

# The KHE Timetabling Platform and Solvers

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

Version 2.4 (January 2020)

# Contents

## Part A: The Platform

<b>Chapter 1. Introduction</b>	2
1.1. Installation and use	2
1.2. The data types of KHE	3
1.3. Common operations	3
1.4. KHE for employee scheduling	5
<b>Chapter 2. Archives and Solution Groups</b>	6
2.1. Archives	6
2.2. Solution groups	8
2.3. Solution sets	9
2.4. Reading archives	10
2.5. Reading archives incrementally	11
2.6. Reading archives from the command line	13
2.7. Writing archives and solution groups	14
<b>Chapter 3. Instances</b>	16
3.1. Creating instances	16
3.2. Visiting and retrieving the components of instances	17
3.3. Constraint density	19
3.4. Times	20
3.4.1. Time groups	20
3.4.2. Times	23
3.5. Resources	25
3.5.1. Resource types	25
3.5.2. Resource groups	26
3.5.3. Resources	29
3.5.4. Resource layers	31
3.5.5. Resource type partitioning	31
3.5.6. Resource similarity and inferring resource partitions	32
3.6. Events	33
3.6.1. Event groups	33
3.6.2. Events	35
3.6.3. Event resources	37
3.6.4. Event resource groups	39

3.7. Constraints	40
3.7.1. Assign resource constraints	44
3.7.2. Assign time constraints	46
3.7.3. Split events constraints	47
3.7.4. Distribute split events constraints	48
3.7.5. Prefer resources constraints	49
3.7.6. Prefer times constraints	51
3.7.7. Avoid split assignments constraints	52
3.7.8. Spread events constraints	54
3.7.9. Link events constraints	55
3.7.10. Order events constraints	56
3.7.11. Avoid clashes constraints	56
3.7.12. Avoid unavailable times constraints	57
3.7.13. Limit idle times constraints	59
3.7.14. Cluster busy times constraints	60
3.7.15. Limit busy times constraints	64
3.7.16. Limit workload constraints	66
3.7.17. Limit active intervals constraints	68
3.7.18. Limit resources constraints	71
<b>Chapter 4. Solutions</b>	<b>74</b>
4.1. Overview	74
4.2. Top-level operations	75
4.2.1. Creation, deletion, and copy	75
4.2.2. Solutions and arenas	76
4.2.3. Simple attributes	77
4.2.4. Diversification	78
4.2.5. Visit numbers	81
4.2.6. Placeholder and invalid solutions	82
4.2.7. Traversing the components of solutions	84
4.3. Complete representation and preassignment conversion	85
4.4. Solution time, resource, and event groups	86
4.5. Meets	87
4.5.1. Splitting and merging	89
4.5.2. Assignment	91
4.5.3. Cycle meets and time assignment	94
4.5.4. Meet domains and bounds	96
4.5.5. Automatic domains	99
4.6. Tasks	100

4.6.1. Assignment	102
4.6.2. Cycle tasks and resource assignment	105
4.6.3. Task domains and bounds	106
4.7. Resource availability	107
4.7.1. Resource availability functions	107
4.7.2. How resource availability is calculated	108
4.7.3. Detailed querying of resource availability	111
4.8. Marks and paths	113
4.9. The solution invariant	116
<b>Chapter 5. Extra Types for Solving</b>	<b>117</b>
5.1. Layer trees	117
5.2. Nodes	118
5.3. Layers	122
5.4. Zones	125
5.5. Taskings	127
5.6. Task sets	128
5.7. Meet sets	131
5.8. Time sets	132
5.9. Resource sets	134
5.10. Time frames	137
<b>Chapter 6. Solution Monitoring</b>	<b>140</b>
6.1. Measuring cost	140
6.2. Monitors	141
6.3. Attaching, detaching, and provably zero fixed cost	144
6.4. Event monitors	145
6.4.1. Split events monitors	145
6.4.2. Distribute split events monitors	146
6.4.3. Assign time monitors	146
6.4.4. Prefer times monitors	147
6.4.5. Spread events monitors	147
6.4.6. Link events monitors	148
6.4.7. Order events monitors	148
6.5. Event resource monitors	149
6.5.1. Assign resource monitors	149
6.5.2. Prefer resources monitors	150
6.5.3. Avoid split assignments monitors	150
6.5.4. Limit resources monitors	151

6.6. Resource monitors	151
6.6.1. Avoid clashes monitors	152
6.6.2. Avoid unavailable times monitors	152
6.6.3. Limit idle times monitors	153
6.6.4. Cluster busy times monitors	154
6.6.5. Limit busy times monitors	157
6.6.6. Limit workload monitors	158
6.6.7. Limit active intervals monitors	160
6.7. Timetable monitors	164
6.7.1. Event timetable monitors	164
6.7.2. Resource timetable monitors	166
6.8. Group monitors	169
6.8.1. Basic operations on group monitors	170
6.8.2. Defects	172
6.8.3. Tracing	173
<b>Chapter 7. Matchings and Evenness</b>	<b>175</b>
7.1. The bipartite matching problem	175
7.2. Setting up	177
7.3. Ordinary supply and demand nodes	179
7.4. Workload demand nodes	181
7.4.1. Constructing workload requirements	182
7.4.2. From workload requirements to workload demand nodes	183
7.5. Diagnosing failure to match	184
7.5.1. Visiting unmatched demand nodes	184
7.5.2. Hall sets	185
7.5.3. Finding competitors	186
7.6. Evenness monitoring	187
<b>Part B: Solvers</b>	
<b>Chapter 8. Introducing Solvers</b>	<b>191</b>
8.1. Keeping track of running time	191
8.2. Options, running time, and time limits	193
8.3. General solving	195
8.4. Parallel solving	197
8.5. Monitor adjustments	200
8.5.1. Propagating unavailable times to resource monitors	200
8.6. Gathering statistics	201
8.6.1. Running time and date	201

8.6.2. Files of tables and graphs	201
8.6.3. Tables	202
8.6.4. Graphs	203
8.7. Exponential backoff	205
<b>Chapter 9. Time-Structural Solvers</b>	<b>208</b>
9.1. Layer tree construction	208
9.1.1. Overview	209
9.1.2. Linking	211
9.1.3. Splitting	212
9.1.4. Layering	213
9.1.5. Merging	214
9.2. Time-equivalence	215
9.3. Layers	216
9.3.1. Layer construction	216
9.3.2. Layer coordination	218
9.4. Runarounds	219
9.4.1. Minimum runaround duration	219
9.4.2. Building runarounds	220
9.5. Rearranging nodes	222
9.5.1. Node merging	222
9.5.2. Node meet splitting and merging	222
9.5.3. Node moving	223
9.5.4. Vizier nodes	223
9.5.5. Flattening	225
9.6. Adding zones	225
9.7. Meet splitting and merging	226
9.7.1. Analysing split defects	226
9.7.2. Merging adjacent meets	227
9.8. Monitor attachment and grouping	228
<b>Chapter 10. Time Solvers</b>	<b>231</b>
10.1. Specification	231
10.2. Helper functions	232
10.2.1. Node assignment functions	232
10.2.2. Kempe and ejecting meet moves	233
10.3. Meet bound groups and domain reduction	240
10.3.1. Meet bound groups	240
10.3.2. Exposing resource unavailability	240

10.3.3. Preventing cluster busy times and limit idle times defects .. ..	241
10.4. Some basic time solvers .. .. .	244
10.5. A time solver for runarounds .. .. .	246
10.6. Extended layer matching with Elm .. .. .	247
10.6.1. Introducing layer matching .. .. .	248
10.6.2. The core module .. .. .	250
10.6.3. Splitting supplies .. .. .	255
10.6.4. Improving node regularity .. .. .	256
10.6.5. Handling irregular monitors .. .. .	257
10.7. Time repair .. .. .	259
10.7.1. Node-regular time repair using layer node matching .. .. .	259
10.7.2. Ejection chain time repair .. .. .	260
10.7.3. Tree search layer time repair .. .. .	260
10.7.4. Meet set time repair and the fuzzy meet move .. .. .	262
10.8. Layered time assignment .. .. .	263
10.8.1. Layer assignments .. .. .	263
10.8.2. A solver for layered time assignment .. .. .	264
10.8.3. A complete time solver .. .. .	267
<b>Chapter 11. Resource-Structural Solvers .. .. .</b>	<b>268</b>
11.1. Task bound groups .. .. .	268
11.2. Task trees .. .. .	268
11.3. Task tree construction .. .. .	269
11.4. Tighten to partition .. .. .	271
11.5. Grouping by resource constraints .. .. .	273
11.5.1. Taskers .. .. .	274
11.5.2. Tasker support for grouping .. .. .	276
11.5.3. Tasker support for profile grouping .. .. .	278
11.5.4. Combinatorial grouping .. .. .	280
11.5.5. Applying combinatorial grouping .. .. .	285
11.5.6. Profile grouping .. .. .	286
11.6. Grouping by resource .. .. .	289
11.7. The task grouper .. .. .	289
11.8. Task finding .. .. .	291
11.8.1. Task finder objects .. .. .	292
11.8.2. Widened task sets .. .. .	292
11.8.3. Other task finding operations .. .. .	296
11.9. Other resource-structural solvers .. .. .	297
11.10. Task groups .. .. .	298

<b>Chapter 12. Resource Solvers</b>	300
12.1. Specification	300
12.2. The resource assignment invariant	300
12.3. Unchecked, checked, ejecting, and Kempe task and task set moves	302
12.4. Resource assignment algorithms	305
12.4.1. Satisfying requested task assignments	305
12.4.2. Most-constrained-first assignment	306
12.4.3. Resource packing	307
12.4.4. Split assignments	308
12.5. Resource matching	308
12.5.1. A solver for resource matching	309
12.5.2. Implementing resource matching	312
12.5.3. Time sweep resource assignment	317
12.5.4. Time sweep with lookahead	318
12.5.5. Resource rematching repair	319
12.6. Ejection chain repair	321
12.7. Resource pair repair	321
12.7.1. The basic function	321
12.7.2. A resource pair solver	322
12.7.3. Partition graphs	322
12.7.4. The implementation of resource pair reassignment	324
12.7.5. A simpler resource pair repair	327
12.7.6. Resource pair run reassignment	329
12.7.7. Resource pair swapping	330
12.8. Trying unassignments	330
12.9. Putting it all together	331
<b>Chapter 13. Ejection Chains</b>	333
13.1. Introduction	333
13.2. Ejector construction	335
13.3. Ejector solving	338
13.4. How to write an augment function	340
13.5. Variants of the ejection chains idea	342
13.5.1. Defect promotion	342
13.5.2. Fresh visit numbers for sub-defects	343
13.5.3. Ejection trees	344
13.5.4. Sorting repairs	348
13.5.5. Adjustment on success	348

13.6. Gathering statistics .. .. .	349
13.6.1. Options for choosing ejectors and schedules .. .. .	349
13.6.2. Statistics for analysing Kempe meet moves .. .. .	349
13.6.3. Statistics describing a single solve .. .. .	349
13.6.4. Statistics describing multiple solves .. .. .	350
13.6.5. Organizing augment and repair types .. .. .	352
13.7. Ejection chain time and resource repair functions .. .. .	353
13.7.1. Obtaining ejector objects .. .. .	356
13.7.2. Limiting the scope of changes .. .. .	357
13.7.3. Correlation problems involving demand defects .. .. .	358
13.7.4. Primary grouping and detaching .. .. .	360
13.7.5. Secondary groupings .. .. .	363
13.7.6. Augment functions .. .. .	364
13.7.7. Repair operations for nurse rostering .. .. .	368
<b>Appendix A. Modules Packaged with KHE .. .. .</b>	<b>369</b>
A.1. Arenas and arrays .. .. .	369
A.1.1. Arenas .. .. .	369
A.1.2. Arena sets .. .. .	370
A.1.3. Arrays .. .. .	371
A.1.4. Version string .. .. .	375
A.1.5. Howard's memory allocator .. .. .	375
A.2. Strings and symbol tables .. .. .	378
A.2.1. Strings .. .. .	378
A.2.2. Abort and assert .. .. .	380
A.2.3. Symbol tables .. .. .	381
A.2.4. Narrow strings and symbol tables .. .. .	384
A.2.5. Pointer tables and groups .. .. .	386
A.3. Variable-length bitsets .. .. .	390
A.4. Shiftable sets .. .. .	392
A.5. Priority queues .. .. .	395
A.6. XML handling with KML .. .. .	397
A.6.1. Representing XML in memory .. .. .	397
A.6.2. Error handling and format checking .. .. .	398
A.6.3. Reading XML files .. .. .	400
A.6.4. Writing XML files .. .. .	403

<b>Appendix B. Implementation Notes</b>	.. .. .	405
B.1. Source file organization	.. .. .	405
B.2. Relations between objects	.. .. .	406
B.3. Kernel operations	.. .. .	407
B.4. Monitor updating	.. .. .	410
B.5. Monitor attachment and unattachment	.. .. .	414
B.6. The limit active intervals monitor	.. .. .	416
B.7. An arena and arena set plan	.. .. .	418
<b>References</b>	.. .. .	420

# 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](mailto: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 2.1 is compiled into the program that calls these functions, their results will be "2.1" and "Version 2.1 (May 2018)".

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. 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 symmetrical: 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.

#### **1.4. 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 just 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.

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 1.3), 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 1.3) 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 limit_busy_recode,
    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 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.6. 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`, `infer_resource_partitions`, and `limit_busy_recode` 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.6.3), passing `echo_fp` to it. The characters read are echoed to `echo_fp` if it is non-`NULL`; it would normally be `NULL`.

## 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 limit_busy_recode, 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 limit_busy_recode, 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`, `-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`, `-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.

`-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` and `-i` 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`. 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,
    bool limit_busy_recode, 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, `infer_resource_partitions` is the subject of Section 3.5.6, and `limit_busy_recode` affects how limit busy times constraints are handled, so is described in Section 3.7.15. `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 *KheInstanceMetaDataTable(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 intersection 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 1.3). 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.

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.

Three functions, which should be called only after the instance is complete, are offered for summarising how complex the task of assigning resources of a given type is. The values of these functions are calculated as the instance is built and kept, 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. Finally,

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

returns the number of points of application of limit resources constraints that have this resource type. See Section 12.5.3 for an application of this function.

### 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 intersection 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 1.3. 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 1.3.

### 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.

### 3.5.6. Resource similarity and inferring resource partitions

Following the general approach introduced in Section 1.3, 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 intersection 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 1.3, 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);
```

`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`).

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.

Also after the instance has ended, function

```
bool KheEventResourceEquivalent(KHE_EVENT_RESOURCE er1,
    KHE_EVENT_RESOURCE er2);
```

may be called to decide whether  $er1$  and  $er2$  are *equivalent*. Two event resources are equivalent

when they lie in the same event, and for every resource  $r$ , assigning  $r$  to  $er1$  has the same cost as assigning  $r$  to  $er2$ , because  $er1$  and  $er2$  are monitored by equivalent constraints: constraints of the same kinds with the same weights and other attributes (domains, basically) that affect cost.

The value returned by `KheEventResourceEquivalent` is based on values computed during `KheInstanceMakeEnd`, so `KheEventResourceEquivalent` is very fast. To ensure that `KheInstanceMakeEnd` itself does not run slowly, only event resources that are adjacent in their events are tested for equivalence, and for their constraints to be pronounced equivalent they must appear in the same order. So when `KheEventResourceEquivalent` returns `true`, the event resources really are equivalent; but when it returns `false`, they may or may not be equivalent.

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. `KheConstraintCostFunction` is the cost function used when calculating the cost of deviations, 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.

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 constraint 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  $\text{KheTimeIndex}(t_i) - \text{KheTimeIndex}(t_0)$  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<sub>i</sub>* 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  $t_0$  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.

`KheInstanceMakeEnd` (Section 3.1) has a `limit_busy_recode` option which affects limit busy times constraints. When it is false they are handled in the usual way. When it is true, some limit busy times constraints are replaced by equivalent cluster busy times constraints when solving. Their monitors are more flexible in some ways; for example, they accept cutoff limits.

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  $t_0$  be the first time in `applies_to_tg`. There is one offset for each time  $t_i$  in `applies_to_tg`, including  $t_0$ , such that when  $\text{KheTimeIndex}(t_i) - \text{KheTimeIndex}(t_0)$  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  $\text{KheTimeIndex}(t_i) - \text{KheTimeIndex}(t_0)$  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<sub>i</sub>*, 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<sub>i</sub>*, 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<sub>i</sub>*, 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. Parameter `as` may be `NULL`; for the effect of passing a non-`NULL` value, see Section 4.2.2 below.

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 `KheGeneralSolve2014` (Section 8.3). 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 1.3. 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 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 not to delete the arenas used by solutions and solvers, but rather, within each thread separately, to recycle them.

This is done by creating one arena set `as` (Appendix A.1.2) per thread, and passing `as` to each call to `KheSolnMake` made by the thread. Then the arenas needed to construct the solution are taken from `as` when it has them, and only created afresh when `as` is empty. When the solution is deleted or made into a placeholder, each arena `a` which is no longer needed is not freed. Instead, it is added to `as` after calling `HaArenaRecycle(a)`. If `as` is passed to other calls to `KheSolnMake` made by the same thread, these arenas will be used to store those solutions.

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 participate in this efficient form of recycling too. Instead of creating an arena afresh by a call to `HaArenaMake`, a solver can call

```
HA_ARENA KheSolnArenaBegin(KHE_SOLN soln, bool large);
```

This will extract an arena from `soln`'s arena set if it is non-NULL and non-empty; otherwise it

will return an arena created by `HaArenaMake`. Either way, it will return an arena whose `large` attribute equals `large`.

The right value for `large` is almost certain to be `false`, because large arenas are intended only for when a very large amount of memory is expected to be used. By nominating a few arenas as `large`, the largest demands for memory are concentrated in a few arenas, reducing the overall demand for memory. At present there is one large arena for the internals of each solution, which is recycled when the solution is deleted or converted into a placeholder.

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);
```

If `soln` has a non-NULL arena set, this calls `HaArenaRecycle(a)` and adds the recycled arena to that set. Otherwise it calls `HaArenaDelete`. This is a convenient interface for solvers to use to obtain the arenas they need, without having to worry about the details of arena recycling.

For completeness, there are functions to set and retrieve a solution's arena set:

```
void KheSolnSetArenaSet(KHE_SOLN soln, HA_ARENA_SET as);
HA_ARENA_SET KheSolnArenaSet(KHE_SOLN soln);
```

Here `as` may be `NULL`. Appendix B.7 documents one use for these functions, although the ordinary user of KHE is unlikely to need 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
0	0	0
1	0	1

d	1	2	3
0	0	0	0
1	0	1	1
2	0	0	2
3	0	1	0
4	0	0	1
5	0	1	2

d	1	2	3	4
0	0	0	0	0
1	0	1	1	1
2	0	0	2	2
3	0	1	0	3
4	0	0	1	0
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
13	0	1	1	1
14	0	0	2	2
15	0	1	0	3
16	0	0	1	0
17	0	1	2	1
18	0	0	0	2
19	0	1	1	3
20	0	0	2	0
21	0	1	0	1
22	0	0	1	2
23	0	1	2	3

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

$$(d / \text{fact}(c - 1)) \% c$$

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
0	0	0
1	0	1

d	1	2	3
0	0	0	0
1	0	1	0
2	0	0	1
3	0	1	1
4	0	0	2
5	0	1	2

d	1	2	3	4
0	0	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	1	1	0
4	0	0	2	0
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

$$(d / \text{fact}(c - 1) + d \% \text{fact}(c - 1)) \% c$$

Delightfully, it produces

d	1	2
0	0	0
1	0	1

d	1	2	3
0	0	0	0
1	0	1	1
2	0	0	1
3	0	1	2
4	0	0	2
5	0	1	0

d	1	2	3	4
0	0	0	0	0
1	0	1	1	1
2	0	0	1	2
3	0	1	2	3
4	0	0	2	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

$$\text{KheSolnVisitNum(soln)} - \text{KheSolnEntityVisitNum}(e) \leq \text{slack}$$

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`.

#### 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.6.2. 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);
```

as usual. 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 1.3). 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. 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.

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);
```

The relationship between meets and meet bounds is a many-to-many one.

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);

```

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,  $w(er)/d(e)$ . This is used when evaluating limit workload constraints, so for efficiency it is calculated just once when the task is created, and stored in the task. 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.

Two tasks are said to be *equivalent* when, if they were assigned and those assignments were swapped, effectively nothing would change. Function

```

bool KheTaskEquivalent(KHE_TASK task1, KHE_TASK task2);

```

returns true when `task1` and `task2` are derived from equivalent event resources according to `KheEventResourceEquivalent` (Section 3.6.3), their enclosing meets must have the same duration and the same assigned time (which could be `NULL`), their domains are equal, and their child tasks are pairwise equivalent. What the tasks are currently assigned to, if anything, has no influence on whether they are equivalent.

Ideally the specification would say that there must be some matching of the two sets of child tasks such that each matched pair is equivalent. However that would require sorting the child tasks in some non-trivial way and has not been implemented. So `KheTaskEquivalent` is similar to `KheEventResourceEquivalent` in that when it returns true, the tasks really are equivalent, but when it returns false, they may or may not be equivalent.

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.

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.

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. When several tasks are linked by an avoid split assignments constraint, assigning all but one of them to that one and fix-

ing 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);
```

The relationship between tasks and task bounds is a many-to-many one.

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 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), and its *available load* is its maximum load minus its current load. Available load could be negative, in which case  $r$  is *overloaded*. In that case, at least one of its resource constraints of non-zero weight must be violated.

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 constraints limit  $r$ 's maximum load 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`.

`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. (The other option which supports caching, which is to calculate them 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 resource availability is calculated

This section explains how `KheResourceMaxBusyTimes` and `KheResourceMaxWorkload` are implemented. We start with `KheResourceMaxBusyTimes`. Owing to caching it does its work only once per resource, so it is more concerned with finding a good limit than running quickly.

A resource's maximum number of busy times depends on its avoid unavailable times, limit busy times, and cluster busy times constraints of non-zero weight, soft as well as hard. There are cases where this number is 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 other, more complicated possibilities. A cluster busy times constraint might limit the number of busy days, and then limit busy times constraints might limit the number of busy times on each day. Or there might be limits on each day or week, which need to be added to give the overall limit.

Possibilities like these explain why `KheResourceMaxBusyTimes` is not always exact. It proceeds as follows, for each resource separately. The following applies to one resource,  $r$ .

An *avail node* is a set of times plus 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.

At various points in the following description, it says that an avail node  $x$  with a given set of times and limit is created. This statement is to be understood as subject to these rules:

- If a time  $t$  is known to lie in an avail node with limit 0 containing just  $t$ , then  $t$  can be omitted from every other node without changing the node's limit. All such times are deleted from  $x$ 's times before  $x$  is created, and before the following rules are applied.
- 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.
- 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  $r$ 's constraints to create avail nodes wherever they can be justified, as follows. These four cases are handled first:

1. Suppose that all events have preassigned times, as occurs in nurse rostering but not high school timetabling. Suppose that at some time  $t$ , all of the event resources of  $r$ 's type in the events running at  $t$  are preassigned resources other than  $r$ . Then  $t$  is unassignable as far as  $r$  is concerned. For each such  $t$ , create one avail node containing limit 0 and  $t$ .
2. If  $r$  is subject to an avoid unavailable times constraint of non-zero weight, then create one avail node for each time of the constraint, containing limit 0 and that time.
3. If  $r$  is subject to a limit busy times constraint of non-zero weight with maximum limit 0, then create one avail node for each time of the constraint, containing limit 0 and that time.
4. If  $r$  is subject to a cluster busy times constraint of non-zero weight with maximum limit 0, then create one avail node for each time in each positive time group of the constraint, containing limit 0 and that time.

Next, the algorithm handles these four cases. It makes two passes over the relevant constraints, because a node derived from one can open the way to nodes derived from others.

5. If  $r$  is subject to a limit busy times constraint of non-zero weight with maximum limit  $m > 0$ , then create one avail node for each time group of the constraint, whose times are the times of the time group, and whose limit is  $m$ .
6. Suppose that  $r$  is subject to cluster busy times constraint  $c$  of non-zero weight with maximum limit  $m > 0$ . For each positive time group  $g$  of  $c$ , define a set of times and a limit as follows. The set of times consists of the times of  $g$ , minus any for which there is an avail node with limit 0 containing just that time. 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, in which case the limit is that node's limit. Then define an avail node as follows. Sort the limits of the positive time groups, as just defined, into decreasing order. The new node's times are the times of the positive time groups, and its limit is the sum of the first  $m$  of the sorted limits.

When history is present, the maximum limit  $m$  is replaced by  $\max(0, m - x_i)$  in accordance with the meaning of history. If  $m < x_i$  the resource is overloaded even if every time group is inactive, but that possibility is not taken into account here.

7. Suppose that  $r$  is subject to cluster busy times constraint  $c$  of non-zero weight with a non-zero minimum limit (including not allowing zero). This may be the same constraint as in the previous point. Then  $c$  may be converted into an equivalent cluster busy times constraint  $c'$  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  $c$  says that  $r$  must be free on at least 8 out of 28 days, then  $c'$  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  $c$  has  $n$  time groups, minimum limit  $m$ , 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  $n - m$ . So altogether the maximum limit comes to  $\max(0, (n - m) - (a_i - x_i))$ .

8. Suppose that  $r$  is subject to a limit workload constraint  $c$  of non-zero weight with maximum limit  $m$ . For each time group  $g$  of  $c$ , proceed as follows. For each time  $t$  of  $g$ , find  $w(t, r)$ , the minimum workload per time that  $r$  could incur when it is busy at  $t$ . Sort the  $w(t, r)$  of  $g$  into increasing order, and let  $k$  be the largest integer such that the sum of the first  $k$  of the  $w(t, r)$  does not exceed  $m$ . Then  $k$  is the largest number of times that  $r$  can be busy within  $g$  without violating  $c$ , so create an avail node containing the times of  $g$  with limit  $k$ .

To find  $w(t, r)$ , proceed as follows. Let  $S$  be the set of all event resources whose type is the type of  $r$ . Make the following definitions:

- $w_{uu}$  is the minimum, over all event resources  $s \in S$  that lie in unpreassigned events and are themselves unpreassigned, of the workload per time of  $s$ .
- $w_{pu}(t)$  is the minimum, over all event resources  $s \in S$  that lie in preassigned events that run during time  $t$ , and are themselves unpreassigned, of the workload per time of  $s$ .
- $w_{up}(r)$  is the minimum, over all event resources  $s \in S$  that lie in unpreassigned events and are themselves preassigned  $r$ , of the workload per time of  $s$ .
- $w_{pp}(r, t)$  is the sum, over all event resources  $s \in S$  that lie in preassigned events that run during time  $t$ , and are themselves preassigned  $r$ , of the workload per time of  $s$ .

Define  $w_{uu}$ ,  $w_{pu}(t)$ ,  $w_{up}(r)$ , and  $w_{pp}(r, t)$  to be  $\infty$  when their defining sets of event resources are empty. Set  $w(r, t)$  to  $w_{pp}(r, t)$  if  $w_{pp}(r, t) < \infty$ , and to  $\min(w_{uu}, w_{pu}(t), w_{up}(r))$  otherwise.

If  $w(r, t) = \infty$ , then  $r$  cannot be busy at time  $t$ , so add an avail node containing  $t$  and limit 0, and proceed as though  $t$  is not present in  $g$ . Also, if  $w_{pp}(r, t) < \infty$ , then  $r$  must be busy at  $t$ , so subtract  $w_{pp}(r, t)$  from  $m$  and proceed as though  $t$  is not present in  $g$ .

This ends the first phase. Its result is a set of avail nodes.

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  $L$  is the sum of their limits.

Let the set of times of the whole cycle be  $C$ . The independent set says that of these  $|C|$

times,  $|S|$  times are subject to limit  $L$ . The remaining  $|C| - |S|$  times are not limited. Overall, then, it places a maximum limit of  $L' = L + |C| - |S|$  on the number of times that  $r$  can be busy.

So the second phase finds an independent set for which  $L'$  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  $L'$  is minimum, and returns that  $L'$  as its result.

`KheResourceMaxWorkload` is simpler because it is affected only by limit workload constraints. It works in the same way as `KheResourceMaxBusyTimes`, finding avail nodes and building independent sets, but the avail nodes come from just one source:

9. For each time group of each limit workload constraint of non-zero weight with a maximum limit, build one avail node containing the times of the time group and the maximum limit.

Only independent sets that cover the whole cycle can be used, since the algorithm knows nothing about workload in the uncovered times. The result is the total limit of the chosen set.

To limit running time on large instances, such as the last few CQ14 instances, the algorithm exits early when 20 candidate independent sets have been tried since the most recent new best.

`KheResourceMaxWorkload` produces an integer despite its return type being `float`, because the maximum limits of limit workload constraints are integers. `KheResourceWorkload` and `KheResourceAvailableWorkload`, on the other hand, can return fractional values.

The cases covered here are not the only possibilities. Limit active intervals constraints 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. 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.

To query the availability of a particular resource from `soln`'s instance, start by calling

```
void KheAvailSolverSetResource(KHE_AVAIL_SOLVER as, KHE_RESOURCE r);
```

This runs the algorithm from the previous section on `r`, keeping the resulting best independent sets, one for busy times and one for workload. It is fairly slow, so it is best if all queries about `r` are made after one call to `KheAvailSolverSetResource`.

After that, several functions become available. To begin with,

```
bool KheAvailSolverMaxBusyTimes(KHE_AVAIL_SOLVER as, int *res);
bool KheAvailSolverMaxWorkload(KHE_AVAIL_SOLVER as, float *res);
```

are the same as `KheResourceMaxBusyTimes` and `KheResourceMaxWorkload` except that they query the avail solver about the set resource.

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 section, 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);
void KheAvailSolverMaxBusyTimesAvailNode(KHE_AVAIL_SOLVER as, 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);
void KheAvailSolverMaxWorkloadAvailNode(KHE_AVAIL_SOLVER as, 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. Details of current load may be found by using 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 next.

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` is assigned to `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*.

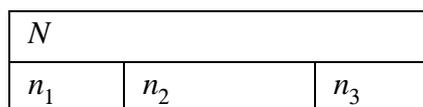
Within each node lie any number of meets. The *node rule*, part of the solution invariant (Section 4.9), imposes a structure on how the meets of nodes may be assigned: if meet is assigned to `target_meet` and lies in node *n*, then *n* has a parent node and `target_meet` lies in that parent node. A layer 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*<sub>1</sub>, *n*<sub>2</sub>, and *n*<sub>3</sub>. The *n*<sub>*i*</sub> 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*<sub>*i*</sub> 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  $\{m_1, m_2\}$ :

$N$		
$n_1$	$n_2$	$n_3$
$m_1$		$m_2$

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):

$N$		
$nm_1$	$n_2$	$n_3$
	$m_2$	$m_3$

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 1.3, 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 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.

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.

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:

```
void KheResourceSetUnion(KHE_RESOURCE_SET rs1, KHE_RESOURCE_SET rs2);
void KheResourceSetIntersect(KHE_RESOURCE_SET rs1, KHE_RESOURCE_SET rs2);
void KheResourceSetDifference(KHE_RESOURCE_SET rs1, KHE_RESOURCE_SET rs2);
```

These update `rs1` to hold its union, intersection, or difference with `rs2`. And functions

```
void KheResourceSetUnionGroup(KHE_RESOURCE_SET rs1,
    KHE_RESOURCE_GROUP rg2);
void KheResourceSetIntersectGroup(KHE_RESOURCE_SET rs1,
    KHE_RESOURCE_GROUP rg2);
void KheResourceSetDifferenceGroup(KHE_RESOURCE_SET rs1,
    KHE_RESOURCE_GROUP rg2);
```

do the same, but with a resource group rather than a resource set. 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`.

Occasionally one needs the cardinality of the results of these set operations, but not the actual sets. For this there is

```
int KheResourceSetUnionCount(KHE_RESOURCE_SET rs1,
    KHE_RESOURCE_SET rs2);
int KheResourceSetIntersectCount(KHE_RESOURCE_SET rs1,
    KHE_RESOURCE_SET rs2);
int KheResourceSetDifferenceCount(KHE_RESOURCE_SET rs1,
    KHE_RESOURCE_SET rs2);
int KheResourceSetSymmetricDifferenceCount(KHE_RESOURCE_SET rs1,
    KHE_RESOURCE_SET rs2);
```

Building the symmetric difference is awkward, so at present there is an operation to find its size, but no operation to find the set itself. And functions

```
int KheResourceSetUnionCountGroup(KHE_RESOURCE_SET rs1,
    KHE_RESOURCE_GROUP rg2);
int KheResourceSetIntersectCountGroup(KHE_RESOURCE_SET rs1,
    KHE_RESOURCE_GROUP rg2);
int KheResourceSetDifferenceCountGroup(KHE_RESOURCE_SET rs1,
    KHE_RESOURCE_GROUP rg2);
int KheResourceSetSymmetricDifferenceCountGroup(KHE_RESOURCE_SET rs1,
    KHE_RESOURCE_GROUP rg2);
```

do the same, but with a resource group rather than a resource set.

Several set queries are available:

```
bool KheResourceSetEqual(KHE_RESOURCE_SET rs1, KHE_RESOURCE_SET rs2);
bool KheResourceSetSubset(KHE_RESOURCE_SET rs1, KHE_RESOURCE_SET rs2);
bool KheResourceSetDisjoint(KHE_RESOURCE_SET rs1, KHE_RESOURCE_SET rs2);
bool KheResourceSetContainsResource(KHE_RESOURCE_SET rs, KHE_RESOURCE r);
```

These return true when `rs1` is equal to `rs2`, when `rs1` is a subset of `rs2`, when `rs1` is disjoint from `rs2`, and 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. `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.

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.

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 6.7) Ordinary and workload demand monitors monitor matchings, and evenness monitors monitor evenness (Chapter 7). Group monitors group together other monitors (Section 6.8). 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`. `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.

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

```
bool KheSolnRetrieveMonitor(KHE_SOLN soln, char *id, KHE_MONITOR *m);
```

This function searches for a monitor whose `Id`, as returned by `KheMonitorId` (see 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 6.8), 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, and that all demand and evenness monitors are detached from the solution, guaranteeing 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. 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 6.7) 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.

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*.

### 6.4.1. 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.

### 6.4.2. 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.

### 6.4.3. 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.

#### 6.4.4. 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.

#### 6.4.5. 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.

### 6.4.6. 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.

### 6.4.7. 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.

## 6.5. 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.

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`.

### 6.5.1. 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. Function

```
void KheAssignResourceMonitorDebug(KHE_ASSIGN_RESOURCE_MONITOR m,
    int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

### 6.5.2. 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. Function

```
void KhePreferResourcesMonitorDebug(KHE_PREFER_RESOURCES_MONITOR m,
    int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

### 6.5.3. 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. 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.

#### 6.5.4. 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`.  
Function

```
void KheLimitResourcesMonitorDebug(KHE_LIMIT_RESOURCES_MONITOR m,
    int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

#### 6.6. 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 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.

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 7.4, are also visited by `KheSolnResourceMonitorCount` and `KheSolnResourceMonitor`. However, the timetable monitor for a resource is not visited by these functions; as explained in Section 6.7, it is retrieved by calling `KheResourceTimetableMonitor`.

### 6.6.1. 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.

### 6.6.2. 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.

### 6.6.3. 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.

#### 6.6.4. 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.

A rough idea of the times monitored by `m` is given by function

```
void KheClusterBusyTimesMonitorRange(KHE_CLUSTER_BUSY_TIMES_MONITOR m,
    KHE_TIME *first_time, KHE_TIME *last_time);
```

It sets `*first_time` and `*last_time` to the first and last times monitored by `m`. In the unlikely event that `m` monitors no times at all, they will be set to `NULL`. These values are calculated the first time that the function is called, and cached in `m`, so that subsequent calls take almost no time.

There are also two functions which 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 cutoff index), 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. Its return value is the value of the condition

```
(*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`.

For the benefit of time sweep algorithms, which may perform better if cluster busy times monitors understand that there is no point in complaining about problems beyond the point that the time sweep has reached, there are functions

```
void KheClusterBusyTimesMonitorSetCutoffIndex(
    KHE_CLUSTER_BUSY_TIMES_MONITOR m, int cutoff_index);
int KheClusterBusyTimesMonitorCutoffIndex(
    KHE_CLUSTER_BUSY_TIMES_MONITOR m);
```

These functions set and retrieve the monitor's *cutoff index*, an integer between 0 and `KheClusterBusyTimesMonitorTimeGroupCount` inclusive, whose effect is explained below. If no cutoff index has been set, `KheClusterBusyTimesMonitorCutoffIndex(m)` returns `KheClusterBusyTimesMonitorTimeGroupCount(m)`. This value cuts off nothing, and should be passed when the aim is to remove a previously set cutoff index.

In practice it will often be easier to call this function:

```
void KheClusterBusyTimesMonitorSetCutoffTime(
    KHE_CLUSTER_BUSY_TIMES_MONITOR m, KHE_TIME cutoff_time);
```

It works out the appropriate cutoff index for ignoring all time groups that contain any time later than `cutoff_time`, and calls `KheClusterBusyTimesMonitorSetCutoffIndex` with that cutoff index. Passing NULL for `cutoff_time` removes any cutoff index.

`KheClusterBusyTimesMonitorSetCutoffTime` examines `m`'s time groups from first

to last, stopping at the first time group that contains a time whose index exceeds the index of `cutoff_time`. The index of that time group is the cutoff index; or if there is no such time group, the cutoff index is `KheClusterBusyTimesMonitorTimeGroupCount(m)`. The function runs much faster than just described when the cutoff times are increasing, as they usually are.

This procedure may seem dubious, given that there is no requirement for the time groups of `m`'s constraint to be added in chronological order. However, cluster busy times monitors sort their time groups into increasing order of the maximum time index in each group.

`KheClusterBusyTimesMonitorSetCutoffIndex` returns true when a non-zero number of time groups is being cut off. `KheClusterBusyTimesMonitorSetCutoffTime` does the same. For example, passing NULL for `cutoff_time` always returns value false. But other cutoff times also return false, when they come after the last time in the last time group.

The general idea is that if a solve is attempting to assign times only up to a certain point in the cycle, then a cutoff index should be set to inform the monitor that there is no point in complaining about things at or beyond that point. This improves the value of the monitor as an influencer of the solve actually under way.

For the record, however, we need to be specific about the effect of a cutoff index. The monitor understands that time groups whose indexes are equal to or larger than the cutoff index are beyond the scope of the current solve. This does not affect busy time groups, which are considered to be active or inactive as usual (depending on their polarity), but it does affect non-busy ones, which are considered to be in an *open* state, that is, not known to be either active or inactive. The monitor then acts conservatively: it considers an open time group to be active when comparing with a minimum limit, and inactive when comparing with a maximum limit. Either way, this makes a violation less likely.

There is also

```
KHE_TIME KheClusterBusyTimesMonitorInitialCutoffTime(
    KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m);
```

This returns the smallest time `t` such that cutting off at `t` is not the same as cutting off at index 0, or NULL if there is no such time.

Suppose the aim is to successively cut `m` off at `t1`, `t2`, and so on to `tn`, the last time. Let `ti = KheClusterBusyTimesMonitorInitialCutoffTime(m)`, and let `tj` be the first time such that `KheClusterBusyTimesMonitorSetCutoffTime(m, tj)` returns false. Then the only calls to set cutoffs that actually need to be made are

```
KheClusterBusyTimesMonitorSetCutoffIndex(m, 0);
KheClusterBusyTimesMonitorSetCutoffTime(m, ti);
...
KheClusterBusyTimesMonitorSetCutoffTime(m, tj);
```

Calls between `t1` and `ti-1` change nothing, and calls after `tj` also change nothing. If there is no `ti`, then cutting off at index 0 is all that is needed.

There is a peculiar but apparently unavoidable asymmetry in the handling of time groups at or after the cutoff 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 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 cutoff 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 cutoff index 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 cutoff index increases these effects become irrelevant anyway.)

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.

### 6.6.5. 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);
```

A rough idea of the times monitored by *m* is given by function

```
void KheLimitBusyTimesMonitorRange(KHE_LIMIT_BUSY_TIMES_MONITOR m,
    KHE_TIME *first_time, KHE_TIME *last_time);
```

It sets `*first_time` and `*last_time` to the first and last times monitored by *m*. In the unlikely

event that  $m$  monitors no times at all, they will be set to `NULL`. These values are calculated the first time that the function is called, and cached in  $m$ , so that subsequent calls take almost no time.

#### 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 *minimum, int *maximum, bool *allow_zero);
```

visit the time groups monitored by  $m$  that are currently defective, in any order. 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  $*tg$ , and  $*minimum$ ,  $*maximum$ , and  $*allow\_zero$  are set to the corresponding values from the constraint; so either the resource is underloaded during  $*tg$  and  $*busy\_count < *minimum$ , or the resource is overloaded during  $*tg$  and  $*busy\_count > *maximum$ . The time groups are the time groups of the constraint, adjusted using `KheTimeGroupNeighbour` by the offset.

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 13.7.3. 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.

### 6.6.6. 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`. Functions

```

int KheLimitWorkloadMonitorDefectiveTimeGroupCount(
    KHE_LIMIT_WORKLOAD_MONITOR m);
void KheLimitWorkloadMonitorDefectiveTimeGroup(
    KHE_LIMIT_WORKLOAD_MONITOR m, int i, KHE_TIME_GROUP *tg,
    float *workload, int *minimum, int *maximum, bool *allow_zero);

```

visit the time groups monitored by `m` that are currently defective, in any order. For each `i`, `*tg` is set to one defective time group, `*workload` is set to the workload of `m`'s resource during `*tg`, and `*minimum`, `*maximum`, and `*allow_zero` are set to the corresponding values from the constraint; so either the resource is underloaded during `*tg` and `*workload < *minimum`, or the resource is overloaded during `*tg` and `*workload > *maximum`. The time groups are the time groups of the constraint, adjusted using `KheTimeGroupNeighbour` by the offset.

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 13.7.3. 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.

### 6.6.7. 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`.

*\*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 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 cutoff index (see below).

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.

For the benefit of time sweep algorithms, which may perform better if active intervals monitors understand that there is no point in complaining about problems beyond the point that the time sweep has reached, there are functions

```
bool KheLimitActiveIntervalsMonitorSetCutoffIndex(
    KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m, int cutoff_index);
int KheLimitActiveIntervalsMonitorCutoffIndex(
    KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m);
```

These functions set and retrieve the monitor's *cutoff index*, an integer between 0 and `KheLimitActiveIntervalsMonitorTimeGroupCount(m)` inclusive, whose effect is explained below. If no cutoff index has been set, `KheLimitActiveIntervalsMonitorCutoffIndex(m)` returns `KheLimitActiveIntervalsMonitorTimeGroupCount(m)`. This value cuts off nothing, and should be passed when the aim is to remove a previously set cutoff index.

In practice it will often be easier to call this function:

```
bool KheLimitActiveIntervalsMonitorSetCutoffTime(
    KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m, KHE_TIME cutoff_time);
```

It works out the appropriate cutoff index for ignoring all time groups that contain any time later than `cutoff_time`, and calls `KheLimitActiveIntervalsMonitorSetCutoffIndex` with that cutoff index. Passing `NULL` for `cutoff_time` removes any cutoff index.

`KheLimitActiveIntervalsMonitorSetCutoffTime` examines `m`'s time groups from first to last, stopping at the first time group that contains a time whose index exceeds the index of `cutoff_time`. The index of that time group is the cutoff index; or if there is no such time group, the cutoff index is `KheLimitActiveIntervalsMonitorTimeGroupCount(m)`. The function runs much faster than just described when the cutoff times are increasing, as they usually are.

`KheLimitActiveIntervalsMonitorSetCutoffIndex` returns `true` when a non-zero number of time groups is being cut off. `KheLimitActiveIntervalsMonitorSetCutoffTime` does the same. For example, passing `NULL` for `cutoff_time` always returns value `false`. But other cutoff times also return `false`, when they come after the last time in the last time group.

The general idea is that if a solve is attempting to assign times only up to a certain point in the cycle, then a cutoff index should be set to inform the monitor that there is no point in complaining about things at or beyond that point. This improves the value of the monitor as an influencer of the solve actually under way.

For the record, however, we need to be specific about the effect of a cutoff index. It influences its monitor in two ways. First, and most simply, active intervals that begin at or after the cutoff index do not attract a cost, no matter how short or long they are. Active intervals that begin before the cutoff index and extend beyond it are not truncated, however, except where the second effect (which we are about to explain) changes the state of some of their time groups.

Second, and more subtly, the monitor understands that time groups whose indexes are equal to or larger than the cutoff index are beyond the scope of the current solve. This does not affect busy time groups, which are considered to be active or inactive as usual (depending on their polarity), but it does affect non-busy ones, which are considered to be in an *open* state, that is, not known to be either active or inactive. The monitor then acts conservatively: it considers an open time group to be active when comparing with a minimum limit, and inactive when comparing with a maximum limit. Either way, this makes a violation less likely.

There is also

```
KHE_TIME KheLimitActiveIntervalsMonitorInitialCutoffTime(
    KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m);
```

This returns the smallest time  $t$  such that cutting off at  $t$  is not the same as cutting off at index 0, or NULL if there is no such time.

Suppose the aim is to successively cut  $m$  off at  $t_1, t_2$ , and so on to  $t_n$ , the last time. Let  $t_i = \text{KheLimitActiveIntervalsMonitorInitialCutoffTime}(m)$ , and let  $t_j$  be the first time such that  $\text{KheLimitActiveIntervalsMonitorSetCutoffTime}(m, t_j)$  returns false. Then the only calls to set cutoffs that actually need to be made are

```
KheLimitActiveIntervalsMonitorSetCutoffIndex(m, 0);
KheLimitActiveIntervalsMonitorSetCutoffTime(m, t_i);
...
KheLimitActiveIntervalsMonitorSetCutoffTime(m, t_j);
```

Calls between  $t_1$  and  $t_{i-1}$  change nothing, and calls after  $t_j$  also change nothing. If there is no  $t_i$ , then cutting off at index 0 is all that is needed.

There is a peculiar but apparently unavoidable asymmetry in the handling of time groups at or after the cutoff 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  $t_g$ . This informs  $m$  that when  $t_g$  is at or after the cutoff 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 cutoff index or anything else. It may change the cost of  $m$ . Function

```
void KheLimitActiveIntervalsMonitorClearNotBusyState(
    KHE_LIMIT_ACTIVE_INTERVALS_MONITOR m, int i);
```

returns  $t_g$  to its default state. (In practice, there is no reason to call this function, because as the cutoff index increases these effects become irrelevant anyway.)

For example, suppose that some resource  $r$  has requested that a certain time group  $t_g$  be kept free. Suppose that  $r$  is subject to a limit active intervals monitor  $m$ , whose  $i$ th time group is a subset of  $t_g$ . 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.

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.

## 6.7. 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.

### 6.7.1. 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 6.8), 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.

### 6.7.2. 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.

Other functions are offered which may be more convenient in some cases. 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. And

```
bool KheResourceTimetableMonitorTimeGroupAvailable(
    KHE_RESOURCE_TIMETABLE_MONITOR rtm, KHE_TIME_GROUP tg,
    bool ignore_nocost, bool certain);
```

returns true when the resource monitored by *rtm* is free at all of the times of *tg*. If *ignore\_nocost* is true, tasks *t* for which `KheTaskNonAssignmentHasCost(t, certain)` returns false are ignored. Function

```
bool KheResourceTimetableMonitorTaskAvailableInFrame(
    KHE_RESOURCE_TIMETABLE_MONITOR rtm, KHE_TASK task, KHE_FRAME frame,
    KHE_TASK ignore_task);
```

is similar but more elaborate. For each time that *task* and its descendants is running, it finds the time group containing that time in *frame*. It returns true when all of those time groups are

available. If `ignore_task` is non-NULL, it ignores any task whose proper root is `ignore_task`. This is useful, for example, when checking whether a swap of the assignments of `task` and `ignore_task` would create no cases of two tasks running in the same time group of frame.

Next come some operations concerned with finding sets of tasks that overlap things:

```
void KheResourceTimetableMonitorAddProperRootTasks(
    KHE_RESOURCE_TIMETABLE_MONITOR rtm, KHE_TIME_GROUP tg,
    bool include_preassigned, KHE_TASK_SET ts);
```

adds to existing task set `ts` the proper root tasks of the tasks of `rtm` that overlap with time group `tg`. It 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. And

```
KHE_BUSY_TYPE KheResourceTimetableMonitorTaskBusyType(
    KHE_RESOURCE_TIMETABLE_MONITOR rtm, KHE_TASK task,
    KHE_FRAME days_frame, KHE_TASK_SET r_ts, bool ignore_nocost,
    bool certain);
KHE_BUSY_TYPE KheResourceTimetableMonitorTaskSetBusyType(
    KHE_RESOURCE_TIMETABLE_MONITOR rtm, KHE_TASK_SET task_set,
    KHE_FRAME days_frame, KHE_TASK_SET r_ts, bool ignore_nocost,
    bool certain);
```

add to existing task set `r_ts` the proper roots of the tasks of `rtm` that are running on the same days as `task` or the tasks of `task_set`, including descendant tasks, using `days_frame` to determine what the days are. They do not add tasks to `r_ts` that are already present. Both functions return a value of type

```
typedef enum {
    KHE_BUSY_NONE = 0,
    KHE_BUSY_SOME = 1,
    KHE_BUSY_ALL = 2
} KHE_BUSY_TYPE;
```

saying whether `rtm` is busy on no day that `task` or `task_set` is running, or one some but not all days, or on all days. If `ignore_nocost`, the calculation of this return value considers tasks `t` for which `KheTaskNonAssignmentHasCost(t, certain)` returns false to be the same as free time. However such tasks are still added to `r_ts`.

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 6.8), 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

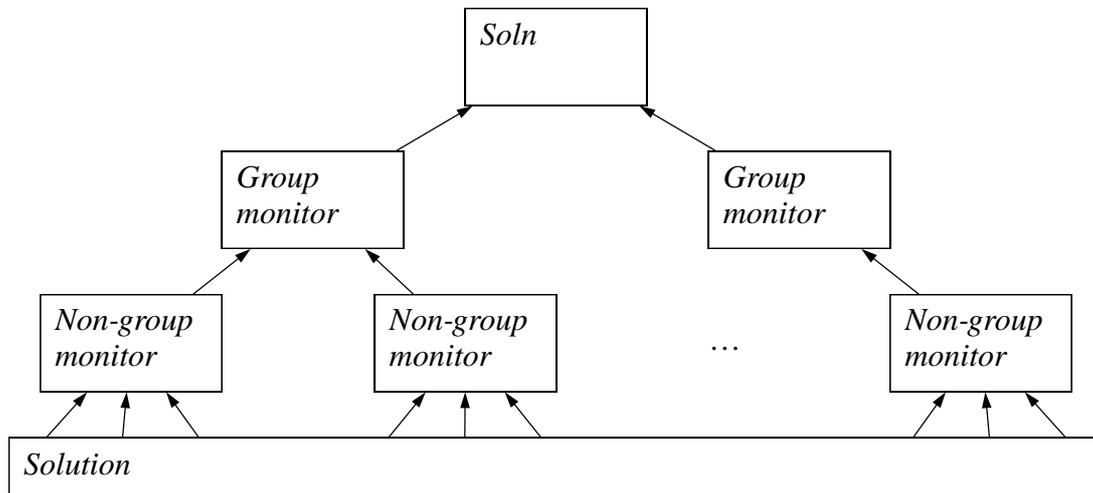
```
void KheResourceTimetableMonitorPrintTimetable(
    KHE_RESOURCE_TIMETABLE_MONITOR rtm, int cell_width, int indent, FILE *fp);
```

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.

## 6.8. 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.

### 6.8.1. 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 KheGroupMonitorDebug(KHE_GROUP_MONITOR gm,
    int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

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. They should attach and detach it from its children, but that has not yet been implemented.

### 6.8.2. 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.

### 6.8.3. 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 made 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. Getting started can be as simple as calling

```
KheSolnMatchingBegin(soln);  
KheSolnMatchingSetWeight(soln, KheCost(1, 0));  
KheSolnMatchingAddAllWorkloadRequirements(soln);  
KheSolnMatchingAttachAllOrdinaryDemandMonitors(soln);
```

after the solution is made a complete representation.

## 7.1. The bipartite matching problem

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 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

(*Smith, 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 8A, 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 7.4). 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. Setting up

By default, a solution contains no matching. To add one, and later to take it away, call

```
void KheSolnMatchingBegin(KHE_SOLN soln);
void KheSolnMatchingEnd(KHE_SOLN soln);
```

`KheSolnMatchingEnd` can be omitted if the matching is needed for the lifetime of the solution, since the matching is deleted when its solution is deleted. There is also

```
bool KheSolnHasMatching(KHE_SOLN soln);
```

which returns `true` when `soln` has a matching. Most of the other operations of this chapter are undefined when no matching is present. Some may abort, others may do nothing.

`KheSolnMatchingBegin` adds exactly one matching to the solution. It is kept up to date thereafter as the solution changes, until `KheSolnMatchingEnd` is called or the solution is deleted. Adding a matching includes adding its demand nodes, each of which is represented by a monitor called a *demand monitor*. Removing a matching includes removing all demand monitors. A demand monitor contributes a cost to the solution just like other monitors do. The cost is 0 when the node is matched, and some non-negative value, set by the user, when it is unmatched.

Demand monitors may be attached and detached individually as usual. Detaching a demand monitor removes its node from the matching graph. Immediately after `KheSolnMatchingBegin` returns, the demand monitors it makes are all detached, so the matching graph has no demand nodes. Convenience functions defined below may be used to attach the demand monitors.

Rather than fiddling around calling `KheSolnHasMatching`, it is conventional to assume that a matching is present when KHE is being used for solving, but not when it is being used only to evaluate solutions. The rationale for this is that by comparison with the overall cost of a solve, it costs virtually nothing, and helps to make the solve environment uniform, if a matching is always present. If it isn't actually wanted, its demand monitors can be detached. On the other hand, when evaluating solutions, at least when just their cost is required, matchings have no use, and if there are many solutions it is best to avoid the memory cost of the demand and supply nodes.

Behind the scenes, a lazy implementation is used: no matching is done until a query operation (for example, a request for the current cost of a demand monitor, or a debug print) occurs, allowing 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.

The cost of one unmatched node is set and retrieved by

```
void KheSolnMatchingSetWeight(KHE_SOLN soln, KHE_COST weight);
KHE_COST KheSolnMatchingWeight(KHE_SOLN soln);
```

For example, a call to

```
KheSolnMatchingSetWeight(soln, KheCost(1, 0));
```

gives each unmatched node a hard cost of 1. The initial weight is 0. A change of weight is reflected immediately in the cost reported by all demand monitors.

Although it would be trivial to allow the user to set the cost of each demand monitor individually, this has not been done, because it might suggest that the matching algorithm is capable of finding the matching which minimizes the total cost of unmatched nodes. In reality, there is no way to make the cost depend on which nodes are unmatched, nor on how appropriate the matching's assignments are. Those would be useful features, since then the cost of assign resources and prefer resources constraints could be reflected in the matching cost, but then a different problem, called *weighted bipartite matching*, would have to be solved, whose algorithm the author considers to be too slow for solving.

In the absence of weighted matching, choosing weight is not easy. The simple choice is `KheCost(1, 0)`, and it may well be the best. Another choice is one which guarantees that the weighted cost of the matching is a lower bound on the ultimate total cost of the violations of all relevant constraints, assuming that more assignments are added without changing the current ones. Each unassigned tixel in the matching must ultimately correspond with either a missing resource assignment at one time, or a resource clash at one time. So a suitable weight is the minimum of the following quantities: for each event resource, the sum of the combined weights of the assign resource constraints that apply to it; and for each resource, the sum of the combined weights of the avoid clashes constraints that apply to it. (Fortunately, both of these constraints incur a cost for each violating tixel.) Function

```
KHE_COST KheSolnMinMatchingWeight(KHE_SOLN soln);
```

works out this value. If there are no event resources and no resources, it returns 0.

The matching has a *type* that may be changed at any moment:

```
KHE_MATCHING_TYPE KheSolnMatchingType(KHE_SOLN soln);
void KheSolnMatchingSetType(KHE_SOLN soln, KHE_MATCHING_TYPE mt);
```

`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 an enhanced 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 cost reported by all attached 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. Ordinary supply and demand nodes

This section explains how most of the supply and demand nodes of the matching, the ones associated with meets, are defined. Since they are linked together with edges that depend on the type of the matching, this section also explains `KHE_MATCHING_TYPE` in detail.

For each offset of a meet `meet` (for each integer between 0 inclusive and the duration of `meet` exclusive), the matching contains  $R$  ordinary supply nodes, where  $R$  is the total number of resources in the instance. If `meet` has duration  $d$ , this is  $dR$  supply nodes altogether. Each models the supply of one resource at one offset. These supply nodes cannot be accessed by the user.

Each task of `meet` contains `KheMeetDuration(meet)` demand nodes, which will be called *ordinary demand nodes* to distinguish them from the workload demand nodes to be defined later. Each models the demand made by its task at one offset. Ordinary demand nodes have type `KHE_ORDINARY_DEMAND_MONITOR` and may be accessed in the usual way by

```
int KheTaskDemandMonitorCount(KHE_TASK task);
KHE_ORDINARY_DEMAND_MONITOR KheTaskDemandMonitor(KHE_TASK task, int i);
```

The first function's value is equal to the duration of the enclosing meet. Like most monitors, these ones cannot be created or deleted by the user. They are created when the task is created, split and merged when it is split and merged, and deleted when it is deleted. Unlike other monitors, they are detached initially. This is so that, by default, KHE monitors only the official cost.

In addition to the operations applicable to all monitors, ordinary demand monitors offer

```
KHE_TASK KheOrdinaryDemandMonitorTask(KHE_ORDINARY_DEMAND_MONITOR m);
int KheOrdinaryDemandMonitorOffset(KHE_ORDINARY_DEMAND_MONITOR m);
```

returning the task that *m* monitors, and its offset within that task. Helper functions

```
void KheSolnMatchingAttachAllOrdinaryDemandMonitors(KHE_SOLN soln);
void KheSolnMatchingDetachAllOrdinaryDemandMonitors(KHE_SOLN soln);
```

ensure that all ordinary demand monitors are attached or detached; they visit every ordinary demand monitor of every task of every meet of *soln*, check whether it is currently attached, then attach or detach it if required. Function

```
void KheOrdinaryDemandMonitorDebug(KHE_ORDINARY_DEMAND_MONITOR m,
int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

Although the list of monitors in a task is fixed, each may be attached or detached individually, and they may be linked by edges to supply nodes in different ways, depending on the matching type, as will now be explained.

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 ordinary 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 non-empty 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. Workload demand nodes

In addition to ordinary demand nodes, matchings may contain *workload demand nodes*, used to take account of avoid unavailable times constraints, limit busy times constraints, and limit workload constraints, collectively called *workload demand constraints* here. For example, suppose the cycle contains 40 times, and teacher *Smith* has a required workload limit of 30 times and is unavailable at time *Mon1*. Then ten workload demand nodes should be created, one demanding supply tixel (*Smith, Mon1*), and the other nine demanding *Smith* at one unrestricted 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 in most cases it is enough to call helper function

```
void KheSolnMatchingAddAllWorkloadRequirements(KHE_SOLN soln);
```

This may be done at any time, and does what is usually wanted. However, it is partly heuristic, so KHE offers the option of controlling the details.

For the purposes of matchings only, a *workload requirement* is a requirement imposed on a resource that it be occupied attending meets for at most a given number of the times of some time group. There are no operations for creating workload demand nodes directly; instead, there are operations for defining workload requirements, and the workload demand nodes are derived from them by KHE behind the scenes, as explained below (Section 7.4.2).

Within a solution at any moment, a sequence of workload requirements is associated with each resource. They may be visited in order by calling

```
int KheSolnMatchingWorkloadRequirementCount(KHE_SOLN soln,
      KHE_RESOURCE r);
void KheSolnMatchingWorkloadRequirement(KHE_SOLN soln, KHE_RESOURCE r,
      int i, int *num, KHE_TIME_GROUP *tg, KHE_MONITOR *m);
```

The first returns the number of workload requirements associated with *r* in *soln*, and the second returns the *i*'th requirement, in the form of a number of times and a time group. If the third return parameter, *\*m*, is non-NULL, it is the *originating monitor*: the monitor that gave rise to this requirement. The originating monitor is stored in workload demand monitors created as a consequence of this requirement, to assist in analysing defects; it is not otherwise used.

Each resource has no workload requirements initially. To change the requirements of resource *r*, begin with a call to

```
void KheSolnMatchingBeginWorkloadRequirements(KHE_SOLN soln, KHE_RESOURCE r);
```

continue with any number of calls to

```
void KheSolnMatchingAddWorkloadRequirement(KHE_SOLN soln,
      KHE_RESOURCE r, int num, KHE_TIME_GROUP tg, KHE_MONITOR m);
```

where *m* may be NULL, and end with a call to

```
void KheSolnMatchingEndWorkloadRequirements(KHE_SOLN soln,
      KHE_RESOURCE r);
```

All three functions must be called, in order. The first clears  $r$ 's workload requirements, the second appends a requirement that  $r$  attend events for at most  $\text{num}$  of the times of  $\text{tg}$  ( $\text{num}$  may not exceed the number of times in  $\text{tg}$ ), and the third replaces any existing workload demand nodes for  $r$  with new ones derived from the workload requirements. The new monitors are attached as they are created. `KheMatchingMonitorSetAllWorkloadRequirements` calls these functions. The sections below describe the calls it makes, and how workload requirements are converted into workload demand nodes.

To delete the workload requirements of  $r$ , along with their workload demand nodes, call

```
void KheSolnMatchingDeleteWorkloadRequirements(KHE_SOLN soln,
      KHE_RESOURCE r);
```

`KheSolnMatchingBeginWorkloadRequirements` does this, as does `KheSolnMatchingEnd` when deleting the whole matching.

The workload demand nodes created by `KheSolnMatchingEndWorkloadRequirements` are monitors of type `KHE_WORKLOAD_DEMAND_MONITOR`. Like other monitors of resources, they appear on the list of monitors visited by functions `KheResourceMonitorCount` and `KheResourceMonitor` from Section 6.6.

In addition to the operations applicable to all monitors, workload demand monitors offer

```
KHE_RESOURCE KheWorkloadDemandMonitorResource(
      KHE_WORKLOAD_DEMAND_MONITOR m);
KHE_TIME_GROUP KheWorkloadDemandMonitorTimeGroup(
      KHE_WORKLOAD_DEMAND_MONITOR m);
KHE_MONITOR KheWorkloadDemandMonitorOriginatingMonitor(
      KHE_WORKLOAD_DEMAND_MONITOR m);
```

These return the resource that the workload demand monitor is for, the time group of the workload requirement that led to  $m$ , and the originating monitor (possibly `NULL`) of the workload requirement that led to  $m$ . Finally, function

```
void KheWorkloadDemandMonitorDebug(KHE_WORKLOAD_DEMAND_MONITOR m,
      int verbosity, int indent, FILE *fp);
```

is like `KheMonitorDebug`, only specific to this type of monitor.

### 7.4.1. Constructing workload requirements

This section explains how `KheSolnMatchingAddAllWorkloadRequirements` works. For each resource  $r$ , it first calls `KheSolnMatchingBeginWorkloadRequirements(soln, r)`, and then visits each required workload demand monitor  $m$  of weight greater than 0 applicable to  $r$ , in order of decreasing weight. What it does with each monitor is explained below. It then finishes its work on  $r$  with a call to `KheSolnMatchingEndWorkloadRequirements(soln, r)`.

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 it calls

```
KheSolnMatchingAddWorkloadRequirement(soln, r, 0,
    KheTimeSingletonTimeGroup(t), m);
```

If  $m$  is a limit busy times monitor with `Maximum` greater than 0, then for each time group  $tg$  in  $m$ 's constraint it calls

```
KheSolnMatchingAddWorkloadRequirement(soln, r, k, tg);
```

where  $k$  is the `Maximum` attribute. 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 `KheSolnMatchingAddWorkloadRequirement` is called once with this time group. The number passed to this call requires careful calculation, involving the workloads of all events. The remainder of this section explains this calculation.

Let  $k$  be the integer eventually passed to `KheSolnMatchingAddWorkloadRequirement`. 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.

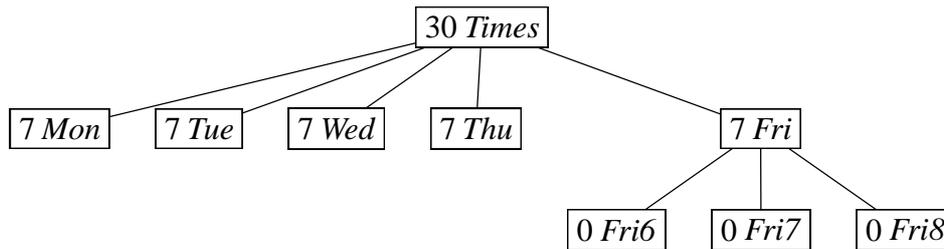
These tests are actually applied to clusters of events known to be running simultaneously, because of required link events constraints or preassignments and other time domain restrictions. Each resource can be assigned to at most one of the event resources of the events of a cluster, so only one of the events, the one whose modelling is least exact, needs to be taken account of.

#### 7.4.2. From workload requirements to workload demand nodes

KHE converts workload requirements to workload demand nodes automatically, during the call to `KheSolnMatchingEndWorkloadRequirements` (defined above). The following explanation of how this is done, adapted from [9], is included for completeness.

When converting workload requirements into workload demand nodes, the relationships between the requirements' sets of times affect the outcome. In general, an exact conversion seems to be possible only when these sets of times satisfy the *subset tree condition*: each pair of sets of times 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 set of times minus its number minus the number of demand nodes contributed by its descendants, or none if this number is negative.

The tree is built by inserting the workload requirements in order, 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 required constraints.

## 7.5. 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.

### 7.5.1. 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.

### 7.5.2. 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 P. 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.

### 7.5.3. 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);
```

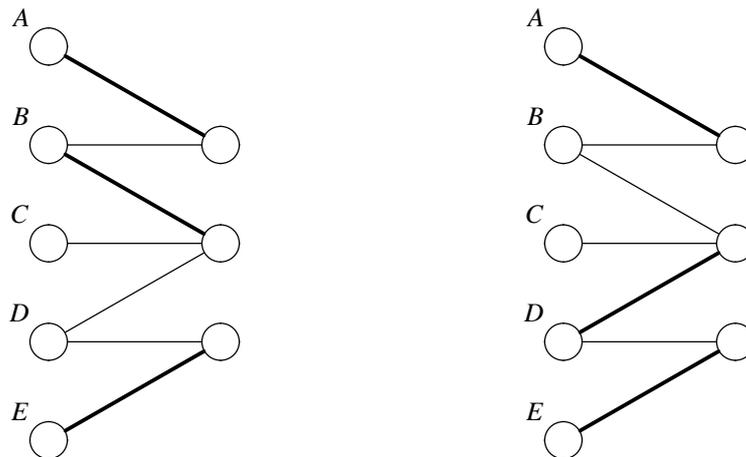
may be used together to visit the competitors of unmatched demand monitor  $m$ :

```
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.

## 7.6. 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 evenness 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 8.3), 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.

# Part B

## Solvers

# Chapter 8. Introducing Solvers

A solver is an operation that makes large-scale changes to a solution. This chapter introduces solvers, defines interfaces for them, presents a few high-level ones, and explains some general ideas related to solving, including setting options and gathering statistics.

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, 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 don't need that.

## 8.1. Keeping track of running time

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.

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. Two functions give access to elapsed time:

```
float KheTimerElapsedTime(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. `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 searches 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. Function

```
bool KheTimerSetTimeLimitReached(KHE_TIMER_SET timer_set);
```

return true if at least one of the timers of `timer_set` has reached its time limit. This is the

logical moment to stop if several time limits are present. 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 when it is no longer needed; instead, delete or recycle 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 8.4) 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`.

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.

Each option is described along with the solver it affects. 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, and they are the natural place to keep

other things which are not options, strictly speaking. In particular, each option 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);
bool KheOptionsTimeLimitReached(KHE_OPTIONS options);
void KheOptionsTimerSetDebug(KHE_OPTIONS options, int verbosity,
    int indent, FILE *fp);
```

`KheOptionsAddTimer` creates a new timer with the given attributes and adds it to the timer set within `options`. `KheOptionsDeleteTimer` deletes the given timer from that timer set; it must be present. `KheOptionsContainsTimer` searches the timer set for a timer with the given tag. `KheOptionsTimeLimitReached` returns true if any of the timer set's time limits have been reached, and `KheOptionsTimerSetDebug` produces a debug print of the timer set of options onto `fp` with the given verbosity and indent. These functions are simple delegations to the corresponding timer set functions.

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. General solving

A *solver* is a function that finds solutions, or partial solutions, to instances. 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 KheGeneralSolve2020(KHE_SOLN soln, KHE_OPTIONS options);
```

This single-threaded general 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.

`KheGeneralSolve2020` assumes that `soln` is as returned by `KheSolnMake`, so it begins

with `KheSolnSplitCycleMeet` and `KheSolnMakeCompleteRepresentation`. Then it calls solvers defined in this guide: it builds a layer tree and task tree, attaches demand monitors, calls `KheCycleNodeAssignTimes` to assign times, and the three `KheTaskingAssignResources` functions to assign resources, ending with `KheSolnEnsureOfficialCost`.

For convenience, `KheGeneralSolve2020` calls

```
KheOptionsSetRunningTime(options, elapsed_time);
```

(Section 8.2) just before returning, where `elapsed_time` is its running time, obtained by calling `KheOptionsTimeLimitNow` (Section 8.2) on a time limit with tag "general" which it creates at its start and deletes at its end. Arguably, this is not quite right, because `soln` has to be created before `KheGeneralSolve2020` is called, and more work could be done on `soln` afterwards. However, callers can easily reset the running time if they wish to.

By convention, options set or consulted directly by `KheGeneralSolve2020` have names beginning with `gs_`. Here is the full list:

`gs_diversifier`

An integer option which, when set, causes `KheGeneralSolve2020` to set the diversifier of the solution it is given to the given value. When omitted, the diversifier retains the value it has when `KheGeneralSolve2020` is called.

`gs_time_limit`

A string option defining a soft time limit for the solve. Enforcement is up to particular solvers; this option merely calls `KheOptionsSetTimeLimit` (Section 8.2). 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.

`gs_matching_off`

A Boolean option which, when "true", instructs `KheGeneralSolve2020` to refrain from installing the global tixel matching (Chapter 7).

`gs_monitor_evenness`

A Boolean option which, when "true", instructs `KheGeneralSolve2020` to install evenness monitors (Section 7.6).

`gs_propagate_unavailable_times_off`

A Boolean option which, when "true", instructs `KheGeneralSolve2020` to omit its usual call to `KhePropagateUnavailableTimes` (Section 8.5.1).

`gs_time_assignment_only`

A Boolean option which, when "true", instructs `KheGeneralSolve2020` to exit early, leaving the solution in its state after time assignment.

`gs_unassignment_off`

A Boolean option which, when "true", instructs `KheGeneralSolve2020` to omit the calls to `KheSolnTryTaskUnAssignments` (Section 12.8) and `KheSolnTryMeetUnAssignments` (Section 10.4) during the cleanup phase.

`gs_event_timetable_monitor`

During the resource assignment phase of `KheGeneralSolve2020`, this option has a value of type `KHE_EVENT_TIMETABLE_MONITOR`, and holds the result of the call

```
KheEventTimetableMonitorMake(soln, KheInstanceFullEventGroup(ins));
```

The monitor is attached. Before and after that phase, the option is either absent or has value `NULL`. The point of this is that this event timetable monitor is expensive to create and probably too expensive to update during time assignment, but it is useful during resource assignment. So this arrangement gives resource assignment algorithms access to a single shared event timetable monitor, at little cost.

`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 `KheGeneralSolve2020` 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 `KheGeneralSolve2020`, 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?'.

`KheGeneralSolve2020` is affected indirectly by many other options, via the solvers it calls.

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 "global"; if not, the running time is omitted), and ends with whatever `fprintf(stderr, fmt, ...)` would produce, followed by a newline.

## 8.4. 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.

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`.

If `as != NULL`, each call to `KheSolnMake` is passed an arena set, as Section 4.2.2 suggests. There is one arena set per thread, with `as` serving one thread and freshly created arena sets serving the others. At the end, the idle arenas in all arena sets other than `as` are moved into `as`, and the arena sets in all solutions kept by the two functions are set to `as` by calls to `KheSolnSetArenaSet` (Section 4.2.2). If further parallel solving of these solutions is attempted, it will be necessary to install distinct arena sets first.

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` make per instance. If `ps_make` is absent, its value is taken to be 1.

`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, the name is extended so that it does not clash with existing solution group names.

If `ps_soln_group` is omitted, or its name is wrong, no solution group is made. When solutions have been found but they are not in the result archive, this is the usual reason.

`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 uses

```
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 `KheGeneralSolve2020`, for example, each holds the wall clock time from when `KheGeneralSolve2020` 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`  $\geq$  `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.

`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.

On the author's quad-core machine, finding 8 solutions by running 8 threads is usually somewhat faster than finding them by running 4 threads. The effect is not large. It is presumably due to the hardware hyper-threading feature, which allows up to two threads to run on each processor in an attempt to improve throughput. But there is also a random element concerning whether two slow solves happen to be allocated to the same thread, so it is hard to be sure.

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.

## 8.5. Monitor adjustments

In this section we present solvers which adjust monitors. At present we have only one to present here, although the monitor grouping solvers from Chapter 13 would also fit.

### 8.5.1. 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 6.6.4) or `KheLimitActiveIntervalsMonitorSetNotBusyState` (Section 6.6.7).

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.

## 8.6. Gathering statistics

KHE offers a module for gathering statistics. It can calculate running times and generate files containing tables in several formats, and graphs in Lout format.

### 8.6.1. Running time and date

To find out how long something takes to run, objects of type `KHE_STATS_TIMER` (the usual pointer to a private record) are used. Each records one moment in time. To create and delete these timer objects, the functions are

```
KHE_STATS_TIMER KheStatsTimerMake(void);
void KheStatsTimerDelete(KHE_STATS_TIMER st);
```

`KheStatsTimerMake` returns a new timer, initialized by calling `KheStatsTimerReset` on it, and `KheStatsTimerDelete` deletes `st`, reclaiming the memory it used. There is also

```
KHE_STATS_TIMER KheStatsTimerCopy(KHE_STATS_TIMER st);
```

which copies `st`, producing a new timer holding the same time as `st`. The other functions are

```
void KheStatsTimerReset(KHE_STATS_TIMER st);
float KheStatsTimerNow(KHE_STATS_TIMER st);
```

`KheStatsTimerReset` resets the time held within `st` to the time when `KheStatsTimerReset` was called. `KheStatsTimerNow` compares the time recorded in `st` (when `KheStatsTimerReset` was last called) with the time now and reports the difference in seconds. Both functions may be called any number of times on the same timer. Any number of timers may be used independently.

Because wall clock times are used, times measured within one thread of a parallel solve will not in general measure the time consumed by that thread. However, a parallel solver can be called between `KheStatsTimerReset` and `KheStatsTimerNow`, and then they will reliably measure the elapsed time of the parallel solve.

Also offered is

```
char *KheStatsDateToday(void);
```

which returns the current date as a string in static memory.

For the sake of compilations that do not have the Unix system functions called by these functions, file `khe.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, all functions will still compile and run without error, but `KheStatsTimerNow` will always return -1.0, and `KheStatsDateToday` will return "?".

### 8.6.2. Files of tables and graphs

The main thing that the stats module does is generate files of tables and graphs. Any number of files may be generated simultaneously (not in parallel, because the stats module has no locking, but by one thread). One file may contain any number of tables and graphs, although only one may be generated at a time within any one file.

To begin and end a file, call

```
void KheStatsFileBegin(char *file_name);
void KheStatsFileEnd(char *file_name);
```

This writes a file called `file_name` in sub-directory `stats` of the current directory (which the user must have created previously). The file is opened by `KheStatsFileBegin` and closed by `KheStatsFileEnd`. To generate the actual tables and graphs, see the following subsections.

### 8.6.3. Tables

To generate tables, make matching pairs of calls to the following functions in between the calls to `KheStatsFileBegin` and `KheStatsFileEnd`:

```
void KheStatsTableBegin(char *file_name, KHE_STATS_TABLE_TYPE table_type,
    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 KheStatsTableEnd(char *file_name);
```

Only one table at a time can be generated into a given file, so a table is not identified separately from its file. The table is begun by `KheStatsTableBegin`, and finished, including being written out to the file, by `KheStatsTableEnd`. 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 `KheStatsTableEnd` is called, these other calls may occur in any order. In particular it is equally acceptable to generate the table row by row or column by column.

The format of the table is specified by `table_type`:

```
typedef enum {
    KHE_STATS_TABLE_PLAIN,
    KHE_STATS_TABLE_LOUT,
    KHE_STATS_TABLE_LATEX
} KHE_STATS_TABLE_TYPE;
```

The choices are plain text, Lout, or LaTeX. Parameter `col_width` determines the width in characters of each column in plain text; 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 KheStatsCaptionMake(char *file_name, char *fmt, ...);
```

at any time between `KheStatsTableBegin` and `KheStatsTableEnd`, 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 `KheStatsTableEnd`.

In any given table, each row except the first (header) row must be declared, by calling

```
void KheStatsRowAdd(char *file_name, 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 KheStatsColAdd(char *file_name, 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 KheStatsAddEntryString(char *file_name, char *row_label,
    char *col_label, char *str);
void KheStatsAddEntryCost(char *file_name, char *row_label,
    char *col_label, KHE_COST cost);
void KheStatsAddEntryTime(char *file_name, char *row_label,
    char *col_label, float time);
void KheStatsAddEntryInt(char *file_name, char *row_label,
    char *col_label, int val);
```

These add an entry to `file_name`'s table 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,

```
KheStatsAddEntryCost(file_name, row_label, col_label, KheSolnCost(soln));
```

adds a solution cost to the table which will be formatted in the standard way.

All strings passed to these functions that require long-term storage are copied, so mutating strings are not a concern. On the other hand, there is no locking, so calls which create tables should be single-threaded, as should calls which modify the same table.

#### 8.6.4. Graphs

To generate graphs in Lout format, make matching pairs of calls to the following functions in between the calls to `KheStatsFileBegin` and `KheStatsFileEnd`:

```
void KheStatsGraphBegin(char *file_name);
void KheStatsGraphEnd(char *file_name);
```

As for tables, only one graph can be generated into a given file at a time, and so the graph is identified by the file name. To set options which control the overall appearance of the graph, call

```
void KheStatsGraphSetWidth(char *file_name, float width);
void KheStatsGraphSetHeight(char *file_name, float height);
void KheStatsGraphSetXMax(char *file_name, float xmax);
void KheStatsGraphSetYMax(char *file_name, float ymax);
void KheStatsGraphSetAboveCaption(char *file_name, char *val);
void KheStatsGraphSetBelowCaption(char *file_name, char *val);
void KheStatsGraphSetLeftCaption(char *file_name, char *val);
void KheStatsGraphSetRightCaption(char *file_name, char *val);
```

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.

A caption can be added by calling the same function as for tables:

```
void KheStatsCaptionMake(char *file_name, char *fmt, ...);
```

at any time between `KheStatsGraphBegin` and `KheGraphTableEnd`.

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

```
void KheStatsDataSetAdd(char *file_name, char *dataset_label,
    KHE_STATS_DATASET_TYPE dataset_type);
```

where `dataset_label` is used to identify the dataset, and `dataset_type` determines how the data are presented. At present the stats module offers just one choice:

```
typedef enum {
    KHE_STATS_DATASET_HISTO
} KHE_STATS_DATASET_TYPE;
```

but the Lout graph package offers many others, so it would not be difficult to expand the choices here. `KHE_STATS_DATASET_HISTO` prints a histogram. The x values of the dataset's points should be increasing integers; the y values are the frequencies. Function

```
void KheStatsPointAdd(char *file_name, char *dataset_label,
    float x, float y);
```

adds a point to a dataset. The points are generated in the order received, so in practice, successive calls to `KheStatsPointAdd` on the same dataset should have increasing x values.

## 8.7. 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 opportunities 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.

# 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 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 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  $\lfloor N \rfloor$  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 functions 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

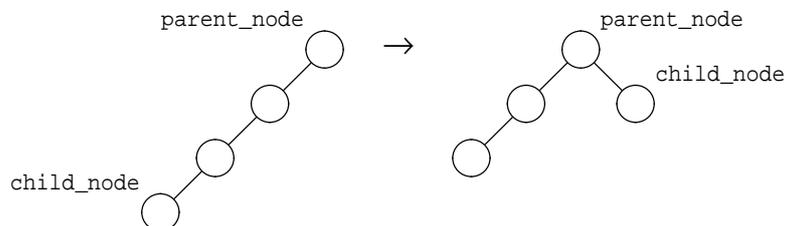
```
bool KheNodeMoveCheck(KHE_NODE child_node, KHE_NODE parent_node);
bool KheNodeMove(KHE_NODE child_node, KHE_NODE parent_node);
```

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.

## 9.8. Monitor attachment and grouping

Sometimes, how monitors are grouped and attached is important: when using ejection chains (Chapter 13), for example, or Kempe and ejecting meet moves (Section 10.2.2). This section lays out some general concepts and conventions for monitor attachment and grouping.

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 should always be used, because it is the efficient way.

Some solvers need specific groupings. The Kempe meet move operation (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 over and over again. 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.

Two monitors (or defects) are *correlated* when they monitor the same thing, not formally usually, 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.

Correlated defects are bad for ejection chains. The cost of each defect separately might not be large enough to end the chain if removed, causing the 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. So correlated monitors should be grouped, whenever possible. These groups are the equivalence classes of the correlation relation, which is clearly an equivalence relation. A grouping of correlated monitors is called a *primary grouping*.

A function which creates a primary grouping works as follows. Monitors not relevant to the grouping remain as they were. Relevant monitors are deleted from any parents they have, and partitioned into groups of correlated monitors. For each group containing two or more monitors, a group monitor called a *primary group monitor* is made, the monitors are made children of it, and it is made a child of the solution object. For each group containing one monitor, that monitor is made a child of the solution, and no group monitor is made. Any group monitors other than

the solution object which lose all their children because of these changes are deleted, possibly causing further deletions of childless group monitors.

A function which deletes a primary grouping visits all monitors relevant to the grouping and deletes those parents of those monitors whose `sub_tag` indicates that they are part of the primary grouping. The deleting is done by calls to `KheGroupMonitorBypassAndDelete`.

Function `KheEjectionChainPrepareMonitors` (Section 13.7.4) creates primary groupings of some correlated monitors, and detaches others, in preparation for ejection chain repair.

*Secondary groupings* are useful groupings that are not primary groupings (that do not group monitors which monitor the same thing). Instead, they group diverse sets of monitors for particular purposes, such as efficient access to defects.

Secondary groupings are often built on primary groupings: if a monitor that a secondary grouping handles is a descendant of a primary group monitor, the primary group monitor goes into the secondary grouping, replacing the individual monitors which are its children.

A secondary grouping makes one group monitor, called a *secondary group monitor*, not many. The secondary 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. A secondary grouping can be removed by passing the secondary group monitor to `KheGroupMonitorDelete`.

Functions for creating secondary 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 13.7.5).

When building secondary groupings, these public functions are often helpful:

```
bool KheMonitorHasParent(KHE_MONITOR m, int sub_tag,
    KHE_GROUP_MONITOR *res_gm);
void KheMonitorAddSelfOrParent(KHE_MONITOR m, int sub_tag,
    KHE_GROUP_MONITOR gm);
void KheMonitorDeleteAllParentsRecursive(KHE_MONITOR m);
```

Consult the documentation in the source code to find out what they do.

It is convenient to have standard values for the sub-tags and sub-tag labels of the group monitors created by grouping functions, both primary and secondary. 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.

# 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 secondary grouping, as defined in Section 9.8. 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 `KheKempMeetMove` 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, `KheKempMeetMove` 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 7.5.3) are used to find the demand monitors competing with them for supply. These can be of four kinds:

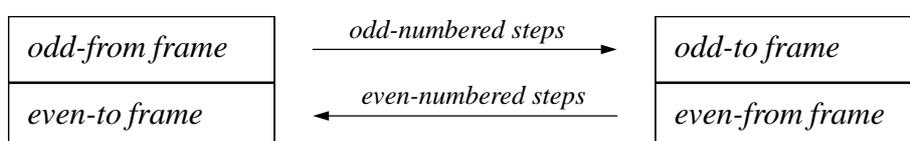
1. A workload demand monitor derived from an avoid unavailable times monitor signals that a preassigned resource has moved to an unavailable time, so fail.
2. Any other workload demand monitor signals a workload overload other than an unavailable time, so ignore it. At a higher level, this defect might 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.

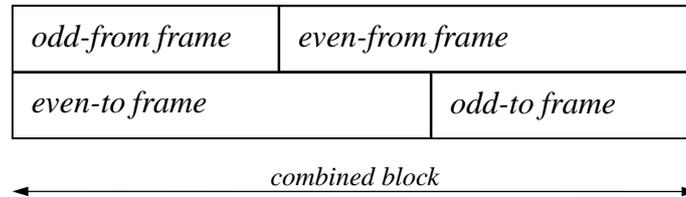
Given the call `KheKempMeetMove(meet, target_meet, offset, ...)`, the target meet of the odd-from frame and the even-to frame is `KheMeetAsst(meet)`, and the target meet of the even-from frame and the odd-to frame is `target_meet`. These may be equal, or not.

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.

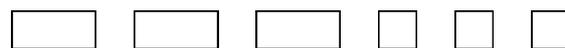
Parameter `sec` is optional (may be `NULL`); a simple choice for it would be any spread events constraint whose number of points of application is maximal. If `sec` is present, the algorithm tries to assign the same number of meets to each of `sec`'s time groups. To see why, consider an example of the 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

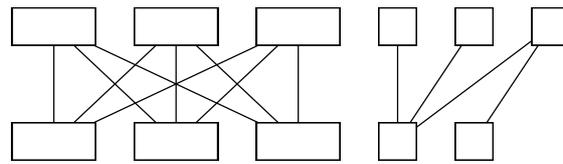
This section introduces layer matching. Later sections describe the implementation. Suppose some layer has three meets of duration 2 and two meets of duration 1, like this:



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 unmatchable, 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  $\text{KheElmSupplyOffset}(s) < \text{offset}$ , then a supply *\*ls* is split off *s* at left, holding the offsets from  $\text{KheElmSupplyOffset}(s)$  inclusive to *offset* exclusive; otherwise *\*ls* is set to *NULL*. If  $\text{offset} + \text{durn} < \text{KheElmSupplyOffset}(s) + \text{KheElmSupplyDuration}(s)$ , then a supply *\*rs* is split off *s* at right, holding the offsets from  $\text{offset} + \text{durn}$  inclusive to  $\text{KheElmSupplyOffset}(s) + \text{KheElmSupplyDuration}(s)$  exclusive; otherwise *\*rs* is set to *NULL*. The original *s* is left with offsets from *offset* inclusive to  $\text{offset} + \text{durn}$  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 13.7.

### 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 13.7), 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.

**10.8.3. A complete time solver**

Time solver

```
bool KheCycleNodeAssignTimes(KHE_NODE cycle_node, KHE_OPTIONS options);
```

combines the ideas of this chapter into one solver that assigns the meets in the proper descendants of `cycle_node`, assumed to be the cycle node.

`KheCycleNodeAssignTimes` first assigns preassigned meets. If all events have preassigned times, according to `KheInstanceAllEventsHavePreassignedTimes`, it does nothing else. Otherwise it assigns times layer by layer using `KheNodeLayeredAssignTimes` (Section 10.8.2). Then it removes any regularity features (zones and interior nodes) installed earlier and returns.

If not all events have preassigned times, this function is influenced by three options:

`ts_cluster_meet_domains`

A Boolean option which, when true, instructs `KheCycleNodeAssignTimes` to cluster meet domains using `KheSolnClusterAndLimitMeetDomains` (Section 10.3.3) before assigning times, and to uncluster them afterwards.

`ts_tighten_domains_off`

A Boolean option which, when true, instructs `KheCycleNodeAssignTimes` to not tighten resource domains (Section 11.9).

`ts_node_repair_off`

A Boolean option which, when true, instructs `KheCycleNodeAssignTimes` to not call `KheEjectionChainNodeRepairTimes` (Section 10.7.2). If it does call it, it calls it twice, before and after removing regularity-enhancing features.

`ts_node_repair_time_limit`

A string option, a soft time limit for each call on `KheEjectionChainNodeRepairTimes`. The format is that accepted by `KheTimeFromString` (Section 8.1): `secs`, or `mins:secs`, or `hrs:mins:secs`. The special value `-` (the default) means ‘set no limit’.

Other options influence it indirectly, via its calls to `KheNodeLayeredAssignTimes`.

# 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. 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.

## 11.5. Grouping by resource constraints

*Grouping by resource constraints* is KHE's term for a method of grouping tasks together, forcing the tasks in each group to be assigned the same resource, when all other ways of assigning resources to those tasks can be shown to have non-zero cost. `KheTaskTreeMake` also does this, but its groups are based on avoid split assignments constraints, whereas grouping by resource constraints makes groups based on resource constraints. The function is

```
bool KheGroupByResourceConstraints(KHE_SOLN soln,
    KHE_RESOURCE_TYPE rt, KHE_OPTIONS options, KHE_TASK_SET ts);
```

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, and for which non-assignment may have a cost (discussed later). If `rt` is `NULL`, it 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.

For each resource type, `KheGroupByResourceConstraints` finds whatever groups it can. 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 non-cycle tasks that are not already assigned to other non-cycle tasks, so it does not disturb existing groups. However it does take existing groups into account, and it will use tasks to which other tasks are assigned in its own groups.

Tasks which are initially assigned a resource (cycle task) participate in grouping. Such a task may have its assignment changed to some other task, but in that case the other task will be assigned the resource. In other words, if one task is assigned a resource initially, and it gets grouped, then its whole group will be assigned that resource afterwards. Two tasks initially assigned different resources will never be grouped together.

If `ts` is non-`NULL`, every task that `KheGroupByResourceConstraints` assigns is added to `ts`. This makes it easy to remove the groups when they are no longer wanted, by running through `ts` and unassigning each of its tasks. `KheTaskSetUnGroup` (Section 5.6) does this.

`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 11.5.4). Values 0 or 1 turn combinatorial grouping off. The default value is 3.

It also calls `KheFrameOption` (Section 5.10) to obtain the common frame, and retrieves the event timetable monitor from option `gs_event_timetable_monitor` (Section 8.3).

`KheGroupByResourceConstraints` groups tasks whenever it can show that not assigning the same resource to all of them must incur a cost. That does not mean that they 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 they usually are, so it makes sense to require them to be, and decide later whether to break up a few groups.

The following subsections describe how `KheGroupByResourceConstraints` works in detail. It has several parts, which are available separately, as we will see. For each resource type, it starts by building a tasker and adding the time groups of the common frame to it as overlap time groups (Section 11.5.1). Then, using this tasker, it performs combinatorial grouping by calling `KheCombGrouping` (Section 11.5.5), and profile grouping by calling `KheProfileGrouping` (Section ??), first with `non_strict` set to `false`, then again with `non_strict` set to `true`.

### 11.5.1. Taskers

A *tasker* is an object of type `KHE_TASKER` that facilitates grouping by resource constraints. We'll see how to create one shortly; but first, we introduce two other types that taskers use.

Taskers deal directly only with proper root tasks (tasks which are either unassigned, or assigned directly to a cycle task, that is, to a resource). Taskers consider two proper root tasks to be equivalent when they have equal domains and assigned resources (possibly `NULL`), and they cover the same set of times. (A task *covers* a time when it, or some task assigned directly or indirectly to it, is running at that time.) Equivalent tasks are interchangeable with respect to resource assignment: they may be assigned the same resources, and their effect on resource constraints is the same. Identifying equivalent tasks is vital in grouping; without it, virtually no group could be shown to be the only zero-cost option.

A *class* is an object of type `KHE_TASKER_CLASS`, representing an equivalence class of tasks (a set of equivalent tasks). Each task known to a tasker lies in exactly one class. The user cannot create these classes; they are created and kept up to date by the tasker.

The tasks of an equivalence class may be visited by

```
int KheTaskerClassTaskCount(KHE_TASKER_CLASS c);
KHE_TASK KheTaskerClassTask(KHE_TASKER_CLASS c, int i);
```

There must be at least one task, because if a class becomes empty, it is deleted by the tasker.

The three attributes that equivalent tasks share may be retrieved by

```
KHE_RESOURCE_GROUP KheTaskerClassDomain(KHE_TASKER_CLASS c);
KHE_RESOURCE KheTaskerClassAsstResource(KHE_TASKER_CLASS c);
KHE_TIME_SET KheTaskerClassTimeSet(KHE_TASKER_CLASS c);
```

These return the domain (from `KheTaskDomain`) that the tasks of `c` share, their assigned resource (from `KheTaskAsstResource`), and the set of times they each cover. The user must not modify the value returned by `KheTaskerClassTimeSet`. Function

```
void KheTaskerClassDebug(KHE_TASKER_CLASS c, int verbosity,
    int indent, FILE *fp);
```

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

The other type that taskers use represents one time. The type is `KHE_TASKER_TIME`. Again, the tasker creates objects of these types, and keeps them up to date. Function

```
KHE_TIME KheTaskerTimeTime(KHE_TASKER_TIME t);
```

returns the time that `t` represents.

The tasks of an equivalence class all run at the same times, and so for each time, either every task of an equivalence class is running at that time, or none of them are. Accordingly, to visit the tasks running at a particular time, we actually visit classes:

```
int KheTaskerTimeClassCount(KHE_TASKER_TIME t);
KHE_TASKER_CLASS KheTaskerTimeClass(KHE_TASKER_TIME t, int i);
```

Each equivalence class appears in one time object for each time that its tasks are running, giving a many-to-many relationship between time objects and class objects. Function

```
void KheTaskerTimeDebug(KHE_TASKER_TIME t, int verbosity,
    int indent, FILE *fp);
```

produces a debug print of `t` onto `fp` with the given verbosity and indent.

We turn now to taskers themselves. To create a tasker, call

```
KHE_TASKER KheTaskerMake(KHE_SOLN soln, KHE_RESOURCE_TYPE rt,
    KHE_TASK_SET task_set, HA_ARENA a);
```

The tasker's attributes may be accessed by

```
KHE_SOLN KheTaskerSoln(KHE_TASKER tr);
KHE_RESOURCE_TYPE KheTaskerResourceType(KHE_TASKER tr);
KHE_TASK_SET KheTaskerTaskSet(KHE_TASKER tr);
HA_ARENA KheTaskerArena(KHE_TASKER tr);
```

A tasker object remains in existence until its arena, `a`, is deleted or recycled.

`KheTaskerMake` gathers into the tasker object all proper root tasks (tasks which are either unassigned, or assigned directly to a cycle task representing a resource) of `soln` whose resource type is `rt`, for which non-assignment may have a cost (see below), and which lie in meets that have an assigned time. The meets' time assignments are assumed to be fixed for the lifetime of

the tasker; if they change, errors will occur. From here on, ‘task’ means one of these tasks, unless stated otherwise.

It seems wrong to group a task for which non-assignment has a cost with a task for which non-assignment has no cost. But what to do about this issue is a puzzle. Simply refusing to group such tasks would not address all the relevant issues, e.g. whether to include both types in profiles. At present, if the instance contains at least one assign resource constraint, then only tasks derived from event resources for which `KheEventResourceNeedsAssignment` (Section 3.6.3) returns `KHE_YES` are considered for grouping. If the instance contains no assign resource constraints, then only tasks derived from event resources for which `KheEventResourceNeedsAssignment` returns `KHE_MAYBE` are considered for grouping. This is basically a stopgap.

Tasks are grouped by calls to `KheTaskMove`, each of which assigns one follower task to a leader task. This removes the follower task from the set of tasks of interest to the tasker, and it usually enlarges the set of times covered by the leader task, placing it into a different equivalence class. The main purpose of the tasker object is to keep track of these changes.

If `task_set` is non-NULL, each follower task assigned during grouping is added to it. This makes it easy to remove the groups later, when they are no longer wanted, by running through `task_set` and unassigning each of its tasks. `KheTaskSetUnGroup` (Section 5.6) does this.

`KheTaskerMake` places its tasks into classes indexed by time. To visit each time, call

```
int KheTaskerTimeCount(KHE_TASKER tr);
KHE_TASKER_TIME KheTaskerTime(KHE_TASKER tr, int i);
```

Here `KheTaskerTimeTime(KheTaskerTime(tr, KheTimeIndex(t))) == t` for all times `t`. `KheTaskerTimeCount(tr)` returns the same value as `KheInstanceTimeCount(ins)`, where `ins` is `tr`’s solution’s instance. From each `KHE_TASKER_TIME` object one can access the classes running at that time, and the tasks of those classes, using functions introduced above.

Finally,

```
void KheTaskerDebug(KHE_TASKER tr, int verbosity, int indent, FILE *fp);
```

produces a debug print of `tr` onto `fp` with the given verbosity and indent.

### 11.5.2. Tasker support for grouping

Taskers keep their classes up to date as tasks are grouped. However, they can’t know by magic that tasks are being grouped. So it’s wrong to call platform operations like `KheTaskAssign` and `KheTaskMove` directly while using a tasker. `KheTaskAddTaskBound` is also out of bounds. Instead, proceed as follows.

A *grouping* is a set of classes used for grouping tasks. A group is made by taking any one task out of each class in the grouping, choosing one to be the leader task, assigning the others (called the followers) to it, and inserting the leader task into some other class appropriate to it, where it is available to participate in other groupings.

When a task is taken out of a class, the class may become empty, in which case the tasker deletes that class. When the follower tasks are assigned to the leader tasks, the set of times covered by it usually changes, and the tasker may need to create a new class object to hold it. So

class objects may be both created and destroyed by the tasker when tasks are grouped.

A tasker may handle any number of groupings over its lifetime, but at any moment there is at most one grouping. The operations for building this *current grouping* are:

```
void KheTaskerGroupingClear(KHE_TASKER tr);
bool KheTaskerGroupingAddClass(KHE_TASKER tr, KHE_TASKER_CLASS c);
bool KheTaskerGroupingDeleteClass(KHE_TASKER tr, KHE_TASKER_CLASS c);
int KheTaskerGroupingBuild(KHE_TASKER tr, int max_num, char *debug_str);
```

These call the platform operations, as well as keeping the tasker up to date.

`KheTaskerGroupingClear` starts off a grouping, clearing out any previous grouping.

`KheTaskerGroupingAddClass`, which may be called any number of times, adds `c` to the current grouping. If there is a problem with this, it returns `false` and changes nothing. These potential problems (there are two kinds) are explained below.

`KheTaskerGroupingDeleteClass` undoes a call to `KheTaskerGroupingAddClass` with the same `c` that returned `true`. Deleting `c` might not be possible, since it might leave the grouping with no viable leader class (for which see below). `KheTaskerGroupingDeleteClass` returns `false` in that case, and changes nothing. This cannot happen if classes are deleted in stack order (last in first out), because each deletion then returns the grouping to a viable previous state.

`KheTaskerGroupingBuild` ends the grouping. It makes some groups and returns the number it made. Each group is either made completely, or not at all. The number of groups made is the minimum of `max_num` and the `KheTaskerClassTaskCount` values for the classes. It then removes all classes from the grouping, like `KheTaskerGroupingClear` does, understanding that some may have already been destroyed by being emptied out by `KheTaskerGroupingBuild`.

It is acceptable to add just one class, in which case the ‘groups’ are just tasks from that class, no assignments are made, and nothing actually changes in the tasker’s data structure. If this is not wanted, then the caller should ensure that `KheTaskerGroupingClassCount` (see below) is at least 2 before calling `KheTaskerGroupingBuild`.

Parameter `debug_str` is used only by debugging code, to say why a group was made. For example, its value might be "combinatorial grouping" or "profile grouping".

At any time, the classes of the current grouping may be accessed by calling

```
int KheTaskerGroupingClassCount(KHE_TASKER tr);
KHE_TASKER_CLASS KheTaskerGroupingClass(KHE_TASKER tr, int i);
```

in the usual way. They will not usually be returned in the order they were added, however; in particular, the class that the tasker currently intends to use as the leader class has index 0.

We now describe the two problems that make `KheTaskerGroupingAddClass` return `false`. The first problem concerns leader tasks. Tasks are grouped by choosing one task as the leader and assigning the others to it. So one of the classes added by `KheTaskerGroupingAddClass` has to be chosen as the one that leader tasks will be taken from (the *leader class*). The tasker does this automatically in a way that usually works well. (It chooses any class whose tasks are already assigned a resource, or if there are none of those, a class whose domain has minimal cardinality, and checks that the first task of each of the other classes can be assigned to the first task of that class without changing any existing resource assignment.) But in rare cases, the domains of two

classes may be such that neither is a subset of the other, or two classes may be initially assigned different resources. `KheTaskerGroupingAddClass` returns `false` in such cases.

The second problem concerns the times covered by the classes. It would not do to group together two tasks which cover the same time, because then, when a resource is assigned to the grouped task, the resource would have a clash. More generally, if a resource cannot be assigned to two tasks on the same day (for example), then it would not do to group two tasks which cover two times from the same day. To help with this, the tasker has functions

```
void KheTaskerAddOverlapFrame(KHE_TASKER tr, KHE_FRAME frame);
void KheTaskerDeleteOverlapFrame(KHE_TASKER tr);
```

`KheTaskerAddOverlapFrame` informs the tasker that a resource should not be assigned two tasks that cover the same time group of `frame`. If this condition would be violated by some call to `KheTaskerGroupingAddClass`, then that call returns `false` and adds nothing. `KheTaskerDeleteOverlapFrame`, which is never needed in practice, removes this requirement.

If overlaps are prevented in this way, the same class cannot be added to a grouping twice. So there is no need to prohibit that explicitly.

When `KheTaskerGroupingAddClass` returns `false`, the caller has two options. One is to abandon this grouping altogether, which is done by not calling `KheTaskerGroupingBuild`. The next call to `KheTaskerGroupingClear` will clear everything out for a fresh start. The other option is to continue with the grouping, finding other classes to add. This is done by making zero or more other calls to `KheTaskerGroupingAddClass`, followed by `KheTaskerGroupingBuild`.

After one grouping is completed, the user may start another. The tasker will have been updated by the previous `KheTaskerGroupingBuild` to no longer contain the ungrouped tasks but instead to contain the grouped ones. They can become elements of new groups.

`KHE_TASKER_CLASS` objects may be created by `KheTaskerGroupingBuild`, to hold the newly created groups, and also destroyed, because empty classes are deleted. So variables of type `KHE_TASKER_CLASS` may become undefined when `KheTaskerGroupingBuild` is called.

Although `KheTaskerGroupingAdd` can be used to check whether a class can be added, it may be convenient to check for overlap in advance. For this there are functions

```
bool KheTaskerTimeOverlapsGrouping(KHE_TASKER_TIME t);
bool KheTaskerClassOverlapsGrouping(KHE_TASKER_CLASS c);
```

`KheTaskerTimeOverlapsGrouping` returns `true` if `t` lies in an overlap time group which is currently covered by a class of the current grouping. `KheTaskerClassOverlapsGrouping` returns `true` if any of the times covered by `c` is already so covered.

### 11.5.3. Tasker support for profile grouping

Taskers also have functions which support profile grouping (Section 11.5.6). To set and retrieve the *profile maximum length*, the calls are

```
void KheTaskerSetProfileMaxLen(KHE_TASKER tr, int profile_max_len);
int KheTaskerProfileMaxLen(KHE_TASKER tr);
```

The profile maximum length can only be set when there are no profile time groups.

To visit the sequence of *profile time groups* maintained by the tasker, the calls are

```
int KheTaskerProfileTimeGroupCount(KHE_TASKER tr);
KHE_PROFILE_TIME_GROUP KheTaskerProfileTimeGroup(KHE_TASKER tr, int i);
```

To make one profile time group and add it to the end of the tasker's sequence, and to delete a profile time group, the calls are

```
KHE_PROFILE_TIME_GROUP KheProfileTimeGroupMake(KHE_TASKER tr,
        KHE_TIME_GROUP tg);
void KheProfileTimeGroupDelete(KHE_PROFILE_TIME_GROUP ptg);
```

The last profile time group is moved to the position of the deleted one, which only makes sense in practice when all the profile time groups are being deleted. So a better function to call is

```
void KheTaskerDeleteProfileTimeGroups(KHE_TASKER tr);
```

which deletes all of *tr*'s profile time groups. They go into a free list in the tasker.

#### Functions

```
KHE_TASKER KheProfileTimeGroupTasker(KHE_PROFILE_TIME_GROUP ptg);
KHE_TIME_GROUP KheProfileTimeGroupTimeGroup(KHE_PROFILE_TIME_GROUP ptg);
```

retrieve a profile time group's tasker and time group.

A profile time group's *cover* is the number of *cover tasks*: tasks that cover the time group, ignoring tasks that cover more than `profile_max_len` profile time groups. This is returned by

```
int KheProfileTimeGroupCover(KHE_PROFILE_TIME_GROUP ptg);
```

The profile time group also keeps track of the *domain cover*: the number of cover tasks with a given domain. Two domains are considered to be equal if `KheResourceGroupEqual` says that they are. To visit the (distinct) domains of a profile time group, in increasing domain size order, the calls are

```
int KheProfileTimeGroupDomainCount(KHE_PROFILE_TIME_GROUP ptg);
KHE_RESOURCE_GROUP KheProfileTimeGroupDomain(KHE_PROFILE_TIME_GROUP ptg,
        int i, int *cover);
```

`KheProfileTimeGroupDomain` returns the domain cover as well as the domain itself. The sum of the domain covers is the cover. There is also

```
bool KheProfileTimeGroupContainsDomain(KHE_PROFILE_TIME_GROUP ptg,
        KHE_RESOURCE_GROUP domain, int *cover);
```

which searches *ptg*'s list of domains for *domain*, returning true and setting *\*cover* to the domain cover if it is found.

`KheProfileTimeGroupDomain` and `KheProfileTimeGroupContainsDomain` may return 0 for *\*cover*, when tasks with a given domain enter the profile and later leave it.

Profile grouping algorithms will group tasks while these functions are being called. The

sequence of profile time groups is unaffected by grouping, but covers and domain covers will change if the grouped tasks cover more than `profile_max_len` profile time groups. The domains of a profile time group may also change during grouping, when tasks with unequal domains are grouped. Altogether it is safest to discontinue a partially completed traversal of the domains of a profile time group when a grouping occurs.

There are also a few functions on tasker classes that relate to profile time groups. First,

```
bool KheTaskerClassCoversProfileTimeGroup(KHE_TASKER_CLASS c,
    KHE_PROFILE_TIME_GROUP ptg);
```

returns true if `c` covers `ptg`. Each class keeps track of the times from profile time groups that it covers. Functions

```
int KheTaskerClassProfileTimeCount(KHE_TASKER_CLASS c);
KHE_TASKER_TIME KheTaskerClassProfileTime(KHE_TASKER_CLASS c, int i);
```

visit these times in an unspecified order.

Function

```
void KheTaskerProfileDebug(KHE_TASKER tr, int verbosity, int indent,
    FILE *fp);
```

prints the profile groups of `tr` onto `fp`, with the classes that cover not more than `profile_max_len` of them.

#### 11.5.4. Combinatorial grouping

Suppose that there are two kinds of shifts (tasks), day and night; that a resource must be busy on both days of the weekend or neither; and that a resource cannot work a day shift on the day after a night shift. Then resources 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 those 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. A time sweep task assignment algorithm (say) cannot look ahead and see such cases coming.

*Combinatorial grouping* implements these ideas. It searches through a space whose elements are sets of classes. For each set of classes  $S$  in the search space, it calculates a cost  $c(S)$ , defined below, and selects a set  $S'$  such that  $c(S')$  is zero, or minimal. It then makes one group by selecting one task from each class and grouping those tasks, and then repeating that until as

many tasks as possible or desired have been grouped.

As formulated here, one application of combinatorial grouping groups one set of classes  $S'$ . In the example above, grouping the Saturday and Sunday night shifts would be one application, then grouping the Saturday and Sunday day shifts would be another.

Combinatorial grouping is carried out by a *combinatorial grouping solver*, made like this:

```
KHE_COMB_SOLVER KheCombSolverMake(KHE_TASKER tr, KHE_FRAME days_frame);
```

It deals with `tr`'s tasks, using memory from `tr`'s arena. Any groups it makes are made using `tr`'s grouping operations, and so are reflected in `tr`'s classes, and in its task set. Parameter `days_frame` would always be the common frame. It is used when selecting a suitable resource to tentatively assign to a group of tasks, to find out what times the resource should be free.

#### Functions

```
KHE_TASKER KheCombSolverTasker(KHE_COMB_SOLVER cs);
KHE_FRAME KheCombSolverFrame(KHE_COMB_SOLVER cs);
```

return `cs`'s tasker and frame.

A `KHE_COMB_SOLVER` object can solve any number of combinatorial grouping problems, one after another. The user loads the solver with one problem's *requirements* (these determine the search space  $S$ ), then requests a solve, then loads another problem and solves, and so on.

It is usually best to start the process of loading requirements into the solver by calling

```
void KheCombSolverClearRequirements(KHE_COMB_SOLVER cs);
```

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 KheCombSolverAddTimeGroupRequirement(KHE_COMB_SOLVER cs,
    KHE_TIME_GROUP tg, KHE_COMB_SOLVER_COVER_TYPE cover);
void KheCombSolverDeleteTimeGroupRequirement(KHE_COMB_SOLVER cs,
    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_SOLVER_COVER_YES,
    KHE_COMB_SOLVER_COVER_NO,
    KHE_COMB_SOLVER_COVER_PREV,
    KHE_COMB_SOLVER_COVER_FREE,
} KHE_COMB_SOLVER_COVER_TYPE;
```

We'll explain this in detail later. `KheCombSolverDeleteTimeGroup` removes the effect of a previous call to `KheCombSolverAddTimeGroup` with the same time group. There must have been such a call, otherwise `KheCombSolverDeleteTimeGroup` aborts.

Any number of requirements that the groups should cover a given class may be added:

```
void KheCombSolverAddClassRequirement(KHE_COMB_SOLVER cs,
    KHE_TASKER_CLASS c, KHE_COMB_SOLVER_COVER_TYPE cover);
void KheCombSolverDeleteClassRequirement(KHE_COMB_SOLVER cs,
    KHE_TASKER_CLASS c);
```

These work in the same way as for time groups, except that care is needed because `c` may be rendered undefined by a solve, if it makes groups which empty `c` out. The safest option after a solve whose requirements include a class is to call `KheCombSolverClearRequirements`.

Three other requirements of quite different kinds may be added:

```
void KheCombSolverAddProfileGroupRequirement(KHE_COMB_SOLVER cs,
    KHE_PROFILE_TIME_GROUP ptg, KHE_RESOURCE_GROUP domain);
void KheCombSolverDeleteProfileGroupRequirement(KHE_COMB_SOLVER cs,
    KHE_PROFILE_TIME_GROUP ptg);
```

and

```
void KheCombSolverAddProfileMaxLenRequirement(KHE_COMB_SOLVER cs);
void KheCombSolverDeleteProfileMaxLenRequirement(KHE_COMB_SOLVER cs);
```

and

```
void KheCombSolverAddNoSinglesRequirement(KHE_COMB_SOLVER cs);
void KheCombSolverDeleteNoSinglesRequirement(KHE_COMB_SOLVER cs);
```

Again, we'll explain the precise effect later. These last three requirements can only be added once: a second call replaces the first, it does not add to it.

There is no need to reload requirements between solves. The requirements stay in effect until they are either deleted individually or cleared out by `KheCombSolverClearRequirements`. The only caveat concerns classes that become undefined during grouping, as discussed above.

The search space of combinatorial solving is defined by all these requirements. First, we need some definitions. A task *covers* a time if it, or a task assigned to it directly or indirectly, runs at that time. A task covers a time group if it covers any of the time group's times. A class covers a time or time group if its tasks do. A class covers a class if it is that class. A set of classes covers a time, time group, or class if any of its classes covers that time, time group, or class.

Now a set  $S$  of classes lies in the search space for a run of combinatorial grouping if:

1. Each class in  $S$  covers at least one of the time groups and classes passed to the solver by the calls to `KheCombSolverAddTimeGroup` and `KheCombSolverAddClass`.

2. For each time group *tg* or class *c* passed to the solver by `KheCombSolverAddTimeGroup` or `KheCombSolverAddClass`, if the accompanying `cover` is `KHE_COMB_SOLVER_COVER_YES`, then *S* covers *tg* or *c*; or if `cover` is `KHE_COMB_SOLVER_COVER_NO`, then *S* does not cover *tg* or *c*; or if `cover` is `KHE_COMB_SOLVER_COVER_PREV`, then *S* covers *tg* or *c* if and only if it covers the time group or class immediately preceding *tg* or *c*; or if `cover` is `KHE_COMB_SOLVER_COVER_FREE`, then *S* is free to cover *tg* or *c*, or not.

If the first time group or class has `cover` `KHE_COMB_SOLVER_COVER_PREV`, this is treated like `KHE_COMB_SOLVER_COVER_FREE`.

Time groups and classes not mentioned may be covered, or not. The difference between this and passing a time group or class with `cover` `KHE_GROUP_SOLVER_COVER_FREE` is that the classes that cover a free time group or class are included in the search space.

3. The classes of *S* may be added to the tasker to form a grouping. There are rare cases where adding the classes in one order will succeed, while adding them in another order will fail. In those cases, whether *S* is included in the search space or not will depend on the (unspecified) order in which the solver chooses to add *S*'s classes to the tasker.
4. If `KheCombSolverAddProfileRequirement(cs, ptg, domain)` is in effect, then *S* contains at least one class that covers *ptg*'s time group, and if `domain != NULL`, that class has that domain.
5. If `KheCombSolverAddProfileMaxLenRequirement(cs)` is in effect, then *S* contains only classes that cover at most `profile_max_len` times from profile time groups.
6. If `KheCombSolverAddNoSinglesRequirement(cs)` is in effect, then *S* contains at least two classes. Otherwise *S* contains at least one class.

That fixes the search space. We now define the cost  $c(S)$  of each set of classes *S* in that space.

The first step is to identify a suitable resource *r*. Take the first class of the tasker grouping made from *S*; this is the class that leader tasks will come from. If it already has an assigned resource (as returned by `KheTaskerClassAsstResource`), use that resource for *r*. Otherwise search the class's domain (as returned by `KheTaskerClassDomain`) for a resource which is free at all of the time groups of the current frame which overlap with the time groups added by calls to `KheCombSolverAddTimeGroupRequirement`. If no such resource can be found, ignore *S*.

The second step is to assign *r* to one task from each class of *S*, except in classes where *r* is already assigned to a task. This is done without informing the tasker, but after the cost is determined these assignments are undone, so the tasker's integrity is not compromised in the end. The cost  $c(S)$  of a set of classes *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 covered by the time groups added by calls to `KheCombSolverAddTimeGroupRequirement`. This second condition is included because we don't want *r*'s global workload, for example, to influence the outcome.

After all the requirements are added, an actual solve is carried out by calling

```
int KheCombSolverSolve(KHE_COMB_SOLVER cs, int max_num,
    KHE_COMB_SOLVER_COST_TYPE ct, char *debug_str);
```

`KheCombSolverSolve` searches the space of all sets of classes  $S$  that satisfy the six conditions, and selects one set  $S'$  of minimal cost  $c(S')$ . Using  $S'$ , it makes as many groups as it can, up to `max_num`, and returns the number it actually made, between 0 and `max_num`. If  $S'$  contains a single class, no groups are made and the value returned is 0.

Parameter `ct` has type

```
typedef enum {
    KHE_COMB_SOLVER_COST_MIN,
    KHE_COMB_SOLVER_COST_ZERO,
    KHE_COMB_SOLVER_COST_SOLE_ZERO
} KHE_COMB_SOLVER_COST_TYPE;
```

If `ct` is `KHE_COMB_SOLVER_COST_MIN`, then  $c(S')$  must be minimal among all  $c(S)$ . If `ct` is `KHE_COMB_SOLVER_COST_ZERO` or `KHE_COMB_SOLVER_COST_SOLE_ZERO`, then  $c(S')$  must also be 0, and in the second case there must be no other  $S$  in the search space such that  $c(S)$  is 0. If these conditions are not met, no groups are made.

Parameter `debug_str` is passed on to `KheTaskerGroupingBuild`. It might be "combinatorial grouping", for example.

An awkward question raised by combinatorial grouping is what to do about *singles*: classes whose tasks already satisfy the requirements, without any grouping. The answer seems to vary depending on why combinatorial grouping is being called, so the combinatorial solver does not have a single way of dealing with singles. Instead it offers three features that help with them.

First, as we have seen, if the set of classes  $S'$  with minimum or zero cost contains only one class, `KheCombSolverSolve` accepts that it is the best but makes no groups from it, returning 0 for the number of groups made.

Second, as we have also seen, `KheCombSolverAddNoSinglesRequirement` allows the user to declare that a set  $S$  whose classes consist of a single class which satisfies all the requirements (a single) should be excluded from the search space. But adding this requirement is not a magical solution to the problem of singles. For one thing, when we need a unique zero-cost set of classes, we may well want to include singles in the search space, to show that grouping is better than doing nothing. For another, there may still be an  $S$  containing one single and another class which covers a time group or class with cover type `KHE_COMB_SOLVER_COVER_FREE`.

Third, after setting up a problem ready to call `KheCombSolverSolve`, one can call

```
int KheCombSolverSingleTasks(KHE_COMB_SOLVER cs);
```

This searches the same space as `KheCombSolverSolve` does, but it does no grouping. Instead, it returns the total number of tasks in sets of classes  $S$  in that space such that  $|S| = 1$ . It returns 0 if `KheCombSolverAddNoSinglesRequirement` is in effect when it is called, quite correctly.

Finally,

```
void KheCombSolverDebug(KHE_COMB_SOLVER cs, int verbosity,
    int indent, FILE *fp);
```

produces the usual debug print of `cs` onto `fp` with the given verbosity and indent.

### 11.5.5. Applying combinatorial grouping

This section describes one way in which the general idea of combinatorial grouping, as just presented, may be applied in practice. This way is implemented by function

```
int KheCombGrouping(KHE_COMB_SOLVER cs, KHE_OPTIONS options);
```

`KheCombGrouping` does what this section describes, and returns the number of groups it made. Before it is called, the common frame should be loaded into `cs`'s tasker as overlap time groups.

Let  $m$  be the value of the `rs_group_by_rc_max_days` option of `options`. Iterate over all pairs  $(f, c)$ , 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  $c$  is a class that covers  $f$ 's first or last time group.

For each pair, set up and run combinatorial grouping with one 'yes' class, namely  $c$ , and one 'free' time group for each of the  $k$  time groups of  $f$ . Set `max_num` to `INT_MAX`, and set `ct` to `KHE_COMB_SOLVER_COST_SOLE_ZERO`. If there is a unique zero-cost way to group a task of  $c$  with tasks on the following  $k - 1$  days, this call will find it and carry out as many groupings as it can.

If  $f$  has  $k$  time groups, each with  $n$  classes, 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  $(f, c)$  pair produces some grouping, then `KheCombGrouping` returns 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 elimination*. This is a refinement that `KheCombGrouping` 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  $T_i$ . `KheCombGrouping` searches for constraints  $C$  that have this effect, and informs its combinatorial grouping solver about what it found by changing the cover types of some time groups from 'free' to 'prev'. 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 elimination 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  $R_j F_j$ , 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  $c_j$ .

### 11.5.6. 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  $T_1, \dots, T_k$  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.

A *maximal task* is a task which covers at least  $C_{max}$  adjacent time groups from  $C$ . Maximal 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.

Let  $n_i$  be the number of tasks that cover  $T_i$ , not including maximal tasks. The  $n_i$  together make up the *profile* of  $C$ . The tasker operations from Section 11.5.1 which support profile grouping make it easy to find the 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  $T_i, \dots, T_{i+C_{min}-1}$  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 maximal ones) that satisfy these conditions already (as returned by `KheCombSolverSingles`). The new groups must group at least 2 tasks each. Some of the time groups may not exist; in that case, omit the non-existent ones but still do the grouping, provided there are at least 2 'yes' time groups. The case for sequences ending at  $j$  is symmetrical.

If  $C$  has no history, we may set  $n_0$  and  $n_{k+1}$  to 0, allowing groups to begin at  $T_1$  and end at  $T_k$ . If  $C$  has history, we do not know how many tasks are running outside  $C$ , so we set  $n_0$  and  $n_{k+1}$  to infinity, preventing groups from beginning at  $T_1$  and ending 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 beginning at  $T_2$ , then 4 groups of length 3 ending at  $T_4$ , incorporating the length 2 groups.

We turn now to four issues of detail.

**History.** If history is present, the first step is to handle it. For each resource  $r_i$  with a history value  $x_i$  such that  $x_i < C_{min}$ , use combinatorial grouping with one 'yes' time group for each of the first  $C_{min} - x_i$  time groups of  $C$  (when these all exist), build one group, and assign  $r_i$  to it. (This idea is not yet implemented; none of the instances available at the time of writing need it.)

**Singles.** We need to consider how singles affect profile grouping. Singles of length  $C_{max}$  or more are ignored, but there may be singles of length  $C_{min}$  when  $C_{min} < C_{max}$ .

The  $n_i - n_{i-1}$  groups that must start at  $T_i$  include singles. Singles are already present, which is similar to saying that they must be made first. So before calling `KheCombSolverSolve` we call

CombSolverSingleTasks to determine  $c_i$ , the number of singles that satisfy the requirements, and then we pass  $n_i - n_{i-1} - c_i$  to KheCombSolverSolve, not  $n_i - n_{i-1}$ , and exclude singles from its search space.

**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 the two groups starting on Monday nights 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 class running at time  $T_i$ , find  $n_{gi}$ , the number of tasks (not including maximal 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, insisting (by calling KheCombSolverAddProfileRequirement) that  $T_i$  be covered by a class whose domain is  $g$ , passing  $m = \min(M, n_{gi} - n_{g(i-1)})$ , then subtract from  $M$  the number of groups actually made. Stop when  $M = 0$  or the list of domains is exhausted.

**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.

One reasonable way of dealing with this problem is the following. First, do not insist on unique zero-cost groupings; instead, accept any zero-cost grouping. This ensures that a reasonable amount of profile grouping will happen. Second, to reduce the chance of making poor choices of zero-cost groupings, limit profile grouping to two cases.

The first case is when each time group  $T_i$  contains a single time, as at the start of this section, where each  $T_i$  contained the time of a night shift. Although we do not insist on unique zero-cost groupings, we are likely to get them in this case, so we call this *strict profile grouping*.

The second case 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. We call this *non-strict profile grouping*.

When  $C_{min} = C_{max}$ , all singles are off-profile. This is easy to see: by definition, a single covers  $C_{min}$  time groups, so it covers  $C_{max}$  time groups, but `profile_max_len` is  $C_{max} - 1$ .

These ideas are implemented by function

```
int KheProfileGrouping(KHE_COMB_SOLVER cs, bool non_strict);
```

It carries out some profile grouping, as follows, and returns the number of groups it makes.

Find all limit active intervals constraints  $C$  whose time groups are all positive and which apply to all resources. Notionally merge pairs of these constraints that share the same time

groups when one has a minimum limit and the other has a maximum limit. Let  $C$  be one of these (possibly merged) constraints such that  $C_{min} \geq 2$ . Furthermore, if `non_strict` is false, then  $C$ 's time groups must all be singletons, while if `non_strict` is true, then  $C_{min} = C_{max}$  must hold.

A constraint may qualify for both strict and non-strict processing. This is true, for example, of a constraint that imposes equal lower and upper limits on the number of consecutive night shifts. Such a constraint will be selected in both the strict and non-strict cases, which is fine.

For each of these constraints, proceed as follows. Set the profile time groups in the tasker to  $T_1, \dots, T_k$ , the time groups of  $C$ , and set the `profile_max_len` attribute to  $C_{max} - 1$ . The tasker will then report the values  $n_i$  needed for  $C$ .

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  $C_{min}$  time groups starting at  $T_i$  (or ending at  $T_j$ ). This involves loading  $T_i, \dots, T_{i+C_{min}-1}$  as 'yes' time groups, and  $T_{i-1}$  and  $T_{i+C_{max}}$  as 'no' time groups, as explained above.

The profile is traversed repeatedly until no points which allow grouping can be found. In the strict grouping case, it is then time to stop, but in the non-strict case it is better to keep grouping, 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  $n_i > n_{i-1}$  cases just before the latest group and  $n_j < n_{j+1}$  cases just after it. Repeat until there is no  $T_i$  where  $n_i > 0$  and combinatorial grouping can build a group.

## 11.6. Grouping by resource

*Grouping by resource* is another kind of task grouping, obtained by calling

```
bool KheTaskingGroupByResource(KHE_TASKING tasking,
                               KHE_OPTIONS options, KHE_TASK_SET ts);
```

Like grouping by resource constraints, 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 `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 (by means of any repair algorithm whatsoever) to assign them to different resources.

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 more elaborate form of grouping, one 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 operations, based on *task finder* objects, which find sets of tasks running on consecutive days, and intended to be moved from one resource to another. It is used by ejection chains when solving nurse rostering problems; it could be used elsewhere.

Task finding treats non-assignment like the assignment of a special resource (represented by `NULL`); it is included in the term 'assignment' unless explicitly excluded. This is very handy: it means that task finding is equally at home finding assigned and unassigned tasks.

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.

The result of a task finding operation is a set of tasks, often represented by a task set object (Section 5.6). The set of tasks satisfies these five *basic conditions*:

1. The set is non-empty. Moving an empty set would change nothing.
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. Every task is unpreassigned. A preassigned task would not move.
4. 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.
5. The set of days that the tasks are running is consecutive. In other words, between the first day and the last there are no *gaps*: days when none of the tasks is running.

A set of tasks that satisfies these conditions will be called a *task run*.

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.

A task  $t$  is *optional* if `KheTaskNeedsAssignment(t)` (Section 4.6.1) returns `false`, meaning that assignment of a resource to  $t$  is optional, because none of its event resource constraints requires it (there are no assign resource constraints, and any limit resources constraints are currently above their minimum limits). Task finding excludes optional tasks from the runs of unassigned tasks that it finds, because assigning resources to them is not a high priority.

A resource is *effectively free* during some set of days if it is `NULL`, or it is not `NULL` and the only tasks it is assigned to on those days are optional tasks. The point of this is that it is always

safe to move some tasks to a resource on days when it is effectively free: if it is NULL, they are simply unassigned, and if it is non-NULL, any optional tasks running on those days can be unassigned, at no cost, before the move is made. Task finding utilizes the effectively free concept and unassigns optional tasks in this way as required.

### 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 8.3) 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.

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:

```
void KheTaskFinderTaskInterval(KHE_TASK_FINDER tf,
    KHE_TASK task, int *first_index, int *last_index);
void KheTaskFinderTaskSetInterval(KHE_TASK_FINDER tf,
    KHE_TASK_SET ts, int *first_index, int *last_index);
```

These set *\*first\_index* and *\*last\_index* to the indexes of the first and last day that task or *ts* is running. If *ts* is empty, *\*first\_index* > *\*last\_index*.

### 11.8.2. Widened task sets

The task finder offers a *widened task set* type, representing a set of tasks assigned a common resource *from\_r*, and divided into three parts: the *core*, a task run passed to the widened task set initially; the *left wing*, a task run lying just before the core in time; and the *right wing*, a task run lying just after the core in time. Widened task sets support moving and swapping the core tasks, plus a variable number of wing tasks, from *from\_r* to another resource.

To create a widened task set with a given core, call

```
bool KheWidenedTaskSetMake(KHE_TASK_FINDER tf, KHE_RESOURCE from_r,
    KHE_TASK_SET from_r_ts, int max_left_wing_count,
    int max_right_wing_count, KHE_WIDENED_TASK_SET *wts);
```

The tasks of `from_r_ts` must be assigned `from_r`, which may be `NULL`, meaning unassigned as usual. When `from_r_ts` satisfies the basic conditions given above, `true` is returned and `*wts` is set to a widened task set with a copy of `from_r_ts` as its core (`from_r_ts` itself is not stored, and the user is free to change it, or delete it, at any time), plus left and right wings containing `max_left_wing_count` and `max_right_wing_count` tasks compatible with the core, or fewer if `KheWidenedTaskSetMake` cannot find suitable tasks.

When `from_r` is not `NULL`, a task is compatible with the core if it is assigned `from_r`. When `from_r` is `NULL`, a task is compatible with the core if it is not optional, it is unassigned, and its domain is similar to those of the tasks of the core, in a sense that we will not define. (When we move the core to some resource, we want the wings to be able to move to that resource too.)

Nothing prevents the user from creating a widened task set with `max_left_wing_count` and `max_right_wing_count` set to 0. Each wing is represented by one array inside the widened task set object. Empty arrays generate no memory allocation calls, so basically all that is wasted is the time spent on rediscovering, once per function call, that the wing is empty.

When a widened task set is no longer needed, it should be deleted, by calling

```
void KheWidenedTaskSetDelete(KHE_WIDENED_TASK_SET wts);
```

This recycles `wts` through a free list in its task finder.

There are also functions which search for widened task sets:

```
bool KheFindMovableWidenedTaskSetRight(KHE_TASK_FINDER tf,
    KHE_RESOURCE from_r, KHE_RESOURCE to_r, int days_first_index,
    KHE_WIDENED_TASK_SET *res_wts);
bool KheFindMovableWidenedTaskSetLeft(KHE_TASK_FINDER tf,
    KHE_RESOURCE from_r, KHE_RESOURCE to_r, int days_last_index,
    KHE_WIDENED_TASK_SET *res_wts);
```

They search right starting at `days_first_index`, and left starting at `days_last_index`, for the first task run whose tasks are assigned `from_r` (or are unassigned with the type of `to_r` when `from_r` is `NULL`) and are moveable to `to_r` as defined by `KheWidenedTaskSetMoveCheck` just below. If such a run is found, they return it as a widened task set with empty wings. When `from_r` is `NULL`, the first task of the run may be an arbitrary unassigned task, but subsequent tasks must be compatible with it, as defined above. The task run is maximal subject to these conditions, given that days before `days_first_index` or after `days_last_index` are out of bounds. The search ends at the first or last day; it does not wrap around.

Move and swap operations are offered. To find out whether a move is possible, call

```
bool KheWidenedTaskSetMoveCheck(KHE_WIDENED_TASK_SET wts,
    KHE_RESOURCE to_r, bool force, int *max_left_count,
    int *max_right_count);
```

If this returns `true`, the core may be moved from `from_r` to `to_r`, along with any num-

ber of initial left and right wing tasks up to `*max_left_count` and `*max_right_count`. It calls `KheTaskMoveCheck` to verify that the tasks will move, except that if `to_r` is `NULL`, this is an unassignment and no checks of tasks initially assigned to `to_r` are needed. If `wts` came from `KheFindMovableWidenedTaskSetRight` or `KheFindMovableWidenedTaskSetLeft`, then there is no need to call `KheWidenedTaskSetMoveCheck`, since the result must be true, with `*max_left_count` and `*max_right_count` equal to 0.

If `force` is false, `KheWidenedTaskSetMoveCheck` also requires `to_r` to be effectively free during the bounding interval of the tasks it moves. If `force` is true, this is not required: non-optional tasks may be unassigned by the move.

To actually carry out a move, call

```
bool KheWidenedTaskSetMove(KHE_WIDENED_TASK_SET wts,
    KHE_RESOURCE to_r, int left_count, int right_count,
    int *from_r_durn_change, int *to_r_durn_change);
```

This moves `wts`'s core tasks, plus its first `left_count` and `right_count` left and right wing tasks, to `to_r`, unassigning `to_r`'s tasks as required. It does not check anything again, it just does the moves. If `KheWidenedTaskSetMoveCheck` has returned true, then this must succeed for any `left_count` and `right_count` such that  $0 \leq \text{left\_count} \leq *max\_left\_count$  and  $0 \leq \text{right\_count} \leq *max\_right\_count$ . It can be undone using marks and paths.

If the move succeeds, `*from_r_durn_change` and `*to_r_durn_change` are set to the change in total duration of the tasks assigned from `from_r` and to `to_r`. Tasks are neither created nor destroyed, so `from_r_durn_change` and `to_r_durn_change` will be equal in absolute value and opposite in sign—that is, unless some of `to_r`'s tasks were unassigned, since that causes the total duration of the tasks assigned from `from_r` and to `to_r` to decrease.

Another move operation is offered:

```
bool KheWidenedTaskSetMovePartial(KHE_WIDENED_TASK_SET wts,
    KHE_RESOURCE to_r, int first_index, int last_index);
```

This is like `KheWidenedTaskSetMove` except that it moves only some tasks: the core tasks with index numbers `first_index` to `last_index` inclusive, and no wing tasks. The implementation is deficient in two respects: duration changes are not calculated, and all of `to_r`'s optional core tasks are unassigned, not just those running at the times of the part of the core that is moved.

These functions may provide suitable values for `first_index` and `last_index`:

```
bool KheWidenedTaskSetFindInitial(KHE_WIDENED_TASK_SET wts,
    int wanted_durn, int *first_index, int *last_index);
bool KheWidenedTaskSetFindFinal(KHE_WIDENED_TASK_SET wts,
    int wanted_durn, int *first_index, int *last_index);
```

`KheWidenedTaskSetFindInitial` searches for an initial sequence of `wts`'s core tasks whose total duration is `wanted_durn`. It sets `*first_index` and `*last_index` to the index of the first and last task in this sequence (`*first_index` is always 0), and it returns true when the duration of the sequence is equal to `wanted_durn`. `KheWidenedTaskSetFindFinal` is the same except that it searches for a final sequence (`*last_index` is always the index of the last task).

To check that a swap is possible, call

```
bool KheWidenedTaskSetSwapCheck(KHE_WIDENED_TASK_SET wts,
    KHE_RESOURCE to_r, bool exact, KHE_TIME_GROUP blocking_tg,
    KHE_MONITOR blocking_m, int *max_left_count, int *max_right_count);
```

This checks whether a swap is possible between the core and `to_r`'s tasks running on the same days as the core. If so it returns `true` and sets `*max_left_count` and `*max_right_count` to the number of initial positions in the wings where the swap is also possible.

In the core, the following checks are made, and if any of them fail, `false` is returned. First, `from_r`'s tasks must be movable to `to_r`, and `to_r`'s tasks must be movable to `from_r`. Then, if `exact` is `true`, `to_r`'s tasks must be running on exactly the same days as `from_r`'s. Furthermore, if `blocking_tg` `!=` `NULL`, none of `to_r`'s tasks may be running during `blocking_tg`, and if `blocking_m` `!=` `NULL`, none of `to_r`'s tasks may be monitored by `blocking_m`. And finally, if `to_r` has at least one task, the first of `from_r`'s tasks must not be equivalent to the first of `to_r`'s tasks. The reasoning here is that if these tasks are equivalent, the swap is about swapping equivalent tasks, which would achieve nothing.

In each element of each wing, the following checks are made, and the first element at which they fail determines `*max_left_count` and `*max_right_count`. First, `from_r`'s tasks must be movable to `to_r`, and `to_r`'s tasks must be movable to `from_r`. Then, if `exact` is `true`, `to_r`'s tasks must be running on exactly the same days as `from_r`'s.

To actually carry out a swap, call

```
bool KheWidenedTaskSetSwap(KHE_WIDENED_TASK_SET wts,
    KHE_RESOURCE to_r, int left_count, int right_count,
    int *from_r_durn_change, int *to_r_durn_change);
```

This moves the core tasks plus the first `left_count` and `right_count` left and right wing tasks to `to_r`, like moving does, but it also moves `to_r`'s tasks and running on the same days from `to_r` to `from_r`. If `to_r` is `NULL` there will be no such tasks. It does not check anything again, it just does the swap. If successful it sets `*from_r_durn_change` and `*to_r_durn_change` in the same way as `move` does. If `KheWidenedTaskSetSwapCheck` has returned `true`, then this must succeed for any `left_count` and `right_count` such that  $0 \leq \text{left\_count} \leq \text{*max\_left\_count}$  and  $0 \leq \text{right\_count} \leq \text{*max\_right\_count}$ .

The return values of `KheWidenedTaskSetMoveCheck` and `KheWidenedTaskSetSwapCheck` may differ, and when they are both `true`, `*max_left_count` and `*max_right_count` may differ. This is because non-optional tasks assigned to `to_r` may prevent the move, but not the swap unless `KheTaskMoveCheck` reports that they cannot move to `from_r`.

There are several aspects of all this that the author finds quite confusing. One is that if `to_r` is effectively free during the core days, both the move and the swap may succeed, and are the same except for how they treat optional tasks (the move unassigns them, the swap moves them to `from_r`). Another is that when `from_r` `==` `NULL`, the code searches for unassigned tasks, whereas when `to_r` `==` `NULL`, it doesn't. Altogether it seems best to try the move first, and to only try the swap if `KheWidenedTaskSetMoveCheck` returns `false`.

Is swapping only reasonable when both resources are non-`NULL`? No. When `to_r` is `NULL`, swapping equals moving except for optional tasks. But when `from_r` is `NULL`, `to_r` must be

non-NULL, and swapping replaces some of `to_r`'s tasks with different tasks that were previously unassigned. While this is not striking, it is different, and cases exist where it would do good.

A widened task set is not kept up to date as the solution changes. If it gets out of date the only option is to delete it and make a fresh one. The four move and swap functions share the work of finding the tasks assigned to `to_r` that are running on the same days as `wts`'s core and wings: if a call on one of these functions for a given `wts` has the same value for `to_r` as the previous call, this shared work is not redone. Care is needed here when the solution is changing.

Finally, a few helper functions. Function

```
void KheWidenedTaskSetInterval(KHE_WIDENED_TASK_SET wts,
    int left_count, int right_count, int *first_index, int *last_index);
```

sets `*first_index` and `*last_index` to the endpoints of the bounding interval of `wts`'s core plus the first `left_count` elements of its left wing and the first `right_count` elements of its right wing—the interval affected by a move or swap.

There are several debug functions for widened task sets. Function

```
void KheWidenedTaskSetDebug(KHE_WIDENED_TASK_SET wts, int left_count,
    int right_count, int verbosity, int indent, FILE *fp);
```

prints `wts` onto `fp` with the given verbosity and indent. Only the first `left_count` and `right_count` left and right wing tasks are printed. The tasks are printed in chronological order, with the core enclosed in brackets if `left_count` or `right_count` is non-zero. Also,

```
void KheWidenedTaskSetMoveDebug(KHE_WIDENED_TASK_SET wts,
    KHE_RESOURCE to_r, int left_count, int right_count,
    int verbosity, int indent, FILE *fp);
void KheWidenedTaskSetMovePartialDebug(KHE_WIDENED_TASK_SET wts,
    KHE_RESOURCE to_r, int first_index, int last_index,
    int verbosity, int indent, FILE *fp);
void KheWidenedTaskSetSwapDebug(KHE_WIDENED_TASK_SET wts,
    KHE_RESOURCE to_r, int left_count, int right_count,
    int verbosity, int indent, FILE *fp);
```

are useful for debugging move, partial move, and swap operations. They assume that the preparation has been done for `to_r` (they do not do it themselves, since that would be wrong if the operation has already been carried out successfully), and print `from_r`, the tasks initially assigned `from_r` that will move, `to_r`, and the tasks initially assigned to `to_r` that will move.

### 11.8.3. Other task finding operations

This section presents some other task finding functions, not directly associated with widened task sets. At present there are only two of these functions:

```

bool KheFindTaskRunRight(KHE_TASK_FINDER tf, KHE_RESOURCE_TYPE rt,
    KHE_RESOURCE from_r, bool separate_need_asst, int days_first_index,
    KHE_TASK_SET res_ts, int *res_first_index, int *res_last_index);
bool KheFindTaskRunLeft(KHE_TASK_FINDER tf, KHE_RESOURCE_TYPE rt,
    KHE_RESOURCE from_r, bool separate_need_asst, int days_last_index,
    KHE_TASK_SET res_ts, int *res_first_index, int *res_last_index);

```

`KheFindTaskRunRight` searches from left to right, starting from the day whose index is `day_first_index`, for the first run of tasks of type `rt` assigned `from_r`. If it finds such a run, it returns `true`, sets `res_ts` to hold the tasks (from left to right), and sets `*res_first_index` and `*res_last_index` to the endpoints of their bounding interval. `KheFindTaskRunLeft` is the same, only searching from right to left, starting from `days_last_index`. The task runs are *maximal*: they cannot be extended on either side, given that tasks running on or before `days_first_index - 1`, or on or after `days_last_index + 1`, are out of bounds.

If `separate_need_asst` is `true`, each run found consists either of tasks for all of which `KheTaskNeedsAssignment` is `true`, or tasks for all of which `KheTaskNeedsAssignment` is `false`: whatever value the first task has, the others must share, and the run ends just before the first that doesn't, or the first free day as usual. Two such runs may occur on adjacent days.

When `from_r` is `NULL`, unassigned tasks are found. The first could be any unassigned task of type `rt` (it is this case that `rt` is needed for), but the others must be compatible with the first, in the sense defined earlier for widened task sets.

### 11.9. Other resource-structural solvers

This section documents some miscellaneous functions that reorganize task trees, represented by taskings. They assume that only unfixed tasks lie in taskings, and they preserve this condition.

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.

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.

### 11.10. Task groups

There are cases where two tasks are interchangeable as far as resource assignment is concerned, because they demand the same kinds of resources at the same times. The *task group* embodies KHE's approach to taking advantage of interchangeable tasks.

The *full task set* of an unfixed task is the task itself and all the tasks assigned to it, directly or indirectly (all its followers), omitting tasks that do not lie in a meet. An unfixed task is *time-complete* if each task of its full task set lies in a meet that has been assigned a time. Two time-complete tasks are *time-equal* if their full task sets have equal cardinality, and the two sets can be sorted so that corresponding tasks have equal starting times, durations, and workloads. Two unfixed tasks are *interchangeable* if they are time-complete and time-equal, and their domains are equal. When two resources are assigned to two interchangeable tasks, either resource can be assigned to either task and it does not matter which is assigned to which. (Exception: if a limit resources constraint contains one of the tasks but not the other, it does matter.)

A *task group* is a set of pairwise interchangeable tasks. Task groups occur naturally when there are linked events, or when time assignments are regular. Virtually any resource assignment algorithm can benefit from task groups. Assigning to a task group rather than to a task eliminates symmetries that can slow down searching. A given resource can only be assigned to one task of a task group, since its tasks overlap in time, so task groups help with estimating realistically how many resources are available, and how much workload is open to a resource.

Objects of type `KHE_TASK_GROUP` hold one set of interchangeable tasks, and objects of type `KHE_TASK_GROUPS` hold a set of task groups. Such a set can be created by calling

```
KHE_TASK_GROUPS KheTaskGroupsMakeFromTasking(KHE_TASKING tasking);
```

It places every task of `tasking` into one task group. The task groups are maximal.

To remove a set of task groups (but not their tasks), call

```
void KheTaskGroupsDelete(KHE_TASK_GROUPS task_groups);
```

To access the task groups, call

```
int KheTaskGroupsTaskGroupCount(KHE_TASK_GROUPS task_groups);
KHE_TASK_GROUP KheTaskGroupsTaskGroup(KHE_TASK_GROUPS task_groups, int i);
```

To access the tasks of a task group, call

```
int KheTaskGroupTaskCount(KHE_TASK_GROUP task_group);
TASK KheTaskGroupTask(KHE_TASK_GROUP task, int i);
```

There must be at least one task in a task group, otherwise the task group would not have been made. Task groups are not kept up to date as the solution changes, so if time assignments are being altered the affected tasks cannot be relied upon to remain interchangeable.

The tasks of a task group have the same total duration, total workload, and domain, and these common values are returned by

```
int KheTaskGroupTotalDuration(KHE_TASK_GROUP task_group);
float KheTaskGroupTotalWorkload(KHE_TASK_GROUP task_group);
KHE_RESOURCE_GROUP KheTaskGroupDomain(KHE_TASK_GROUP task_group);
```

`KheTaskGroupTotalDuration` is the value of `KheTaskTotalDuration` shared by the tasks, not the sum of their durations; and similarly for `KheTaskGroupTotalWorkload`.

For the convenience of algorithms that use task groups, function

```
int KheTaskGroupDecreasingDurationCmp(KHE_TASK_GROUP tg1,
    KHE_TASK_GROUP tg2);
```

is a comparison function that may be used to sort task groups by decreasing duration.

Because the tasks of a task group are interchangeable, it does not matter which of them is assigned when assigning resources to them. This makes the following functions possible:

```
int KheTaskGroupUnassignedTaskCount(KHE_TASK_GROUP task_group);
bool KheTaskGroupAssignCheck(KHE_TASK_GROUP task_group, KHE_RESOURCE r);
bool KheTaskGroupAssign(KHE_TASK_GROUP task_group, KHE_RESOURCE r);
void KheTaskGroupUnAssign(KHE_TASK_GROUP task_group, KHE_RESOURCE r);
```

`KheTaskGroupUnassignedTaskCount` returns the number of unassigned tasks in `task_group`; `KheTaskGroupAssignCheck` checks whether `r` can be assigned to a task of `task_group` (by finding the first unassigned task and checking there); `KheTaskGroupAssign` is the same, only it actually makes the assignment, using `KheTaskAssign`, if it can; and `KheTaskGroupUnAssign` finds a task of `task_group` currently assigned `r`, and unassigns that task.

The tasks of a task group may have different constraints, in which case assigning one may change the solution cost differently from assigning another. This is handled heuristically as follows. The first time `KheTaskGroupAssign` returns `true`, it tries assigning `r` to each task of the task group, notes the solution cost after each, and sorts the tasks into increasing order of this cost. Then it and all later calls assign the first unassigned task in this order.

The usual debug functions are available:

```
void KheTaskGroupDebug(KHE_TASK_GROUP task_group, int verbosity,
    int indent, FILE *fp);
void KheTaskGroupsDebug(KHE_TASK_GROUPS task_groups, int verbosity,
    int indent, FILE *fp);
```

print `task_group` and `task_groups` onto `fp` with the given verbosity and indent.

# 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.*

Assignments are permitted 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. This strategy is followed by KHE's high-level resource solvers (Section 12.9). They set this option, so it is futile for the end user to set it when using these functions.

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 may be 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.

### 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.

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 8.3), 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. 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 task groups (Section 11.10), not individual tasks. Each task of a task group is assignable by the same resources, so one list of suitable resources is kept per task group. At each step, a task group 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 task groups is rechecked. If it proves to be no longer assignable to some of them, their priorities are changed. The task groups are held in a priority queue (Section A.5), which allows their queue positions to be updated efficiently when their priorities change.

### 12.4.3. 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 clustered into task groups (Section 11.10). 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 task groups 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 task groups 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 task groups are taken in decreasing order of the

maximum, over all tasks  $t$  of the task group, of  $\text{KheMeetDemand}(m)$ , 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.

#### 12.4.4. Split assignments

After solver functions such as  $\text{KheMostConstrainedFirstAssignResources}$  (Section 12.4.2) and  $\text{KheEjectionChainRepairResources}$  (Section 12.6) 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  $\text{tasking}$  while maintaining the resource assignment invariant. Any tasks which were unassigned to begin with are replaced in  $\text{tasking}$  by their child tasks.

At the core of  $\text{KheFindSplitResourceAssignments}$  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,  $\text{KheFindSplitResourceAssignments}$  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. 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 12.5.2. 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.

### 12.5.1. 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 `rmds` 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 8.3), 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, bool preserve_existing);
```

(for `preserve_existing`, see below), 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.

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. If `preserve_existing` is true, it is unassigned;
4. 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  $T$ ;
5. It is not derived from a preassigned event resource;
6. Its assignment is not fixed (by `KheTaskAssignFix`, or because it is a cycle task);
7. 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 12.5.2.

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 (for example, when `preserve_existing` is true), 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.

### 12.5.2. 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  $(g_i, w_i d_i)$  works well, and when it is low, adding preference  $(G \cup \{r_0\} - g_i, w_i d_i)$  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  $(g_i, w_i d_i)$  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  $(G \cup \{r_0\} - g_i, w_i d_i)$  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  $(g_i, w_i d_i)$  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  $(g_i, w_i d_i)$  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.

### 12.5.3. 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 12.5.1.

`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’, and this is the default value.

`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 12.5.4). The default value, 0, produces no lookahead.

`rs_time_sweep_preserve_existing_off`

A Boolean option which, when `true`, causes existing assignments to not be preserved, by passing `false` to `KheResourceMatchingSolverSolve` for `preserve_existing`.

`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.

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.

#### 12.5.4. 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 11.5.4). 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`.

### 12.5.5. 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.

## 12.6. 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 13.7.

## 12.7. 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.

Most of this section is devoted to function `KheResourcePairReassign`, which carries out this idea while trying to save time by detecting symmetries. The last subsection is devoted to a simpler function, `KheResourcePairSimpleReassign`, which is better suited to nurse rostering.

### 12.7.1. 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.

### 12.7.2. 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 partition. 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).

### 12.7.3. 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.

#### 12.7.4. 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  $r_1$  or  $r_2$ , 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  $r_1$  to the nodes of one part and  $r_2$  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  $r_1$  and  $r_2$  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.

### 12.7.5. A simpler resource pair repair

This section describes `KheResourcePairSimpleReassign`, a simpler resource pair repair function, which is suited to nurse rostering applications. It does not attempt to find symmetries, which are rare in instances whose events all have duration 1. Instead it offers a way of limiting the search space to just one segment of the timetables of the two resources:

```
bool KheResourcePairSimpleReassign(KHE_SOLN soln,
    KHE_RESOURCE r1, KHE_RESOURCE r2, KHE_FRAME frame, int fi, int li,
    bool resource_invariant, int max_assignments);
```

It tries all combinations of reassignments of `r1` and `r2` to the tasks assigned to them in `frame` between indexes `fi` and `li` inclusive. If `resource_invariant` is true, only assignments satisfying the resource invariant are acceptable.

Each resource is assigned to at most one task per time group of `frame`, so there are two choices at each time group (or 1 when neither resource is assigned), and the size of the search space is at most 2 to the power `KheFrameTimeGroupCount(frame)`. However, the function stops after making `max_assignments` assignments in total. If it finds an improvement, it changes `soln` to it and returns true, otherwise it keeps the original assignments and returns false.

A convenient way to invoke `KheResourcePairSimpleReassign` repeatedly is

```
bool KheResourcePairSimpleRepair(KHE_SOLN soln, KHE_OPTIONS options);
```

It returns true if any of its calls to `KheResourcePairSimpleReassign` return true. It obtains `frame` and `resource_invariant` from `options`. The following options are also consulted and determine the other parameters:

`rs_pair_off`

A Boolean option which, when true, turns resource simple pair repair off.

`rs_pair_select`

This option determines which pairs of resources are tried. Its value may be "none", meaning that no pairs are tried, giving another way to turn this repair off; or "all", meaning that for each resource type, all pairs are tried; or "adjacent", meaning that each adjacent

pair of resources (0 and 1, 1 and 2, 2 and 3, and so on) in each resource type is tried; or "defective" (the default), meaning that for each resource type, all pairs of resources for which at least one of the resources has a defective resource monitor are tried.

`rs_pair_parts`

The value of `KheFrameTimeGroupCount(frame)` on each call. For example, setting this value to 7 (the default) reassigns one week.

`rs_pair_start, rs_pair_increment`

The value of `frame`'s start index on the first call, and how much it is incremented by on each subsequent call. The default values are 0 and `rs_pair_parts`.

`rs_pair_max`

The value of `max_assignments`. Its default value is 1000000, which is fine when `rs_pair_parts` is 7 but may need to be reduced when it is larger.

Resource pair repair runs very quickly when the default values are used, as would be expected given that the search space has size at most  $2^7$  per pair. On tests run by the author it found several improvements, enough to justify its modest running time. Setting `rs_pair_parts` to 14 gave some further improvement, but it also slowed the solves down noticeably.

Another way to invoke `KheResourcePairSimpleReassign` repeatedly is

```
bool KheResourcePairSimpleBusyRepair(KHE_SOLN soln, KHE_OPTIONS options);
```

This is like `KheResourcePairSimpleRepair` except that it makes a different set of calls to `KheResourcePairSimpleReassign`, and its options have different names: `rs_bpair_off`, `rs_bpair_select`, `rs_bpair_parts`, `rs_bpair_start`, `rs_bpair_increment`, and `rs_bpair_max`. These are like the corresponding options of `KheResourcePairSimpleRepair`, except `rs_bpair_select` has no effect, and the default value of `rs_bpair_parts` is 14.

For each resource, `KheResourcePairSimpleBusyRepair` obtains its number of busy times and maximum busy times from the functions documented in Section 4.7. It pairs the most overloaded resource with the most underloaded one, the second most overloaded with the second most underloaded, and so on, and calls `KheResourcePairSimpleReassign` on each pair. Each resource participates in at most one call to `KheResourcePairSimpleReassign`, so the number of these calls is very much smaller than the number made by `KheResourcePairSimpleRepair`, and accordingly `rs_bpair_parts` can reasonably be larger than `rs_pair_parts`.

When tested by the author, `KheResourcePairSimpleBusyRepair` produced very promising pairs, but failed to improve the solution, both when `rs_bpair_parts` had the default value 14, and when it was increased to 21, with `rs_bpair_increment` set to 7. At that point a small but noticeable amount of time was being consumed by some pairs. Pairing each overloaded resource with the most underloaded resource also failed to find any improvement. Accordingly, the default value of `rs_bpair_off` has been set to true.

Also available are

```
bool KheResourceTripleSimpleReassign(KHE_SOLN soln,
    KHE_RESOURCE r1, KHE_RESOURCE r2, KHE_RESOURCE r3, KHE_FRAME frame,
    int fi, int li, bool resource_invariant, int max_assignments);
```

and

```
bool KheResourceTripleSimpleRepair(KHE_SOLN soln, KHE_OPTIONS options);
```

These are just like `KheResourcePairSimpleReassign` and `KheResourcePairSimpleRepair`, only reassigning three resources rather than two. `KheResourceTripeSimpleRepair` consults options `rs_triple_off`, `rs_triple_select`, `rs_triple_parts`, `rs_triple_start`, `rs_triple_increment`, and `rs_triple_max`. These are like the corresponding options for resource pairs, except that `rs_triple_select` has default value "none", not "defective".

The author undertook one test of resource triple repair. It ran after resource pair repair, but still it found five improvements. The search space has size  $6^7$  per resource triple when `rs_triple_parts` is 7. This is not impossibly large, but trying all triples containing at least one defective resource is very slow. The author's test took several hours (the instance was INRC1-LH03, with about 50 nurses) and convinced him that resource triple repair is not suitable for routine use. It may be suitable for repairing a few very bad resources.

#### 12.7.6. Resource pair run reassignment

This section describes yet another resource pair reassignment algorithm, again for nurse rostering. The basic function is

```
bool KheResourcePairRunReassign(KHE_SOLN soln, KHE_OPTIONS options,
    KHE_RESOURCE r1, KHE_RESOURCE r2, int fi, int li,
    bool resource_invariant, int max_assignments);
```

This is similar to earlier functions in that it tries to reassign `r1` and `r2` to the tasks currently assigned those resources in the range of days `fi` to `li` so as to improve the solution, returning true if it succeeds. As before, `resource_invariant` says whether the resource invariant is to apply, and `max_assignments` limits the number of assignments that may be tried.

The difference is that this function calls `KheFindTaskRunRight` (Section 11.8.3) repeatedly, to group the tasks assigned a given resource into runs of tasks on consecutive days. Each run is taken to be indivisible, as though its tasks were grouped together. This makes for a smaller search space, allowing `fi` and `li` to cover larger intervals, possibly the whole cycle.

Again similarly to earlier functions,

```
bool KheResourcePairRunRepair(KHE_SOLN soln, KHE_OPTIONS options);
```

may be called to make many calls on `KheResourcePairRunReassign` for different pairs of resources `r1` and `r2`. There is also a time saving because `KheResourcePairRunReassign` finds the runs for each resource just once, unless one of its calls to `KheResourcePairRunReassign` is successful, in which case it finds the runs for the two resources again.

`KheResourcePairRunRepair` consults options `rs_rpair_off`, `rs_rpair_select`, `rs_rpair_parts`, `rs_rpair_start`, `rs_rpair_increment`, and `rs_rpair_max`. These are as for `KheResourcePairSimpleRepair` above except that the default value of `rs_rpair_parts` is 28, much larger than for `rs_pair_starts`. This is because `fi` and `li` can reasonably cover much larger intervals when reassigning runs, as explained earlier.

### 12.7.7. Resource pair swapping

Yet another way to repair two resources is to simply swap their timetables:

```
bool KheResourcePairSwap(KHE_SOLN soln,
    KHE_RESOURCE r1, KHE_RESOURCE r2, KHE_FRAME frame, int fi, int li,
    bool resource_invariant);
```

It attempts to move all tasks of `soln` lying (partly or wholly) between indexes `fi` and `li` inclusive in `frame` initially assigned `r1` to `r2` and vice versa. If `resource_invariant` is true, only assignments satisfying the resource invariant are acceptable. If all these assignments succeed and the solution cost is reduced, `KheResourcePairSwap` leaves `soln` in the new state and returns true. Otherwise it leaves `soln` unchanged and returns false.

A convenient way to invoke `KheResourcePairSwap` repeatedly is

```
bool KheResourcePairSwapRepair(KHE_SOLN soln, KHE_OPTIONS options);
```

It returns true if any of its calls to `KheResourcePairSwap` return true. It obtains `frame` and `resource_invariant` from `options`. The following options are also consulted and determine the other parameters:

`rs_swap_off`

A Boolean option which, when true, turns resource swap repair off.

`rs_swap_select`

This option determines which pairs of resources are tried. Its values are the same as those for `rs_pair_select`.

`rs_swap_parts`

The value of `fi - li + 1` on each call. For example, setting this value to 7 reassigns one week. The default value is `KheTimeGroupTimeCount(frame)`, which means that the entire timetables of the two resources are swapped.

`rs_swap_start, rs_swap_increment`

The value of `fi` on the first call, and how much it is incremented by on each subsequent call. The default values are 0 and `rs_swap_parts`.

*report on its effectiveness still to do*

### 12.8. Trying unassignments

KHE's solvers assume that it is always a good thing to assign a resource to a task. However, occasionally there are cases where cost can be reduced by unassigning a task, because the cost of the resulting assign resource defect is less than the cost of the defects introduced by the assignment. As some acknowledgement of these anomalous cases, KHE offers

```
bool KheSolnTryTaskUnAssignments(KHE_SOLN soln);
```

for use at the end. It tries unassigning each proper root task of `soln` in turn. If any unassign-

ment reduces the cost of `soln`, it is not reassigned. The result is `true` if any unassignments were 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.

## 12.9. Putting it all together

This section presents functions which assemble the pieces described in previous sections.

Three structural decisions face a resource solver. Should it work with split assignments? Should it preserve the resource assignment invariant? Should it respect the domains of tasks? It is easy to write solvers that can be used with any combination of these decisions, as follows.

Get unsplit assignments by building a task tree with avoid split assignments jobs. Allow split assignments by calling `KheTaskingAllowSplitAssignments` (Section 11.9). Either way, a solver assigns resources to unfixed tasks, without knowing or caring if they have followers.

By enclosing each attempt to change the solution in `KheAtomicTransactionBegin` and `KheAtomicTransactionEnd` (Section 12.2), a solver can preserve the resource assignment invariant, or not, depending on the value of a Boolean parameter.

If domains are to be respected, do nothing; if not, then before running the solver, call `KheTaskingEnlargeDomains` (Section 11.9) to enlarge them to the full set of resources.

A sequence of three functions,

```
bool KheTaskingAssignResourcesStage1(KHE_TASKING tasking,
    KHE_OPTIONS options);
bool KheTaskingAssignResourcesStage2(KHE_TASKING tasking,
    KHE_OPTIONS options);
bool KheTaskingAssignResourcesStage3(KHE_TASKING tasking,
    KHE_OPTIONS options);
```

packages this chapter's ideas into a three-stage solver which assigns resources to the tasks of `tasking`. Called in order, they take a 'progressive corruption' approach to the decisions just described: they are spotless at first, but they slide into the gutter towards the end.

`KheTaskingAssignResourcesStage1` begins by setting option `"rs_invariant"` to `true`. Then it assigns resources to the unassigned unfixed tasks of `tasking`, using the assignment algorithm indicated by the `rs_constructor` option, as detailed below. This is followed by a call to a private function, called the 'repair part' here, which tries several kinds of repairs, including `KheResourceRematch` (Section 12.5.5), `KheEjectionChainRepairResources` (Section 12.6), and, in the employee scheduling model, `KheResourcePairSimpleRepair` (Section 12.7.5).

After this, the great majority of the tasks, probably, have been assigned resources. There are no split assignments, the resource assignment invariant is preserved, and domains are respected.

`KheTaskingAssignResourcesStage2` does nothing if the instance contains no avoid split assignments constraints. Otherwise, it calls `KheFindSplitResourceAssignments` to build split assignments, and `KheTaskingAllowSplitAssignments` to permit all tasks, assigned or not, to be split. It then calls the repair part. Ejection chain repair will try to remove split assignments

(it has always been able to, but there has been nothing to trigger it until now), and it also tries to assign unassigned tasks, even at the cost of splitting assignments that were previously unsplit.

`KheTaskingAssignResourcesStage3` is very corrupt indeed. It turns the resource assignment invariant off, enlarges domains by calling `KheTaskingEnlargeDomains`, then runs the repair part yet again. Enlarging domains makes sense only at the very end, and will help only if any resource is better than none. Because the resource assignment invariant is removed, this stage should be run only after the first two stages have been run *for each resource type*.

The options consulted by the three functions directly are

#### `rs_constructor`

This option determines which resource solver `KheTaskingAssignResourcesStage1` calls to construct the initial resource assignment. If it is "none", then no solver is called: no assignments are made, and the repair stages have to find assignments as well as repair them. This is not likely to work well, although it is worth trying. If it is "most\_constrained", then `KheMostConstrainedFirstAssignResources` (Section 12.4.2) is called. If it is "resource\_packing", then `KheResourcePackAssignResources` (Section 12.4.3) is called. If it is "time\_sweep", then `KheTimeSweepAssignResources` (Section 12.5.3) is called. If it is "auto" (the default), then one of these functions is called, depending on the model and whether there are avoid split assignments constraints. Finally, if it is "requested\_only", then only `KheSolnAssignRequestedResources` (Section 12.4.1) is called.

#### `rs_group_by_resource`

This option, when true, causes the repair part of `KheTaskingAssignResourcesStage1` to be executed twice, first in the usual way, and then with the tasks grouped by resource using `KheTaskingGroupByResource` (Section 11.6). The grouping is then removed.

#### `rs_repair_off`

This option, when true, causes the repair part to do nothing in all three stages, leaving just the initial construction, including any repair steps within the construction algorithms.

#### `rs_repair1_off`, `rs_repair2_off`, `rs_repair3_off`

These three options, when true, cause stage 1, 2, or 3 of the repair part to do nothing.

#### `rs_repair_rematch_off`

This option, when true, turns off rematching repair in the repair parts.

#### `rs_repair_ejection_off`

This option, when true, turns off ejection chain repair in the repair parts.

#### `rs_repair_time_limit`

A string option defining a soft time limit for the repair part of each stage. The format is the one accepted by `KheTimeFromString` (Section 8.1): `secs`, or `mins:secs`, or `hrs:mins:secs`, or the special value `-`, meaning 'no limit', which is the default value.

Many other options influence the solvers called by the three functions. All three functions set the `rs_invariant` option, making it futile for the user to do so if they are used.

# Chapter 13. Ejection Chains

Ejection chains are sequences of repairs that generalize the augmenting paths from bipartite matching. They are due to Glover [3], who applied them to the travelling salesman problem.

## 13.1. Introduction

An ejection chain algorithm targets one defect and tries a set of alternative *repairs* on it. A repair could be a simple move or swap, or something arbitrarily complex. It removes the defect, but may introduce new defects. If no new defects of significant cost appear, that is success. If just one significant new defect appears, the method calls itself recursively to try to remove it; in this way a chain of coordinated repairs is built up. 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. It can also try to remove all the new defects, although that is not often useful.

Corresponding to the well-known function for finding an augmenting path in a bipartite graph, starting from a given 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 it 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`. The precondition implies that removing *d* without adding new defects would be one way to 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 ways that *d* could be repaired. For each repair, it applies it and

receives the set of new defects introduced by that repair, 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  $\text{cost}(s)$  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  $\text{cost}(s)$  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 design problems in combining it with ejection trees (Section 13.5.3). It is no great 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 depth, as described next.

The tree searched by `Augment` as presented may easily grow to exponential size, which is not the intention. The author has tried 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 depth of recursion to a fixed constant, perhaps 3 or 4. The maximum depth 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 depth parameter increased each time. Another idea is to use a small depth on the first iteration of the main loop (see below), and increase it on later iterations.

The second method is the one 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  $\text{cost}(s)$ . 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.

When there are several defect types, several `Augment` algorithms are needed, one for each defect type, dynamically dispatched on the type. The repairs are usually applied directly, rather than indirectly via objects built to represent them.

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.

## 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. The framework uses visit numbers (Section 4.2.5), in the conventional way.

To support statistics gathering (Section 13.6), a single ejector object may be re-used many times, even on different instances (although not in parallel—an ejector object is highly mutable and cannot be shared by two or more threads). This makes it important to distinguish between those parts of the ejector which are constant throughout its lifetime, and those parts which vary from solve to solve. This section is concerned with the constant parts.

An ejector is constructed by a sequence of calls beginning with

```
KHE_EJECTOR KheEjectorMakeBegin(HA_ARENA a);
```

and ending with

```
void KheEjectorMakeEnd(KHE_EJECTOR ej);
```

Its constant parts are set by calling the functions given in this section, between these two calls. After `KheEjectorMakeEnd` is called the constant parts cannot be changed.

The ejector is stored in arena `a`, which is accessible by calling

```
HA_ARENA KheEjectorArena(KHE_EJECTOR ej);
```

There is no function to delete an ejector; as usual, it is deleted when its arena is deleted.

The constant parts of an ejector object are a sequence of *major schedules* and a set of *augment functions*. The major schedules control the detailed behaviour of each solve, while the augment functions are callback functions passed in by the user, containing the repairs.

A major schedule is represented by an object of type `KHE_EJECTOR_MAJOR_SCHEDULE`. It contains a sequence of *minor schedules*. A minor schedule is represented by an object of type `KHE_EJECTOR_MINOR_SCHEDULE`. It contains two attributes: *maximum depth* and *may-revisit*. The entire main loop of the algorithm, which repeatedly tries to augment out of each defect until no further improvements can be found, is repeated once for each major schedule in order. An ejector with no major schedules does nothing. Within each 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. A major schedule with no minor schedules does nothing.

An ejector's major schedules may be accessed at any time by

```
int KheEjectorMajorScheduleCount(KHE_EJECTOR ej);
KHE_EJECTOR_MAJOR_SCHEDULE KheEjectorMajorSchedule(KHE_EJECTOR ej, int i);
```

in the usual way. To begin and end adding one major schedule, call

```
void KheEjectorMajorScheduleBegin(KHE_EJECTOR ej);
void KheEjectorMajorScheduleEnd(KHE_EJECTOR ej);
```

Its minor schedules are added by calls to `KheEjectorMinorScheduleAdd` (described below) in

between these calls, and may then be accessed by calling

```
int KheEjectorMajorScheduleMinorScheduleCount(KHE_EJECTOR_MAJOR_SCHEDULE ejm);
KHE_EJECTOR_MINOR_SCHEDULE KheEjectorMajorScheduleMinorSchedule(
    KHE_EJECTOR_MAJOR_SCHEDULE ejm, int i);
```

in the usual way.

Minor schedules are added to a major schedule by calls to

```
void KheEjectorMinorScheduleAdd(KHE_EJECTOR ej,
    int max_depth, bool may_revisit);
```

in between the calls to `KheEjectorMajorScheduleBegin` and `KheEjectorMajorScheduleEnd`. The attributes of a minor schedule may be retrieved by

```
int KheEjectorMinorScheduleMaxDepth(KHE_EJECTOR_MINOR_SCHEDULE ejms);
bool KheEjectorMinorScheduleMayRevisit(KHE_EJECTOR_MINOR_SCHEDULE ejms);
```

The `max_depth` attribute determines the maximum depth of recursion (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.

Each *tree* (rooted at one augment call in the main loop) gets a new global visit number, making it free to change any part of the solution. When `may_revisit` is false, each part of the solution may be changed by at most one of the recursive calls within one tree; when it is true, each part may be changed by any number of them, although only once along any one chain.

Here are some examples. To allow up to two repairs on any chain, with revisiting:

```
KheEjectorMajorScheduleBegin(ej);
KheEjectorMinorScheduleAdd(ej, 2, true);
KheEjectorMajorScheduleEnd(ej);
```

To allow arbitrary-depth recursion, but no revisiting:

```
KheEjectorMajorScheduleBegin(ej);
KheEjectorMinorScheduleAdd(ej, INT_MAX, false);
KheEjectorMajorScheduleEnd(ej);
```

It is a bad idea to set `max_depth` to a large value and `may_revisit` to true in the same schedule, because the algorithm will then usually take exponential time. But setting `max_depth` to a small constant, or setting `may_revisit` to false, or both, guarantees polynomial time. Another interesting idea is *iterative deepening*, in which several depths are tried. For example,

```

KheEjectorMajorScheduleBegin(ej);
KheEjectorMinorScheduleAdd(ej, 1, true);
KheEjectorMinorScheduleAdd(ej, 2, true);
KheEjectorMinorScheduleAdd(ej, 3, true);
KheEjectorMinorScheduleAdd(ej, INT_MAX, false);
KheEjectorMajorScheduleEnd(ej);

```

tries maximum depth 1, then 2, then 3, and finishes with arbitrary depth.

Here are two faster ways to add schedules:

```

void KheEjectorAddDefaultSchedules(KHE_EJECTOR ej);
void KheEjectorSetSchedulesFromString(KHE_EJECTOR ej,
    char *ejector_schedules_string);

```

`KheEjectorAddDefaultSchedules` adds some major and minor schedules, chosen by the author as reasonable defaults. At present it does this:

```

KheEjectorMajorScheduleBegin(ej);
KheEjectorMinorScheduleAdd(ej, 1, true);
KheEjectorMajorScheduleEnd(ej);
KheEjectorMajorScheduleBegin(ej);
KheEjectorMinorScheduleAdd(ej, INT_MAX, false);
KheEjectorMajorScheduleEnd(ej);

```

`KheEjectorSetSchedulesFromString` interprets its string as a sequence of instructions for adding schedules to `ej`, and follows the instructions. It supports tests that compare schedules.

The string contains a sequence of one or more major schedules separated by commas. Each major schedule consists of a sequence of minor schedules, each represented by two characters. The first character of each minor schedule is a digit or `u`, and defines the depth limit; `u` means unlimited. The second is either `+`, meaning with revisiting, or `-`, meaning without it. For example, "`1+, u-`" defines two major schedules. The first has one minor schedule with depth limit 1 and revisiting; the second has one minor schedule with unlimited depth and no revisiting.

The other constant part of an ejector object is the set of augment functions, one function for each kind of defect that the user wants the ejector to repair. These augment functions are written by the user, as described in Section 13.4, 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 13.6), 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_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` is also a child monitor of `continue_gm`.

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_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_FRAME KheEjectorFrame(KHE_EJECTOR ej);
KHE_EVENT_TIMETABLE_MONITOR KheEjectorEventTimetableMonitor(KHE_EJECTOR ej);
KHE_SOLN KheEjectorSoln(KHE_EJECTOR ej);
KHE_COST KheEjectorTargetCost(KHE_EJECTOR ej);
KHE_EJECTOR_MAJOR_SCHEDULE KheEjectorCurrMajorSchedule(KHE_EJECTOR ej);
KHE_EJECTOR_MINOR_SCHEDULE KheEjectorCurrMinorSchedule(KHE_EJECTOR ej);
bool KheEjectorCurrMayRevisit(KHE_EJECTOR ej);
int KheEjectorCurrDepth(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`.

`KheEjectorFrame` is the frame (Section 5.10) that `ej` passes to `KheEjectingTaskMove` (Section 12.3), just `KheFrameOption(options, "gs_common_frame", ins)` (Section 5.10).

`KheEjectorEventTimetableMonitor` is the value of the `gs_event_timetable_monitor` option (Section 8.3, or NULL if absent. When it is present, some augment functions use it to find events running at the same time as a given event; when it is absent, those functions omit repairs that depend on finding those events.

`KheEjectorSoln` is `start_gm`'s and `continue_gm`'s solution. `KheEjectorTargetCost` is the cost that the chain must improve on to succeed ( $c$  in the abstract code above): the cost that the solution had when `Augment` was most recently called from the main loop, except when ejection trees are in use, as explained in Section 13.5.3. `KheEjectorCurrMajorSchedule` is the current major schedule. `KheEjectorCurrMinorSchedule` is the current minor schedule,

and `KheEjectorCurrMayRevisit` is its `may_revisit` attribute—frequently used by augment functions, as will be seen. `KheEjectorCurrDepth` is the current depth (1 when the augment function was called from the main loop, 2 when 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 `gs_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 that debug prints should be indented by; this is twice the current depth.

### 13.4. How to write an augment function

An augment function has type

```
typedef bool (*KHE_EJECTOR_AUGMENT_FN)(KHE_EJECTOR ej, KHE_MONITOR d);
```

The parameters are the ejector `ej` passed to `KheEjectorSolve`, and the defect `d` that the augment function is supposed to repair. It is a precondition that `d` have non-zero cost and removing that cost would make for a successful augment. The return value is supposed to be equal to the value returned by the most recent call to `KheEjectorRepairEnd` below, or false if there have been no repairs. The ejector does not use this value, since it already knows what the most recent call to `KheEjectorRepairEnd` returned. However, it is convenient to have it available for when one augment function calls another.

Augment functions often look like this, although not necessarily exactly:

```
bool ExampleAugment(KHE_EJECTOR ej, 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 boilerplate code

```
if( KheEjectorCurrMayRevisit(ej) )
    KheEntityUnVisit(e);
```

marks the entity unvisited if revisiting is allowed.

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 drop all visiting code if desired, producing this:

```
bool ExampleAugment(KHE_EJECTOR ej, 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 will be 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`, 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 of them is not needed.

Functions `KheEjectorRepairBegin` and `KheEjectorRepairEnd` are supplied by KHE:

```
void KheEjectorRepairBegin(KHE_EJECTOR ej);
bool KheEjectorRepairEndLong(KHE_EJECTOR ej, int repair_type,
    bool success, int max_sub_chains, bool save_and_sort,
    void (*on_success_fn)(void *on_success_val), void *on_success_val);
bool KheEjectorRepairEnd(KHE_EJECTOR ej, int repair_type, bool success);
```

`KheEjectorRepairEnd` and `KheEjectorRepairEndLong` are the same except that `KheEjectorRepairEnd`, the best choice in most circumstances, relieves the user of the burden of supplying the usual values for the last four parameters, namely 1, false, NULL, and NULL. If other values are wanted for any of these parameters, then `KheEjectorRepairEndLong` must be

called. For `max_sub_chains` see Section 13.5.3; for `save_and_sort` see Section 13.5.4; and for `on_success_fn` and `on_success_val` see Section 13.5.5. ‘`KheEjectorRepairEnd`’ means ‘`KheEjectorRepairEnd` or `KheEjectorRepairEndLong`’ from here on.

Calls to `KheEjectorRepairBegin` and `KheEjectorRepairEnd` must occur in matching pairs. A call to `KheEjectorRepairBegin` informs `ej` that a repair is about to begin, and the following call to `KheEjectorRepairEnd` informs it that that repair has just ended. The repair is undone and redone (if 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 13.6). 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. The other parameters are ignored in that case. 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 decided that there is no point in trying more repairs for the current defect. Its reason for this is not the business of the augment function; it must return `true` immediately, without trying any more repairs (it is an error not to do this). In that case it does not matter whether the entity is marked unvisited or not before exit.

## 13.5. Variants of the ejection chains idea

This section presents some variants of the basic ejection chains idea.

### 13.5.1. Defect promotion

Successful chains begin by repairing a defect which is one of `start_gm`’s children, and 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`. These come from the trace object in the usual way. 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`.

### 13.5.2. 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 at depth 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( KheEjectorCurrDepth(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( KheEjectorCurrDepth(ej) == 1 )
    KheSolnNewGlobalVisit(soln);
```

to be executed within each call to `KheEjectorRepairBegin`, opening up the entire solution, not just to each sub-defect at depth 1, but to each repair of each sub-defect at depth 1.

### 13.5.3. Ejection trees

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` 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 `KheEjectorCurrDepth(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

$$\text{Potential}(d) = \text{KheMonitorCost}(d) - \text{KheMonitorLowerBound}(d)$$

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 6.8.3). 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 6.8.3).

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( NotAtDepthLimit() )
        {
            {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 depth 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

$$\text{sub\_c} = c + \text{GPotential}(d_2) + \dots + \text{GPotential}(d_n);$$

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

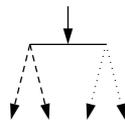
```
if( Success(s, sub_c, sub_x) )
    continue;
if( cost(s) - GPotential(di) >= sub_c )
    break;
```

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.

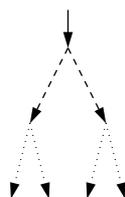
`Success(s, c, x)` implies `Success(s, sub_c, sub_x)` throughout `Augment`, because `sub_c >= c` and `sub_x` is a subset of `x`. This cannot be used to simplify `Augment`, but it does have one or two interesting consequences. For example, it applies transitively down through all active calls to `Augment`, 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. There is no reason to think that it would be better.

### 13.5.4. Sorting repairs

Each repair is usually followed immediately by recursive calls which extend the chain, where applicable. Setting the `save_and_sort` parameter of `KheEjectorRepairEndLong` 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) || (NotAtDepthLimit() &&
    cost(s) - (GPotential(d1) + ... + GPotential(dn)) < c)
```

holds after the repair, in the terminology of Section 13.5.3.

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.

### 13.5.5. Adjustment on success

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, since the domains of the meets are not supposed to 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 it is not undone.

`KheEjectorRepairEnd` returns `true` to tell the user's augment function to not generate any more repairs. Although this is often because the repair just ended was successful, it is not so always. So it would be a mistake to use this `true` result as the signal to do this kind of work.

Instead, cases like this may be handled by passing non-NULL values for the `on_success_fn` and `on_success_val` parameters of `KheEjectorRepairEndLong`. If the ejector subsequently decides to not undo that repair, it will then call

```
on_success_fn(on_success_val)
```

and the user can ensure that this removes meet bounds or whatever is wanted.

The call to `on_success_fn` should not change the cost of the solution; in practice it is limited to enlarging domains, unfixing, and so on. It works with all kinds of repairs and options, including `save_and_sort`. It is made at a time when the associated repair has been done or redone and not undone, but by no means directly after it is done or redone, since there may be a long chain to execute after that before success can be established. It is an error to assume that the state of the solution when `on_success_fn` is called is its state at the end of the repair.

## 13.6. 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.

### 13.6.1. 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 "ejector1" and "ejector2", as explained in Section 13.7.1; 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, using helper function `KheEjectorSetSchedulesFromString` (Section 13.2).

### 13.6.2. 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 ejection 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.

### 13.6.3. 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.

#### 13.6.4. 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 13.4), 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. When the `save_and_sort` option is in use, not all saved repairs are counted, only those redone after sorting. 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`.

### 13.6.5. 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.

### 13.7. Ejection chain time and resource repair functions

Previous sections have described ejectors in general. This section describes how ejectors are put to use in three ejection chain time and resource repair functions:

```
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);
```

`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 useful 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`.

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.

Although these functions target different parts of the solution, they share much of their implementation. In particular, they call the same augment functions, although the detailed behaviour of those functions depends on several options.

Here is the full list of options consulted or set by these functions. By convention, these options all have names beginning with `es_`. The most important options from the point of view of the user are those that he can reasonably set:

`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`. For full details on this method of obtaining ejector objects, see Section 13.7.1. While there is no pressing need for the user to set these options, since they

will be set the first time they are needed, 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_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 at depth 1 on the ejection chain, i.e. only 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).

#### `es_no_limit_busy_sequences`

A Boolean option which, when `true`, instructs augment functions that repair limit busy times defects to move only single assignments, not sequences of assignments.

#### `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.

#### `es_max_augments`

An upper limit on the number of augments tried on each attempt to repair a main loop defect. The default value is 120, which may be more effective than a larger number, especially when there is a time limit, because it helps the search to not waste time on lost causes.

`es_max_repairs`

An upper limit on the number of repairs tried on each attempt to repair a main loop defect. The default value is `INT_MAX`.

`es_no_promote_defects`

A Boolean option which, when `true`, instructs an ejector not to promote defects (Section 13.5.1).

`es_fresh_visits`

A Boolean option which, when `true`, instructs an ejector to make fresh visits (Section 13.5.2).

`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_task_seq_swaps_off`

A Boolean option which, when `true`, causes each move of a sequence of tasks on adjacent days to not try to find a second sequence of tasks in the other direction.

`es_group_limit_resources_off`

A Boolean option which, when `true`, instructs `KheEjectionChainPrepareMonitors` (Section 13.7.4) to not group limit resources monitors for the same events and resource types. Grouping these monitors is not exact, which is why this option is offered to turn it off, but it may be helpful anyway, to reduce the density of these constraints.

`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_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. Calling this number  $m$ , and supposing the task set being moved already covers  $s$  time groups, the maximum is  $\max(m, s)$ . The default value is 4.

`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.

The following options are set by KHE's functions, making it futile for the user to set them:

`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. `KheGeneralSolve2020` 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_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_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`.

All three functions require certain monitor groupings, but they set them up and remove them themselves. It is reasonable to worry about the time it takes to set up these group monitors. To investigate this question, the author ran just the group monitor setup and removal parts of function `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.

### 13.7.1. Obtaining ejector objects

The first issue is where the ejector objects come from. They need to be loaded with the right augment functions, and they need to be available after the solve, to give access to their statistics. KHE offers a function for creating an ejector object with its augment functions loaded:

```
KHE_EJECTOR KheEjectionChainEjectorMake(KHE_OPTIONS options, HA_ARENA a);
```

This returns a new ejector object, ready to solve with, stored in arena `a`. Its schedules come from the `es_schedules` option of `options`, and its augment functions are the ones defined by KHE. However, the three ejection chain solvers obtain their ejector objects indirectly, by calling

```
KHE_EJECTOR KheEjectionChainEjectorOption(KHE_OPTIONS options, char *key);
```

This retrieves the object from `options` with the given `key`, casts it to `KHE_EJECTOR`, and returns it; or if there is no such object, it calls `KheEjectionChainEjectorMake(options, a)` to create one, adds it to `options` under the given `key`, and returns it. The ejector's lifetime is the lifetime of `options`, because `a` is `KheOptionsArena(options)`.

For `KheEjectionChainNodeRepairTimes` and `KheEjectionChainLayerRepairTimes`

the key is "es\_ejector1"; for `KheEjectionChainRepairResources` it is "es\_ejector2". This allows for collecting two sets of statistics, for time assignment and resource assignment.

### 13.7.2. 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 constructed 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 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.

The options object contains an `es_repair_times` option, which when `true` allows repairs that assign and move meets, and an `es_repair_resources` option, 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 has caused problems. So the options object contains an `es_limit_node` option, 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.

### 13.7.3. Correlation problems involving demand defects

Section 9.8 discusses the problem of correlated monitors, and how it can be solved by grouping. Demand monitors obviously correlate with avoid clashes monitors: when there is a clash, there will be both an avoid clashes defect and an ordinary demand defect. They also correlate with avoid unavailable times, limit busy times, and limit workload monitors: when there is a hard avoid unavailable times defect, for example, there will also be a demand defect. This section explores several ways of handling these correlations, beginning with grouping.

*Group the correlated monitors.* Grouping is the ideal solution for correlation problems, but it does not work here. There are two reasons for this.

First, although every avoid clashes defect has a correlated ordinary demand defect, unless the resource involved is preassigned there is no way to predict which monitors will be correlated, since that depends on which resource is assigned to the demand monitors' tasks.

Second, grouping workload demand monitors with the resource monitors they are derived from has a subtle flaw. A demand defect is really the whole set of demand monitors that compete for the insufficient supply. (These sets are quite unpredictable and cannot themselves be grouped.) A workload demand defect shows up, not in the workload demand monitor itself,

but in an ordinary demand monitor that competes with it. This is because the last demand tixel to change its domain and require rematching is the one that misses out on the available supply, and workload demand monitors never change their domains. So this grouping still leaves two correlated defects ungrouped: the group monitor and the unmatched ordinary demand monitor.

If grouping does not work, then one of the correlated monitors has to be detached or otherwise ignored. There are several ways to do this.

*Detach demand monitors.* This does not work, because no-one notices that six Science meets are scheduled simultaneously when there are only five Science laboratories, and the resulting time assignment is useless.

*Attach demand monitors but exclude them from continue group monitors.* This prevents correlated monitors from appearing on defect lists, but both costs continue to be added to the solution cost, so removing the resource defect alone does not produce an overall improvement. The chain terminates in failure; the ejector cannot see that repairing the resource defect could work.

*Attach demand monitors but exclude them from the solution and continue group monitors; add a limit monitor holding them.* Then other monitors will be repaired, but no chain which increases the total number of demand defects will be accepted. This has two problems.

First, it can waste time constructing chains which fall at the last hurdle when it is discovered that they increase demand defects. This is particularly likely during time repair: the six Science meets problem could well occur and pass unnoticed for a long time.

Second, although it prevents demand defects from increasing, it does not repair them. This rules it out for time repair, which is largely about repairing demand defects, but it may not matter for resource repair. Resource repair cannot reduce the number of demand defects unless it moves meets: merely assigning resources reduces the domains of demand tixels, which cannot reduce demand defects. Even moving meets is unlikely to reduce demand defects during resource repair, since many of those moves will have been tried previously, during time repair.

*Detach correlated resource monitors.* Instead of detaching demand monitors, detach the resource monitors they correlate with. Each kind of resource monitor is considered below. The main danger here is *inexactness*: if some detached resource monitor is not modelled exactly by the demand monitors that replace it, then some defects go undetected and unrepaired.

*Detach all avoid clashes monitors.* For every avoid clashes defect there is an ordinary demand defect. The only inexactness is that avoid clashes monitors may have any weights, whereas demand monitors have equal weight, usually 1 (hard). But avoid clashes constraints usually have weight 1 (hard) or more, so this does not seem to be a problem in practice, given that, as the specification says, hard constraint violations should be few in good solutions.

*Detach avoid unavailable times monitors that give rise to workload demand monitors.* These are monitors with weight at least 1 (hard). The modelling here is exact apart from any difference in hard weight, so again there is no problem in practice.

*Detach limit busy times monitors that give rise to workload demand monitors.* These are monitors with weight at least 1 (hard) which satisfy the subset tree condition (Section 7.4.2). Apart from the possible difference in hard weight, this is exact except for one problem: limit busy times constraints can impose both a lower and an upper limit on resource availability in some set of times, and workload demand monitors do not model lower limits at all.

This can be fixed by (conceptually) breaking each limit busy times monitor into two, an

underload monitor and an overload monitor, and detaching 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 6.6.5) 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.

*Detach 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 (Section 7.4.1). This problem can be solved as follows.

Consider a resource with a hard limit workload monitor and some hard 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 6.6.6) 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 to lie on successful chains anyway, 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.

To summarize all this: there is some choice during resource repair, but detaching resource monitors (with ceilings) always works, and it is the only method that works during time repair.

#### 13.7.4. Primary grouping and detaching

To install and remove the primary groupings used by ejection chains, call

```
void KheEjectionChainPrepareMonitors(KHE_SOLN soln,
    KHE_OPTIONS options);
void KheEjectionChainUnPrepareMonitors(KHE_SOLN soln);
```

This includes detaching some resource monitors, as in the plan evolved in Section 13.7.3. This section explains exactly what `KheEjectionChainPrepareMonitors` does.

`KheEjectionChainPrepareMonitors` partitions 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 attached 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.* Like split events monitors, these are usually handled structurally, so it does nothing with them. They usually have provably zero fixed cost, so are already detached.

*Order events monitors.* For each order events monitor, it finds the sequence of classes that hold the two events it monitors. It groups attached 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.

Next, `KheEjectionChainPrepareMonitors` 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 primary 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.

Students who follow the same curriculum have the same timetable. So for each resource

type `rt` such that a call to `KheResourceTypeDemandIsAllPreassigned(rt)` (Section 3.5.1) shows that its resources are all preassigned, `KheEjectionChainPrepareMonitors` groups the resource monitors of `rt`'s resources 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.* It 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.

`KheEjectionChainPrepareMonitors` also treats some resource monitors according to the plan from Section 13.7.3, whether they are grouped or not:

- It detaches all attached avoid clashes monitors.
- For each attached avoid unavailable times monitor `m` for which a workload demand monitor with originating monitor `m` is present (all hard ones, usually), it detaches `m`.
- For each attached limit busy times monitor `m` for which a workload demand monitor with originating monitor `m` is present (all hard ones satisfying the subset tree condition of Section 7.4.2, usually), if `m`'s lower limit is 0 it detaches `m`, otherwise it sets `m`'s ceiling attribute to its maximum attribute, by calling `KheLimitBusyTimesMonitorSetCeiling`.
- For each attached limit workload monitor `m` for which a workload demand monitor with originating monitor `m` is present (all hard ones, usually), it sets `m`'s ceiling attribute to the cycle length minus the number of workload demand monitors for `m`'s resource (regardless of origin), or 0 if this is negative, by calling `KheLimitWorkloadMonitorSetCeiling`.

Section 13.7.3 has the reasoning.

Finally, `KheEjectionChainPrepareMonitors` groups demand monitors as follows. If a limit monitor containing these monitors is wanted, a separate call is needed (Section 13.7.5).

*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_MEET_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.

*Workload demand monitors.* These remain ungrouped. As explained in Section 13.7.3, workload demand defects appear only indirectly, as competitors of ordinary demand defects.

### 13.7.5. Secondary groupings

Section 9.8 introduces the concept of secondary groupings. The three ejection chain functions need secondary groupings built on primary groupings for their start group monitors (but not their continue group monitors, since they use the solution object for that), and other secondary groupings for their limit monitors. These are the subject of this section.

`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 primary 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 primary 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. Primary groupings are irrelevant to limit monitors, so these last two functions take no account of them.

### 13.7.6. Augment functions

The augment functions passed to the ejector are private to KHE. This section explains what they do in detail. It may be somewhat out of date.

The augment functions consult several options. The `es_repair_times`, `es_limit_node`, and `es_repair_resources` options are particularly important because they limit the scope of repairs. They cannot be set by the user—or rather, they can, but that would be futile because they are reset within the main functions. Any repair which assigns or moves a meet consults `es_repair_times`, and only proceeds if it is true. It tries moving each ancestor of the meet, since that will also move the original meet; but if `es_limit_node` is non-NULL, it omits moves of meets lying within nodes which are not proper descendants of `es_limit_node`. Any repair which assigns or moves a task consults `es_repair_resources`, and only proceeds if it is true.

Here is the full list of repair operations executed by KHE's augment functions.

*Node swaps*, which use `KheNodeMeetSwap` (Section 10.2.1) to swap the assignments of the meets of two nodes. If the `es_nodes_before_meets` option is true, then if node swaps are tried at all, they are tried before (rather than after) meet moves.

*Basic and ejecting meet assignments and moves and Kempe meet moves*, which move meets (Section 10.2.2). Where it is stated that a Kempe meet move is tried, it is in fact tried only when the `es_kempe_moves` option (Section 13.7) is true, or it is `large_layers` and the meet to be moved lies in at least one layer whose duration is at least 80% of the duration of the cycle. Where it is stated that an ejecting meet assignment or move is tried, a basic meet assignment or move is tried instead when the `es_no_ejecting_moves` option is true.

*Fuzzy meet moves*, which move meets in a more elaborate way (Section 10.7.4). These are not mentioned below, but they are tried after Kempe and ejecting meet moves, although only when the `es_fuzzy_moves` option is true and the current depth is 1.

*Split moves*, which split a meet into two and Kempe-move one of the fragments, and *merge*

*moves*, which Kempe-move one meet to adjacent to another and merge the two fragments. As well as being used to repair split defects, split 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 `es_split_moves` option is `true` and the current depth is 1. These Kempe meet moves are not influenced by the `es_kempe_moves` option.

*Ejecting task assignments and moves*, which assign or move a task to a given resource and then unassign any clashing tasks (Section 12.3).

*Ejecting task-set moves*, which use ejecting task moves to move a set of tasks to a common resource, succeeding only if all of the moves that change anything succeed.

*Meet-and-task moves*, which Kempe-move a meet at the same time as moving one of its tasks, succeeding only if both moves succeed.

Each repair is enclosed in calls to `KheEjectorRepairBegin` and `KheEjectorRepairEnd` as usual. In the more complex cases, such as the last two on the list above, the `success` argument of `KheEjectorRepairEnd` is set to `true` only if all parts of the repair succeed. Some of the more complex repairs are tried only when the current depth is 1, that is, when the defect being repaired is truly present, not introduced by some other repair.

Some alternative repairs are naturally tried one after another. The ejecting task moves of a given task to each resource in its domain is one example. Here are three less obvious, but nevertheless very useful sequences of alternative repairs.

A *Kempe/ejecting meet move* is a sequence 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 and split moves may follow the Kempe and ejecting meet moves, as explained above.

A *resource underload repair* for resource  $r$  and time group  $g$  is a sequence of alternative repairs which aim to increase the number of times  $r$  is busy within  $g$ . Unless  $g$  is the whole cycle, for each task assigned  $r$  whose *overlap* (the number of times it is running within  $g$ ) is less than its duration, it tries all ejecting meet moves of the task's meet which increase its overlap. After that it tries an ejection tree repair like the one described below for cluster busy times defects, which aims to empty out the entire time group—a quite different way to remove the defect.

A *resource overload repair* for resource  $r$  and time group  $g$  is a sequence of alternative repairs which aims to decrease the number of times  $r$  is busy within  $g$ . First, for each task assigned but not preassigned  $r$  whose overlap is non-zero, it tries all ejecting task moves of the task to its domain's resources. Then, for each task assigned (including preassigned)  $r$  whose overlap is non-zero, it tries all ejecting meet moves of the task's meet which decrease the overlap.

Wherever possible, sequences of alternative repairs change the starting point of the traversal of the alternatives on each call. For example, when trying alternative resources, the code is

```

for( i = 0; i < KheResourceGroupResourceCount(rg); i++ )
{
    index = (KheEjectorCurrAugmentCount(ej) + i) %
        KheResourceGroupResourceCount(rg);
    r = KheResourceGroupResource(rg, index);
    ... try a repair using r ...
}

```

The first resource tried depends on the number of augments so far, an essentially random number. This simple idea significantly decreases final cost and run time.

Following is a description of what each augment function does when given a non-group monitor with non-zero cost to repair. When given a group monitor with non-zero cost, since the elements of a group all monitor the same thing in reality, the augment function takes any individual defect from the group and repairs that defect.

*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 bring 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.

*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. It will not be difficult to find suitable meet moves in the future.

*Assign resource defects.* For each monitored unassigned task, all ejecting assignments of the task to resources in its domain are tried. Then if the `es_repair_times` option permits, all combinations of a Kempe meet move of the enclosing meet and an ejecting assignment of the task to resources in its domain are tried.

*Prefer resources defects.* For each monitored task assigned an unpreferred resource, all ejecting moves of the task to preferred resources are tried.

*Avoid split assignments defects.* For each resource participating in a split assignment there is one repair: all involved tasks assigned that resource are unassigned, all involved tasks' domains are restricted to the other participating resources, and a set of ejection chains is tried, each of

which attempts to reassign one of the unassigned tasks. The repair succeeds only if all these chains succeed, making an ejection tree, not a chain. This is expensive and unlikely to work, so it is only tried when the defect is a main loop defect or only one task needs to be unassigned.

*Avoid clashes defects.* Avoid clashes monitors are detached during ejection chain repair, since their work is done by demand monitors. So there are no avoid clashes defects.

*Avoid unavailable times defects.* A resource overload repair (see above) for the monitored resource and the unavailable times is tried.

*Limit idle times defects.* For each task assigned the monitored resource at the start or end of a ‘day’ (actually, a time group being monitored), each Kempe/ejecting meet move of that task’s meet such that the initial meet move reduces the number of idle times for that resource is tried. Calculating this condition is not trivial, but the augment function does it exactly. Task moves could help to repair limit idle times defects for unpreassigned resources, but in current data sets the resources involved are usually preassigned, so task moves are not currently being tried.

After the repairs just given are tried, if the repair has depth 1 (if the defect was not created by a previous repair on the current chain), a complex repair is tried which eliminates all idle times for one resource on one day. Take the meets assigned that resource on that day. A retimetabling of those meets on that day with no clashes and no idle times is defined by a starting time for the first meet and a permutation of the meets (their chronological order in the assignment). If there are  $k$  meets and  $s$  starting times that don’t put the last meet off the end of the day, then there are  $s \cdot k!$  retimetablings in total. In practice this is a moderate number. For safety, only a limited number of retimetablings is tried, by switching to a single permutation at each new node after a fixed limit (currently 1000) is reached.

*Cluster busy times defects.* If the resource is busy in too few monitored time groups, all ejecting meet moves are tried which move a meet which is not the only meet in a monitored time group (either because every monitored time group it overlaps with overlaps with at least one other meet, or because it does not overlap with any monitored time groups) to a monitored time group in which it is. If the resource is busy in too many monitored time groups, then for each monitored time group  $\tau_g$  containing at least one meet, if the depth is 1 or  $\tau_g$  contains exactly one meet, all the meets in  $\tau_g$  are unassigned, and  $\tau_g$  and all monitored time groups containing no meets are removed from the domains of all meets assigned the resource. This is an ejection tree repair if more than one meet is unassigned: all of the unassigned meets must be reassigned for success. The `on_success_fn` parameter of `KheEjectorRepairEndLong` is used to ensure that the domains are restored on success as well as on failure.

*Limit busy times defects.* For each set of times when the resource is underloaded (resp. overloaded), a resource underload (resp. overload) repair of the resource at those times is tried.

*Limit workload defects.* If the resource is overloaded, a resource overload repair is tried, taking the set of times to be the entire cycle. There is currently no repair for underloads.

*Ordinary and workload demand defects.* If the defect has a workload demand monitor competitor (possibly itself, although workload demand monitors rarely fail to match), a resource overload repair is tried for the workload demand monitor’s resource and the domain of its originating monitor. If the defect has no workload demand defect competitor, all ejecting meet moves of competitor meets to anywhere different are tried; but meets within vizier nodes are not moved in this case, since that would only shift the problem to a different time.

### 13.7.7. Repair operations for nurse rostering

*This section is obsolete. It will be rewritten eventually. Meanwhile, see my KHE18 nurse rostering solver paper.*

Apart from the absence of time assignment, nurse rostering is characterized by constraints that relate what a resource does on one day to what it does on adjacent days. Because of them, it makes sense to use repairs which move small sets of tasks, from adjacent days, as a block from one resource to another. Many authors of nurse rostering solvers have come to this conclusion.

Suppose that a defect arises that might be repaired by an augment that moves `task` from `r1` to `r2`, as carried out by `KheTaskMoveAugment`. Either of `r1` and `r2`, but not both, may be `NULL`. The idea offered here is that in the nurse rostering case, `KheTaskMoveAugment` should reinterpret a request to move `task` as a request to move several tasks from adjacent days, including `task`.

Which tasks should be chosen? Initially, they should all be assigned to whatever `task` is assigned to initially (possibly `NULL`), otherwise the rationale for this kind of move is lost. The number of adjacent days to include tasks from is harder to be sure about, except that, as we search in each direction from `task`'s day, we are obliged to stop after we reach the first or last day, and if there is no task assigned what `task` is assigned to initially. Furthermore, if `r2` is already busy on some day, it is not likely to be useful to give it another task on that day. So the search should stop at the first case of that. So we have a different set of moves to make for each `r2`.

Our plan is to move just `task` to begin with, then `task` plus one day, then `task` plus 2 days, and so on up to `k` days, where `k` is a parameter that we can choose, by means of an option perhaps. We will try moving all sets of from 1 to `k` days that include `task`. Clearly, `k` must be small, say 4 at most. Only `task` will be marked visited and checked for being visited.

The set of possible repairs here has two dimensions, one indexed by sets of days to move, the other by `r2`. We could iterate in any order that maximizes efficiency. It would be good, for example, to build task sets as required, and remember them from one resource to the next.

We also need to be able to handle 'negative moves', where we have a time group rather than a task and the idea is to move something out of that time group or into it. This is done now by `KheNurseOverload`, but not very effectively.

# 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 and arrays

*Note – the Ha library has been flown in from another project of the author’s, called Howard. Its header file is `howard_a.h`.*

This section describes Howard’s Ha library, which provides arenas and extensible arrays.

### 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. Starting from Version 2.1, all memory allocated by KHE is arena memory.

For creating and deleting arenas, the operations are

```
HA_ARENA HaArenaMake(bool large);
```

`HaArenaMake` creates an arena. Parameter `large` does nothing except get itself stored in the arena, where it can be retrieved by calling

```
bool HaArenaLarge(HA_ARENA a);
```

It is a hint that the arena may become (very) large. Function

```
void HaArenaDelete(HA_ARENA a);
```

deletes `a`, freeing all its arena memory. Also,

```
void HaArenaClear(HA_ARENA a);
```

freees most of `a`’s memory, returning it to its state immediately after `HaArenaMake`.

The author has had some unpleasant experiences with the `malloc` memory allocator supplied with his Linux system on large runs with multiple threads, and he has concluded that recycling memory via `malloc` and `free` is best avoided. Accordingly, function

```
HaArenaRecycle(HA_ARENA a);
```

is provided, which is similar to `HaArenaClear` in that it tells `a` that all of its objects except `a` itself are no longer wanted, but different in that it does not free `a`’s memory. Instead, it retains it and uses it for subsequent allocations.

`HaArenaRecycle` is usually used in conjunction with a free list of arenas. When an arena is needed, it is taken from this list if possible, otherwise it is created by `HaArenaMake`. When an

arena *a* is no longer needed, `HaArenaRecycle(a)` is called and *a* is added to the free list. In this way, memory is efficiently recycled without calling `free`. Functions `KheArenaSetArenaBegin` and `KheArenaSetArenaEnd` (Appendix A.1.2) do this.

### 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 and initialized to zero. 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.1.5.

Arenas obtain their memory from `malloc` (actually `calloc`). As long as `calloc` continues to supply memory, an arena will continue to supply memory to the user. If a request for memory from `calloc` fails, then `HaArenaMake` and `HaAlloc` return `NULL`, and `HaMake` sets *res* to `NULL`.

The memory pointed to by a variable *a* of type `HA_ARENA` is arena memory from arena *a*. This memory is freed along with the rest of the arena memory when the arena is deleted.

`HaAlloc` initializes the memory to zero for two reasons. First, an uninitialized object field can cause a program to behave differently each time it runs. If all object memory is initialized to zero, an uninitialized field is still a bug, but at least the program behaves the same each time it runs. Second, `HaArrayContains` (Section A.1.3) compares generic objects using `memcmp`. When these objects are structs with gaps in them, this will only be correct if the gaps are equal.

The cost of calling `HaArenaMake` and `HaArenaDelete` is small enough to allow many small arenas to come and go. A call to `HaArenaMake` generates one call to `calloc` requesting 14 words of memory, a few internal function calls which will certainly be inlined, and about 15 initializing assignments. A call to `HaArenaDelete` generates one call to `free` and three assignments when the arena is empty, growing logarithmically (i.e. negligibly slowly) as the amount of memory allocated in the arena increases. Section A.1.5 has more detail.

### A.1.2. Arena sets

An *arena set* is a set of arenas. It is a convenient place to store recycled arenas while they are waiting for new uses. To create a new, empty arena set, call

```
HA_ARENA_SET HaArenaSetMake(HA_ARENA a);
```

The arena set (but not its arenas) lies in arena *a* and will be freed when *a* is freed.

In practice, what is mainly wanted is the ability to add an arena and extract one:

```
void HaArenaSetAddArena(HA_ARENA_SET as, HA_ARENA arena);
HA_ARENA HaArenaSetLastAndDelete(HA_ARENA_SET as);
```

`HaArenaSetAddArena` adds *arena* to *as*, and `HaArenaSetLastAndDelete` deletes and returns the last arena from *as*. Also useful are

```
int HaArenaSetArenaCount(HA_ARENA_SET as);
HA_ARENA HaArenaSetArena(HA_ARENA_SET as, int i);
```

which return the number of arenas in `as`, and the `i`th arena, counting from 0 as usual.

Typically, there is one arena set per thread. When a thread terminates, the arenas of its arena set need to be passed on to the arena set of the parent thread. For this there is

```
void HaArenaSetMerge(HA_ARENA_SET dest_as, HA_ARENA_SET src_as);
```

It moves the arenas of `src_as` to `dest_as`, leaving `src_as` empty.

Since arena sets are mainly for recycling arenas, two convenience functions are offered:

```
HA_ARENA HaArenaSetArenaBegin(HA_ARENA_SET as, bool large);
```

```
void HaArenaSetArenaEnd(HA_ARENA_SET as, HA_ARENA a);
```

If `as` is non-NULL, `HaArenaSetArenaBegin` returns a fresh arena, extracted from `as` if there is one with the given `large` attribute, or else newly created, and `HaArenaSetArenaEnd` calls `HaArenaRecycle(a)` then adds `a` to `as`. If `as` is NULL, the two functions just call `HaArenaMake` and `HaArenaDelete`.

For completeness, a few other operations are provided:

```
void HaArenaSetClear(HA_ARENA_SET as);
```

```
void HaArenaSetDropFromEnd(HA_ARENA_SET as, int n);
```

```
void HaArenaSetDeleteArena(HA_ARENA_SET as, HA_ARENA arena);
```

```
bool HaArenaSetContainsArena(HA_ARENA_SET as, HA_ARENA arena, int *pos);
```

```
void HaArenaSetDebug(HA_ARENA_SET as, int verbosity, int indent,  
FILE *fp);
```

These clear `as` back to the empty set, remove the last `n` arenas from `as`, delete `arena` (which must be present) from `as`, return `true` when `arena` lies in `as`, and produce 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*  $\geq 0$  and that *src\_i* and *dest\_i* are at least 0 and at most `HaArrayCount(a) - len`. It is used by `HaArrayAdd` above and `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 delete the *i*th 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. Version string

Macro `HA_HOWARD_VERSION` is a wide character string defining the current version of Howard. For example, its value was

```
L"Howard Version 1.0 (June 2011)"
```

at the time of writing.

#### A.1.5. 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.

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 width is at least the width of a pointer. `HaArenaMake` checks this condition and aborts if it does not hold, since the implementation depends on it.

*Resizable arena memory* is arena memory returned by functions `HaResizableAlloc` and `HaResizableReAlloc`, defined below. Given a pointer to resizable arena memory, the allocator can deduce what arena it is from and what its size is (in bytes):

```
HA_ARENA HaResizableArena(void *resizable);
size_t HaResizableSize(void *resizable);
```

The value of `HaResizableSize` may be larger than the size requested when `resizable` was allocated. The functions for allocating and freeing resizable arena memory are

```
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.

Like ordinary arena memory, resizable arena memory is aligned suitably for any kind of data. Resizable arena memory is not initialized to zero, however.

The remainder of this section describes the implementation of the arena memory allocator, including how it tries to avoid various problems that memory allocators are prone to.

An arena obtains its memory from `calloc`. A piece of memory given to an arena by `calloc` 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.

An arena cannot satisfy all the block requests it receives out of one chunk. So it calls `calloc` more than once, and maintains a linked list of all the chunks it receives. The arena object contains a pointer to the most recently obtained chunk; this chunk's first  $A$  bytes hold a pointer to the next most recently obtained chunk, and so on. The second  $A$  bytes in each chunk hold the total number of words in the chunk, and the total number of words not yet allocated. Thus the memory overhead per chunk is `calloc`'s overhead, plus  $2A$  bytes to hold the singly linked

list of chunks and the two integers.

The linked list serves two purposes. First, when the arena is deleted, its memory is freed by traversing the list and freeing each chunk. The arena object itself, and 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 a block is required, 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, for two reasons. First, 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, as it may do if there is one arena per function. Second, a request for a block whose size is larger than the chunk size could not be satisfied.

Accordingly, the chunks obtained from `calloc` vary in size, as follows. The arena contains a *normal size* field. Whenever a new chunk is needed, the normal size is first increased. Then, if the request is for less memory than the normal size, a chunk of the normal size is obtained from `calloc` and placed first in the chunk list, and the request is satisfied from within that chunk. If the request is for as much as or more memory than the normal size, a chunk of the requested size is obtained from `calloc` and placed second in the chunk list, and the request is satisfied from this chunk, which it fills completely.

The normal sizes are  $(2^n - 2)A$ , for  $n = 4$ ,  $n = 5$ , and so on. This ensures that memory is not wasted on unnecessarily large chunks. When `calloc`'s overhead is added, chunk sizes will probably be powers of 2, and as  $n$  increases, a thread-aware `calloc` will allocate chunks on cache boundaries. Even if `calloc` is not thread-aware, each arena should be accessed by only one thread, which alone should give reasonable cache behaviour.

When an arena is recycled, its chunks are moved to a different singly linked list, the *recycled chunk list*, and their memory is reset to 0. The chunk holding the arena object itself is treated somewhat differently: it stays on the main chunk list, and only the part of its memory that does not hold the arena object is reset to 0. When allocating memory, if the current chunk is too full, then before making a new chunk, the recycled chunk list is examined. If it is non-empty, its first chunk becomes the current chunk and the whole process is restarted. Otherwise a new chunk is obtained from `calloc` as usual.

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  $R_0 A$  (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  $R_1 A$ ; 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 (type `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 later, by calling either `HaArrayFree(ac)` or, equivalently, `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 `vwprintf` 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 `wcslon`; 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.

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 returned, and freed by

```
void HpTableInit(TABLE_X table, HP_HASH_FN key_hash_fn,
    HP_EQUAL_FN key_equal_fn, HA_ARENA a);
HP_HASH_FN HpTableKeyHashFn(TABLE_X table);
HP_EQUAL_FN HpTableKeyEqualFn(TABLE_X table);
HA_ARENA HpTableArena(TABLE_X table);
void HpTableFree(TABLE_X table);
```

where `HP_HASH_FN` and `HP_EQUAL_FN` are the types of the two user-supplied functions. 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)

```

Finally,

```

int HpTableSize(TABLE_X table);

```

returns the size of the table.

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, HA_ARENA a);
HP_HASH_FN HpGroupKeyHashFn(HP_GROUP group);
HP_EQUAL_FN HpGroupKeyEqualFn(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. 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.4) 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.4. 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.

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 *i*th 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.5. 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.

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.6. 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.6.1. 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.6.2. 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.6.3. 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.6.1 may be used to traverse `*res`. Otherwise, `*ke` is set to an error object (Appendix A.6.2) 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 requires 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 descendants. 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.6.4. 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 (`&lt;` 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 `KheABCCount` 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:

<code>KheMeetKernelSetBack</code>	<code>KheTaskKernelSetBack</code>
<code>KheMeetKernelAdd</code>	<code>KheTaskKernelAdd</code>
<code>KheMeetKernelDelete</code>	<code>KheTaskKernelDelete</code>
<code>KheMeetKernelSplit</code>	<code>KheTaskKernelSplit</code>
<code>KheMeetKernelMerge</code>	<code>KheTaskKernelMerge</code>
<code>KheMeetKernelMove</code>	<code>KheTaskKernelMove</code>
<code>KheMeetKernelAssignFix</code>	<code>KheTaskKernelAssignFix</code>
<code>KheMeetKernelAssignUnFix</code>	<code>KheTaskKernelAssignUnFix</code>
<code>KheMeetKernelAddMeetBound</code>	<code>KheTaskKernelAddTaskBound</code>
<code>KheMeetKernelDeleteMeetBound</code>	<code>KheTaskKernelDeleteTaskBound</code>
<code>KheMeetKernelSetAutoDomain</code>	
<code>KheMeetBoundKernelAdd</code>	<code>KheTaskBoundKernelAdd</code>
<code>KheMeetBoundKernelDelete</code>	<code>KheTaskBoundKernelDelete</code>
<code>KheMeetBoundKernelAddTimeGroup</code>	
<code>KheMeetBoundKernelDeleteTimeGroup</code>	<code>KheNodeKernelSetBack</code>
	<code>KheNodeKernelAdd</code>
<code>KheLayerKernelSetBack</code>	<code>KheNodeKernelDelete</code>
<code>KheLayerKernelAdd</code>	<code>KheNodeKernelAddParent</code>
<code>KheLayerKernelDelete</code>	<code>KheNodeKernelDeleteParent</code>
<code>KheLayerKernelAddChildNode</code>	<code>KheNodeKernelSwapChildNodesAndLayers</code>
<code>KheLayerKernelDeleteChildNode</code>	<code>KheNodeKernelAddMeet</code>
<code>KheLayerKernelAddResource</code>	<code>KheNodeKernelDeleteMeet</code>
<code>KheLayerKernelDeleteResource</code>	
	<code>KheZoneKernelSetBack</code>
	<code>KheZoneKernelAdd</code>
	<code>KheZoneKernelDelete</code>
	<code>KheZoneKernelAddMeetOffset</code>
	<code>KheZoneKernelDeleteMeetOffset</code>

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.

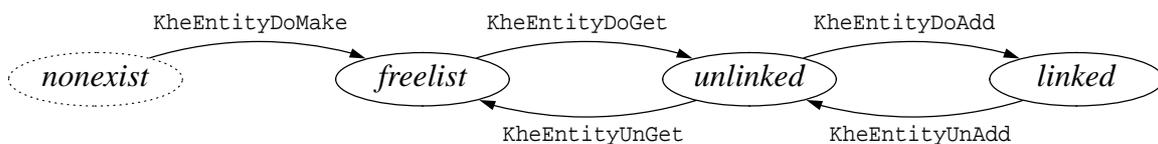
A redo must be identical to the original operation, because both can be inverted by calling `KheEntityKernelOpUndo` and removing one record from the solution path. So there are no `KheEntityKernelOpRedo` functions; `KheEntityKernelOp` functions are called instead.

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.

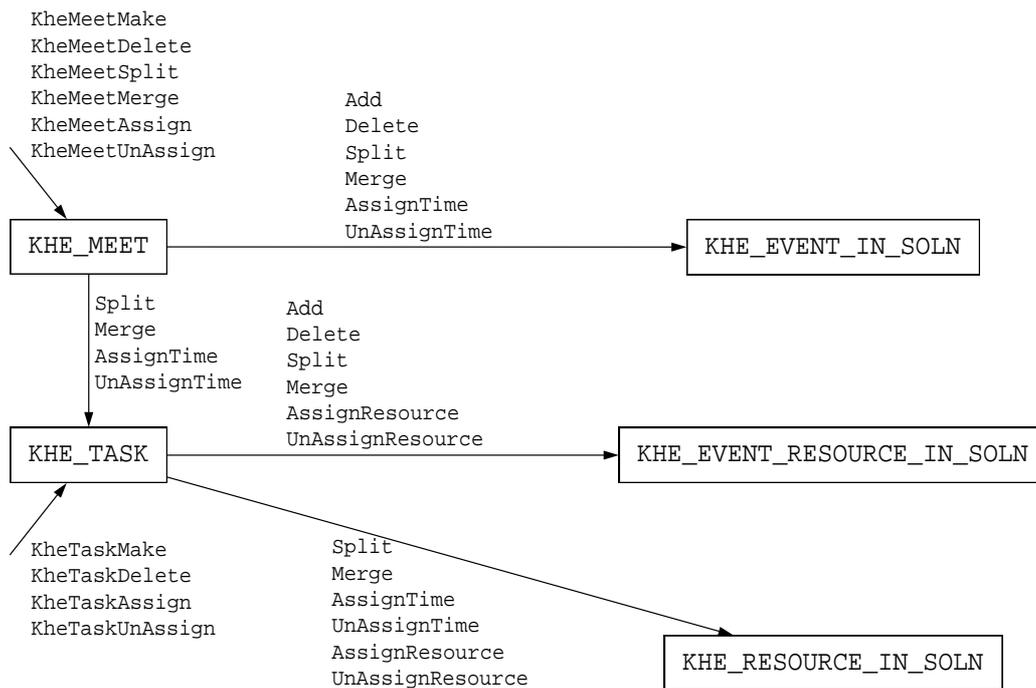
### B.4. Monitor updating

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

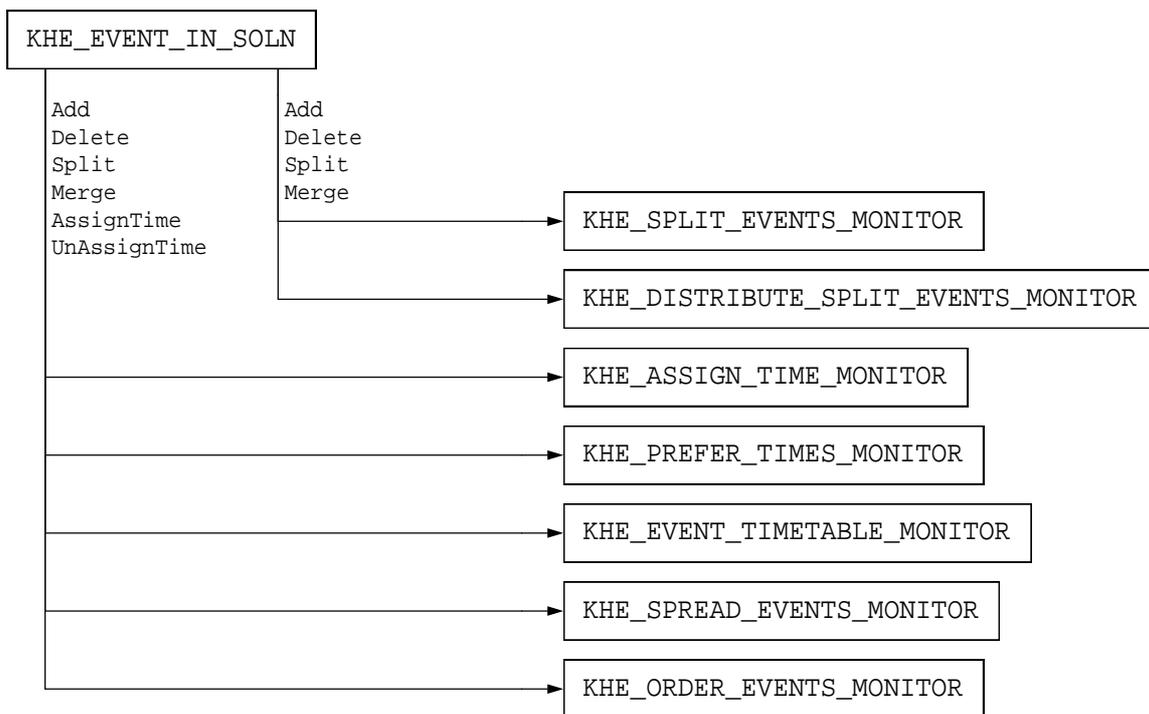
Six originate in KHE\_MEET objects, four in KHE\_TASK objects. From there their impulses flow to objects of three private types:



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.

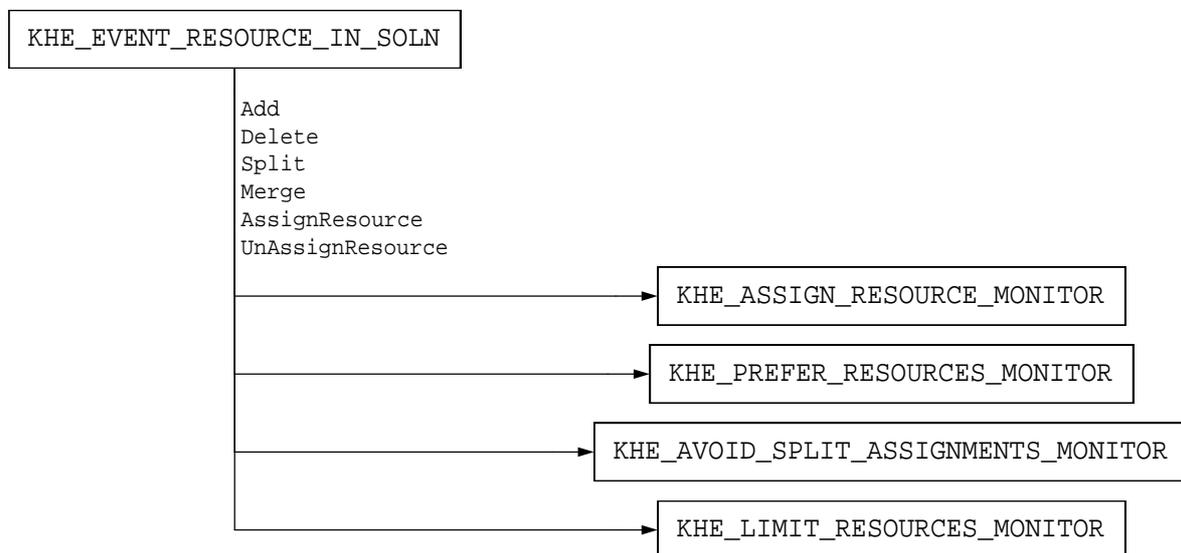
Event timetables are used by link events constraints, which need to know the times when the event’s meets are running, ignoring clashes, which is just what timetables offer.

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

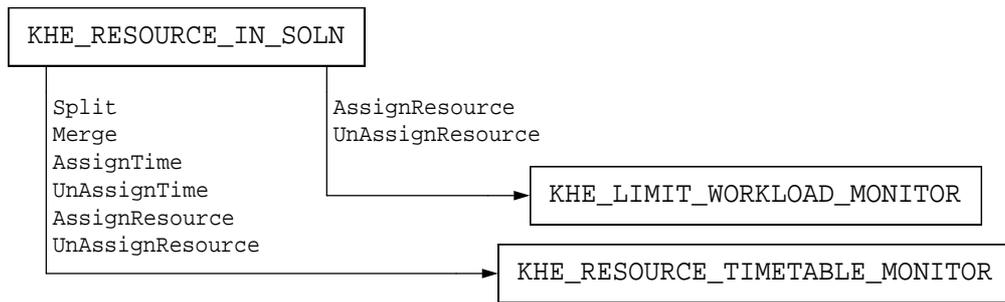


None of these monitors cares about time assignments and unassignments. Assign resource monitors and prefer resources monitors are very simple, reporting cost depending on whether the tasks passed to them are assigned or not.

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.

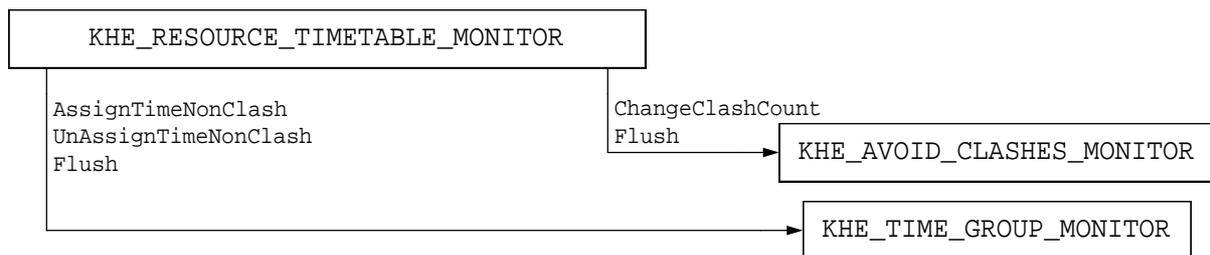
The connections leading out of `KHE_RESOURCE_IN_SOLN` are



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

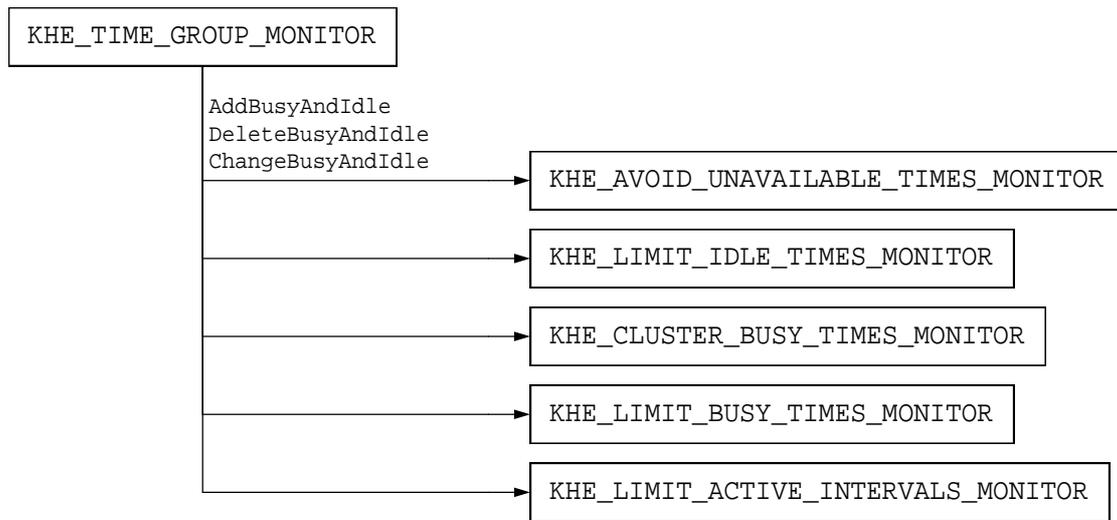


An avoid clashes monitor is notified whenever the number of meets at any one time increases to more than 1 or decreases from more than 1 (operation `ChangeClashCount` above). It uses these notifications to maintain its deviation. It updates the solution when a `Flush` is received from the timetable at 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`:



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

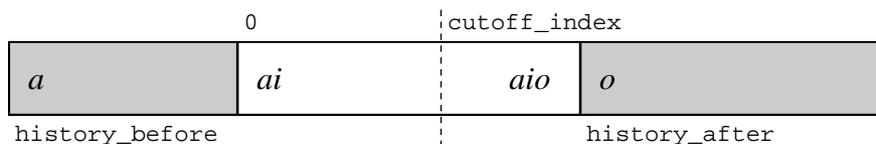
Monitors can be quite lengthy to implement, given the many state-changing operations they need to accept. However they are usually straightforward, once one understands the basic structure of taking a state change in, producing a new cost, and reporting it if it changed.

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.

# 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).