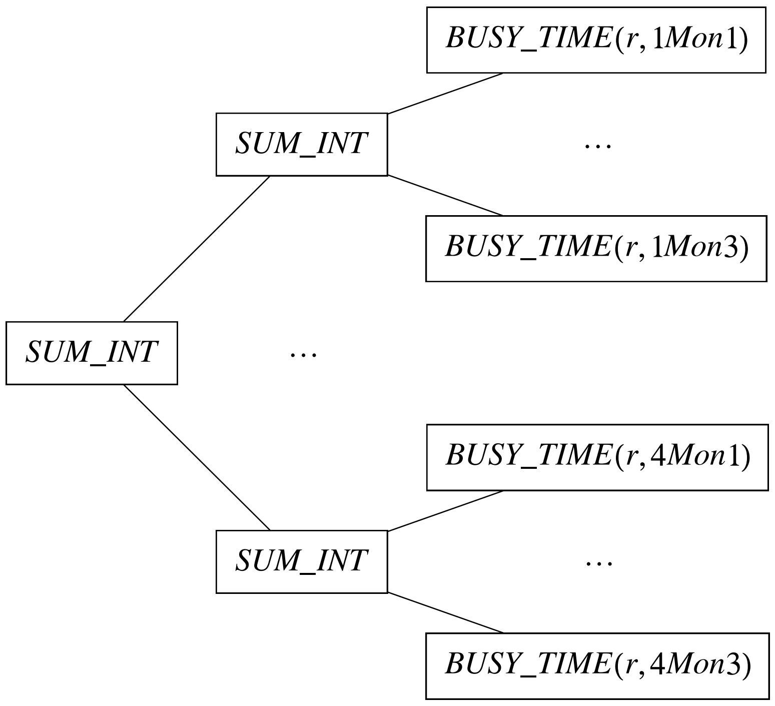


Within each *OR*, if a day's times are all present, their *BUSY_TIME* expressions are replaced by a *BUSY_DAY* expression, saving time. Negative time groups become



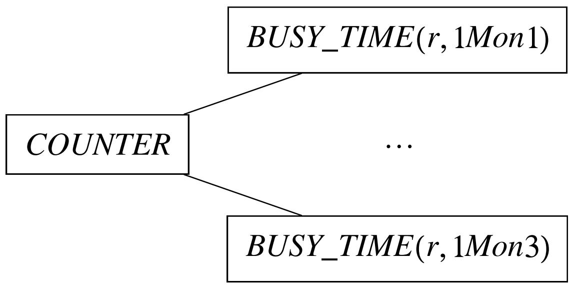
Again, *FREE_DAY* expressions may replace *FREE_TIME* expressions. And when an *OR* or *AND* expression has exactly one child, the *OR* or *AND* expression is omitted.

**Limit busy times monitors**. A limit busy times monitor may monitor several time groups, like a cluster busy times monitor, but a deviation is calculated for each time group separately:



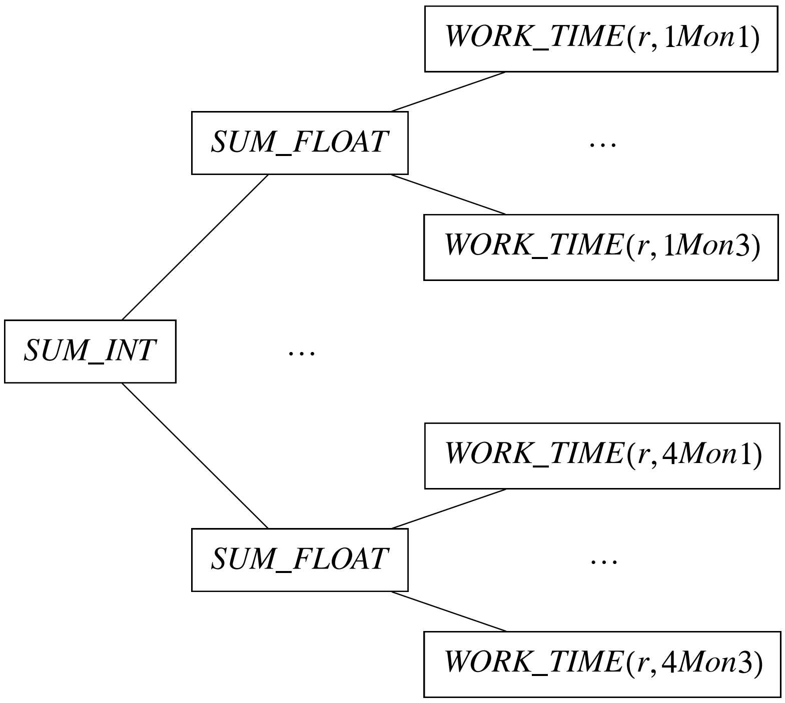
The lower *SUM_INT* expressions, described earlier as case (4) sum expressions, have deviation calculations but not cost calculations. The higher *SUM_INT*, called case (1), includes cost but only a trivial deviation (maximum limit 0). If any of the time groups contains a full day's worth of times, their *BUSY_TIME* expressions are replaced by one *BUSY_DAY* expression.

If there is only one time group or the cost function is linear, the tree is broken up into one tree for each time group. Each of these trees has the form



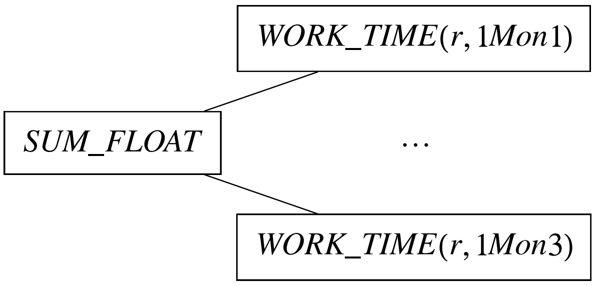
We can replace *SUM_INT* by *COUNTER* here because the children all have value 0 or 1, and we do it because it allows tabulated dominance to be used. This is case (2). As before, a day's worth of *BUSY_TIME* expressions are replaced by a *BUSY_DAY* expression. If the tree as a whole requires a maximum of one busy time on one day, it is omitted, since all solutions produced by this solver have that property.

**Limit workload monitors**. These are the same as limit busy times monitors, except that they keep track of a `float` workload rather than an `int` number of busy times:



The *SUM_FLOAT* expressions, which are case (4), include deviation calculations but not cost calculations. The *SUM_INT* expression, which is case (1), includes the cost calculation, but only a trivial deviation calculation (maximum limit 0). If any of the time groups contains a full day's worth of times, their *WORK_TIME* expressions are replaced by one *WORK_DAY* expression.

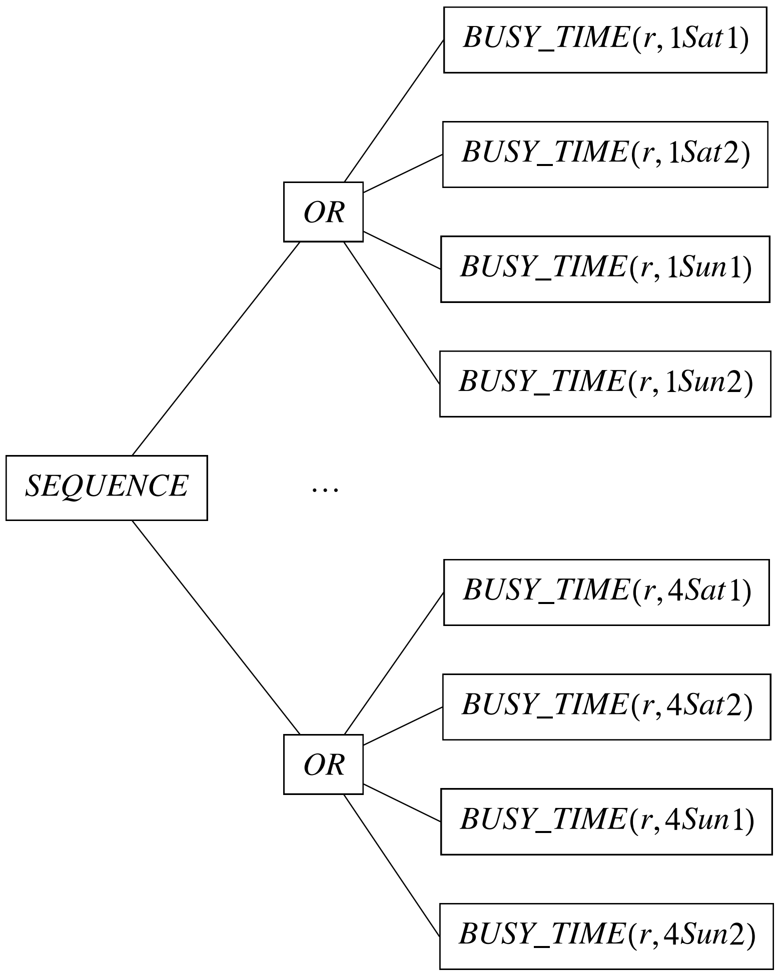
As for limit busy times monitors, if there is only one time group or the cost function is linear, the tree is broken up into one tree for each time group. Each of these trees has the form



The remaining *SUM_FLOAT* expression, which is case (3), takes over the cost calculation. As before, a day's worth of *WORK_TIME* expressions are replaced by a *WORK_DAY* expression.

When a time group spans more than one day, its *SUM_FLOAT* expression needs to store floating point numbers in signatures. This is done by having each position in each signature have type `KHE_DRS_VALUE`, an untagged union of `int` and `float`. The context is used to select the appropriate value from the union.

**Limit active intervals monitors**. These have the same data as cluster busy times monitors, without allow zero. Only the root expression is different:



As for counter monitors, negative time groups produce *AND* and *FREE_TIME* expressions, and times making up complete days become *BUSY_DAY* and *FREE_DAY* expressions. However, when an *OR* or *AND* expression has exactly one child, we do not omit it as we do for cluster busy times monitors. The two reasons for this are explained in Appendix D.8.11.

### D.12.3. Solving

At last, we are ready for the main solving functions. As explained earlier, a solve has three steps: opening, searching, and closing. Here is the function which opens a solve:

```
void KheDrsSolveOpen(KHE_DYNAMIC_RESOURCE_SOLVER drs, bool testing,
  bool priqueue, bool extra_selection, bool expand_by_shifts,
  bool correlated_exprs, int daily_expand_limit, int daily_prune_trigger,
  int resource_expand_limit, int dom_approx,
  KHE_DRS_DOM_KIND main_dom_kind, bool cache,
  KHE_DRS_DOM_KIND cache_dom_kind, KHE_DRS_PACKED_SOLN *init_soln)
{
  KHE_DRS_DAY_RANGE ddr;  KHE_DRS_DAY day;  int i, j, rcount;
  KHE_DRS_EXPR e;  KHE_DRS_RESOURCE dr;  KHE_RESOURCE r;

  /* initialize fields that vary with the solve */
  drs->solve_priqueue = priqueue;
  drs->solve_extra_selection = extra_selection;
  drs->solve_expand_by_shifts = expand_by_shifts;
  drs->solve_correlated_exprs = correlated_exprs;
  drs->solve_daily_expand_limit = daily_expand_limit;
  drs->solve_daily_prune_trigger = daily_prune_trigger;
  drs->solve_resource_expand_limit = resource_expand_limit;
  drs->solve_dom_approx = dom_approx;
  drs->solve_dom_test_type =
    KheDomKindCheckConsistency(main_dom_kind, cache, cache_dom_kind);
  drs->solve_init_cost = drs->solve_start_cost = KheSolnCost(drs->soln);
  KheDrsResourceSetClear(drs->open_resources);
  HaArrayClear(drs->open_days);
  HaArrayClear(drs->open_shifts);
  HaArrayClear(drs->open_exprs);

  /* open selected resources */
  *init_soln = KheDrsPackedSolnBuildEmpty(drs);
  rcount = KheResourceSetResourceCount(drs->selected_resource_set);
  for( i = 0;  i < rcount;  i++ )
  {
    r = KheResourceSetResource(drs->selected_resource_set, i);
    dr = HaArray(drs->all_resources, KheResourceResourceTypeIndex(r));
    KheDrsResourceOpen(dr, KheDrsResourceSetCount(drs->open_resources),
      *init_soln, drs);
    KheDrsResourceSetAddLast(drs->open_resources, dr);
  }

  ... code omitted here (see below) ...
}
```

First, the solve options are copied into the solver, and the sets of open resources, days, shifts, and expressions are cleared. Then the selected resources are opened by calls to `KheDrsResourceOpen`, and added to `drs->open_resources`. Then the code that was omitted above is run; here it is:

```
    /* open selected days */
    HaArrayForEach(drs->selected_day_ranges, ddr, i)
      for( j = ddr.first;  j <= ddr.last;  j++ )
      {
        day = HaArray(drs->all_days, j);
        KheDrsDayOpen(day, ddr, HaArrayCount(drs->open_days), main_dom_kind,
          cache, cache_dom_kind, drs);
        HaArrayAddLast(drs->open_days, day);
      }

    /* open the shifts and mtasks on selected days */
    HaArrayForEach(drs->selected_day_ranges, ddr, i)
      for( j = ddr.first;  j <= ddr.last;  j++ )
      {
        day = HaArray(drs->all_days, j);
        KheDrsDayOpenShifts(day, ddr, drs);
      }

    /* sort drs->open_exprs by postorder index, then open them */
    HaArraySort(drs->open_exprs, &KheDrsExprPostorderCmp);
    HaArrayForEach(drs->open_exprs, e, i)
      KheDrsExprOpen(e, drs);
    HaArrayForEach(drs->open_exprs, e, i)
      KheDrsExprNotifySigners(e, drs);
```

It opens the selected days, shifts, mtasks, and expressions. All days must be open before any mtasks are opened, because opening a task includes assigning open day indexes to its task on day objects, and only open days have those. The two-stage process for opening expressions is discussed elsewhere.

After opening comes searching, but we'll look at closing first:

```
void KheDrsSolveClose(KHE_DYNAMIC_RESOURCE_SOLVER drs,
  KHE_DRS_PACKED_SOLN soln, bool check_rerun_costs)
{
  KHE_DRS_DAY day;  int i, j;  KHE_DRS_EXPR e;  KHE_MONITOR m;
  KHE_DRS_TASK_ON_DAY dtd;  KHE_DRS_PACKED_SOLN_DAY rd;
  KHE_DRS_RESOURCE dr;  KHE_DRS_MONITOR dm;

  /* traverse soln, closing assigned tasks */
  HaArrayForEachReverse(soln->days, rd, i)
    HaArrayForEach(rd->prev_tasks, dtd, j)
      if( dtd != NULL )
      {
        dr = KheDrsResourceSetResource(drs->open_resources, j);
        KheDrsTaskClose(dtd->encl_dt, dr);
      }

  /* close the open days, including closing unassigned tasks */
  HaArrayForEach(drs->open_days, day, i)
    KheDrsDayClose(day, drs);

  /* close the open expressions */
  HaArrayForEach(drs->open_exprs, e, i)
    KheDrsExprClose(e, drs);

  /* close the open resources */
  KheDrsResourceSetForEach(drs->open_resources, dr, i)
    KheDrsResourceClose(dr, drs);

  /* close drs */
  KheDrsResourceSetClear(drs->open_resources);
  HaArrayClear(drs->open_days);
  HaArrayClear(drs->open_shifts);
  HaArrayClear(drs->open_exprs);

  /* optionally check rerun costs */
  if( check_rerun_costs )
    HaArrayForEach(drs->all_monitors, dm, i)
      KheDrsMonitorCheckRerunCost(dm);

  /* check that DRS soln cost equals KHE soln cost */
  HnAssert(KheSolnCost(drs->soln) == soln->cost,
    "KheDrsSolveClose internal error: soln %.5f != packed %.5f",
    KheCostShow(KheSolnCost(drs->soln)), KheCostShow(soln->cost));
}
```

This closes everything that was previously opened. Parameter `soln` says which solution to install into the KHE platform: a new best solution, or the original. Tasks assigned by `soln` are closed first, more than once if they are multi-day tasks. Open but unassigned tasks are closed later, when their days are closed. Assigned tasks get closed at least twice, but as we saw in Appendix D.5.1, `KheDrsTaskClose` can safely close a task more than once: only the first call does anything.

Here now is the function for carrying out the search:

```
bool KheDrsSolveSearch(KHE_DYNAMIC_RESOURCE_SOLVER drs,
  bool testing, KHE_DRS_PACKED_SOLN *final_soln)
{
  KHE_DRS_DAY prev_day, next_day;  KHE_DRS_SOLN root_soln, soln;
  KHE_DRS_SOLN_LIST soln_list, root_soln_list;
  int i, made_count, undominated_count, kept_count;

  /* priority search is different */
  if( drs->solve_priqueue )
    return KheDrsSolvePrioritySearch(drs, testing, final_soln);

  /* do the search */
  root_soln_list = soln_list = KheDrsSolnListMake(drs);
  root_soln = KheDrsSolnMake(NULL, drs->solve_start_cost, drs);
  KheDrsSolnListAddSoln(soln_list, root_soln, drs);
  KheDrsAddStats(drs, testing, soln_list, NULL, 0, 0, 0);
  prev_day = NULL;
  HaArrayForEach(drs->open_days, next_day, i)
  {
    KheDrsSolnListExpand(soln_list, prev_day, next_day, drs);
    made_count = next_day->soln_made_count;
    soln_list = KheDrsDayGatherSolns(next_day, drs);
    undominated_count = KheDrsSolnListCount(soln_list);
    KheDrsSolnListSortAndReduce(soln_list, drs);
    kept_count = KheDrsSolnListCount(soln_list);
    KheDrsAddStats(drs, testing, soln_list, next_day, made_count,
      undominated_count, kept_count);
    prev_day = next_day;
  }

  /* set *final_soln, to NULL if there isn't one */
  if( KheDrsSolnListCount(soln_list) == 1 )
  {
    soln = KheDrsSolnListFirstSoln(soln_list);
    KheDrsPrintSearchStatistics(soln, drs);
    *final_soln = KheDrsPackedSolnBuildFromSoln(soln, drs);
  }
  else
    *final_soln = NULL;

  /* free root_soln (others later) and return true if have final soln */
  KheDrsSolnFree(root_soln, drs);
  KheDrsSolnListFree(root_soln_list, drs);
  return *final_soln != NULL;
}
```

First, if the priority queue is in use it passes its job on to a completely different function. Then it makes `root_soln_list`, containing just the root solution, the one not lying in any day. Then, for each open day `next_day`, it calls `KheDrsSolnListExpand` (Appendix D.11) to build a new solution set, by trying all ways to expand the solutions of `soln_list` by one day. Then `KheDrsDayGatherSolns` traverses this solution set and adds each solution to a new `soln_list`, which is then sorted and optionally reduced in size by `KheDrsSolnListSortAndReduce`, ready for the next iteration. At the end, if there is a solution in `soln_list` we have a new best solution, so it calls `KheDrsPackedSolnBuildFromSoln` to convert this into a packed solution, and returns `true`. Otherwise it returns `false`.

To complete our presentation of solving, we'll skip forward in the source file to the function called by the user to carry out a solve:

```
bool KheDynamicResourceSolverSolve(KHE_DYNAMIC_RESOURCE_SOLVER drs,
  bool priqueue, bool extra_selection, bool expand_by_shifts,
  bool shift_pairs, bool correlated_exprs, int daily_expand_limit,
  int daily_prune_trigger, int resource_expand_limit, int dom_approx,
  KHE_DRS_DOM_KIND main_dom_kind, bool cache,
  KHE_DRS_DOM_KIND cache_dom_kind)
{
  KHE_COST cost;  jmp_buf env;  bool res;

  if( setjmp(env) == 0 )
  {
    KheSolnJmpEnvBegin(drs->soln, &env);
    res = KheDynamicResourceSolverDoSolve(drs, priqueue, extra_selection,
      expand_by_shifts, shift_pairs, correlated_exprs, daily_expand_limit,
      daily_prune_trigger, resource_expand_limit, dom_approx,
      main_dom_kind, cache, cache_dom_kind, false, &cost);
    KheSolnJmpEnvEnd(drs->soln);
  }
  else
  {
    KheSolnJmpEnvEnd(drs->soln);
    res = false;
  }
  return res;
}
```

As shown, this code ensures that if memory runs out during the solve, the resulting long jump will return here, then it calls `KheDynamicResourceSolverDoSolve`, with the extra `false` argument to indicate that this is a real solve and not a test:

```
bool KheDynamicResourceSolverDoSolve(KHE_DYNAMIC_RESOURCE_SOLVER drs,
  bool priqueue, bool extra_selection, bool expand_by_shifts,
  bool shift_pairs, bool correlated_exprs, int daily_expand_limit,
  int daily_prune_trigger, int resource_expand_limit, int dom_approx,
  KHE_DRS_DOM_KIND main_dom_kind, bool cache,
  KHE_DRS_DOM_KIND cache_dom_kind, bool test_only, KHE_COST *cost)
{
  int rcount, i;  KHE_RESOURCE r;  KHE_INTERVAL ddr;  KHE_TIMER timer;
  KHE_DRS_PACKED_SOLN init_soln, new_best_soln, junk;  KHE_COST init_cost;
  char buff[20];  KHE_TIME_GROUP tg1, tg2;

  /* open, search, close, and possibly rerun */
  init_cost = KheSolnCost(drs->soln);
  KheDrsSolveOpen(drs, true, priqueue, extra_selection, expand_by_shifts,
    shift_pairs, correlated_exprs, daily_expand_limit, daily_prune_trigger,
    resource_expand_limit, dom_approx, main_dom_kind, cache,
    cache_dom_kind, &init_soln);
  if( !KheDrsSolveSearch(drs, true, &new_best_soln) )
  {
    /* no new best; close using init_soln */
    KheDrsSolveClose(drs, init_soln, false);
    *cost = KheSolnCost(drs->soln);
  }
  else if( RERUN )
    ... omitted ...
  else
  {
    /* have new best solution, close using that */
    KheDrsSolveClose(drs, new_best_soln, false);
    *cost = KheSolnCost(drs->soln);
  }

  /* delete init_soln and (if present) new_best_soln */
  KheDrsPackedSolnDelete(init_soln, drs);
  if( new_best_soln != NULL )
    KheDrsPackedSolnDelete(new_best_soln, drs);

  /* clear out selections ready for a fresh set of resources and days */
  KheResourceSetClear(drs->selected_resource_set);
  HaArrayClear(drs->selected_day_ranges);

  /* return true if the solution has been improved */
  HnAssert(KheSolnCost(drs->soln) <= init_cost,
    "KheDynamicResourceSolverDoSolve internal error: new %.5f > old %.5f",
    KheCostShow(KheSolnCost(drs->soln)), KheCostShow(init_cost));
  return KheSolnCost(drs->soln) < init_cost;
}
```

It calls on `KheDrsSolveOpen`, `KheDrsSolveSearch`, and `KheDrsSolveClose`, and manages two packed solutions, `init_soln` holding the initial solution, and `new_best_soln` holding the new best solution if `KheDrsSolveSearch` finds one. For the `RERUN` code see Appendix D.12.4.

### D.12.4. Testing

In general it is not possible to compare the cost of a solver solution with a KHE cost, because incomplete solutions have no KHE cost. But when a new best solution is found, it is complete. If it is installed into the KHE platform its cost can be compared with the KHE cost and should be equal to it. This important correctness check is made at the end of every successful solve.

If the check fails, the solver has calculated the cost of one or more constraints incorrectly. But working out which constraints are wrong is not easy. To help with this, the solver offers a `RERUN` compiler flag and a `rerun` field in the solver holding a packed solution. If the `RERUN` flag is 1 and the `rerun` field of the solver is non-`NULL`, the solve is a *rerun.* This means that instead of trying many different assignments, only the assignments from `drs->rerun` are tried. This reduces the amount of debug output, while still executing the code that led to the incorrect cost.

We have already seen the key piece of code here, within `KheDrsResourceOnDayIsFixed` (Appendix D.10.3). There, if the current run is a rerun, every resource on day is reported to have a fixed assignment, which `KheDrsResourceOnDayIsFixed` retrieves from `drs->rerun`. Previously omitted code from `KheDynamicResourceSolverDoSolve` does the rest:

```
if( !KheDrsSolveSearch(drs, true, &new_best_soln) )
{
  /* no new best; close using init_soln */
  KheDrsSolveClose(drs, init_soln, false);
  *cost = KheSolnCost(drs->soln);
}
else if( RERUN )
{
  /* close using init_soln */
  KheDrsSolveClose(drs, init_soln, false);

  /* rerun new_best_soln (drs is closed on new_best_soln after this) */
  KheDrsRerun(drs, priqueue, extra_selection, expand_by_shifts, shift_pairs,
    correlated_exprs, daily_expand_limit, daily_prune_trigger,
    resource_expand_limit, dom_approx, main_dom_kind, cache,
    cache_dom_kind, new_best_soln);
  *cost = KheSolnCost(drs->soln);

  /* if test only, return to init_soln */
  if( test_only )
  {
    KheDrsSolveOpen(drs, false, priqueue, extra_selection, expand_by_shifts,
      shift_pairs, correlated_exprs, daily_expand_limit, daily_prune_trigger,
      resource_expand_limit, dom_approx, main_dom_kind, cache,
      cache_dom_kind, &junk);
    KheDrsPackedSolnDelete(junk, drs);
    KheDrsSolveClose(drs, init_soln, false);
  }
}
```

If a new best solution is found and a rerun is wanted, the solve is closed using `init_soln`, returning the solver to the initial state. Then `KheDrsRerun` is called to carry out the rerun. We'll see this function in a moment. It leaves the solver with the new best solution reinstalled. Finally, if we are only testing, we open for solving and immediately close again with the initial solution. This returns the solver once again to the initial state, which is what is wanted when testing.

Here is `KheDrsRerun`:

```
void KheDrsRerun(KHE_DYNAMIC_RESOURCE_SOLVER drs, bool priqueue,
  bool extra_selection, bool expand_by_shifts, bool shift_pairs,
  bool correlated_exprs, int daily_expand_limit, int daily_prune_trigger,
  int resource_expand_limit, int dom_approx,
  KHE_DRS_DOM_KIND main_dom_kind, bool cache,
  KHE_DRS_DOM_KIND cache_dom_kind, KHE_DRS_PACKED_SOLN soln)
{
  KHE_DRS_PACKED_SOLN init_soln2, new_best_soln2;
  int i;  KHE_DRS_MONITOR dm;

  /* carry out the open, search, and close of the rerun */
  drs->rerun_soln = soln;
  HaArrayForEach(drs->all_monitors, dm, i)
    KheDrsMonitorInitRerunCost(dm, drs);
  KheDrsSolveOpen(drs, false, priqueue, extra_selection, expand_by_shifts,
    shift_pairs, correlated_exprs, daily_expand_limit, daily_prune_trigger,
    resource_expand_limit, dom_approx, main_dom_kind, cache,
    cache_dom_kind, &init_soln2);
  if( !KheDrsSolveSearch(drs, false, &new_best_soln2) )
    HnAbort("KheDrsRerun internal error (rerun failed to find new best)");
  HnAssert(soln->cost == new_best_soln2->cost,
    "KheDrsRerun internal error (rerun new best has different cost)");
  KheDrsSolveClose(drs, new_best_soln2, true);
  drs->rerun_soln = NULL;

  /* delete the packed solutions made by this function */
  KheDrsPackedSolnDelete(init_soln2, drs);
  KheDrsPackedSolnDelete(new_best_soln2, drs);
}
```

This begins by setting `drs->rerun` to its `soln` parameter to signal to the rest of the code that this is a rerun using `soln`. Then it initializes the rerun cost fields in the monitors; we'll come to those shortly. After that we have the usual open-search-close sequence, followed by `drs->rerun = NULL` to indicate that the rerun is over. Some tidying up ends the function.

We mentioned earlier the problem of finding out which code has calculated an incorrect cost. This is simplified by the *rerun cost expressions*. Each descendant of type `KHE_DRS_EXPR_COST` contains a `monitor` field of type `KHE_DRS_MONITOR`:

```
typedef struct khe_drs_monitor_rec *KHE_DRS_MONITOR;

struct khe_drs_monitor_rec {
  KHE_MONITOR           monitor;
  KHE_COST              rerun_open_and_search_cost;
  KHE_COST              rerun_open_and_close_cost;
  KHE_DRS_EXPR          sample_expr;
};
```

as we saw in Appendix D.7.2. It contains the monitor plus two costs used only during reruns. Whenever an expression containing monitor `dm` reports a cost when opening or searching, it also adds the cost it reports to `dm->rerun_open_and_search_cost`; and whenever it reports a cost when opening or closing, it also adds the cost it reports to `dm->rerun_open_and_close_cost`. These reports are made by calls, which we have omitted in our presentations so far, to `KheDrsMonitorInfoUpdateRerunCost`. This happens only during reruns.

At the end of a rerun, when a new best solution is installed, we should have

```
dm->rerun_open_and_search_cost == KheMonitorCost(dm->monitor)
```

and

```
dm->rerun_open_and_close_cost == KheMonitorCost(dm->monitor)
```

`KheDrsSolveClose` checks these conditions, by calling `KheDrsMonitorInfoCheckRerunCost`, which we omitted before. This prints debug output pointing to the failures, if there are any. In this way we can track down the misbehaving expressions.

Some monitors give rise to multiple expression trees, when they are broken into independent parts. That's fine; the different expression trees share the same monitor object.

The solver also offers debug code to help with working out what is going wrong. We won't detail it here, but one can name a particular cost expression (one previously found to be going wrong) by setting the `RERUN_MONITOR_ID` compiler flag, and this will produce debug output during the rerun which shows how the cost of that expression is calculated during opening, closing, and evaluating on each day. On a regular run this would be incomprehensible, but on a rerun there is just the one search path to follow.

When testing, the solver will also collect statistics about the current solve. These are stored in fields of `KHE_DYNAMIC_RESOURCE_SOLVER` that we omitted before:

```
#if TESTING
  KHE_TIMER                    timer;
  HA_ARRAY_INT                 solns_made_per_day;
  HA_ARRAY_INT                 table_size_per_day;
  HA_ARRAY_FLOAT               running_time_per_day;
  HA_ARRAY_INT                 ancestor_freq;
  HA_ARRAY_INT                 dominates_freq;
  int                          max_open_day_index;
#endif
```

Skipping details, there is a `KheDrsAddStats` function, whose calls we have omitted, that adds values to these statistics; and then the public `KheDynamicResourceSolverSolveStatsCount` and `KheDynamicResourceSolverSolveStats` functions return these values to the user.

# References

[1] R. Ahuja, Ö. Ergun, J. Orlin, and A. Punnen. A survey of very large-scale neighbourhood search techniques. *Discrete Applied Mathematics* **123**, 75–102 (2002).

[2] J. Csima and C. C. Gotlieb. Tests on a computer method for constructing school timetables. *Communications of the ACM* **7**, 160–163 (1964).

[3] Fred Glover. Ejection chains, reference structures and alternating path methods for traveling salesman problems. *Discrete Applied Mathematics* **65**, 223–253 (1996).

[4] C. C. Gotlieb. The construction of class-teacher timetables. In *Proc. IFIP Congress*, pages 73–77, 1962.

[5] Peter de Haan, Ronald Landman, Gerhard Post, and Henri Ruizenaar. A case study for timetabling in a Dutch secondary school. In *Practice and Theory of Automated Timetabling VI (Sixth International Conference, PATAT2006, Czech Republic, August 2006, Selected Papers)*, pages 267–279. Springer Lecture Notes in Computer Science 3867, 2007.

[6] Jeffrey H. Kingston. The KTS high school timetabling web site (Version 1.4), September 2006. URL *http://www.it.usyd.edu.au/~jeff*.

[7] Jeffrey H. Kingston. Hierarchical timetable construction. In *Practice and Theory of Automated Timetabling VI (Sixth International Conference, PATAT2006, Brno, Czech Republic, August 2006, Selected Papers)*, pages 294–307. Springer Lecture Notes in Computer Science 3867, 2007.

[8] Jeffrey H. Kingston. The KTS high school timetabling system. In *Practice and Theory of Automated Timetabling VI (Sixth International Conference, PATAT2006, Czech Republic, August 2006, Selected Papers)*, pages 308–323. Springer Lecture Notes in Computer Science 3867, 2007.

[9] Jeffrey H. Kingston. Resource assignment in high school timetabling. In *PATAT2008 (Seventh international conference on the Practice and Theory of Automated Timetabling, Montreal, August 2008)*, 2008.

[10] Jeffrey H. Kingston. Modelling history in nurse rostering. In *PATAT 2018 (Twelfth international conference on the Practice and Theory of Automated Timetabling, Vienna, August 2018)*, pages 97–111, 2018.

[11] Jeffrey H. Kingston, Gerhard Post, and Greet Vanden Berghe. A unified nurse rostering model based on XHSTT. In *PATAT 2018 (Twelfth international conference on the Practice and Theory of Automated Timetabling, Vienna, August 2018)*, pages 81–96, 2018.

[12] Carol Meyers and James B. Orlin. Very large-scale neighbourhood search techniques in timetabling problems. In *Practice and Theory of Automated Timetabling VI (Sixth Interna-*

*tional Conference, PATAT2006, Brno, Czech Republic, August 2006, Selected Papers)*, pages 24–39. Springer Lecture Notes in Computer Science 3867, 2007.

[13] Samad Ahmadi, Sophia Daskalaki, Jeffrey H. Kingston, Jari Kyngäs, Cimmo Nurmi, Gerhard Post, David Ranson, and Henri Ruizenaar. An XML format for benchmarks in high school timetabling. In *PATAT08 (Seventh international conference on the Practice and Theory of Automated Timetabling, Montreal, August 2008)*, 2008.

[14] D. de Werra. Construction of school timetables by flow methods. *INFOR – Canadian Journal of Operations Research and Information Processing* **9**, 12–22 (1971).