A User’s Guide to the NRConv Nurse Rostering Converter

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

Version 2.6 (March 2021)

Contents

Chapter 1. Introduction

This document describes NRConv, a program for converting instances of nurse rostering problems from various existing formats into a common XML format called XESTT.

In NRConv's view of the world, there are three kinds of models. They all model instances of nurse rostering problems, and solutions to those instances, but they do it in different ways.

A *source model* is one of the models (concretely, file formats) that NRConv converts. At present NRConv can convert four source models: the Curtois 'original instances' model, the first international nurse rostering competition model, the second international nurse rostering competition model, and the Curtois-Qu 2014 model. However, NRConv is designed to minimize the work needed to add new source models.

The *intermediate model*, also called the *NRC model* after the software platform that implements it, represents the concepts underlying source models, including metadata, days, shifts, cover, patterns, and so on. If something in some source model is not in the intermediate model, then either the intermediate model needs to be extended, or else the source model is beyond the scope of NRConv. The intermediate model also includes its own versions of the XESTT concepts of archive (a set of instances and solution groups) and solution group (a set of solutions). There is no file format for the intermediate model; its values are held only in memory.

Finally, there is the *target model*, XESTT. Its main concepts are archives, instances, times, resources, events, constraints, solution groups, and solutions. It lies in memory, in objects defined by the KHE platform which implements XESTT, and it also has a file format version. The user of NRConv does not need to be a KHE expert, because NRConv uses it only behind the scenes.

There are two parts to NRConv, a *platform* called NRC and the *converters*. NRC defines many types and functions, culminating in type `NRC_ARCHIVE`, representing a set of instances and a set of solution groups in the intermediate model, and function `NrcArchiveWrite`, which converts an `NRC_ARCHIVE` object into a target model archive and writes it by calling `KheArchiveWrite`.

Each converter is a function which reads instance and solution files in one source model and converts what it reads into intermediate model objects. Since the intermediate model understands all the source model concepts, this conversion should be straightforward, a matter of reading the source model file and generating calls to the NRC platform. After this is done, the converter completes the conversion to the target model by calling `NrcArchiveWrite`.

NRConv is written in C, and is packaged with the author's KHE distribution, which is a gzipped tar file. To install it you need to get that file, unzip it, untar it, modify `makefile` slightly to say where you want the final `nrconv` binary to be copied to, and run `make`. After compiling, execute the `nrconv` binary with no command line options to get a comprehensive usage message.

Part A

The NRC Nurse Rostering Model

Chapter 2. NRC Archives and Solution Groups

The subject of this part is the NRC nurse rostering model. As explained in the introduction, the NRC model is an intermediate model, lying between the source models, which are existing nurse rostering models such as the First International Timetabling Competition model, and the target model, XESTT. By calling the functions defined in this part, a converter can convert instances and solutions in its model into the intermediate model; and then one call on `NrcArchiveWrite` converts that into XESTT and writes it as an XESTT archive.

This chapter describes types `NRC_ARCHIVE` and `NRC_SOLN_GROUP`, representing archives (sets of instances and solution groups) and solution groups (sets of solutions) in the intermediate model. Nurse rostering data formats do not seem to have features for grouping instances and solutions, beyond, say, the use of a zip file containing several instances. So these features of NRC are copied directly from the corresponding XESTT features.

2.1. Archives

An archive is 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

```
NRC_ARCHIVE NrcArchiveMake(char *id, HA_ARENA_SET as);
```

Here `id` is an identifier for the archive, and `as` is an arena set, used to gain efficient access to heap memory. You can safely pass `NULL` for arena set; to find out how to pass a non-`NULL` value, and why you might want to do that, read the KHE User's Guide. Function

```
char *NrcArchiveId(NRC_ARCHIVE archive);
```

returns the Id attribute.

Archive metadata may be set and retrieved by calling

```
void NrcArchiveSetMetaData(NRC_ARCHIVE archive, char *name,
  char *contributor, char *date, char *description, char *remarks);
void NrcArchiveMetaData(NRC_ARCHIVE archive, char **name,
  char **contributor, char **date, char **description, char **remarks);
```

where `remarks`, being optional, may be `NULL`.

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 NrcArchiveAddInstance(NRC_ARCHIVE archive, NRC_INSTANCE ins);
```

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

```
void NrcArchiveDeleteInstance(NRC_ARCHIVE archive, NRC_INSTANCE ins);
```

`NrcArchiveDeleteInstance` 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 NrcArchiveInstanceCount(NRC_ARCHIVE archive);
NRC_INSTANCE NrcArchiveInstance(NRC_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 NrcArchiveRetrieveInstance(NRC_ARCHIVE archive, char *id,
  NRC_INSTANCE *ins);
```

If `archive` contains an instance with the given `id`, this function sets `ins` to that instance and returns `true`; otherwise it leaves `*ins` untouched and returns `false`. In the same way,

```
int NrcArchiveSolnGroupCount(NRC_ARCHIVE archive);
NRC_SOLN_GROUP NrcArchiveSolnGroup(NRC_ARCHIVE archive, int i);
bool NrcArchiveRetrieveSolnGroup(NRC_ARCHIVE archive, char *id,
  NRC_SOLN_GROUP *soln_group);
```

visit the solution groups of an archive, and retrieve 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 NrcSolnGroupMake(NRC_ARCHIVE archive, char *id,
  NRC_SOLN_GROUP *soln_group);
```

Parameter `archive` is compulsory. The solution group will be added to the archive. Parameter `id` is an identifier for the solution group. If the operation is successful, then `true` is returned with `*soln_group` set to the new solution group; if it is unsuccessful (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 retrieve these attributes, call

```
NRC_ARCHIVE NrcSolnGroupArchive(NRC_SOLN_GROUP soln_group);
char *NrcSolnGroupId(NRC_SOLN_GROUP soln_group);
```

Solution group metadata may be set and retrieved by calling

```
void NrcSolnGroupSetMetaData(NRC_SOLN_GROUP soln_group,
  char *contributor, char *date, char *description, char *publication,
  char *remarks);
void NrcSolnGroupMetaData(NRC_SOLN_GROUP soln_group,
  char **contributor, char **date, char **description, char
**publication,
  char **remarks);
```

where `publication` and `remarks`, being optional, may be `NULL`.

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

```
void NrcSolnGroupAddSoln(NRC_SOLN_GROUP soln_group, NRC_SOLN soln);
void NrcSolnGroupDeleteSoln(NRC_SOLN_GROUP soln_group, NRC_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 NrcSolnGroupSolnCount(NRC_SOLN_GROUP soln_group);
NRC_SOLN NrcSolnGroupSoln(NRC_SOLN_GROUP soln_group, int i);
```

Solutions have no Ids, so there is no `NrcSolnGroupRetrieveSoln` function. When solution `i` is deleted, `NrcSolnGroupSolnCount` decreases by 1, solution `i+1` becomes solution `i`, and so on.

2.3. Writing archives

To convert an archive to XESTT format and write it to a file, call

```
void NrcArchiveWrite(NRC_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. The initial tag will be `<EmployeeScheduleArchive>`.

`NrcArchiveWrite` converts the NRC archive into a KHE archive and writes that archive using `KheArchiveWrite`.

At present, NRC does not check that the names of its entities are distinct. If two entities with conflicting names are given to NRC, they will be accepted at the time, but `NrcArchiveWrite` will exit with an error message when the KHE conversion reports the problem.

Chapter 3. NRC Instances and Solutions

An *instance* is a particular case of the nurse rostering problem, for a particular ward and a particular period of time. A *solution* is a solution to a particular instance, saying which workers to assign to which shifts. This chapter describes the `NRC_INSTANCE` and `NRC_SOLN` data types, which represents instances and solutions as defined by the NRC model.

3.1. Overview

NRC instances contain days, shifts, workers, and constraints (unlike KHE instances, which contain times, resources, events, and constraints). These data types are strongly interconnected, so we begin by giving informal definitions of most of them, as an overview.

`NRC_INSTANCE` represents one *instance* of the nurse rostering problem: one case of it, for a particular hospital ward and a particular interval of time.

`NRC_DAY` represents one *day*: not a generic day like 'Friday', but a particular day like 'Friday 18 November 2016', although its calendar date need not be known to the instance.

`NRC_DAY_SET` represents one *day-set*, which is a set (actually a sequence) of days. An example of a day-set is the *cycle*, containing all the days of the instance.

`NRC_DAY_SET_SET` represents one *day-set set*: a set (again, actually a sequence) of day-sets. An example of a day-set set is the one containing one day-set for each day of the week. Its first day-set contains all the Sundays, its second contains all the Mondays, and so on.

`NRC_SHIFT_TYPE` represents a *shift type*: a generic shift like the day shift or night shift.

`NRC_SHIFT_TYPE_SET` represents a *shift-type set*: a set of shift types.

`NRC_SHIFT` represents a *shift*, which is a particular shift such as the night shift on 18 November 2016. Each shift is characterised by a day plus a shift type.

`NRC_SHIFT_SET` represents a *shift-set*: a set of shifts. For example, the shifts whose day is 18 November 2016 form a shift-set, as do the shifts whose shift type is the night shift.

`NRC_SHIFT_SET_SET` represents a *shift-set set*, which is a set of shift-sets.

`NRC_WORKER` represents one *worker*, NRC's term for a nurse or employee.

`NRC_WORKER_SET` represents one *worker-set*: a set of workers. The leading examples of worker-sets are the sets of workers that share a particular contract, and the sets of workers that share a particular skill. Indeed, contracts and skills are represented in NRC by worker-sets.

`NRC_WORKER_SET_SET` represents one *worker-set set*: a set of worker-sets. For example, the set of all contracts is a worker-set set.

`NRC_WORKER_SET_TREE` is a tree of worker-sets, where children are subsets of their parents, and siblings are disjoint. It is used when analysing overlapping cover requirements.

`NRC_DEMAND` represents one *demand*, describing a demand for one worker made by a shift, including the penalty for when a worker is not assigned, and the default penalty to apply when a worker without the appropriate skill is assigned. One shift may have any number of demands.

`NRC_DEMAND_SET` represents one *demand-set*, a set of demands. One can add a demand-set to a shift, which is the same as adding each of its demands separately, only more convenient.

`NRC_POLARITY` is used by patterns and constraints to say that what counts about a shift-set is whether a worker is busy for at least one of its shifts, or free for all of them.

`NRC_PATTERN` represents a *pattern*: a sequence of shift-sets, each with a polarity. A pattern may be used to define a worker constraint which, for example, applies a penalty when the pattern appears within a worker's timetable.

`NRC_CONSTRAINT` represents one *worker constraint*: a rule about what a worker may do, which if broken in some solution adds a penalty to that solution's cost.

`NRC_SOLN` represents one *solution*: a collection of assignments to the demands of the shifts of one instance.

3.2. Debug functions

Many of NRC's entities have functions included to help with debugging. These functions all work in the same way. Their interface has this form:

```
void NrcEntityDebug(NRC_ENTITY e, int indent, FILE *fp);
```

where `NRC_ENTITY` stands for an NRC type like `NRC_SHIFT`, `NRC_WORKER`, and so on. This produces a debug print of `e` onto file `fp`, which must be open for writing 8-bit characters.

If `indent >= 0`, the print will be indented `indent` spaces and occupy one or more complete lines (that is, it will end with a newline). One debug function often calls another, in which case it adds 2 to the indent, producing a neatly formatted result.

If `indent < 0`, the print will not be indented and will contain no newlines. It will usually be abbreviated, perhaps by printing just the object's name rather than its contents.

3.3. Instance objects

This section describes instance objects: how to create them, and how to visit their components. (Whenever NRC makes an object that is part of an instance, that object is not only created and returned to the user, it is also added to the instance. So one can visit every object via functions on the instance.) Operations for creating components appear in later sections.

3.3.1. Creation, metadata, and archives

To make a new, empty instance, start by calling

```
NRC_INSTANCE NrcInstanceMakeBegin(char *id, char *worker_word,
  HA_ARENA_SET as);
```

Parameter `id` is an identifier identifying the instance, returned by

```
char *NrcInstanceId(NRC_INSTANCE ins);
```

Parameter `worker_word` is the name to give to the single XESTT resource type. Good choices are `"Worker"`, `"Nurse"`, `"Employee"`, and so on. This value is also used when NRC decides to assign its own names to the workers (Section 3.8). Parameter `as` may be `NULL`, or it may be an arena set, as for `NrcArchiveMake`; indeed, it would almost certainly be the same arena set.

After adding all the elements of the instance, but before adding any of its solutions, call

```
void NrcInstanceMakeEnd(NRC_INSTANCE ins);
```

NRC will abort if this is omitted.

Functions

```
HA_ARENA NrcInstanceArenaBegin(NRC_INSTANCE ins);
void NrcInstanceArenaEnd(NRC_INSTANCE ins, HA_ARENA a);
```

provide a convenient interface for obtaining and releasing a memory arena, recycled through the arena set passed to `NrcInstanceMake`. Consult the KHE User's Guide for more information about memory arenas, arena sets, and memory management generally.

NRC needs to know what penalty is wanted when a worker is assigned twice to the same shift. The default value is

```
NrcPenalty(true, 1, NRC_COST_FUNCTION_LINEAR, ins)
```

which, as Section 3.9.1 explains, makes avoiding clashes a hard constraint with weight 1.[1] If this is not correct for some reason, it can be changed, and the value retrieved, by

```
void NrcInstanceSetAvoidClashesPenalty(NRC_INSTANCE ins, NRC_PENALTY p);
NRC_PENALTY NrcInstanceAvoidClashesPenalty(NRC_INSTANCE ins);
```

There is also

```
NRC_PENALTY NrcInstanceZeroPenalty(NRC_INSTANCE ins);
```

which is a convenient way to obtain a penalty with weight 0.

Instance metadata may be set and retrieved by calling

```
void NrcInstanceSetMetaData(NRC_INSTANCE ins, char *name,
  char *contributor, char *date, char *country, char *description,
  char *remarks);
void NrcInstanceMetaData(NRC_INSTANCE ins, char **name,
  char **contributor, char **date, char **country, char **description,
  char **remarks);
```

where `remarks`, being optional, may be `NULL`.

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 `NrcArchiveAddInstance` and `NrcArchiveDeleteInstance` from Section 2.1. To visit the

[1]When a constraint which limits a worker to a most one shift per day is added, what it actually does is limit the number of times during that day when the worker may be busy to one. A worker who is assigned twice to the same shift is still only busy for one time, so NRC must generate a separate constraint, called an avoid clashes constraint, which prevents this. It does this without being asked, using this penalty for the constraint's penalty.

archives containing a given instance, call

```
int NrcInstanceArchiveCount(NRC_INSTANCE ins);
NRC_ARCHIVE NrcInstanceArchive(NRC_INSTANCE ins, int i);
```

in the usual way.

3.3.2. Day names, days, day-sets, and day-set sets

By default, the seven days of the week have their usual English names. To change this, call

```
void NrcInstanceSetDayNames(NRC_INSTANCE ins, char *short_names,
  char *long_names);
```

The first parameter contains the short names of the seven days, separated by colons, and beginning with the name of Sunday (which is how the Unix `mktime` function does it). The second parameter is the same except that it contains long names. For example, the call

```
NrcInstanceSetDayNames(ins, "Sun:Mon:Tue:Wed:Thu:Fri:Sat",
  "Sunday:Monday:Tuesday:Wednesday:Thursday:Friday:Saturday");
```

does not need to be made, because it sets the day names to their default values.

Do not try to use `NrcInstanceResetDayNames` to begin the cycle on a day of the week other than Sunday. Setting parameter `first_day_index` of `NrcCycleMake` (Section 3.4) to a value other than 0 is the right way to do that.

To retrieve the day names, use

```
int NrcInstanceDayNameCount(NRC_INSTANCE ins);
char *NrcInstanceShortDayName(NRC_INSTANCE ins, int i);
char *NrcInstanceLongDayName(NRC_INSTANCE ins, int i);
```

The first day name (the one for Sunday) has index 0, the second (for Monday) has index 1, and so on. `NrcInstanceDayNameCount` always returns 7.

The day objects created make up a day-set called the *cycle*. For visiting them, see Section 3.3.3 below. To visit all the day-sets created within an instance, including the cycle, use

```
int NrcInstanceDaySetCount(NRC_INSTANCE ins);
NRC_DAY_SET NrcInstanceDaySet(NRC_INSTANCE ins, int i);
```

The day-sets are numbered from 0, so the code for visiting them all is

```
for( i = 0;  i < NrcInstanceDaySetCount(ins);  i++ )
{
  ds = NrcInstanceDaySet(ins, i);
  ... visit ds ...
}
```

This is a standard arrangement throughout NRC. Similarly, call

```
int NrcInstanceDaySetSetCount(NRC_INSTANCE ins);
NRC_DAY_SET_SET NrcInstanceDaySetSet(NRC_INSTANCE ins, int i);
```

to visit all the day-set sets created within `ins`.

3.3.3. The cycle and the days of the week

The *cycle* (the sequence of all the days of the instance) is a day-set stored in the instance. Once a cycle has been added, by calling `NrcCycleMake` or `NrcCalendarCycleMake` (Section 3.4.2), the following operations become available. To retrieve the entire cycle as a day-set, call

```
NRC_DAY_SET NrcInstanceCycle(NRC_INSTANCE ins);
```

To visit the days in chronological order, call

```
int NrcInstanceCycleDayCount(NRC_INSTANCE ins);
NRC_DAY NrcInstanceCycleDay(NRC_INSTANCE ins, int i);
```

`NrcInstanceCycleDayCount` returns the number of days, and `NrcInstanceCycleDay` returns the `i`'th day. As usual in NRC, counting starts from 0, so the code to visit each day is

```
for( i = 0;  i < NrcInstanceCycleDayCount(ins);  i++ )
{
  day = NrcInstanceCycleDay(ins, i);
  ... visit day ...
}
```

There is also

```
bool NrcInstanceCycleRetrieveDay(NRC_INSTANCE ins, char *ymd,
  NRC_DAY *d);
```

which retrieves a day from the cycle by its calendar date. If the cycle contains a day whose year-month-day name is `ymd`, this function sets `*d` to one such day and returns `true`; if not, it sets `*d` to `NULL` and returns `false`. The date string must contain three non-negative integers separated by hyphens; a copy, normalized to a four-digit year and two-digit month and day, is used when retrieving.

At the same time a cycle is added to an instance, day-sets representing the 7 days of the week are also added. These are stored in a day-set set, and may be retrieved in that form by

```
NRC_DAY_SET_SET NrcInstanceDaysOfWeek(NRC_INSTANCE ins);
```

They may also be visited individually by calling

```
int NrcInstanceDaysOfWeekDaySetCount(NRC_INSTANCE ins);
NRC_DAY_SET NrcInstanceDaysOfWeekDaySet(NRC_INSTANCE ins, int i);
```

`NrcInstanceDaysOfWeekDaySetCount` returns the number of day-sets representing days of the week (always 7), and `NrcInstanceDaysOfWeekDaySet` returns the `i`'th of these day-sets, counting from 0 as usual. There is also

```
bool NrcInstanceDaysOfWeekRetrieveDaySet(NRC_INSTANCE ins,
  char *long_name, NRC_DAY_SET *ds);
```

which retrieves one of these day sets by long name. Irrespective of how the cycle was created, the first of these day-sets holds the Sunday days, the second holds the Monday days, and so on.

3.3.4. Shift types and shift-type sets

Functions for creating and querying shift types and shift-type sets are given in Section 3.6. To retrieve all the shift types of the instance as a shift-type set, call

```
NRC_SHIFT_TYPE_SET NrcInstanceAllShiftTypes(NRC_INSTANCE ins);
```

To visit the shift types of an instance one by one, call

```
int NrcInstanceShiftTypeCount(NRC_INSTANCE ins);
NRC_SHIFT_TYPE NrcInstanceShiftType(NRC_INSTANCE ins, int i);
```

counting from 0 in the usual way. To retrieve a shift type by name, call

```
bool NrcInstanceRetrieveShiftType(NRC_INSTANCE ins, char *name,
  NRC_SHIFT_TYPE *st);
```

As usual, if there is a shift type with the given name, this sets `*st` to that shift type and returns `true`, otherwise it sets `*st` to `NULL` and returns `false`. Function

```
bool NrcInstanceRetrieveShiftTypeByLabel(NRC_INSTANCE ins, char *label,
  NRC_SHIFT_TYPE *st);
```

is the same, except that it searches for a shift type with a non-`NULL` label equal to `label`.

To visit the shift-type sets of an instance, call

```
int NrcInstanceShiftTypeSetCount(NRC_INSTANCE ins);
NRC_SHIFT_TYPE_SET NrcInstanceShiftTypeSet(NRC_INSTANCE ins, int i);
```

in the usual way. If a shift-type set has a non-`NULL` name, it may be retrieved by calling

```
bool NrcInstanceRetrieveShiftTypeSet(NRC_INSTANCE ins, char *name,
  NRC_SHIFT_TYPE_SET *sts);
```

in the usual way.

3.3.5. Shifts, shift-sets, and shift-set sets

The shifts of an instance are stored in the instance as a shift-set. To retrieve this shift-set call

```
NRC_SHIFT_SET NrcInstanceAllShifts(NRC_INSTANCE ins);
```

To visit the shifts one by one, call

```
int NrcInstanceShiftCount(NRC_INSTANCE ins);
NRC_SHIFT NrcInstanceShift(NRC_INSTANCE ins, int i);
```

as usual. (In many cases, however, a better way to visit each shift is to visit each day, and then visit each shift on that day.) Also, functions

```
NRC_SHIFT_SET NrcInstanceDailyStartingShiftSet(NRC_INSTANCE ins);
NRC_SHIFT_SET NrcInstanceWeeklyStartingShiftSet(NRC_INSTANCE ins);
```

return the set of shifts which are first in each day, and first in each week. These are useful values for the `starting_ss` parameter of `NrcConstraintMake` (Section 3.9.8). To visit all shift-sets, call

```
int NrcInstanceShiftSetCount(NRC_INSTANCE ins);
NRC_SHIFT_SET NrcInstanceShiftSet(NRC_INSTANCE ins, int i);
```

and to visit all shift-set sets, call

```
int NrcInstanceShiftSetSetCount(NRC_INSTANCE ins);
NRC_SHIFT_SET_SET NrcInstanceShiftSetSet(NRC_INSTANCE ins, int i);
```

in the usual way. There are also

```
NRC_SHIFT_SET_SET NrcInstanceDaysShiftSetSet(NRC_INSTANCE ins);
```

which returns a shift-set set containing one shift-set for each day of the cycle, holding the shifts of that day, and

```
NRC_SHIFT_SET_SET NrcInstanceShiftsShiftSetSet(NRC_INSTANCE ins);
```

which returns a shift-set set containing one shift-set for each shift, holding that shift.

3.3.6. Workers, worker-sets, and worker-set sets

Functions for creating and querying workers, worker-sets, and worker-set sets are given in Section 3.8. The workers of an instance are held in a worker-set in the instance called the *staffing*. It may be retrieved by calling

```
NRC_WORKER_SET NrcInstanceStaffing(NRC_INSTANCE ins);
```

For convenience, its elements may be visited directly by

```
int NrcInstanceStaffingWorkerCount(NRC_INSTANCE ins);
NRC_WORKER NrcInstanceStaffingWorker(NRC_INSTANCE ins, int i);
```

There is also

```
bool NrcInstanceStaffingRetrieveWorker(NRC_INSTANCE ins,
  char *name, NRC_WORKER *w);
```

which retrieves a worker with the given name from the staffing, setting `*w` to that worker and returning `true` if successful, and setting `*w` to `NULL` and returning `false` otherwise.

It can be convenient sometimes to have access to an empty worker set:

```
NRC_WORKER_SET NrcInstanceEmptyWorkerSet(NRC_INSTANCE ins);
```

Adding a worker to the result of this function will cause strange errors.

To visit all the worker-sets of an instance (including the staffing), call

```
int NrcInstanceWorkerSetCount(NRC_INSTANCE ins);
NRC_WORKER_SET NrcInstanceWorkerSet(NRC_INSTANCE ins, int i);
```

in the usual way. There is also

```
bool NrcInstanceRetrieveWorkerSet(NRC_INSTANCE ins, char *name,
  NRC_WORKER_SET *ws);
```

which retrieves a worker-set with the given name from the instance, setting `*ws` to that worker-set and returning `true` if successful, and setting `*ws` to `NULL` and returning `false` otherwise.

To visit all the worker-set sets of an instance, call

```
int NrcInstanceWorkerSetSetCount(NRC_INSTANCE ins);
NRC_WORKER_SET_SET NrcInstanceWorkerSetSet(NRC_INSTANCE ins, int i);
```

in the usual way.

3.3.7. Contracts and skills

Several nurse rostering models offer *contracts*, which are sets of constraints. A worker can be made subject to a contract, which means that the contract's constraints apply to that worker. At least one model allows a worker to be subject to more than one contract.

In NRC, a worker-set is used to model each contract. Its name is the name of the contract, perhaps with `"Contract-"` prepended, and its members are the workers subject to the contract. The contract's constraints are not stored in the worker-set. Instead, each constraint has a worker-set attribute saying which workers it applies to. When there are contracts, this would be the worker-set for the contract that the constraint lies within. When there are no contracts, it would be something else—the set of all workers returned by `NrcInstanceStaffing` above, perhaps, or `NrcWorkerSingletonWorkerSet(w)`, the worker-set containing just worker `w`.

Each NRC instance holds a worker-set set called the *contracts*, intended to hold the set of all contracts, although what it actually holds is up to the user. This worker-set set is not consulted when generating an instance; it is there only for the convenience of the user. (Exception: for documentation, a KHE resource group is generated for each contract, even if it is not used.)

To add a contract to the contracts, call

```
void NrcInstanceContractsAddContract(NRC_INSTANCE ins,
  NRC_WORKER_SET contract_ws);
```

To retrieve the contracts as a worker-set set, call

```
NRC_WORKER_SET_SET NrcInstanceContracts(NRC_INSTANCE ins);
```

To visit the contracts one by one, call

```
int NrcInstanceContractsContractCount(NRC_INSTANCE ins);
NRC_WORKER_SET NrcInstanceContractsContract(NRC_INSTANCE ins, int i);
```

To retrieve a contract by name, call

```
bool NrcInstanceContractsRetrieveContract(NRC_INSTANCE ins,
  char *name, NRC_WORKER_SET *contract_ws);
```

If the contracts contain a contract with the given name, this sets `*contract_ws` to a contract with that name and returns `true`, otherwise it sets `*contract_ws` to `NULL` and returns `false`.

It is acceptable to define the contract at one time and add workers to it later. Indeed, this is what usually happens, given that contracts are defined at one point in the source file and workers declare their adherence to a contract at another.

Most nurse rostering models offer *skills*. A skill is some capability that a worker has, such as being a senior nurse, or a CPR expert. A worker may have any number of skills. A demand for a worker may require that a worker with a particular skill be assigned. If a worker without that skill is assigned, there is a penalty, which may vary depending on which worker is assigned.

Again, in NRC a worker-set is used to model each skill. Its name is the name of the skill, and its elements are the workers who have that skill. Each demand has an optional worker-set attribute specifying the skill that workers satisfying that demand should have.

Each NRC instance holds a worker-set set called the *skills*, intended to hold the set of all skills, although what it actually holds is up to the user. This worker-set set is not consulted when generating an instance; it is there only for the convenience of the user. (Exception: for documentation, a KHE resource group is generated for each skill, even if it is not used.)

Operations entirely analogous to those for the contracts are offered for the skills:

```
void NrcInstanceSkillsAddSkill(NRC_INSTANCE ins,
  NRC_WORKER_SET skill_ws);
NRC_WORKER_SET_SET NrcInstanceSkills(NRC_INSTANCE ins);
int NrcInstanceSkillsSkillCount(NRC_INSTANCE ins);
NRC_WORKER_SET NrcInstanceSkillsSkill(NRC_INSTANCE ins, int i);
bool NrcInstanceSkillsRetrieveSkill(NRC_INSTANCE ins,
  char *name, NRC_WORKER_SET *skill_ws);
```

The contracts and skills may well be the only worker-sets the user needs. There is nothing to prevent a worker from lying within two or more contracts, or two or more skills. Indeed, this is quite normal, at least for skills.

3.3.8. Demands, demand-sets, patterns, and constraints

Functions for creating and querying demands, demand-sets, patterns, pattern sets, and constraints are given in Section 3.9. All these objects are stored in the instance. Demands may be visit-ed by

```
int NrcInstanceDemandCount(NRC_INSTANCE ins);
NRC_DEMAND NrcInstanceDemand(NRC_INSTANCE ins, int i);
```

and demand-sets may be visited by

```
int NrcInstanceDemandSetCount(NRC_INSTANCE ins);
NRC_DEMAND_SET NrcInstanceDemandSet(NRC_INSTANCE ins, int i);
```

in the usual way. Patterns may be visited by

```
int NrcInstancePatternCount(NRC_INSTANCE ins);
NRC_PATTERN NrcInstancePattern(NRC_INSTANCE ins, int i);
```

in the usual way, and patterns with a non-`NULL` name may be retrieved by calling

```
bool NrcInstanceRetrievePattern(NRC_INSTANCE ins, char *name,
  NRC_PATTERN *p);
```

The stored patterns are not used privately by NRC; in particular, they do not become unwanted unless they are added to worker constraints. Pattern sets are may be visited by

```
int NrcInstancePatternSetCount(NRC_INSTANCE ins);
NRC_PATTERN_SET NrcInstancePatternSet(NRC_INSTANCE ins, int i);
```

Pattern sets have no names so there is no retrieve operation.

Demand constraints and worker constraints are stored in the instance, and may be visited by

```
int NrcInstanceDemandConstraintCount(NRC_INSTANCE ins);
NRC_DEMAND_CONSTRAINT NrcInstanceDemandConstraint(NRC_INSTANCE ins,
  int i);
```

and

```
int NrcInstanceConstraintCount(NRC_INSTANCE ins);
NRC_CONSTRAINT NrcInstanceConstraint(NRC_INSTANCE ins, int i);
```

in the usual way.

3.4. Days

In informal discourse, a day could be a specific day, such as 23 July 2016, or it could be a day of the week, such as Friday. In NRC and this documentation, the term *day* always refers to a specific day, represented by an object of type `NRC_DAY`.

3.4.1. The cycle and the days of the week

There is no NRC function for creating one day. Instead, there is a function for creating the *cycle*, the set of all days of an instance:

```
void NrcCycleMake(NRC_INSTANCE ins, int day_count, int first_day_index);
```

`NrcCycleMake` adds to `ins` a cycle of `day_count` days. Parameter `first_day_index` says which day of the week the first day is on: 0 means that the first day is a Sunday, 1 means that the first day is a Monday, and so on.

Although the days created by `NrcCycleMake` are specific days, they are not associated with calendar dates. To get days with calendar dates, call `NrcCalendarCycleMake` instead:

```
bool NrcCalendarCycleMake(NRC_INSTANCE ins,
  char *start_ymd, char *end_ymd, char **err_str);
```

`NrcCalendarCycleMake` creates a cycle with first day `start_ymd` and last day `end_ymd`, where the two strings are given in `YYYY-MM-DD` format. (Actually, the format is just three non-negative integers separated by hyphens; but the dates actually stored are normalized to the format shown.) The Unix `mktime` function is used to find out which day of the week `start_ymd` is on, and other important facts such as the number of days in each month. The two days are arbitrary except that there must be at least one day in the cycle. Value `true` is returned if successful; otherwise `false` is returned and `err_str` is set to an error message explaining what went wrong. This will be some problem with `start_ymd` or `end_ymd`, such as being formatted wrongly or specifying a non-existent date.

`NrcCycleMake` and `NrcCalendarCycleMake` may only be called after the last call to `NrcShiftTypeMake` (Section 3.6.1). A call to `NrcShiftTypeMake` after the cycle is made will cause NRConv to exit with an error message.

3.4.2. Days

A *day* in NRC is an object of type `NRC_DAY` representing a specific day, such as Monday 21 November 2016, although the calendar date of the day need not be known.

There is no function for creating an individual day. Instead, functions `NrcCycleMake` and `NrcCalendarCycleMake` (Section 3.4.1) are called, to make all the days at once. The days created by these functions can be accessed using functions `NrcInstanceCycleDayCount` and `NrcInstanceCycleDay` (Section 3.3.3).

The basic attributes of a day may be found by calling

```
NRC_INSTANCE NrcDayInstance(NRC_DAY d);
char *NrcDayYMD(NRC_DAY d);
char *NrcDayShortName(NRC_DAY d);
char *NrcDayLongName(NRC_DAY d);
```

`NrcDayInstance` returns the enclosing instance. `NrcDayYMD` returns the date of day `d` in `YYYY-MM-DD` format if the day was created by `NrcCalendarCycleMake`, or `"-"` otherwise. `NrcDayShortName` and `NrcDayLongName` return the name of the day in either short and long form, consisting of a week number followed by a short or long day name as defined by `NrcInstanceSetDayNames` (Section 3.3.2). For example, `1Mon` is the short name of the day which represents the Monday of the first week of the cycle.

Several functions return integer indexes defining the position of the day in the cycle:

```
int NrcDayIndexInCycle(NRC_DAY d);
int NrcDayWeekInCycle(NRC_DAY d);
int NrcDayIndexInWeek(NRC_DAY d);
```

`NrcDayIndexInCycle` returns the index of `d` in the cycle: `0` for the first day, 1 for the second, and so on. `NrcDayWeekInCycle` returns the number of the week that `d` lies in. The first 7 days of the cycle have week number 1, the second 7 days have week number 2, and so on, irrespective of the day of the week the cycle begins on. `NrcDayIndexInWeek` returns the index of `d`'s day of the week: 0 for Sunday, 1 for Monday, and so on (which is how the Unix `mktime` function does it). NRC does not have an `NRC_WEEK` data type.

Functions

```
NRC_DAY NrcDayPrev(NRC_DAY d);
NRC_DAY NrcDayNext(NRC_DAY d);
```

return the day preceding `d` in the cycle, or `NULL` when `d` is the first day, and the day following `d` in the cycle, or `NULL` when `d` is the last day.

A surprisingly useful function is

```
NRC_DAY_SET NrcDayDayOfWeek(NRC_DAY d);
```

which returns `d`'s day of the week. NRC does not have a data type for day of the week. Instead, a day of the week is represented by an `NRC_DAY_SET` object holding a set of days—all the days that fall on the same day of the week. The day set objects returned by `NrcDayDayOfWeek` are created by `NrcCycleMake` and `NrcCalendarCycleMake` while they are creating the days; the user does not need to create them. Another function that returns a day-set is

```
NRC_DAY_SET NrcDaySingletonDaySet(NRC_DAY d);
```

The result is a a day-set containing just `d`.

Function

```
NRC_SHIFT_SET NrcDayShiftSet(NRC_DAY d);
```

returns a shift-set containing the shifts of `d`. There is one of these for each shift type, in the order that the shift types were added to the instance. For convenience,

```
int NrcDayShiftCount(NRC_DAY d);
NRC_SHIFT NrcDayShift(NRC_DAY d, int i);
```

can be used to visit this shift-set's shifts directly. Function

```
NRC_SHIFT NrcDayShiftFromShiftType(NRC_DAY d, NRC_SHIFT_TYPE st);
```

returns the shift with day `d` and type `st`, i.e. `NrcDayShift(d, NrcShiftTypeIndex(st))`, and

```
NRC_SHIFT_SET NrcDayShiftSetFromShiftTypeSet(NRC_DAY d,
  NRC_SHIFT_TYPE_SET sts);
```

does this for each element of `sts`, producing a shift-set. There is also

```
NRC_SHIFT_SET_SET NrcDayShiftSetSet(NRC_DAY d);
```

which returns a shift-set set containing one shift-set for each shift on day `d`, namely the singleton shift-set holding that shift.

To produce a debug print of day `d`, call

```
void NrcDayDebug(NRC_DAY d, int indent, FILE *fp);
```

This works as explained in Section 3.2.

3.4.3. Day-sets

A *day-set* is a set (more precisely, a sequence) of days. Although any days can make up a day-set, the most likely combinations are adjacent days (for example, the days of a weekend) and days that share the same day of the week (for example, the set of all Mondays in the cycle).

`NrcCycleMake` and `NrcCalendarCycleMake` make day-sets: one holding the cycle as a whole, and one for each of the seven days of the week (the day-set of all Mondays, the day-set of all Tuesdays, and so on). Day-sets can also be created by the user, by calling

```
NRC_DAY_SET NrcDaySetMake(NRC_INSTANCE ins, char *short_name,
  char *long_name);
void NrcDaySetAddDay(NRC_DAY_SET ds, NRC_DAY d);
```

in the usual way. Both names must be non-`NULL`. Functions

```
NRC_INSTANCE NrcDaySetInstance(NRC_DAY_SET ds);
char *NrcDaySetShortName(NRC_DAY_SET ds);
char *NrcDaySetLongName(NRC_DAY_SET ds);
```

return the attributes of day-set `ds`, functions

```
int NrcDaySetDayCount(NRC_DAY_SET ds);
NRC_DAY NrcDaySetDay(NRC_DAY_SET ds, int i);
```

visit the days in the order they were inserted,

```
bool NrcDaySetContainsDay(NRC_DAY_SET ds, NRC_DAY d);
```

returns `true` when `ds` contains `d`, and

```
bool NrcDaySetRetrieveDay(NRC_DAY_SET ds, char *ymd, NRC_DAY *d);
```

retrieves a day from `ds` by its `ymd` value. The date string must contain three non-negative integers separated by hyphens; a copy, normalized to a four-digit year and two-digit month and day, is used when retrieving. Function

```
bool NrcDaySetsOverlap(NRC_DAY_SET ds1, NRC_DAY_SET ds2);
```

returns `true` when `ds1` and `ds2` have a non-empty intersection. And

```
NRC_DAY_SET NrcDaySetDifference(NRC_DAY_SET ds1, NRC_DAY_SET ds2);
```

returns a new day-set containing the days of `ds1` that are not in `ds2`.

Function

```
NRC_SHIFT_SET NrcDaySetShiftSet(NRC_DAY_SET ds);
```

returns a shift-set containing all the shifts on all the days of `ds`, while

```
NRC_SHIFT_SET NrcDaySetStartingShiftSet(NRC_DAY_SET ds);
```

returns a shift-set containing the first shift on each day of `ds`. This helps when constructing suitable values for the `starting_ss` parameter of `NrcConstraintMake` (Section 3.9.8). If any of the shift-sets of the days of `ds` is empty, `NrcDaySetStartingShiftSet` aborts. Also,

```
NRC_SHIFT_SET_SET NrcDaySetShiftSetSet(NRC_DAY_SET ds);
```

returns a shift-set set containing one shift-set for each day of `ds`, holding that day's shifts. And

```
NRC_SHIFT_SET NrcDaySetShiftSetFromShiftTypeSet(NRC_DAY_SET ds,
  NRC_SHIFT_TYPE_SET sts);
```

returns a shift-set containing all shifts whose day is in `ds` and whose shift type is in `sts`.

Function

```
void NrcDaySetDebug(NRC_DAY_SET ds, int indent, FILE *fp);
```

produces a debug print of `ds` onto `fp`, as explained in Section 3.2.

3.4.4. Day-set sets

A *day-set set* is a set (more precisely, a sequence) of day-sets. For example, `1Sat` is a day, `{1Sat, 1Sun}` is a day-set representing one weekend, and

```
{{1Sat, 1Sun}, {2Sat, 2Sun}, {3Sat, 3Sun}, {4Sat, 4Sun}}
```

is a day-set set representing (possibly) the set of all weekends in the cycle.

To create a day-set set and add day-sets to it, the functions are

```
NRC_DAY_SET_SET NrcDaySetSetMake(NRC_INSTANCE ins, char *short_name,
  char *long_name);
void NrcDaySetSetAddDaySet(NRC_DAY_SET_SET dss, NRC_DAY_SET ds);
```

Functions

```
NRC_INSTANCE NrcDaySetSetInstance(NRC_DAY_SET_SET dss);
char *NrcDaySetSetShortName(NRC_DAY_SET_SET dss);
char *NrcDaySetSetLongName(NRC_DAY_SET_SET dss);
```

return the attributes of a day-set set, and

```
int NrcDaySetSetDaySetCount(NRC_DAY_SET_SET dss);
NRC_DAY_SET NrcDaySetSetDaySet(NRC_DAY_SET_SET dss, int i);
bool NrcDaySetSetRetrieveDaySet(NRC_DAY_SET_SET dss,
  char *long_name, NRC_DAY_SET *ds);
```

are used to visit the day-sets of a day-set set in the order they were inserted, and to retrieve a day-set from a day-set set by long name. And

```
void NrcDaySetSetDebug(NRC_DAY_SET_SET dss, int indent, FILE *fp);
```

produces a debug print of `dss` onto `fp`, as explained in Section 3.2.

3.5. Time intervals

A *time interval* is an interval of time, consisting of a start time and an end time, measured in seconds since midnight. These are not absolute intervals like ‘10am to 11am on 7 April 2007’, but rather generic ones, like ‘10am to 11am’.

As is common in nurse rostering, an end time may be smaller than a start time, meaning that the interval spans midnight. Strictly speaking, then, what is representable is a time interval that starts at any time of day and ends less than 24 hours later, on the same day or the following day.

Before considering operations on time intervals proper, here are two more basic functions. The first converts a string in `HH:MM:SS` or `HH:MM` format into an integer number of seconds:

```
bool NrcHMSToSecs(char *hms, int *res);
```

It returns `true` and sets `*res` to the number of seconds if successful, and returns `false` and sets `*res` to `-1` if not successful (because `hms` has a format problem). The second function,

```
char *NrcSecsToHMS(int secs, HA_ARENA a);
```

converts an integer number of seconds to a string in `HH:MM:SS` format stored in arena `a`.

To create a time interval, call

```
NRC_TIME_INTERVAL NrcTimeIntervalMake(int start_secs, int end_secs,
  NRC_INSTANCE ins);
```

The new object will be stored in the arena of `ins` and deleted when the instance is deleted. There are `24 * 60 * 60` seconds in a day, so `start_secs` and `end_secs` must satisfy

```
0 <= start_secs < 24 * 60 * 60
0 < end_secs <= 24 * 60 * 60
```

Disallowing `start_secs == 24 * 60 * 60` and `end_secs == 0` ensures that there is only one way to represent any non-empty interval, including intervals that start or end at midnight.

Function

```
bool NrcTimeIntervalMakeFromHMS(char *start_hms, char *end_hms,
  NRC_TIME_INTERVAL *res, NRC_INSTANCE ins);
```

converts `start_hms` and `end_hms` using `NrcHMSToSecs` above, and makes a time interval from them, returning `true` and setting `*res` to it if successful, and returning `false` and setting `*res` to `NULL` if unsuccessful, because `start_hms` or `end_hms` has a format problem or is out of range. As occurs in instances, `end_hms` may be `00:00:00`, which is taken to mean `24:00:00`.

The two basic queries are

```
int NrcTimeIntervalStartSecs(NRC_TIME_INTERVAL ti);
int NrcTimeIntervalEndSecs(NRC_TIME_INTERVAL ti);
```

There are also set operations on time intervals:

```
bool NrcTimeIntervalEqual(NRC_TIME_INTERVAL ti1, NRC_TIME_INTERVAL ti2);
bool NrcTimeIntervalDisjoint(NRC_TIME_INTERVAL ti1, NRC_TIME_INTERVAL ti2);
bool NrcTimeIntervalSubset(NRC_TIME_INTERVAL ti1, NRC_TIME_INTERVAL ti2);
```

These return `true` when `ti1` and `ti2` are equal, or disjoint, or when `ti1` is a subset of `ti2`. These operations consider intervals to be *open*, that is, to not include their endpoints. So when one interval's end time equals another's start time, the two intervals are disjoint. They also understand what it means when `end_secs < start_secs`, and act accordingly. Function

```
NRC_SHIFT_SET NrcTimeIntervalShiftSet(NRC_TIME_INTERVAL ti, NRC_DAY d);
```

returns a shift-set containing those shifts which have time intervals which intersect with (are not disjoint with) time interval `ti` on day `d`. The shifts of the result do not necessarily all come from day `d`; some may come from the previous day if they span midnight and there is a previous day, while others may come from the next day if `ti` spans midnight and there is a next day. Finally,

```
char *NrcTimeIntervalShow(NRC_TIME_INTERVAL ti, HA_ARENA a);
```

returns a string, stored in arena `a`, representing time interval `ti`.

3.6. Shift types

In informal discourse, a shift could be a specific shift, such as the night shift on 23 July 2016, or it could be a generic shift, such as the night shift. In NRC, a generic shift is called a *shift type*, and a specific shift is called a *shift*. There is one shift of each shift type on each day.

3.6.1. Shift types

A *shift type* is a generic shift, such as 'the day shift' or 'the night shift'. To make one, call

```
NRC_SHIFT_TYPE NrcShiftTypeMake(NRC_INSTANCE ins, char *name,
  int workload);
```

This creates a new shift type object with the given name and workload, adds it to `ins`, and returns it. The workload is a value in arbitrary units (for example, in minutes), describing how much work a shift of this type is. It is only needed when there are worker constraints that limit total workload. When it is not needed, `NRC_NO_WORKLOAD` (a synonym for `1`) should be passed.

Some shift types have a label as well as a name. Function

```
void NrcShiftTypeUseLabelInEventName(NRC_SHIFT_TYPE st, char *label);
```

adds a label to `st` and ensures that it is used in event names.

To retrieve the attributes of a shift type, call

```
NRC_INSTANCE NrcShiftTypeInstance(NRC_SHIFT_TYPE st);
char *NrcShiftTypeName(NRC_SHIFT_TYPE st);
char *NrcShiftTypeLabel(NRC_SHIFT_TYPE st);
int NrcShiftTypeWorkload(NRC_SHIFT_TYPE st);
int NrcShiftTypeIndex(NRC_SHIFT_TYPE st);
```

`NrcShiftTypeLabel` returns `NULL` if `NrcShiftTypeUseLabelInEventName` has not been called on `st`. `NrcShiftTypeIndex` returns `st`'s index in the instance (0 for the first shift type added, 1 for the second, and so on). There is also

```
NRC_SHIFT_TYPE_SET NrcShiftTypeSingletonShiftTypeSet(NRC_SHIFT_TYPE st);
```

which returns a shift-type set containing just `st`.

Two functions give access to the shifts of type `st`:

```
NRC_SHIFT_SET NrcShiftTypeShiftSet(NRC_SHIFT_TYPE st);
NRC_SHIFT_SET_SET NrcShiftTypeShiftSetSet(NRC_SHIFT_TYPE st);
```

`NrcShiftTypeShiftSet` returns a shift-set containing all shifts which have `st` for their shift type, in increasing day order. `NrcShiftTypeShiftSetSet` is similar, except that each shift lies in its own singleton shift set.

There is no `NrcShiftTypeDebug` function, because a shift type is essentially just its name. Use `NrcShiftTypeName` instead.

There is one shift for each shift type on each day. Shift types must be added before days. Later, when days are added, by `NrcCycleMake` or `NrcCalendarCycleMake` (Section 3.4), one shift of each type is added to each day.

A shift type may optionally contain a time interval. It is not used by NRC, but it may be useful to the user. The operations for setting and retrieving it are

```
void NrcShiftTypeAddTimeInterval(NRC_SHIFT_TYPE st, NRC_TIME_INTERVAL ti);
NRC_TIME_INTERVAL NrcShiftTypeTimeInterval(NRC_SHIFT_TYPE st);
```

`NrcShiftTypeTimeInterval` returns `NULL` when `st`'s time interval has not been set.

The author hesitated over whether to include shift types, since a shift-set with the name of the shift type does most of what a shift type does. However, to construct that shift-set one needs some way to identify what is common to its shifts—a name, presumably; but then that is the name of a shift type, not of a shift. More importantly, with shift types available, NRC can make the individual shifts itself, and guarantee a uniform structure of one shift of each type on each day. This uniformity matters, for example, when implementing patterns. If some of these shifts do not need to be covered on some days (for example, if the early shift does not need to be staffed on Sundays), one can just add no demands to those shifts.

3.6.2. Shift-type sets

A *shift-type set* is a set of shift types. It is created in the usual way:

```
NRC_SHIFT_TYPE_SET NrcShiftTypeSetMake(NRC_INSTANCE ins, char *name);
void NrcShiftTypeSetAddShiftType(NRC_SHIFT_TYPE_SET sts, NRC_SHIFT_TYPE st);
```

The name is optional (may be `NULL`). Its attributes are returned by

```
NRC_INSTANCE NrcShiftTypeSetInstance(NRC_SHIFT_TYPE_SET sts);
char *NrcShiftTypeSetName(NRC_SHIFT_TYPE_SET sts);
```

and its shift types are visited by

```
int NrcShiftTypeSetShiftTypeCount(NRC_SHIFT_TYPE_SET sts);
NRC_SHIFT_TYPE NrcShiftTypeSetShiftType(NRC_SHIFT_TYPE_SET sts, int i);
```

There are also

```
bool NrcShiftTypeSetContainsShiftType(NRC_SHIFT_TYPE_SET sts,
  NRC_SHIFT_TYPE st);
```

which returns `true` when `sts` contains `st`,

```
bool NrcShiftTypeSetEqual(NRC_SHIFT_TYPE_SET sts1,
  NRC_SHIFT_TYPE_SET sts2);
```

which returns `true` when `sts1` and `sts2` contain the same shift types, and

```
bool NrcShiftTypeSetDisjoint(NRC_SHIFT_TYPE_SET sts1,
  NRC_SHIFT_TYPE_SET sts2);
```

which returns `true` when they are disjoint. Finally,

```
NRC_SHIFT_TYPE_SET NrcShiftTypeSetMerge(NRC_SHIFT_TYPE_SET sts1,
  NRC_SHIFT_TYPE_SET sts2);
```

returns a new shift-type set containing the set union of `sts1` and `sts2`. Shift-type sets are mainly used when constructing patterns (Section 3.9.6).

3.7. Shifts

3.7.1. Shifts

There are no functions for creating individual shifts. Instead, NRC automatically creates one shift, of type `NRC_SHIFT`, for each shift type on each day. These shifts can be accessed as elements of the shift-sets returned by `NrcShiftTypeShiftSet` and `NrcDayShiftSet`, or using `NrcInstanceShiftCount` and `NrcInstanceShift`.

To retrieve the basic attributes of a shift, call

```
NRC_INSTANCE NrcShiftInstance(NRC_SHIFT s);
NRC_DAY NrcShiftDay(NRC_SHIFT s);
NRC_SHIFT_TYPE NrcShiftType(NRC_SHIFT s);
```

The shift's index in the list of all shifts held by the instance is

```
int NrcShiftIndex(NRC_SHIFT s);
```

This is equal to the shift's day's index multiplied by the number of shift types, plus the shift's shift type's index. In other words, as indexes increase we run through the shifts of the first day in shift type order, then the shifts of the second day, and so on.

Function

```
char *NrcShiftName(NRC_SHIFT s);
```

returns the name that `s` will have in the converted instance, based on the day and shift type.

There is also

```
NRC_SHIFT_SET NrcShiftSingletonShiftSet(NRC_SHIFT s);
```

which returns a shift-set containing just `s`.

A shift has an optional workload, measured in arbitrary units, for example minutes:

```
int NrcShiftWorkload(NRC_SHIFT s);
```

The workload of a shift is the workload of its shift type, and this cannot be changed.

To add a demand (Section 3.9.3) to a shift, specifying that a worker, optionally with a certain skill, needs to be assigned to that shift, call

```
void NrcShiftAddDemand(NRC_SHIFT s, NRC_DEMAND d);
```

Any number of demands may be added in this way; their total number is the total number of workers demanded by the shift. There are also the convenience functions

```
void NrcShiftAddDemandMulti(NRC_SHIFT s, NRC_DEMAND d, int multiplicity);
void NrcShiftAddDemandSet(NRC_SHIFT s, NRC_DEMAND_SET ds);
```

`NrcShiftAddDemandMulti` adds `d` to `s` `multiplicity` times, while `NrcShiftAddDemandSet` adds the demands of `ds` individually to `s`. The fact that these demands arrived in a group is not remembered. To visit the demands of shift `s`, call

```
int NrcShiftDemandCount(NRC_SHIFT s);
NRC_DEMAND NrcShiftDemand(NRC_SHIFT s, int i);
```

as usual. Demands are immutable objects and may be shared by several shifts, and added to the same shift several times, when several workers with the same characteristics are wanted. In fact, behind the scenes NRC forces you to share demands: the demand object returned by `NrcDemandMake` is only new when there is no existing demand with the same attributes. This helps to reduce the size of the generated XESTT file, as it turns out.

Workers can be preassigned to shifts by calling

```
void NrcShiftAddPreassignment(NRC_SHIFT s, NRC_WORKER w);
```

To visit these preassignments, call

```
int NrcShiftPreassignmentCount(NRC_SHIFT s);
NRC_WORKER NrcShiftPreassignment(NRC_SHIFT s, int i);
```

as usual. This works in simple cases, but be warned that the implementation is rough and ready. At present there is no way to indicate that the preassigned worker should fill a particular role or satisfy the demand for a particular skill. And if the preassignments cannot all be included in the XESTT event which is the result of converting the shift, NRC aborts with an error message.

Function

```
void NrcShiftDebug(NRC_SHIFT s, int indent, FILE *fp);
```

produces a debug print of `s` onto `fp`, as explained in Section 3.2.

3.7.2. Shift-sets

A shift-set is a set of shifts. To make a shift-set, use

```
NRC_SHIFT_SET NrcShiftSetMake(NRC_INSTANCE ins, char *name);
void NrcShiftSetAddShift(NRC_SHIFT_SET ss, NRC_SHIFT s);
```

as usual; or to add the shifts of another shift-set `ss2` to shift-set `ss` all at once, call

```
void NrcShiftSetAddShiftSet(NRC_SHIFT_SET ss, NRC_SHIFT_SET ss2);
```

To retrieve the attributes, call

```
NRC_INSTANCE NrcShiftSetInstance(NRC_SHIFT_SET ss);
char *NrcShiftSetName(NRC_SHIFT_SET ss);
int NrcShiftSetShiftCount(NRC_SHIFT_SET ss);
NRC_SHIFT NrcShiftSetShift(NRC_SHIFT_SET ss, int i);
```

There is also

```
bool NrcShiftSetContainsShift(NRC_SHIFT_SET ss, NRC_SHIFT s);
```

which returns `true` when `ss` contains `s`.

Function

```
bool NrcShiftSetUniform(NRC_SHIFT_SET ss, int *offset);
```

returns `true` when `ss` is uniform: when successive shifts are the same distance apart. A shift set containing the shifts of one day is uniform, but so is a shift set containing the first shift on each day. If the shift set is uniform, `*offset` is set to the gap between successive shifts.

A different kind of uniformity is tested by

```
bool NrcShiftSetsUniform(NRC_SHIFT_SET ss1, NRC_SHIFT_SET ss2, int *offset);
```

Here `ss1` and `ss2` do not have to be uniform in the previous sense. Instead, `ss2` has to be equal to `ss1` except shifted by `*offset`. Finally,

```
bool NrcShiftSetsEqual(NRC_SHIFT_SET ss1, NRC_SHIFT_SET ss2);
```

is a conventional equality test, equivalent to uniformity with an offset of 0.

The shifts of a shift-set may be arbitrary, but often they will be all the shifts of a particular day, or all the shifts with a particular shift type. These shift-sets are constructed automatically by NRC, and are obtained by calling `NrcDayShiftSet` and `NrcShiftTypeShiftSet`.

Function

```
void NrcShiftSetDebug(NRC_SHIFT_SET ss, int indent, FILE *fp);
```

produces a debug print of `ss` onto `fp` with the given indent, as described in Section 3.2.

3.7.3. Shift-set sets

A shift-set set is a set of shift-sets. To make a shift-set set, call

```
NRC_SHIFT_SET_SET NrcShiftSetSetMake(char *name);
void NrcShiftSetSetAddShiftSet(NRC_SHIFT_SET_SET sss, NRC_SHIFT_SET ss);
```

To retrieve the name and the shift-sets, call

```
char *NrcShiftSetSetName(NRC_SHIFT_SET_SET sss);
int NrcShiftSetSetShiftSetCount(NRC_SHIFT_SET_SET sss);
NRC_SHIFT_SET NrcShiftSetSetShiftSet(NRC_SHIFT_SET_SET sss, int i);
```

in the usual way.

Function

```
NRC_SHIFT_SET NrcShiftSetSetStartingShiftSet(NRC_SHIFT_SET_SET sss);
```

returns a shift-set containing the first shift from each shift-set of `sss`. This helps when constructing values for the `starting_ss` parameter of `NrcConstraintMake` (Section 3.9.8). If any of the shift-sets of `sss` is empty, `NrcShiftSetSetStartingShiftSet` aborts. Function

```
void NrcShiftSetSetDebug(NRC_SHIFT_SET_SET sss, int indent, FILE *fp);
```

produces a debug print of `sss` onto `fp`, as explained in Section 3.2.

3.8. Workers

A *worker* is one person capable of filling shifts. The term ‘worker’ has been preferred to ‘nurse’ because it is more general, and to ‘employee’ because it is shorter.

3.8.1. Workers

To create a worker with a given name, call

```
NRC_WORKER NrcWorkerMake(NRC_INSTANCE ins, char *name);
```

The new object is added to `ins` and returned. The basic attributes may be retrieved by

```
NRC_INSTANCE NrcWorkerInstance(NRC_WORKER w);
char *NrcWorkerName(NRC_WORKER w);
```

as usual. There is also

```
char *NrcWorkerConvertedName(NRC_WORKER w);
```

This returns the name that will be used to identify `w` in the converted instances and solutions. This could be just `NrcWorkerName(w)`, but it could also be the value of the `worker_word` parameter of `NrcInstanceMake` followed by a number, if NRC decides that `NrcWorkerName(w)` is not suitable as it stands (if it begins with a digit, or is very long).

The author would have preferred to omit this function. But sometimes one wants to create an entity with the name of worker `w` as part of its name, and then `NrcWorkerConvertedName(w)` is best, not `NrcWorkerName(w)`, since NRC does not convert the names of other entities.

Function

```
int NrcWorkerIndex(NRC_WORKER w);
```

which returns the index of `w` in the instance (the number of previously created workers), and

```
NRC_WORKER_SET NrcWorkerSingletonWorkerSet(NRC_WORKER w);
```

which returns a worker-set (Section 3.8.2) containing just `w`.

To say that a worker wants a particular shift, shift-set, or day off, call

```
extern void NrcWorkerAddShiftOff(NRC_WORKER w, NRC_SHIFT s, NRC_PENALTY p);
extern void NrcWorkerAddShiftSetOff(NRC_WORKER w, NRC_SHIFT_SET ss,
  NRC_PENALTY p);
extern void NrcWorkerAddDayOff(NRC_WORKER w, NRC_DAY d, NRC_PENALTY p);
```

Requesting a shift-set or day off means requesting that no shift of that shift-set or day be assigned. The `p` parameters give the penalty for failing to satisfy the request. These penalties may differ from one request to another, even for the same worker. Similarly, to say that a worker wants a particular shift, shift-set, or day on, call

```
void NrcWorkerAddShiftOn(NRC_WORKER w, NRC_SHIFT s, NRC_PENALTY p);
void NrcWorkerAddShiftSetOn(NRC_WORKER w, NRC_SHIFT_SET ss, NRC_PENALTY p);
void NrcWorkerAddDayOn(NRC_WORKER w, NRC_DAY d, NRC_PENALTY p);
```

Requesting a shift-set or day on means requesting that some shift of that shift-set or day be assigned, without caring which. By calling `NrcShiftAddPreassignment` (Section 3.7.1), a resource may also be preassigned to a shift—essentially an unbreakable shift-on request. Also,

```
void NrcWorkerAddStartDay(NRC_WORKER w, NRC_DAY d, NRC_PENALTY p);
void NrcWorkerAddEndDay(NRC_WORKER w, NRC_DAY d, NRC_PENALTY p);
```

say that `w` is only available starting at day `d`, or ending at day `d`. The given penalties apply to every assignment before the start day or after the end day.

Some models include *history*: information about what a worker did before the current instance began. NRC offers these two functions for handling a worker's history:

```
void NrcWorkerAddHistory(NRC_WORKER w, char *name, int value);
bool NrcWorkerRetrieveHistory(NRC_WORKER w, char *name, int *value);
```

`NrcWorkerAddHistory` associates `value` with `name` within `w`, and `NrcWorkerRetrieveHistory` sets `*value` to the value associated with `name` within `w`, or returns `false` if there is no value associated with `name`. For example,

```
NrcWorkerAddHistory(w, "WeekendsWorked", 10);
```

says that `w` has worked 10 weekends before the current instance begins.

NRC does not understand what the names mean or use history itself; it simply stores it so that the user can retrieve it later when generating constraints (Section 3.9.10).

Although only integer values may be stored, it is easy to get around this. For example, the leading model which includes history mainly uses integers, but it does include one shift type. This may be stored as the index of the shift type in the instance.

Function

```
void NrcWorkerDebug(NRC_WORKER w, int indent, FILE *fp);
```

produces a debug print of `w` onto `fp`, as explained in Section 3.2.

3.8.2. Worker-sets

A worker-set is a set of workers. To create a new, empty worker-set, call

```
NRC_WORKER_SET NrcWorkerSetMake(NRC_INSTANCE ins, char *name);
```

where `name` is a unique name for the worker-set (it may not be `NULL`). To retrieve these two attributes, call

```
NRC_INSTANCE NrcWorkerSetInstance(NRC_WORKER_SET ws);
char *NrcWorkerSetName(NRC_WORKER_SET ws);
```

To add a worker to a worker-set, call

```
void NrcWorkerSetAddWorker(NRC_WORKER_SET ws, NRC_WORKER w);
```

A worker-set may contain any number of workers, and a worker may lie in any number of worker-sets. There is also

```
void NrcWorkerSetAddWorkerSet(NRC_WORKER_SET ws1, NRC_WORKER_SET ws2);
```

which adds the workers of `ws2` to `ws1`.

To visit the workers of a worker-set, call

```
int NrcWorkerSetWorkerCount(NRC_WORKER_SET ws);
NRC_WORKER NrcWorkerSetWorker(NRC_WORKER_SET ws, int i);
```

with the first worker having index 0 as usual. However, the workers are not stored in the order they are added; they are stored in order of increasing `NrcWorkerIndex`. This is done to facilitate comparisons between worker sets to see whether they have the same workers. There is also

```
    bool NrcWorkerSetContainsWorker(NRC_WORKER_SET ws, NRC_WORKER w);
```

This returns `true` when `ws` contains `w`. And there is also

```
    bool NrcWorkerSetRetrieveWorker(NRC_WORKER_SET ws, char *name,
      NRC_WORKER *w);
```

If `ws` contains a worker with the given name, it sets `*w` to one such worker and returns `true`. Otherwise it sets `*w` to `NULL` and returns `false`.

There is nothing to prevent the same worker from being added twice. Function

```
    bool NrcWorkerSetHasNoDuplicates(NRC_WORKER_SET ws);
```

returns `true` when this has not occurred in `ws`.

Functions

```
    bool NrcWorkerSetEqual(NRC_WORKER_SET ws1, NRC_WORKER_SET ws2);
    bool NrcWorkerSetDisjoint(NRC_WORKER_SET ws1, NRC_WORKER_SET ws2);
    bool NrcWorkerSetSubset(NRC_WORKER_SET ws1, NRC_WORKER_SET ws2);
```

return `true` when `ws1`'s set of workers is equal to `ws2`'s (the name, and the order in which the workers were added, may differ), when `ws1` and `ws2` have no workers in common, and when every worker in `ws1` is also in `ws2`.

In principle, NRC should offer the usual set operations on worker sets. At present, only one is implemented:

```
    NRC_WORKER_SET NrcWorkerSetComplement(NRC_WORKER_SET ws);
```

It returns a worker set containing the workers not in `ws`. Care is needed because this function uses a particular convention for naming these worker sets, which is to add a '`!`' at the start of the name of the uncomplemented set (the 'complement name'). Here are the details.

If `ws` contains all workers, the result is `NrcInstanceEmptyWorkerSet(ins)`. And if `ws` is empty, then the result is `NrcInstanceStaffing(ins)`. In these cases names do not matter.

If the name of `ws` does not begin with '`!`', then `NrcWorkerSetComplement` tries to retrieve a worker set with the complement name from the instance. If it succeeds, it assumes that this is the complement and returns it. If it fails, it builds the complement worker-set worker by worker and adds it, with the complement name, to the instance.

If the name of `ws` begins with '`!`', then `NrcWorkerSetComplement` tries to retrieve a worker set with the same name minus the '`!`' from the instance. If it fails, it aborts. Otherwise it assumes that this is the complement and returns it.

Function

```
    void NrcWorkerSetDebug(NRC_WORKER_SET ws, int indent, FILE *fp);
```

produces a debug print of `ws` onto `fp`, as explained in Section 3.2.

NRC will abort if an attempt is made to create two worker-sets with the same name. Most worker-sets represent skills and contracts (Section 3.3.6), and name clashes are easily avoided there. The result of `NrcWorkerSingletonWorkerSet(w)` is created the first time it is called for;

its name is `w`'s name. NRC also creates its own worker-sets, one with name `"RG:All"` holding all workers, and others to do with penalizing the assignment of workers to shifts they are not qualified for. These last have obscure names that do not clash with each other and are unlikely to clash with others. In short, you can forget about worker-set name clashes until you get one.

Worker-sets have a useful property: they can be used (passed to worker constraints, for example) before workers are added to them. This applies to all worker-sets: the staffing worker-set (holding all workers), contract and skill worker-sets, whatever. As long as the workers are added eventually, before `NrcArchiveWrite`, everything works; but see Section 5.4 for a caveat.

3.8.3. Worker-set sets

A *worker-set set* is a set of worker-sets. To define one, call functions

```
NRC_WORKER_SET_SET NrcWorkerSetSetMake(NRC_INSTANCE ins);
NRC_INSTANCE NrcWorkerSetSetInstance(NRC_WORKER_SET_SET wss);
void NrcWorkerSetSetAddWorkerSet(NRC_WORKER_SET_SET wss, NRC_WORKER_SET ws);
```

in the usual way. To visit the elements of a worker-set set, call functions

```
int NrcWorkerSetSetWorkerSetCount(NRC_WORKER_SET_SET wss);
NRC_WORKER_SET NrcWorkerSetSetWorkerSet(NRC_WORKER_SET_SET wss, int i);
```

The first element has index 0 as usual. There is also

```
bool NrcWorkerSetSetRetrieveWorkerSet(NRC_WORKER_SET_SET wss,
  char *name, NRC_WORKER_SET *ws);
```

If `wss` contains a worker-set with the given name, this function sets `*ws` to that worker-set and returns `true`. Otherwise it sets `*ws` to `NULL` and returns `false`. Function

```
void NrcWorkerSetSetDebug(NRC_WORKER_SET_SET wss, int indent, FILE *fp);
```

produces a debug print of `wss` onto `fp`, as explained in Section 3.2.

3.8.4. Worker-set trees

A *worker-set tree* is a tree whose nodes are sets of worker-sets from a common instance `ins`, subject to the following conditions:

- The worker-sets that share any given node contain the same workers;
- The root of the tree contains `NrcInstanceStaffing(ins)`, the set of all workers;
- The worker-sets of a node are subsets of the worker-sets of its parent, if any;
- The worker-sets of sibling nodes are disjoint.

This structure turns out to be useful when disentangling cover requirements for overlapping skills, although the preferred approach at present is to leave them entangled and use limit resources constraints. To create a worker-set tree, call

```
NRC_WORKER_SET_TREE NrcWorkerSetTreeMake(NRC_INSTANCE ins);
```

The result has one node, containing one worker-set: `NrcInstanceStaffing(ins)`. To add a worker-set to a tree, call

```
bool NrcWorkerSetTreeAddWorkerSet(NRC_WORKER_SET_TREE wst,
  NRC_WORKER_SET ws, NRC_WORKER_SET *incompatible_ws);
```

This will search `wst` for the unique appropriate place to insert `ws`, returning `true` and inserting `ws` there if that place exists, and returning `false` and not inserting `ws` otherwise. In the latter case, `*incompatible_ws` is set to an incompatible worker-set, one which prevents insertion of `ws` because `ws` and `*incompatible_ws` are not disjoint, and nor is either a subset of the other.

To visit the worker-sets stored in the root of worker-set tree `wst`, call

```
int NrcWorkerSetTreeRootWorkerSetCount(NRC_WORKER_SET_TREE wst);
NRC_WORKER_SET NrcWorkerSetTreeRootWorkerSet(NRC_WORKER_SET_TREE wst,
  int i);
```

as usual. To visit the children of `wst`, call

```
int NrcWorkerSetTreeChildCount(NRC_WORKER_SET_TREE wst);
NRC_WORKER_SET_TREE NrcWorkerSetTreeChild(NRC_WORKER_SET_TREE wst,
  int i);
```

in the usual way. The tree lies in its instance's arena and will be freed when its instance is freed. Finally,

```
void NrcWorkerSetTreeDebug(NRC_WORKER_SET_TREE wst, int indent, FILE *fp);
```

produces a debug print of `wst` onto `fp` with the given indent.

3.9. Constraints

A *constraint* is a rule which constrains a solution. If the rule is violated by some solution, a cost is added to the solution's cost. The cost of a solution is the total cost of all constraint violations.

A constraint may be *hard*, meaning that any cost is added to a total called the *hard cost*, or *soft*, meaning that it is added to a total called the *soft cost*. So the cost of a solution is actually this pair of values. A solution with non-zero hard cost is often considered infeasible.

In nurse rostering there are two kinds of constraints: *demand constraints* (often called *cover constraints*), which specify the numbers and skills of workers needed by shifts, and *worker constraints*, which give rules that the timetables of individual workers must follow: a limit on the number of shifts worked, no morning shift on the day after a night shift, and so on. Both kinds are described in this section, although in the NRC model they are not closely related.

3.9.1. Penalties and costs

The cost of a violation of a constraint is a function of two things: the *degree of violation* of the constraint, also called the *deviation*, and the *penalty* associated with the constraint.

Most constraints are not simply violated or not; rather, they are violated to some degree. For example, if some constraint requires worker `Smith` to work at most 18 shifts, the degree of violation is the amount by which the number of shifts `Smith` works exceeds 18, or 0 if it does not. The degree of violation is always a non-negative integer.

The *penalty* is a triple of values: `hard`, a Boolean value which is `true` when the constraint is hard, and `false` when it is soft (in NRC, every constraint can be hard or soft); `weight`, a non-negative integer; and `cost_fn`, a value of type

```
typedef enum {
  NRC_COST_FUNCTION_STEP,
  NRC_COST_FUNCTION_LINEAR,
  NRC_COST_FUNCTION_QUADRATIC
} NRC_COST_FUNCTION;
```

Let the degree of violation (a non-negative number) be x. When $x = 0$, the cost is 0. When $x > 0$, the cost depends on the cost function. When `cost_fn` is `NRC_COST_FUNCTION_STEP` it is w, the penalty weight. When `cost_fn` is `NRC_COST_FUNCTION_LINEAR` (the usual value) the cost is wx. When `cost_fn` is `NRC_COST_FUNCTION_QUADRATIC` the cost is wx^2. This cost is added to the total hard cost or soft cost, depending on whether `hard` is `true` or `false`.

A penalty is represented by a non-`NULL` value of type `NRC_PENALTY`. It would ordinarily be created by a function called `NrcPenaltyMake`, but a briefer name has been chosen in this case:

```
NRC_PENALTY NrcPenalty(bool hard, int weight,
  NRC_COST_FUNCTION cost_fn, NRC_INSTANCE ins);
```

A negative `weight` causes `NrcPenalty` to abort, while if `weight > 1000`, it is silently reduced to `1000` (the largest legal XESTT weight). Also, if `weight == 0`, `hard` is silently set to `false` and `cost_fn` to `NRC_COST_FUNCTION_LINEAR`. To create a penalty with weight zero, it may be easier to call `NrcInstanceZeroPenalty` (Section 3.3.1) than to make a fresh one using `NrcPenalty`.

The attributes of a penalty may be retrieved by calls to

```
bool NrcPenaltyHard(NRC_PENALTY p);
int NrcPenaltyWeight(NRC_PENALTY p);
NRC_COST_FUNCTION NrcPenaltyCostFn(NRC_PENALTY p);
```

as usual. There is also

```
bool NrcPenaltyEqual(NRC_PENALTY p1, NRC_PENALTY p2);
```

which returns `true` when `p1` and `p2` have equal attributes, and

```
bool NrcPenaltyLessThan(NRC_PENALTY p1, NRC_PENALTY p2);
```

which returns `true` when `p1` is less than `p2`. `NrcPenaltyLessThan` is somewhat complicated. It aborts if `p1` and `p2` have different cost functions. It returns `true` when their hardnesses differ and `p1`'s is soft, and when their hardnesses are equal and `p1`'s weight is less than `p2`'s.

The sum of two penalties is returned by

```
NRC_PENALTY NrcPenaltyAdd(NRC_PENALTY p1, NRC_PENALTY p2,
  NRC_INSTANCE ins);
```

If either `p1` or `p2` has weight 0, `NrcPenaltyAdd` returns the other. Otherwise, if one of `p1` and `p2` is hard and the other is soft, then `NrcPenaltyAdd` returns the hard one. This is arguably not exact, but the inexactness does not matter. Otherwise, if the cost functions differ, `NrcPenaltyAdd` aborts. Otherwise, the result is a new penalty object, stored in `ins`'s arena, whose hardness and cost function are those of `p1` and `p2`, and whose weight is the sum of the weights of `p1` and `p2`, reduced as usual to 1000 if necessary.

To help with debugging, functions

```
char *NrcCostFnShow(NRC_COST_FUNCTION cost_fn);
char *NrcPenaltyShow(NRC_PENALTY p);
```

display a cost function or penalty as a string, for example `"step"` and `"h10q"`. `NrcPenaltyShow` begins with `"h"` or `"s"`, follows with the weight, and ends with `"s"`, `"q"`, or nothing, the latter meaning `NRC_COST_FUNCTION_LINEAR`.

3.9.2. Bounds

A *bound* is a set of integer minimum or maximum limits, together with penalties which are to be applied when the value of some quantity (not specified by the bound itself) falls short of a minimum limit or exceeds a maximum limit. Together, these define a function which maps each non-negative value of the quantity to a non-negative cost.

For several good reasons, bounds are not as general as they could be. For example, they do not implement arbitrary piecewise linear functions. Instead, they focus on things needed in practice. To create a bound which initially limits nothing (defining the zero function), call

```
NRC_BOUND NrcBoundMake(NRC_INSTANCE ins);
```

The instance may be retrieved by

```
NRC_INSTANCE NrcBoundInstance(NRC_BOUND b);
```

To add a minimum, maximum, or preferred limit to a bound, call

```
bool NrcBoundAddMin(NRC_BOUND b, int min_value, bool allow_zero,
  NRC_PENALTY below_min_penalty);
bool NrcBoundAddMax(NRC_BOUND b, int max_value,
  NRC_PENALTY above_max_penalty);
bool NrcBoundAddPreferred(NRC_BOUND b, int preferred_value,
  NRC_PENALTY below_preferred_penalty,
  NRC_PENALTY above_preferred_penalty);
```

Each of these functions can be called at most once on a given bound `b`. Each returns `true` when the limit is consistent with all other limits previously added to `b`.

`NrcBoundAddMin` says that if the value `x` of the quantity being limited is below `min_value`, then penalty `below_min_penalty` applies to `min_value - x`. When `allow_zero` is `true`, if `x` is 0, then, as a special case, no penalty applies.

`NrcBoundAddMax` says that if the value `x` of the quantity being limited is above `max_value`, then penalty `above_max_penalty` applies to `x - max_value`.

`NrcBoundAddPreferred` says that if the value `x` is below `preferred_value`, then penalty `below_preferred_penalty` applies to `preferred_value - x`, while if it is above `preferred_value`, then `above_preferred_penalty` applies to `x - preferred_value`.

When there is both a minimum value and a maximum value, `false` is returned if the maximum value is less than the minimum value.

When there is both a minimum and a preferred value, `false` is returned if the preferred value is smaller than the minimum value. The preferred penalty applies from the preferred value exclusive down to the minimum value inclusive. This interval is empty when the preferred value equals the minimum value. Below the minimum value, only the minimum penalty applies.

When there is both a maximum and a preferred value, `false` is returned if the preferred value is larger than the maximum value. The preferred penalty applies from the preferred value exclusive up to the maximum value inclusive. This interval is empty when the preferred value equals the maximum value. Above the maximum value, only the maximum penalty applies.

A preferred limit is not just a minimum limit plus a maximum limit; it applies with lower priority than a minimum or maximum limit, as can be seen when its value is equal to a minimum or maximum limit. This is one of the good reasons why the bound type is not more general.

There is one last rule. When there is both a preferred limit and a maximum limit, both penalties must be linear, and the weight of the maximum penalty must be substantially larger than the weight of the preferred penalty (for how much larger, see below.) As usual, `false` is returned when these conditions do not hold. A corresponding rule applies when there is both a preferred limit and a minimum limit.

This rule is made because, above the maximum limit, both constraints will be violated in XESTT, whereas in a bound, only the maximum penalty applies. (This is to agree with the Curtois original instances.) Accordingly, the weight of the maximum penalty must be reduced by the weight of the preferred penalty, and a constant adjustment must be added, using a constraint with a step cost function. The Appendix to this section has the details.

For convenience, there are also functions that make a bound object and add one limit to it:

```
NRC_BOUND NrcBoundMakeMin(int min_value, bool allow_zero,
  NRC_PENALTY below_min_penalty, NRC_INSTANCE ins);
NRC_BOUND NrcBoundMakeMax(int max_value,
  NRC_PENALTY above_max_penalty, NRC_INSTANCE ins);
NRC_BOUND NrcBoundMakePreferred(int preferred_value,
  NRC_PENALTY below_preferred_penalty,
  NRC_PENALTY above_preferred_penalty, NRC_INSTANCE ins);
```

The first limit cannot be inconsistent, so these functions do not need to return a Boolean result. Further limits may be added as usual. And functions

```
bool NrcBoundMin(NRC_BOUND b, int *min_value, bool *allow_zero,
  NRC_PENALTY *below_min_penalty);
bool NrcBoundMax(NRC_BOUND b, int *max_value,
  NRC_PENALTY *above_max_penalty);
bool NrcBoundPreferred(NRC_BOUND b, int *preferred_value,
  NRC_PENALTY *below_preferred_penalty,
  NRC_PENALTY *above_preferred_penalty);
```

return `true` when a minimum, maximum, or preferred limit has been added to `b`, setting the other parameters to its attributes if so. Finally, to display a bound for debugging there is

```
char *NrcBoundShow(NRC_BOUND b);
```

The result is stored in static memory and will be overwritten by the next call to `NrcBoundShow`.

Appendix. Here is what needs to be done when there is both a preferred limit and a maximum limit. We assume that both penalties are linear, and that either both are hard or both are soft. The desired penalty function is

$$\begin{aligned} f(x) &= 0 && x \le x_1 \\ &= w_1(x - x_1) && x_1 < x \le x_2 \\ &= w_2(x - x_2) && x_2 < x \end{aligned}$$

where x_1 is the preferred limit, w_1 is the penalty weight for exceeding x_1, x_2 is the maximum limit, w_2 is the penalty weight for exceeding x_2, and x, x_1, and x_2 are integers.

We can't just use the obvious two constraints, because the cost will be $w_1(x - x_1) + w_2(x - x_2)$ for $x_2 < x$, not $w_2(x - x_2)$. Our first attempt at a fix is to reduce w_2 by w_1 to compensate, giving

$$\begin{aligned} g_1(x) &= 0 && x \le x_1 \\ &= w_1(x - x_1) && x_1 < x \le x_2 \\ &= w_1(x - x_1) + (w_2 - w_1)(x - x_2) && x_2 < x \end{aligned}$$

This is implementable with two constraints. The first two parts are correct, but the third is

$$\begin{aligned} w_1(x - x_1) + (w_2 - w_1)(x - x_2) &= w_1x - w_1x_1 + w_2(x - x_2) - w_1x + w_1x_2 \\ &= w_2(x - x_2) + w_1(x_2 - x_1) \end{aligned}$$

which differs from the needed $w_2(x - x_2)$ by the constant value $w_1(x_2 - x_1)$.

A constant can be compensated for using a constraint with a step cost function; but $w_1(x_2 - x_1)$ is positive, so the weight would need to be negative, which is not allowed. To fix this problem, we start the second constraint from $x_2 + 1$ rather than x_2:

$$\begin{aligned} g_2(x) &= 0 && x \le x_1 \\ &= w_1(x - x_1) && x_1 < x \le x_2 + 1 \\ &= w_1(x - x_1) + (w_2 - w_1)(x - (x_2 + 1)) && x_2 + 1 < x \end{aligned}$$

Now the third part is

$$\begin{aligned} w_1(x - x_1) + (w_2 - w_1)(x - (x_2 + 1)) &= w_1x - w_1x_1 + w_2x - w_2x_2 - w_2 - w_1x + w_1x_2 + w_1 \\ &= w_2(x - x_2) + w_1(x_2 - x_1 + 1) - w_2 \end{aligned}$$

which gives a negative constant as desired, provided w_2 is sufficiently large. This leads to

$$\begin{aligned} g_3(x) &= 0 && x \le x_1 \\ &= w_1(x - x_1) && x_1 < x \le x_2 + 1 \\ &= w_1(x - x_1) + (w_2 - w_1)(x - (x_2 + 1)) + (w_2 - w_1(x_2 - x_1 + 1)) && x_2 + 1 < x \end{aligned}$$

However, there is now a problem at $x = x_2 + 1$: we have $f(x_2 + 1) = w_2(x_2 + 1 - x_2) = w_2$, whereas $g_3(x_2 + 1) = w_1(x_2 + 1 - x_1)$. But this can be fixed by including $x_2 + 1$ in the step, giving

$$\begin{aligned} g_4(x) &= 0 && x \le x_1 \\ &= w_1(x - x_1) && x_1 < x \le x_2 \\ &= w_1(x - x_1) + (w_2 - w_1(x_2 - x_1 + 1)) && x_2 < x \le x_2 + 1 \\ &= w_1(x - x_1) + (w_2 - w_1(x_2 - x_1 + 1)) + (w_2 - w_1)(x - (x_2 + 1)) && x_2 + 1 < x \end{aligned}$$

This is implementable by three constraints: a linear constraint with maximum limit x_1 and weight w_1; a step constraint with maximum limit x_2 and weight $w_2 - w_1(x_2 - x_1 + 1)$; and a linear constraint with maximum limit $x_2 + 1$ and weight $w_2 - w_1$.

As a final check, we verify that $g_4(x) = f(x)$. For $x \le x_1$, both functions are 0. For $x_1 < x \le x_2$, both functions are $w_1(x - x_1)$. For $x_2 < x \le x_2 + 1$, that is, for $x = x_2 + 1$, we have $f(x) = f(x_2 + 1) = w_2(x_2 + 1 - x_2) = w_2$, while $g_4(x) = w_1(x_2 + 1 - x_1) + (w_2 - w_1(x_2 - x_1 + 1)) = w_2$. Finally, for $x_2 + 1 < x$, we have $f(x) = w_2(x - x_2)$, while

$$\begin{aligned} g_4(x) &= w_1(x - x_1) + (w_2 - w_1(x_2 - x_1 + 1)) + (w_2 - w_1)(x - (x_2 + 1)) \\ &= w_1x - w_1x_1 + w_2 - w_1x_2 + w_1x_1 - w_1 + w_2(x - x_2) - w_2 - w_1x + w_1x_2 + w_1 \\ &= w_2(x - x_2) \end{aligned}$$

So there we are.

There are a few obvious special cases. When $x_1 = x_2$, replacing x_1 by x_2 in $f(x)$ gives

$$\begin{aligned} f(x) &= 0 && x \le x_2 \\ &= w_1(x - x_2) && x_2 < x \le x_2 \\ &= w_2(x - x_2) && x_2 < x \end{aligned}$$

The second case is empty, so this reduces to a single constraint; the preferred limit can be ignored. This is true for all penalty functions. When the weight given above for the step constraint is negative, that constraint cannot be implemented and this conversion fails. When it is 0 the step constraint can be omitted as usual.

Essentially the same analysis applies when there is both a minimum limit and a preferred limit. Assume for now that `allow_zero` is not requested. The desired penalty function is

$$\begin{aligned} f(x) &= 0 && x \ge x_1 \\ &= w_1(x_1 - x) && x_1 > x \ge x_2 \\ &= w_2(x_2 - x) && x_2 > x \end{aligned}$$

where x_1 is the preferred limit, w_1 is the penalty weight for falling short of x_1, x_2 is the minimum limit, w_2 is the penalty weight for falling short of x_2, and x, x_1, and x_2 are integers. Function

$$\begin{aligned} h_4(x) &= 0 && x \ge x_1 \\ &= w_1(x_1 - x) && x_1 > x \ge x_2 \\ &= w_1(x_1 - x) + (w_2 - w_1(x_1 - x_2 + 1)) && x_2 > x \ge x_2 - 1 \\ &= w_1(x_1 - x) + (w_2 - w_1(x_1 - x_2 + 1)) + (w_2 - w_1)((x_2 - 1) - x) && x_2 - 1 > x \end{aligned}$$

does the trick. It is implementable by three constraints: a linear constraint with minimum limit x_1 and weight w_1; a step constraint with minimum limit x_2 and weight $w_2 - w_1(x_1 - x_2 + 1)$; and a linear constraint with minimum limit $x_2 - 1$ and weight $w_2 - w_1$.

Constraints with 0 limits are dropped before applying this formula. Even then, $x_2 - 1 = 0$ is possible, in which case the third constraint does nothing and is omitted.

It does not matter whether or not the preferred limit requests `allow_zero`, because it is overridden by the minimum limit. If the minimum limit requests `allow_zero`, that is easily handled by requesting `allow_zero` in each generated constraint.

A different approach to the general problem of multiple limits would be to extend XESTT to allow cost functions which are arbitrary piecewise step, linear and quadratic functions. However, that would substantially increase the complexity of XESTT for the sake of an uncommon case which adds very little, considered from the point of view of the institution being modelled.

In instance file `Musa.ros`, lines 306 and 453 form a case where there is a minimum value and a preferred value, with $x_1 = 3$, $w_1 = 5$, $x_2 = 1$, and $w_2 = 7$. This fails the formula given above, but it is somewhat illogical, because it gives values $f(3) = 0$, $f(2) = 5$, $f(1) = 10$, and $f(0) = 7$. Rather than reject the instance, NRConv chooses to ignore the minimum limit (with a warning message) and carry on. `Musa.ros` contains 14 essentially identical instances of this problem.

3.9.3. Demands

A *demand* is an object of type `NRC_DEMAND`, defining a request for one worker to be assigned to an unspecified shift, and the penalties to apply when the assignment is defective.

In this section, the term *worker assignment* will denote the assignment, or non-assignment, of a worker to fulfill the request represented by a demand object. If there are W workers, there are $W + 1$ distinct worker assignments: one for each worker, and one representing non-assignment. A demand object represents a function that associates a penalty with each worker assignment.

Adding a demand to a shift is done separately, by `NrcShiftAddDemand` (Section 3.7.1). One demand may be added any number of times to one shift, or several shifts. Demand objects are immutable after creation, so doing it this way is safe. It helps the implementation to reduce the length of the generated file. The penalties apply independently to each occurrence of the demand object. To apply penalties across sets of demands, use demand constraints (Section 3.9.5).

Some models specify an optimum number of workers but no maximum. For them, one must decide on some maximum (twice the optimum, perhaps), and add that many demands. This is because XESTT requires each event to contain a definite fixed number of event resources.

Creation of a demand object begins with a call to

```
NRC_DEMAND NrcDemandMakeBegin(NRC_INSTANCE ins);
```

Initially, every worker assignment has a penalty with weight 0; no distinction is made between this and having no penalty at all. Then comes a sequence of calls to *penalizer functions* which associate penalties with worker assignments; we'll return to them in a moment. After they are done, creation is ended, including marking `d` as immutable, by a compulsory call to

```
void NrcDemandMakeEnd(NRC_DEMAND d);
```

Any subsequent calls to `d`'s penalizer functions, or to `NrcDemandMakeEnd`, will abort.

We turn now to the penalizer functions. There are three of them:

```
void NrcDemandPenalizeNonAssignment(NRC_DEMAND d,
  NRC_PENALTY_TYPE ptype, NRC_PENALTY p);
void NrcDemandPenalizeWorkerSet(NRC_DEMAND d, NRC_WORKER_SET ws,
  NRC_PENALTY_TYPE ptype, NRC_PENALTY p);
void NrcDemandPenalizeNotWorkerSet(NRC_DEMAND d, NRC_WORKER_SET ws,
  NRC_PENALTY_TYPE ptype, NRC_PENALTY p);
```

`NrcDemandPenalizeNonAssignment` associates `p` with non-assignment, so that `p` is incurred if no assignment is made to the demand. `NrcDemandPenalizeWorkerSet` associates `p` with each worker assignment of a worker from `ws`, so that `p` is incurred if an assignment of a worker from `ws` is made to the demand. `NrcDemandPenalizeNotWorkerSet` associates `p` with each worker assignment of a worker not from `ws` (but not with non-assignment), so that `p` is incurred if an assignment of a worker not from `ws` is made to the demand. As usual, worker sets whose workers will be added later may be passed to these functions.

In all three functions, parameter `ptype` has type

```
typedef enum {
  NRC_PENALTY_REPLACE,
  NRC_PENALTY_ADD,
  NRC_PENALTY_UNIQUE
} NRC_PENALTY_TYPE;
```

and says how to combine the new penalty with the existing penalty for each affected worker assignment. If `ptype` is `NRC_PENALTY_REPLACE`, the new penalty replaces the existing one; if `ptype` is `NRC_PENALTY_ADD`, the new and existing penalties are added using `NrcPenaltyAdd` (Section 3.9.1); and if `ptype` is `NRC_PENALTY_UNIQUE`, the existing penalty must have weight 0 (otherwise the program aborts) and is replaced. `NRC_PENALTY_UNIQUE` does not mean that the new penalty cannot be replaced later.

For example, suppose that the penalty for non-assignment is `p`, and the penalty for assigning a worker from outside worker set `ws1` is `p1`. Then the appropriate calls are

```
NrcDemandPenalizeNonAssignment(d, NRC_PENALTY_UNIQUE, p);
NrcDemandPenalizeNotWorkerSet(d, ws1, NRC_PENALTY_UNIQUE, p1);
```

Or suppose we prefer that the demand not be assigned at all. The call is

```
NrcDemandPenalizeWorkerSet(d, NrcInstanceStaffing(ins),
  NRC_PENALTY_UNIQUE, p);
```

This says that any assignment of an actual worker attracts penalty `p`.

The First International Nurse Rostering Competition associates a penalty with each contract. This is to be applied whenever any nurse subject to that contract is unpreferred. Suppose there are two contracts, one for nurses `ws1` with penalty `p1`, the other for nurses `ws2` with penalty `p2`. Then if the preferred worker set for some demand is `ws`, the appropriate calls are

```
NrcDemandPenalizeNonAssignment(d, NRC_PENALTY_UNIQUE, hard_p);
NrcDemandPenalizeWorkerSet(d, ws1, NRC_PENALTY_UNIQUE, p1);
NrcDemandPenalizeWorkerSet(d, ws2, NRC_PENALTY_UNIQUE, p2);
NrcDemandPenalizeWorkerSet(d, ws, NRC_PENALTY_REPLACE, p0);
```

This first installs penalty `hard_p` for non-assignment (in INRC1 this is a hard cost of 1). It then adds a penalty for each worker in each contract, after which all worker assignments should have a penalty. But then it replaces the penalties for the preferred workers by `p0`, a zero penalty.

Calling `NrcDemandPenalizeWorkerSet` with an empty worker set does nothing (unless workers are added to the worker set later). Calling `NrcDemandPenalizeNotWorkerSet` with a worker set containing every worker also does nothing.

Calling `NrcDemandPenalizeWorkerSet` with a worker set containing every worker is the same as calling `NrcDemandPenalizeNotWorkerSet` with an empty worker set: both penalize the assignment of any worker. When this is wanted, it is best to pass the sets provided by NRC: `NrcInstanceStaffing(ins)` and `NrcInstanceEmptyWorkerSet(ins)` from Section 3.3.6. Doing that will make the generated XESTT file easier to read.

Behind the scenes, a unique name is created for each demand, and used in the generated XESTT file as part of the names of roles and event groups. The name is based on the calls made when constructing the demand. For example,

```
NA=h1+NW0=s100
```

is the name of a demand in which non-assignment has hard cost 1 (`NA=h1`) and assigning a worker not from worker set `0` has soft cost 100 (`NW0=s100`). This name is returned by

```
char *NrcDemandName(NRC_DEMAND d);
```

but this only works after `NrcDemandMakeEnd(d)` has been called.

The instance that a demand is for may be retrieved by

```
NRC_INSTANCE NrcDemandInstance(NRC_DEMAND d);
```

Its penalizers may be retrieved by

```
int NrcDemandPenalizerCount(NRC_DEMAND d);
void NrcDemandPenalizer(NRC_DEMAND d, int i, NRC_PENALIZER_TYPE *pt,
  NRC_WORKER_SET *ws, NRC_PENALTY_TYPE *ptype, NRC_PENALTY *p);
```

where `NRC_PENALIZER_TYPE` is

```
typedef enum {
  NRC_PENALIZER_NON_ASSIGNMENT,
  NRC_PENALIZER_WORKER_SET,
  NRC_PENALIZER_NOT_WORKER_SET
} NRC_PENALIZER_TYPE;
```

Its members correspond to the three functions given above. `NrcDemandPenalizer` sets `*ws` to `NULL` when `*pt` is `NRC_PENALIZER_NON_ASSIGNMENT`.

As we know, `d` holds one penalty for each worker, and one for non-assignment. Function

```
NRC_PENALTY NrcDemandWorkerPenalty(NRC_DEMAND d, NRC_WORKER w);
```

returns the penalty associated with worker `w`, or `NrcInstanceZeroPenalty` if no penalty has been associated with `w`. Passing `NULL` for `w` returns the penalty for non-assignment. It may only be called after `NrcDemandMakeEnd(d)` has been called and before conversion to XESTT.

Finally, function

```
void NrcDemandDebug(NRC_DEMAND d, int multiplicity, int indent, FILE *fp);
```

produces a debug print of `d` onto `fp` with the given indent, as explained in Section 3.2. As a convenience to some callers, if `multiplicity` is at least 2 it is included in the print.

3.9.4. Demand-sets

Demand-sets are sets of demands, defined in the usual way:

```
NRC_DEMAND_SET NrcDemandSetMake(NRC_INSTANCE ins);
void NrcDemandSetAddDemand(NRC_DEMAND_SET ds, NRC_DEMAND d);
```

There is also the trivial helper function

```
void NrcDemandSetAddDemandMulti(NRC_DEMAND_SET ds, NRC_DEMAND d,
  int multiplicity);
```

which calls `NrcDemandSetAddDemand(ds, d)` `multiplicity` times. It is quite normal for one demand object to be added to a demand set multiple times, this way or otherwise; it just means that that many identical demands are being made.

Demand-sets make it easy to build up the total requirements for one shift into a single object. There might be a minimum, preferred, and maximum number of workers required, similarly to what is represented by a bound. This function develops this idea:

```
NRC_DEMAND_SET NrcDemandSetMakeFromBound(NRC_INSTANCE ins,
  NRC_BOUND b, int count, NRC_WORKER_SET preferred_ws,
  NRC_PENALTY not_preferred_penalty);
```

It returns a new demand-set containing `count` demands. It would be too tedious to describe all the cases, but suppose the bound contains a `min_value`, `preferred_value`, and `max_value`. Then the first `min_value` demands penalize non-assignment by the penalty for being below `min_value`; the next `preferred_value - min_value` demands also penalize non-assignment, but by the penalty for being below `preferred_value`; the next `max_value - preferred_value` demands penalize assignment of a worker, by the penalty for being above `preferred_value`; and the last `count - max_value` demands also penalize assignment of a worker, but by the penalty for being above `max_value`. In addition, if `preferred_ws` is non-`NULL`, all the demands penalize the assignment of a worker not in `preferred_ws` by `not_preferred_penalty`. The penalties must all be linear, and all `allow_zero` attributes must be `false`.

For example, the following handles the Second International Nurse Rostering Competition, whose shifts have a hard minimum cover and a soft optimum (preferred) cover:

```
p1 = NrcPenalty(true, 1, NRC_COST_FUNCTION_LINEAR, ins);
p2 = NrcPenalty(false, 30, NRC_COST_FUNCTION_LINEAR, ins);
p3 = NrcPenalty(false, 0, NRC_COST_FUNCTION_LINEAR, ins);
b = NrcBoundMakeMin(min_cover, false, p1);
NrcBoundAddPreferred(b, opt_cover, p2, p3);
dms = NrcDemandSetMakeFromBound(ins, b, opt_cover * 2, ws, p1);
```

This is much easier and clearer than adding the demands individually. To add a maximum cover, for example, just add a maximum value to the bound.

The instance that a demand-set is for is returned by

```
NRC_INSTANCE NrcDemandSetInstance(NRC_DEMAND_SET ds);
```

To visit the demands of a demand-set, in the order they were added, call

```
int NrcDemandSetDemandCount(NRC_DEMAND_SET ds);
NRC_DEMAND NrcDemandSetDemand(NRC_DEMAND_SET ds, int i);
```

The same demand may be returned multiple times; there is no memory in the demand-set of how its demands were added. Function

```
void NrcDemandSetDebug(NRC_DEMAND_SET ds, int indent, FILE *fp);
```

produces a debug print of `ds` onto `fp`, as explained in Section 3.2.

3.9.5. Demand constraints

The usual way to specify the workers required by shifts is to add demands to the shifts, as just explained. However, some constraints (notably in the Curtois original instances) apply to the whole set of workers that attend some shift, or even to the set of workers that attend a set of shifts (when the constraint is on the workers that attend during some time period). In some cases these constraints can be decomposed into constraints on individual demands, but not always.

For these awkward cases an alternative approach is offered. First, specify a maximum number of workers that may be assigned to each shift, using unconstrained demands like this:

```
NrcInstanceDemandBegin(ins);
dm = NrcInstanceDemandEnd(ins);
dms = NrcDemandSetMake(ins);
NrcDemandSetAddDemandMulti(dms, dm, total_cover);
NrcShiftAddDemandSet(s, dms);
```

or equivalently like this:

```
dms = NrcDemandSetMakeFromBound(ins, NrcBoundMake(ins), total_cover,
  NULL, NULL);
NrcShiftAddDemandSet(s, dms);
```

This ensures that shift `s` can be assigned up to `total_cover` workers, without any constraint on how many assignments there should be, or on the assigned workers' skills. Second, call

```
NRC_DEMAND_CONSTRAINT NrcDemandConstraintMake(NRC_BOUND b,
  NRC_SHIFT_SET ss, NRC_WORKER_SET ws, char *name);
```

This adds a constraint to `ss`'s instance which limits the total number of demands from an arbitrary set of shifts that may be assigned workers from a given set. Functions

```
NRC_BOUND NrcDemandConstraintBound(NRC_DEMAND_CONSTRAINT dc);
NRC_SHIFT_SET NrcDemandConstraintShiftSet(NRC_DEMAND_CONSTRAINT dc);
NRC_WORKER_SET NrcDemandConstraintWorkerSet(NRC_DEMAND_CONSTRAINT dc);
char *NrcDemandConstraintName(NRC_DEMAND_CONSTRAINT dc);
```

retrieve the four attributes of demand constraint `dc`.

Parameter `b` is a bound object (Section 3.9.2) which specifies any combination of minimum, maximum, or preferred limits on the number of assignments. It may be a freshly created object with no limits when passed to `NrcDemandConstraintMake`; the limits may be added to it later.

Parameter `ss` is the set of shifts whose assignments are being constrained. For example, if the constraint is on a single shift `s`, then `ss` would be `NrcShiftSingletonShiftSet(s)`.

Parameter `ws` is the set of workers. The constraint is only interested in assignments of these workers; assignments of other workers do not influence it. If the constraint is on the total number of assignments irrespective of skill, then `ws` would be `NrcInstanceStaffing(ins)`.

Parameter `name` does not influence the meaning of the constraint; rather, it helps to identify the constraint in evaluation prints. It does not have to be unique. It should indicate what is being constrained but not the actual limits, because NRC adds that information to the name.

There is also the usual debug function,

```
void NrcDemandConstraintDebug(NRC_DEMAND_CONSTRAINT dc,
  int indent, FILE *fp);
```

which produces a debug print of `dc` onto `fp` with the given indent.

NRC demand constraints are converted to XESTT limit resources constraints, or in some cases to assign resource and prefer resources constraints, which are preferable because they constrain each event resource independently—a fact which can have practical consequences, for example for time sweep assignment algorithms. For the details, consult Section 4.3.

3.9.6. Patterns

Patterns are used in several nurse rostering models to say that some sequences of shift types—a morning shift following a night shift, for example—should be forbidden, or penalized.

Given that a worker takes at most one shift per day, the choices on each day are one of its shifts or nothing. Denote the shift types by `1`, `2`, and `3`, and denote the absence of a shift (a free day) by `0`. Using regular expression notation, an arbitrary subset of these choices is represented by enclosing them in brackets. For example, `[02]` means 'a shift of type 2 or nothing.' In source models, a *pattern* is a sequence of these *terms*, representing the choices on consecutive days. For example, `[3][1]` means 'an early shift following a night shift'.

A pattern *matches* a worker's timetable at any day where the worker has a sequence of shifts or days off, each of which matches the corresponding term of the pattern. For example, if a worker's timetable contains a shift of type `3` on day `1Wed` and a shift of type `1` on day `1Thu`, then pattern `[3][12]` matches that timetable at `1Wed`.

The empty term `[]` is allowed, but it never matches, which makes it useless in practice. On the other hand, `[0123]` says that we don't care what happens on that day, which can be useful.

NRC supports patterns. A pattern is created by

```
NRC_PATTERN NrcPatternMake(NRC_INSTANCE ins, char *name);
```

where `name` is optional (may be `NULL`). To retrieve the attributes, call

```
NRC_INSTANCE NrcPatternInstance(NRC_PATTERN p);
char *NrcPatternName(NRC_PATTERN p);
```

To add a term to a pattern, call

```
void NrcPatternAddTerm(NRC_PATTERN p, NRC_SHIFT_TYPE_SET sts,
  NRC_POLARITY po);
```

As shown, a term consists of a shift-type set and a *polarity*, of type

```
typedef enum {
  NRC_NEGATIVE,
  NRC_POSITIVE
} NRC_POLARITY;
```

These attributes will be explained shortly. To visit the terms of a pattern, call

```
int NrcPatternTermCount(NRC_PATTERN p);
void NrcPatternTerm(NRC_PATTERN p, int i, NRC_SHIFT_TYPE_SET *sts,
  NRC_POLARITY *po);
```

in the usual way. There is also

```
bool NrcPatternIsUniform(NRC_PATTERN p);
```

which returns `true` when `p` is *uniform*: when all its terms contain equal shift-type sets and equal polarities. Unwanted patterns can be implemented more efficiently when they are uniform.

Type `NRC_POLARITY` has just one operation:

```
NRC_POLARITY NrcPolarityNegate(NRC_POLARITY po);
```

which returns negative for positive and positive for negative.

A term `t` matches the timetable of worker `w` on day `d` if either `t`'s polarity is `NRC_POSITIVE` and `w` works a shift on day `d` whose type is one of the types of `t`'s shift-type set, or else `t`'s polarity is `NRC_NEGATIVE` and `w` does not work a shift on day `d` whose type is one of the types of `t`'s shift-type set. This second case includes not working at all. A pattern matches `w`'s timetable on day `d` if its first term matches on day `d`, its second term matches on the day after `d`, and so on.

A term which does not contain `0` is represented by a shift-type set containing its shift types, with polarity `NRC_POSITIVE`. A term which contains `0` is represented by a shift-type set containing the complement of the term's non-`0` shift types, with polarity `NRC_NEGATIVE`. The reader can easily verify that this does what is wanted. In particular, the don't-care term `[0123]` is represented by an empty shift-type set with negative polarity. This term always matches.

Creating a pattern does not make it unwanted: it has to be added to a worker constraint, using function `NrcConstraintAddPattern` (Section 3.9.8). The constraint holds information about the cost of violations, and which days the pattern is allowed to match with.

All patterns are stored in the instance, and are accessible by `NrcInstancePatternCount`, `NrcInstancePattern`, and `NrcInstanceRetrievePattern` (Section 3.3.8). Function

```
void NrcPatternDebug(NRC_PATTERN p, int indent, FILE *fp);
```

produces a debug print of `p` onto `fp`, as explained in Section 3.2.

3.9.7. Pattern sets

A *pattern set* is a set of patterns. To make an initially empty pattern set, call

```
NRC_PATTERN_SET NrcPatternSetMake(NRC_INSTANCE ins);
```

Functions

```
NRC_INSTANCE NrcPatternSetInstance(NRC_PATTERN_SET ps);
int NrcPatternSetIndexInInstance(NRC_PATTERN_SET ps);
```

return the pattern set's instance, and its index in the instance.

To add a pattern to a pattern set, call

```
void NrcPatternSetAddPattern(NRC_PATTERN_SET ps, NRC_PATTERN p);
```

This may be done any number of times. To visit the patterns of a pattern set, call

```
int NrcPatternSetPatternCount(NRC_PATTERN_SET ps);
NRC_PATTERN NrcPatternSetPattern(NRC_PATTERN_SET ps, int i);
```

in the usual way.

The function that makes pattern sets non-trivial is

```
NRC_PATTERN_SET NrcPatternSetReduce(NRC_PATTERN_SET ps);
```

This returns a new pattern set whose patterns match at the same points as `ps`'s patterns do, but usually using fewer patterns. To see the use for this, consider this excerpt from one of the recent Curtois and Qu instances:

```
a4,720,a1|a2|a3|a4|d1|d2|d3|d4|d5
d1,480,a1|a2|a3
d2,480,a1|a2|a3
d3,600,a1|a2|a3|a4
d4,720,a1|a2|a3|a4|d1|d2|d3|d4|d5
```

Ignoring the workload limits, this gives for each shift type (not all are shown here) a list of other shift types that may not follow it, so it defines a set of patterns. The reduced set is

```
a4|d4,720,a1|a2|a3|a4|d1|d2|d3|d4|d5
d1|d2,480,a1|a2|a3
d3,600,a1|a2|a3|a4
```

These patterns match at the same points as the originals, but there are fewer of them, leading to smaller generated instances. The function merges pairs of patterns with the same length and equal elements and polarities except at one place, where the elements must be disjoint and the polarities must be equal.

Instead of building then reducing, it may be simpler to reduce while building, by calling

```
void NrcPatternSetMergePattern(NRC_PATTERN_SET ps, NRC_PATTERN p);
```

It is like `NrcPatternSetAddPattern` except that it first tries to merge `p` into an existing pattern of `ps`, only adding it as a separate pattern as a last resort. Finally,

```
void NrcPatternSetDebug(NRC_PATTERN_SET ps, int indent, FILE *fp);
```

produces a debug print of `ps` in the usual way.

3.9.8. Worker constraints

A *worker constraint* is a constraint on the timetables of individual workers. Worker constraints all seem to have a similar form: they contain a set of shift-sets, and for each worker they determine whether the worker is busy or free during each shift-set, calculate totals of busy and free shift-sets, and assign a penalty proportional to the amount by which the totals fall short of a given minimum limit or exceed a given maximum limit. There are variations: when there is a non-zero minimum limit, in some cases a total of 0 is nevertheless not penalized; sometimes the total workload (measured in minutes, say) is limited rather than the number of shifts; some constraints are repeated along the cycle (every weekend, for example); some apply to sequences of consecutive shift-sets, others are just concerned with totals, not with ordering; and so on. Nevertheless NRC offers a single interface for all worker constraints.

A shift is *busy* for a worker when the worker works that shift. A shift is *free* for a worker when it is not busy for the worker. A shift-set is busy for a worker when at least one of its shifts is busy for the worker. A shift-set is free for a worker when it is not busy for the worker.

When a shift-set is added to a constraint, a polarity (Section 3.9.6) is added with it, saying whether the shift-set is to be treated *positively* or *negatively*. Informally, we say that a shift-set is *positive* when its associated polarity is positive, and *negative* otherwise. However, it is the usage of the shift-set within the constraint which is positive or negative, not the shift-set itself.

When a constraint is applied to a particular worker, a shift-set within it is *active* when it is positive and busy for the worker, or negative and free for the worker. Otherwise it is *inactive*. The constraint calculates the total number of active shift-sets, and compares it with the limits. If all shift-sets are positive, this constrains busy shift-sets; if all are negative, it constrains free shift-sets. Mixtures of positive and negative are legal, and useful for implementing unwanted patterns.

To create a worker constraint, initially with no shift-sets, call

```
NRC_CONSTRAINT NrcConstraintMake(NRC_INSTANCE ins, char *name,
  NRC_WORKER_SET ws, NRC_CONSTRAINT_TYPE type, NRC_BOUND bound,
  NRC_SHIFT_SET starting_ss);
```

This returns a new constraint and also adds it to `ins`. The type should really be called `NRC_WORKER_CONSTRAINT`, but `NRC_CONSTRAINT` is shorter. To retrieve the attributes, call

```
NRC_INSTANCE NrcConstraintInstance(NRC_CONSTRAINT c);
char *NrcConstraintName(NRC_CONSTRAINT c);
NRC_WORKER_SET NrcConstraintWorkerSet(NRC_CONSTRAINT c);
NRC_CONSTRAINT_TYPE NrcConstraintType(NRC_CONSTRAINT c);
NRC_BOUND NrcConstraintBound(NRC_CONSTRAINT c);
NRC_SHIFT_SET NrcConstraintStartingShiftSet(NRC_CONSTRAINT c);
```

Parameter `name` is the name given to XESTT constraints derived from this constraint. A good choice here is an informal source model description, expressed positively, that is, as what is wanted rather than what is to be avoided: `"At most 4 busy weekends"` and so on. Names mainly appear as entries in tables of defects, where there are other entries giving the details, so a short, informal phrase is best. There is no need for names to be distinct. NRC will ensure that the XESTT constraints it generates have distinct Ids, which is a different thing.

Parameter `ws` says which workers the constraint applies to. For example, `ws` could hold the workers who share a contract containing this constraint. If the constraint is for a single worker `w`, then `ws` is `NrcWorkerSingletonWorkerSet(w)` (Section 3.8.1). Worker constraints which are equal apart from `ws` are merged by `NrcArchiveWrite` into a single XESTT constraint.

Parameter `type` determines what is constrained, and has type

```
typedef enum {
  NRC_CONSTRAINT_ACTIVE,
  NRC_CONSTRAINT_CONSECUTIVE,
  NRC_CONSTRAINT_WORKLOAD
} NRC_CONSTRAINT_TYPE;
```

`NRC_CONSTRAINT_ACTIVE` means that the constraint is on the number of active shift-sets;

`NRC_CONSTRAINT_CONSECUTIVE` means that the constraint is on the number of consecutive active shift-sets;[1] and `NRC_CONSTRAINT_WORKLOAD` means that the constraint is on the total workload of the shifts of one shift-set.

Parameter `bound` says whether the constraint is a minimum limit, a minimum limit in which zero is allowed, or a maximum limit, or a combination, and includes penalties for when the limit is violated (Section 3.9.2).

Lastly, `starting_ss` repeats the constraint along the cycle. If it is `NULL`, the constraint is applied only once. If it is non-`NULL`, the distance from its first shift to each of its other shifts defines a distance along the cycle to repeat the constraint. For example, if `starting_ss` holds the first shift of each day, the constraint repeats on every day. `NrcInstanceDailyStartingShiftSet` and `NrcInstanceWeeklyStartingShiftSet` (Section 3.3.5), `NrcDaySetStartingShiftSet` (Section 3.4.3), and `NrcShiftSetSetStartingShiftSet` (Section 3.7.3) return most of the starting shift-sets needed in practice.

When `starting_ss` is used, the shift-sets added to the constraint must define only the earliest occurrence of the constraint. Some of `starting_ss`'s shifts may place the constraint at points of the cycle where some parts of it go off the end. Such shifts are legal but are ignored.

One may use `starting_ss` with the consecutive limit types, to get constraints such as a limit on the number of consecutive days worked within each four-week interval. But these never seem to occur in source models, perhaps because they are very artificial at the boundaries.

After creating the constraint, add shift-sets to it by calling

```
void NrcConstraintAddShiftSet(NRC_CONSTRAINT c,
  NRC_SHIFT_SET ss, NRC_POLARITY po);
void NrcConstraintAddShiftSetSet(NRC_CONSTRAINT c,
  NRC_SHIFT_SET_SET sss, NRC_POLARITY po);
```

any number of times, arbitrarily intermixed. This adds the shift-sets, each accompanied by a polarity, either one at a time or many at once. There is also

```
void NrcConstraintAddPattern(NRC_CONSTRAINT c, NRC_PATTERN p, NRC_DAY d);
```

A pattern is a sequence of shift-type sets with polarities, and `NrcConstraintAddPattern` simply makes the corresponding sequence of calls to `NrcConstraintAddShiftSet`, converting each shift type set `sts` into a shift set, by calling `NrcDayShiftSetFromShiftTypeSet(d, sts)` for the shift type set of the first term, and similarly using successive days for successive terms.

To visit the shift-sets and polarities added to a constraint `c`, call

```
int NrcConstraintShiftSetCount(NRC_CONSTRAINT c);
void NrcConstraintShiftSet(NRC_CONSTRAINT c, int i,
  NRC_SHIFT_SET *ss, NRC_POLARITY *po);
```

as usual. The constraint does not remember whether the shift-sets and polarities were added individually, or using shift-set sets, or using patterns.

One common form of constraint, the unwanted pattern, is already implemented:

[1]The author learned of this approach to constraining consecutive subsequences from Gerhard Post.

```
NRC_CONSTRAINT NrcUnwantedPatternConstraintMake(NRC_INSTANCE ins,
  char *name, NRC_WORKER_SET ws, NRC_PENALTY penalty, NRC_PATTERN p,
  NRC_DAY_SET starting_ds);
```

The first three parameters and the return value are as for `NrcConstraintMake`. Parameter `penalty` is the penalty to apply when the pattern is violated. The last two parameters give the unwanted pattern and the set of days on which it may begin (pass `NrcInstanceCycle(ins)` if it may begin on any day). Here `p` must contain at least one term, and `starting_ds` must contain at least one day.

Function

```
void NrcConstraintDebug(NRC_CONSTRAINT c, int indent, FILE *fp);
```

produces a debug print of `c` onto `fp`, as explained in Section 3.2.

Behind the scenes, NRConv implements an important optimization called *condensing*, which converts sets of constraints of type `NRC_CONSTRAINT_ACTIVE` into constraints of type `NRC_CONSTRAINT_CONSECUTIVE` when the shift-sets are uniform (when they repeat regularly along the cycle). When source files implement minimum and maximum limits on the number of busy days using patterns, condensing changes them back into constraints which limit the numbers directly. The implementation has been done with care and produces an exact result whenever it is applied. The new constraints have the old names with `" (condensed)"` appended.

3.9.9. Examples of worker constraints

this section is out of date, it needs a makeover

This section presents some examples of worker constraints. Many more may be found in the source code. For reference, here is the interface of `NrcConstraintMake` from Section 3.9.8:

```
NRC_CONSTRAINT NrcConstraintMake(NRC_INSTANCE ins, char *name,
  NRC_WORKER_SET ws, NRC_CONSTRAINT_TYPE type, NRC_BOUND bound,
  NRC_SHIFT_SET starting_ss);
```

Let `ins` be an NRC instance. To say that all staff should work at most one shift per day:

```
c = NrcConstraintMake(ins, "Single assignment per day",
  NrcInstanceStaffing(ins), p, NRC_LIMIT_MAX, 1,
  NrcInstanceDailyStartingShiftSet(ins));
NrcConstraintAddShiftSetSet(c,
  NrcDayShiftSetSet(NrcInstanceCycleDay(ins, 0), NRC_POSITIVE);
```

Looking along the arguments of `NrcConstraintMake`, `c` applies to all workers, has penalty `p`, maximum limit 1 (not consecutive), and repeats each day. The second statement defines the first point where it applies: to the shifts of the first day of the cycle. The shift-set set ensures that each shift is added in its own shift-set, so that the usual limit on the number of busy shift-sets becomes a limit on the number of busy shifts. It would be wrong to pass in a single shift-set containing all the shifts of the day, but NRConv works out that this is a simple case and generates a limit busy times constraint rather than a cluster busy times constraint.

To impose a maximum workload limit of 28 shifts:

```
c = NrcConstraintMake(ins, "At most 28 shifts",
  NrcInstanceStaffing(ins), p, NRC_LIMIT_MAX, 28, NULL);
NrcConstraintAddShiftSetSet(c, NrcInstanceShiftsShiftSetSet(ins),
  NRC_POSITIVE);
```

Here c applies to all workers, has penalty p, is not consecutive, and does not repeat. The second statement adds each shift, again in its own shift-set.

Making a pattern p unwanted is very easy:

```
c = NrcConstraintMake(ins, "Unwanted pattern", NrcInstanceStaffing(ins),
  p, NRC_LIMIT_MAX, NrcPatternTermCount(p) - 1,
  NrcInstanceDailyStartingShiftSet(ins));
NrcConstraintAddPattern(c, p, NrcInstanceCycleDay(ins, 0));
```

The constraint applies to all workers, has penalty p, has a maximum limit of one less than the pattern length (not consecutive), and starts afresh each day. The last line adds shift-sets defining the first occurrence of the pattern, beginning on the first day of the cycle.

NRConv offers NrcUnwantedPatternConstraintMake, its own implementation of unwanted patterns, documented near the end of Section 3.9.8. It is rather more complex than the code above, mainly because, if the pattern is uniform (has the same shift-set and the same polarity at every term) and may begin on any day, it optimizes by generating a limit active intervals constraint rather than a cluster busy times constraint which repeats on each day:

```
NRC_CONSTRAINT NrcUnwantedPatternConstraintMake(NRC_INSTANCE ins,
  char *name, NRC_WORKER_SET ws, NRC_PENALTY penalty, NRC_PATTERN p,
  NRC_DAY_SET starting_ds)
{
  NRC_CONSTRAINT res;  NRC_SHIFT_TYPE_SET sts;  NRC_POLARITY po;  int i;
  NRC_DAY d;  NRC_SHIFT_SET ss;

  MAssert(NrcPatternTermCount(p) > 0,
    "NrcUnwantedPatternConstraintMake:  empty pattern");
  MAssert(NrcDaySetDayCount(starting_ds) > 0,
    "NrcUnwantedPatternConstraintMake:  empty starting_ds");
  if( NrcDaySetDayCount(starting_ds) == NrcInstanceCycleDayCount(ins)
      && NrcPatternIsUniform(p) )
  {
    /* uniform pattern, single limit active intervals constraint */
    res = NrcConstraintMake(ins, name, ws, penalty,
      NRC_LIMIT_MAX_CONSECUTIVE, NrcPatternTermCount(p)-1, NULL);
    NrcPatternTerm(p, 0, &sts, &po);
    for( i = 0;  i < NrcInstanceCycleDayCount(ins);  i++ )
    {
      d = NrcInstanceCycleDay(ins, i);
      ss = NrcDayShiftSetFromShiftTypeSet(d, sts);
      NrcConstraintAddShiftSet(res, ss, po);
    }
  }
  else
  {
    /* non-uniform pattern, repeating cluster busy times constraint */
    res = NrcConstraintMake(ins, name, ws, penalty, NRC_LIMIT_MAX,
      NrcPatternTermCount(p)-1, NrcDaySetStartingShiftSet(starting_ds));
    NrcConstraintAddPattern(res, p, NrcDaySetDay(starting_ds, 0));
  }
  return res;
}
```

This function could be written by an NRC user; it does not use any behind-the-scenes features.

Constraints involving weekends need to know when the weekends are. One way to express this is as a day-set set, each day-set of which contains the days of one weekend, in chronological order. Such a day-set set can be built using NRC's functions for building day-sets and day-set sets, following whatever rule the format uses to define weekends, and shared by all constraints concerning weekends. Assuming that `weekends_dss` is such a day-set set, the following code places a maximum limit of 3 on the number of consecutive weekends that worker `w` can work:

```
NRC_CONSTRAINT c;  int i;  NRC_DAY_SET ds;
c = NrcConstraintMake(ins, "At most 3 consecutive weekends",
  NrcWorkerSingletonWorkerSet(w), p, NRC_LIMIT_MAX_CONSECUTIVE, 3, NULL);
for( i = 0;  i < NrcDaySetSetDaySetCount(weekends_dss);  i++ )
{
  ds = NrcDaySetSetDaySet(weekends_dss, i);
  NrcConstraintAddShiftSet(c, NrcDaySetShiftSet(ds), NRC_POSITIVE);
}
```

The constraint applies to `w` only, has penalty `p`, has maximum limit 3, and is consecutive. There is one shift-set for each weekend, containing the shifts of that weekend.

3.9.10. Adding history to worker constraints

Some constraints need to be influenced by the history of the workers whose timetables they constrain. This can be done by first calling

```
void NrcConstraintAddHistory(NRC_CONSTRAINT c, int history_before,
  int history_after);
```

once, then

```
void NrcConstraintAddHistoryWorker(NRC_CONSTRAINT c, NRC_WORKER w,
  int value);
```

at most once for each worker `w` in `c`'s worker-set. The `history_before`, `history_after`, and `value` values are the a_i, c_i, and x_i values from Jeff Kingston's paper on history.

Functions `NrcWorkerAddHistory` and `NrcWorkerRetrieveHistory` from Section 3.8.1 make it easy to store history values in workers. However, they do not automatically pass these values on to constraints. Code like this is needed for that:

```
c = NrcConstraintMake(..., ws, ...);
NrcConstraintAddHistory(c, ...);
for( i = 0;  i < NrcWorkerSetWorkerCount(ws);  i++ )
{
  w = NrcWorkerSetWorker(ws, i);
  if( NrcWorkerRetrieveHistory(w, "WeekendsWorked", &v) && v > 0 )
    NrcConstraintAddHistoryWorker(c, w, v);
}
```

Accordingly, NRC offers helper function

```
void NrcConstraintAddHistoryAllWorkers(NRC_CONSTRAINT c,
  int history_before, int history_after, char *name);
```

whose body is the call to `NrcConstraintAddHistory` plus the loop, with `name` for `"WeekendsWorked"` and `NrcConstraintWorkerSet(c)` for `ws`. Most cases are best handled by `NrcConstraintAddHistoryAllWorkers`, but when its simple approach is not sufficient, one can fall back on `NrcConstraintAddHistory` and `NrcConstraintAddHistoryWorker`.

History may not be added to a constraint with a starting shift-set. It is just too hard to assign a reasonable meaning to it.

3.10. Solutions

This section describes solutions. A solution is a collection of assignments to the demands of the shifts of one instance. It is a very simple thing, making this a very short section.

To create a new solution for a given instance, call

```
NRC_SOLN NrcSolnMake(NRC_INSTANCE ins, HA_ARENA_SET as);
```

where `as` is as for `NrcInstanceMake`. To retrieve the instance, call

```
NRC_INSTANCE NrcSolnInstance(NRC_SOLN soln);
```

A solution contains an optional description, giving its provenance. To set and retrieve it, call

```
void NrcSolnSetDescription(NRC_SOLN soln, char *description);
char *NrcSolnDescription(NRC_SOLN soln);
```

`NrcSolnDescription` returns `NULL` when there is no description.

A solution also contains an optional running time, giving the time in seconds that it took to find. To set this value and retrieve it, call

```
void NrcSolnSetRunningTime(NRC_SOLN soln, float running_time);
float NrcSolnRunningTime(NRC_SOLN soln);
```

`NrcSolnRunningTime` returns `-1.0` when no running time has been passed, meaning ‘absent’.

A newly created solution does not lie in any archives. To add it to an archive, the user must first ensure that that archive has a solution group, by calling `NrcSolnGroupMake` (Section 2.2). Then the solution may be added to the solution group, by calling `NrcSolnGroupAddSoln`.

Internally, a solution is just a collection of assignments of workers to the demands of shifts. Each demand accepts at most one assignment. To add an assignment, call

```
void NrcSolnAddAssignment(NRC_SOLN soln, NRC_SHIFT s, int i,
  NRC_WORKER w);
```

This assigns `w` to demand `i` of `s`, where `0 <= i < NrcShiftDemandCount(s)`. The assignment is in `soln`, not in the instance; the instance does not change. `NrcSolnAddAssignment` aborts if an attempt is made to assign a second worker to the same demand.

To inspect an existing assignment, call

```
NRC_WORKER NrcSolnAssignment(NRC_SOLN soln, NRC_SHIFT s, int i);
```

This returns `NULL` when no assignment has been made. Unassigned demands are acceptable within solutions, although they usually incur a penalty, depending on the demand’s penalties.

Chapter 4. Implementation Notes

This chapter contains notes on the more complicated parts of the NRC implementation. It is here mainly for the author's benefit; users of NRC do not have to read it.

4.1. Optimizing worker constraints

The worker constraints created by calls to `NrcConstraintMake`, called just *constraints* here, are not mapped to XESTT constraints in a simple one-to-one manner. Instead, a sequence of optimizations is applied, aiming to reduce the size of the generated XESTT file by combining constraints where possible, and to reduce the density of constraints by replacing whole sets of `NRC_CONSTRAINT_ACTIVE` constraints that combine to limit the number of consecutive busy or free days (etc.) by `NRC_CONSTRAINT_CONSECUTIVE` constraints that apply these limits directly.

These optimizations are carried out by `NrcInstanceConvertWorkerConstraints`, a private function which calls on various functions in files `nrc_instance.c`, `nrc_constraint.c`, and `nrc_condensed.c`. The remainder of this section is basically a step-by-step account of what `NrcInstanceConvertWorkerConstraints` does.

The *attributes* of a constraint are its worker set, its type (active, consecutive, or workload), its bound, its starting shift-set, its shift-sets (including their polarities), and its history. It also has a name, but that does not affect optimization and is not an attribute for present purposes. When two constraints are merged into one, their names are merged in a way that preserves everything in both names but eliminates most repetition.

`NrcInstanceConvertWorkerConstraints` has three phases. In order of execution they are *condensing*, *bound merging*, and *worker set merging*. After these phases are complete, the surviving constraints are mapped to XESTT constraints in a simple one-to-one manner, the only complication being that constraints of type active are generated as limit busy times constraints where possible, and as cluster busy times constraints otherwise.

Bound merging and worker set merging are easy. When two constraints have equal attributes except that one has a maximum limit and the other has a minimum limit, they are merged by bound merging. When two or more constraints have equal attributes except that they apply to different worker sets, they are merged by worker set merging.

Actually there is one wrinkle here. A constraint's history after value is only referenced when there is a minimum limit, as Jeff Kingston's paper on history makes clear. So 'equal attributes' may be refined to mean that if one of the constraints has no minimum limit, then the history after attributes need not be compared. If the constraints are merged, the history after attribute of the result should come from a constraint with a minimum limit, if there is one.

It remains to explain condensing. In the Curtois original instances, constraints which limit the number of consecutive busy or free days do not do so directly. Instead, they use patterns to specify limits on the number of busy or free days, not necessarily consecutive, that may occur in certain time windows. This 'encoding' of the constraints is a bad thing, because it leads to many overlapping constraints where just one would do, slowing down constraint evaluation and confusing solvers that attempt to understand a solution's defects, as opposed to merely observing its cost. Condensing detects such constraints and 'decodes' them back to the unencoded form.

Condensing applies only to constraints of type active which have a maximum limit (only) whose value is one less than the number of shift-sets. Each shift-set must be a copy of the previous one, only shifted a certain offset along the cycle (typically one day, but any offset is acceptable), and these offsets must be all equal. Any starting shift-set must have its times equally spaced along the cycle with this same offset. It does not have to cover the whole cycle. The shift-sets' polarities must either be all equal, or all equal except the last, or all equal except the first and last. Respectively, these polarities are what one finds in patterns that impose a maximum limit, an exact limit at the start of the cycle, and an exact limit not at the start of the cycle.

The constraints satisfying these conditions are partitioned into *bags*. Two constraints lie in the same bag when they have the same hardness, the same worker set, the same polarity (ignoring the ends), and the same first shift-set and offset. Also, constraints whose polarities impose maximum limits go into different bags from constraints whose polarities impose exact limits.

Bags of constraints whose polarities impose maximum limits are easy to handle. One consecutive constraint is made for each constraint, with the shift-sets implied by the original shift-sets and starting shift-set. For example, if the original shift-sets are the first four days, and the starting shift-set contains the first shift on each day (possibly minus the last three), then the shift-sets are the whole set of days. No history is needed.

Bags of constraints whose polarities impose exact limits are harder to handle. Some of the constraints may apply at the start of the cycle, others not at the start of the cycle. The exact length penalized may also vary.

The first step is to pair each constraint which applies at the start of the cycle with a constraint which does not apply at the start but otherwise gives the same penalty to sequences of the same length. Any constraint which applies only at the start of the cycle but cannot be paired in this way is left untouched and ultimately generates the usual uncondensed XESTT constraint.

The pairs are then sorted into decreasing order of the exact length penalized. If there is one pair for each length from some number down to 1, and the penalty costs are non-decreasing as the length decreases, then these constraints are replaced by one or more consecutive constraints that generate the same costs.

Rather than giving a tedious general explanation, consider this example from Curtois original instance `GPost`. Sequences of length 3 have penalty 1, sequences of length 2 have penalty 4, and sequences of length 1 have penalty 100. These are mapped into two consective constraints, one with minimum limit 4 and penalty 1 with a quadratic cost function, the other with minimum limit 2 and penalty 91 with a linear cost function. Starting at the largest exact length, the algorithm is to try quadratic first, then linear, then step, and see how much penalty is left after applying this cost function. If these residues are non-negative and non-decreasing, the function is accepted and the algorithm moves on to the next pair with positive residue. Otherwise the function is rejected and the next function is tried. The algorithm cannot reject all functions, because, since the penalties are non-decreasing, step at least must work. As a special case, when there is only one pair left, all three functions work, and linear is chosen.

Finally, consider history in the condensed constraint. Let its minimum limit be L.

Suppose there is an interval of length less than L at the start. If there were patterns that match with this interval, then the history before value is 0. If not, the history before value is L for each resource. It was not mentioned above, but condensing is only applied if, within a given bag, either each pair contains two constraints (one for the start and one for the rest), or else each pair contains one constraint (for the rest, not for the start).

Suppose that there is an interval of length less than L at the end. No penalty should be applied in this case, because none of the original patterns match with this interval. So history after value L is assigned to each resource. If instances appear with patterns that do match at the end, then the algorithm will have to be revised, analogously to what happens at the start now.

4.2. Converting demands into XESTT constraints

This section explains how demand objects are converted into XESTT assign resource and prefer resources constraints.

When a demand is added to a shift, the demand records that fact as well as the shift. When converting the demand, this makes it easy to determine which events, and which event resources within those events, are derived from the demand, and hence which event groups and roles the constraints are to apply to. The main issue, then, is working out which constraints are needed.

In certain special cases, basically those which can be modelled by at most one assign resource constraint plus at most one prefer resources constraint, the needed XESTT constraints are generated directly. Otherwise, the conversion uses the following fully general algorithm.

A demand records the calls on penalizer functions it receives. The first step is to break each call into a set of requests to associate one penalty with one worker assignment (including non-assignment). The penalty type says how to combine penalties for one worker assignment: sum, replace, or abort. At the end there is one penalty, possibly zero, for each worker assignment.

As explained earlier, the sum of a hard penalty and a soft penalty is the hard penalty. This may be inexact, but in nurse rostering at least the inexactness does not matter. We can't add them. Even if NRC used combined costs like KHE does, there would still be no way to represent the combined cost in an XESTT file.

Partition the worker assignments into groups, where the assignments in group G_i all have the same penalty, p_i. Place non-assignment into its own group. Then,

- For each group of workers G_i whose penalty is non-zero, generate one prefer resources constraint whose set of preferred resources is $W - G_i$, where W is the set of all workers, and whose penalty is p_i. This is correct: it penalizes assignments of G_i but nothing else.

- For the group G_i representing non-assignment, if its penalty is non-zero, generate an assign resource constraint with that penalty. This penalizes non-assignment and nothing else.

Whatever assignment or non-assignment is made, at most one constraint is violated.

4.3. Optimizing demand constraints

NRC offers two ways to define cover constraints (constraints on how many nurses should attend each shift, and what skills they need):

- Demand objects, which constrain each request for one nurse independently of the others. They are converted into XESTT assign resource and prefer resources constraints.
- Demand constraints, which constrain multiple requests simultaneously. They are converted into XESTT limit resources constraints, except as explained below.

There is an argument for using demand constraints only: one method is better than two, and demand constraints can do everything that demand objects do. The counter-argument is that it is better for solving if demands are constrained independently. For example, it allows a solver to decide, for each task separately, whether not assigning that task would incur a cost.

NRConv helps to resolve this dilemma by detecting cases where demand constraints can be replaced by equivalent demand objects, and performing those replacements just before the conversion to XESTT. So the user can use demand constraints where convenient, avoiding an error-prone manual replacement by demand objects while still gaining their advantages.

For example, the following appears in Curtois original instance `Azaiez.xml`:

```
<DayOfWeekCover>
  <Day>Sunday</Day>
  <Cover><Shift>1</Shift><Min>3</Min></Cover>
  <Cover><Skill>0</Skill><Shift>1</Shift><Min>1</Min></Cover>
</DayOfWeekCover>
```

The user of NRC will express this with two demand constraints, which NRC will convert into demand objects: one requesting a nurse with skill 0, and at least two requesting any nurse.

There must be nothing approximate about any replacements done here: the result must be strictly equivalent to the original. However, defining equivalence is an issue. A solution to an instance made with demand constraints merely needs to assign workers to shifts; by the way the constraints work, it does not matter which tasks within the shifts are assigned. But it does matter when the solution is to an instance with demand objects.

For example, consider a shift that prefers four nurses, but will accept three or five, with a penalty. When this shift is converted without using demand objects, all five tasks are subject to the same limit resources constraint, and it does not matter which tasks receive the assignments. But when the shift is converted using demand objects, the first four tasks have penalties for non-assignment, and the fifth task has a penalty for assignment, and solutions that assign four workers need to nominate the first four tasks as the ones receiving the assignments.

So is the converted instance really equivalent to the original? Our answer is that when converting a solution, we are given the workers to assign and the shift to assign them to, but not the tasks, and we need to find the best assignment. If we do that, then the converted instance is equivalent, but solvers for the converted instance have an extra job to do: find the best tasks to assign workers to within each shift.

A conversion which converts demand constraints into demand objects will be considered correct when the best assignment of workers to tasks in each shift attracts the same cost as when demand constraints are used.

The question is whether, for a particular shift s, the demand constraints c_i that refer to s can be replaced by demand objects. If any of the c_i also refer to other shifts, the case seems hopeless and we fail to convert. So we assume now that the c_i constrain only s. Each c_i constrains all the demands of s, not just some, since that is all that `NrcDemandConstraintMake` offers.

Each demand constraint c_i places a bound b_i on the number of demands of s that may be assigned workers from a given worker set w_i, which could be all workers but need not be. So we may consider the constraints on s to be a set of pairs (b_i, w_i) for $1 \le i \le k$. We assume also that the total number of demands, N, is given. This is needed because XESTT requires that a particular, fixed number of event resources appear in each event; N is that number. We might offer a function which deduces a reasonable value of N from a shift's demand constraints; but ultimately the user is best placed to determine N, based on what existing solutions need, perhaps.

The algorithm for converting the c_i and N into demand objects is as follows. It may fail at several points, in which case we fail to convert s's demand constraints into demands; they remain as demand constraints and are subsequently converted into XESTT limit resources constraints.

The first step is to transform the c_i to simplify their structure. Each bound b_i contains optional minimum, maximum, and preferred limits, with associated penalties. A preferred limit is two limits, a minimum and a maximum, whose values are equal. So we replace the c_i by a set of triples of the form $min(v_i, w_i, c_i)$ and $max(v_i, w_i, c_i)$, where v_i is the limit value, w_i is the worker set, and c_i is the penalty to apply for each worker over or under the limit.

Let W be the set of all workers, and let w_0 be a set of workers containing just one element, a special worker representing non-assignment. Transform each maximum limit $max(v_i, w_i, c_i)$ into the equivalent minimum limit $min(N - v_i, W \cup w_0 - w_i, c_i)$. Saying that at most v_i workers from w_i are wanted is equivalent to saying that at least $N - v_i$ workers from $W \cup w_0 - w_i$ are wanted.

So the first step yields a set of minimum limits $min(v_i, w_i, c_i)$, where w_i may include w_0. The second step makes these limits, plus the artificial limit $min(N, W \cup w_0, 0)$, into nodes in a tree T_s. T_s is like the tree KHE builds when converting workload requirements into workload demand nodes, although that tree limits times, not workers. The nodes of T_s satisfy these conditions:

1. If node $n_i = min(v_i, w_i, c_i)$ is the parent of node $n_j = min(v_j, w_j, c_j)$, then $w_j \subseteq w_i$ and $v_j \le v_i$.

2. If nodes $n_j = min(v_j, w_j, c_j)$ and $n_k = min(v_k, w_k, c_k)$ are siblings, then $w_j \cap w_k = \emptyset$.

The algorithm for building T_s is as follows. Sort the minimum limits into non-increasing $|w_i|$ order; break ties using non-increasing v_i order. Take each limit in order, make it into a node, and insert it into T_s. The artificial limit $min(N, W \cup w_0, 0)$ comes first in this order, and its insertion is a special case: it becomes the root. Subsequent insertions of a new node y assume that the insertion is to take place below a given node p. Initially, p is the root. Then,

- If the first condition holds between one of p's children q and y, insert y below q.
- Otherwise, if y's set of workers is disjoint from all of p's children's, make y a child of p.
- Otherwise, fail to convert.

It is easy to see that if this algorithm does not fail, then the tree it builds must satisfy the two conditions. By sorting the limits, we ensure that y could never be the parent of a previously inserted node, showing that if a tree exists at all, this algorithm will not fail.

The third and final step traverses T_s in postorder, generating demand objects along the way.

At each node $n_i = min(v_i, w_i, c_i)$, generate $v_i - V_i$ demand objects, where V_i is the total number of demand objects generated at proper descendants of n_i. The point here is that all the demands generated below n_i are demands for workers which are elements of w_i, so they count towards what n_i is demanding. If $v_i - V_i$ is negative, fail to convert.

It remains to associate penalties with worker assignments in the generated demand objects. Take any node n_i and consider the v_i demand objects generated at or below n_i. (These are easy to find during the postorder traversal, since immediately after generating the $v_i - V_i$ demand objects at n_i, they are the v_i most recently generated demand objects.) Each of these demand objects is supposed to incur penalty c_i if its assignment is not an element of w_i. Accordingly we call

```
NrcDemandPenalizeNotWorkerSet(d, $w_i - w_0$, NRC_PENALTY_ADD, $c_i$);
```

on each of these demand objects `d`, being careful to do so only once per distinct object. If w_i does not include w_0, then we also need to call

```
NrcDemandPenalizeNonAssignment(d, NRC_PENALTY_ADD, $c_i$);
```

After all demands are created and all penalties are added, all the demand objects are made immutable by calls to `NrcDemandMakeEnd`, so that no further changes are possible.

Consider the example from `Azaiez.xml` given earlier, and suppose $N = 5$ and w is the set of workers with skill 0. Then T_s has root node $min(5, W, 0)$, that node has one child $min(3, W, c_1)$, and that node has one child $min(1, w, c_2)$, where c_1 and c_2 are given elsewhere in the file. The postorder traversal will generate one demand, with cost c_2 for a nurse outside w and $c_1 + c_2$ for non-assignment, then two demands, with cost c_1 for non-assignment, and finally another two demands, with no costs.

Part B

The NRConv Executable

Chapter 5. NRConv and its Converters

This chapter describes NRConv and its converters, passing silently over routine things. The material on each converter assumes that the reader has a detailed knowledge of the source model being converted. For NRConv usage information, type `nrconv` with no arguments.

As a rough guide to the complexity of the code, here are line counts for the C source files:

Lines	*Files*
5092	`coi.c`, `coi_cover.c`, and `coi_limit.c`
1669	`inrc1.c`
1464	`inrc2.c`
1836	`cq14.c`

The format of the Curtois original instances is considerably more complex than the others.

5.1. Instance models and solution models

NRConv has *instance models* and *solution models*, which define the format of source instances and solutions. Whenever it reads a source instance or solution file, it has already been informed (via the `-i` and `-s` command-line flags) which model the file follows.

An instance model is defined by a text file called an *instance model file.* For example:

```
InstanceSourceFormat: inrc1.xml
Contributor: The organizers of INRC1
Date:
Country:
Description:
Remarks: converted from INRC1 format by NRConv
```

Each line consists of an identifier followed by a colon followed by any number of spaces followed by an optional value. The lines must appear in the order shown.

`InstanceSourceFormat` gives the name of the format in which the source instances which follow this model are expressed. There is a fixed set of valid names, which at present is

```
coi.xml inrc1.xml inrc2.xml cq14.txt
```

but which is easy to expand (see the top of file `ins_model.c`). There is nothing to prevent different instance models from using the same source format.

The remaining lines give values for the metadata fields of each instance. If there is no date, as above, then the date that NRConv runs is substituted. All these fields are to be taken as default values. If an instance contains more informative metadata values, they may replace these ones.

A solution model is defined by a text file called a *solution model file.* For example:

```
SolnSourceFormat: inrc1-soln.xml
LinkageToInstance: internal
LinkageToSolnGroup: first
Keep: best
SolnGroup: GOAL
Contributor: The GOAL team
Date:
Description:
Publication: http://www.goal.ufop.br/nrp/
Remarks: converted from INRC1 format by NRConv
```

As before, each line consists of an identifier followed by a colon followed by any number of spaces followed by an optional value, and the lines must appear in the order shown.

The `SolnSourceFormat` line gives the name of the format in which the source solutions are expressed. There is a fixed set of valid names, which at present is

```
coi-soln.xml inrc1-soln.xml inrc2-soln.xml cq14-soln.xml
```

but which is easy to expand (see the top of file `soln_model.c`). There is nothing to prevent different solution models from using the same source format.

The `LinkageToInstance` line defines how to determine which instance a solution is for. Its value may be `internal`, meaning that the solution file contains this information, or `external`, meaning that it doesn't. In the latter case, the longer form of the `-s` command line flag must be used to supply this information.

The `LinkageToSolnGroups` line defines how to determine which solution group a solution should go into. There is a fixed set of values for this, which at present is

```
first cq14
```

but which could expand in the future. Value `first` places each solution into the first solution group defined in this file (which would usually be the only one), while `cq14` uses a complex rule based on the values of the `<Algorithm>` and `<CpuTime>` elements of the solution files.

The `Keep` line says how many solutions to keep for each instance in each solution group. Acceptable values are `all` to keep all solutions, and `best` to keep only one solution, the best.

The `SolnGroup` line defines a solution group with the given name, which will be added to the archive. The following lines define its metadata fields. They must all be present, in the order shown. If there is already a solution group with the given name, it is not added again, but NRConv checks that it has the same metadata values as the ones given here, and aborts if not.

To include multiple solution groups, start again after `Remarks` with another `SolnGroup` line, and so on. Every solution goes into exactly one solution group, so if there are any solutions at all, there must be at least one solution group.

5.2. The Curtois original instances

Curtois pioneered the assembly of instances from around the world, and their expression in a common format. Published at

```
http://www.cs.nott.ac.uk/~psztc/NRP/
```

under the heading ‘Original instances’, there are 28 of these instances, with 66 solutions. Following Curtois, we omit instance `HED01b` (it is very similar to instance `HED01`) and its solution, so the converter converts 27 instances and 65 solutions, to archive `COI`, using instance source format `coi.xml` and solution source format `coi-soln.xml`, which are implemented by functions in NRConv source files `coi.c`, `coi_limit.c`, and `coi_cover.c`.

Instances `ERMGH.ros`, `CHILD.ros`, `ERRVH.ros`, and `MER.ros` use the `<TimePeriod>` feature, which gives cover requirements for periods of the day rather than for each shift type. Several other instances contain multiple cover requirements (for particular skills) which apply to the same shifts. Accordingly, for all the Curtois original instances generally, NRConv produces a generous number of extra event resources (the exact number is documented in `coi_cover.c`), and constrains them using limit resources constraints.

The exception is shifts for which the `<AutoAllocate>` attribute is `false`, meaning that the shift can only be preassigned, not assigned by a solver. Each such shift is given the exact number of event resources required to hold the preassignments, and these event resources are preassigned in the generated instance.

All instances use *pattern constraints* (element `<Match>`), which place minimum and maximum limits on the number of occurrences of the elements of an arbitrary set of patterns. These are not convertible in general. NRConv analyses them into three convertible cases, and omits instances with constraints outside these cases (none of the 27 instances is omitted).

Case 1. The pattern constraint has maximum limit 0 but is otherwise arbitrary. Then the patterns within this constraint are unwanted patterns and are handled as such.

Case 2. Each pattern of the constraint either contains a single term, or a sequence of terms all containing `0`, or it is one of the last three patterns, and these together match busy weekends, as in `Sat:[123][123]`, `Sat:[0][123]`, and `Sat:[123][0]`, assuming three shifts per day. The constraint is otherwise arbitrary. It is converted to a resource contraint whose minimum and maximum limits are those of the pattern constraint, and whose time groups express its terms.

A pattern containing a single term is easily expressible using one time group for each starting day. For example, `[12]` is converted to

```
{1Mon1, 1Mon2}
{1Tue1, 1Tue2}
{1Wed1, 1Wed2}
...
```

and these time groups are added to the resource constraint.

A pattern whose terms all contain `0` is converted to one negative time group for each starting day. Consider limiting the number of free weekends, counting a Friday night shift (shift 3) as part of the following weekend. The pattern is `Fri:[012][0][0]`. The time groups are

```
{1Fri3, 1Sat1, 1Sat2, 1Sat3, 1Sun1, 1Sun2, 1Sun3}*
{2Fri3, 2Sat1, 2Sat2, 2Sat3, 2Sun1, 2Sun2, 2Sun3}*
...
```

Each time group contains the complement of each term, on successive days.

Patterns which match busy weekends are easily represented by positive time groups

```
{1Sat1, 1Sat2, 1Sat3, 1Sun1, 1Sun2, 1Sun3}
{2Sat1, 2Sat2, 2Sat3, 2Sun1, 2Sun2, 2Sun3}
```

and so on. NRConv looks for this exact case; it does not attempt to generalize it in any way.

Case 3. NRConv is hard-wired to generate certain XESTT constraints when it reaches certain file positions. This allows it to handle source constraints that fall outside the cases above but which are nevertheless convertible. For example, the pattern constraints at lines 107–148 of `ERMGH.ros` penalize cases of two consecutive busy weekends. The instance begins on a Sunday and ends on a Saturday, and the constraint for the last two weekends does not fit the cases given above, so NRConv generates a hard-wired limit active intervals constraint for the whole set. As it turns out, every hard-wired case except the one for instance `HED01.ros` described below concerns limiting the number of consecutive busy weekends, to one, two, or three.

An awkward pattern constraint occurs at line 95 of instance `ORTEC02.ros`. The problem patterns, `Sat:[NEDL][NEDL]`, `Sat:[0][NEDL]`, and `Sat:[NEDL][0]`, aim to match busy weekends, but they omit the ‘on vacation’ shift `V`, so they leave some busy weekends (`Sat:DV`, for example) unmatched. They should be `Sat:[NEDL][NEDL]`, `Sat:[0V][NEDL]`, and `Sat:[NEDL][0V]`, which are convertible, and in fact they are equivalent to them because the vacation shift can only be preassigned to a nurse, not assigned by the solver, and a hand check shows that cases like `Sat:DV` cannot arise. So the conversion is hard-wired here.

Instance `HED01.ros` (and also the omitted `HED01b.ros`) utilizes *conditional constraints*, which require one pattern to match if another does. These cannot be converted in general, but those in these instances can be. For example, some require all the shifts taken by a nurse in Week 1 to have the same type, which is expressible by a cluster busy times constraint with maximum limit 1 and time groups

```
{1Mon1, 1Tue1, 1Wed1, 1Thu1, 1Fri1, 1Sat1, 1Sun1}
{1Mon2, 1Tue2, 1Wed2, 1Thu2, 1Fri2, 1Sat2, 1Sun2}
{1Mon3, 1Tue3, 1Wed3, 1Thu3, 1Fri3, 1Sat3, 1Sun3}
```

Again, NRConv is hard-wired to generate suitable constraints at these file positions.

The Ikegami instances contain constraints that require sequences of night shifts to be separated by at least 6 days, expressed by unwanted patterns penalizing each occurrence of two night shifts separated by 5, 4, 3, 2, or 1 days (of anything). NRConv could express them in the same way, but to reduce the constraint density it makes them a special case and expresses them by a single limit active intervals constraint with, for each day, one negative time group containing the night shift on that day, and minimum limit 6, with history to ensure that a sequence at the start of the cycle is not penalized. This penalizes the same cases as the unwanted patterns, but with a different cost in general. However, good solutions do not violate these constraints (each has weight 100, which is more than the total cost of good solutions), so in practice the amount by which violations are penalized does not matter. It is true that sequences of, say, 7 consecutive night shifts are penalized by the original formulation but not by the converted one, but there are other constraints, again with weight 100, which limit resources to at most 6 night shifts.

The Curtois original instances have undocumented features (such as `<Preferred>` limits alongside `<Min>` and `<Max>`, `<CoverWeights>`, and the format of the `<CpuTime>` attribute of solutions) and undocumented interactions (such as how `<DateSpecificCover>` overrides `<DayOfWeekCover>`). Explaining them all is beyond our scope. There are also documented features which do not appear in the instances. For the most part these are not implemented; NRC will print warning messages and omit instances that contain them.

5.3. The First International Nurse Rostering Competition model

The First International Nurse Rostering Competition has a simple XML format, documented at

```
http://www.kuleuven-kortrijk.be/nrpcompetition
```

NRConv converts it using models `inrc1.xml` and `inrc1-soln.xml`.

Cover constraints appear as numbers of nurses wanted for each shift, for each skill. NRC demand constraints are not needed. There is a long list of resource constraints. It all maps easily into NRC, except as described now.

`<AlternativeSkillCategory>` defines the penalty to apply when a nurse is assigned to a shift without having the required skill. This allows each nurse to have a different penalty, which is a problem for XESTT, since its prefer resources constraint (the obvious target when converting) associates the penalty with the shift, giving all unqualified nurses the same penalty.

This problem is solved as follows. For each skill s_i and each distinct non-zero weight w_j for `<AlternativeSkillCategory>`, let $S(s_i, w_j)$ be the set of all nurses n such that the assignment of n to a place requiring skill s_i should attract penalty w_j. Let N be the set of all nurses. For each place requiring skill s_i, define one prefer resources constraint for each non-zero weight w_j such that $S(s_i, w_j)$ is non-empty, with weight w_j and set of nurses $N - S(s_i, w_j)$. In practice this produces just one or two prefer resources constraints per skill.

There are instance files (for example, `long01.xml`), which assign skills to nurses whose `<AlternativeSkillCategory>` constraint is turned off. We interpret this to mean that skills defects are to be ignored for those nurses.

`<CompleteWeekends>` requires a nurse to work on each day of the weekend, or none. This is implemented for the first weekend by a cluster busy times constraint with one time group for each day of the weekend, containing the times of that day, and minimum limit equal to the number of days or else 0. For example, if there are two days in the weekend, with five times each day, the constraint has time groups

```
{1Sat1, 1Sat2, 1Sat3, 1Sat4, 1Sat5}
{1Sun1, 1Sun2, 1Sun3, 1Sun4, 1Sun5}
```

and minimum limit 2 or else 0. This is then repeated for each weekend.

When weekends have three or more days, it is possible to work on the first and last days and be free in between them. The competition assigns a higher cost for such cases than for other cases of incomplete weekends. Although several instances do have three-day weekends, NRConv does not implement this refinement. It can be done using unwanted patterns.

`<IdenticalShiftTypesDuringWeekend>` requires a nurse to either work the same shift on each day of the weekend, or to be free on all days. This is expressed for the first weekend by a cluster busy times constraint with one time group for each time of day, containing the weekend's times of that time of day, and maximum limit 1. For example, for the two-day weekend with five times per day, the time groups would be

```
{1Sat1, 1Sun1}
{1Sat2, 1Sun2}
{1Sat3, 1Sun3}
{1Sat4, 1Sun4}
{1Sat5, 1Sun5}
```

This is then repeated for each weekend. Clearly, if shifts of two types are busy during one weekend, two time groups will be active and there will be a violation.

Although this is logically correct, the competition evaluator does more: it treats violations of `<CompleteWeekends>` as violations of this constraint as well. The cluster busy times constraint just given does not do this.

One possible alternative is a limit busy times constraint with the same time groups as the cluster busy times constraint, but with a minimum limit of 2 or else 0 applied to each. This will penalize the case where (for example) `1Sat1` is busy but `1Sun1` is not, and vice versa. The problem here is that if `1Sat1` and `2Sun4` are both busy, there will be two violations, not one.

The XESTT `Step` cost function could be used to solve this problem, but unfortunately the intermediate model treats the time groups of limit busy times constraints as independent of one another, which they are not when the `Step` cost function is used.

So NRConv generates two constraints for each identical shift types constraint: the cluster busy times constraint, plus the equivalent of a complete weekends constraint. When there is already a complete weekends constraint it merges the two, adding their weights together, provided they have the same hardness and cost function.

`<TwoFreeDaysAfterNightShifts>` requires that on the two days after a night shift, a nurse should either have the day off or else work another night shift. Assuming three shifts per day, with `3` being the night shift, our solution makes patterns `[3][12][12]`, `[3][12][03]`, and `[3][0][12]` unwanted. On the second-last day of the cycle, only `[3][12]` is unwanted.

A violation on both days should cost more than a violation on one, so `[3][12][12]` should get double weight. However, the competition evaluator does not do this, so we assign the same weight to all patterns. We can then merge the first two, producing unwanted patterns `[3][12]` and `[3][0][12]`.

5.4. The Second International Nurse Rostering Competition model

The Second International Nurse Rostering Competition, which is documented at

```
http://mobiz.vives.be/inrc2/
```

has a similar format to the first, although with fewer resource constraints. NRConv converts it using models `inrc2.xml` and `inrc2-soln.xml`.

The main innovation here is that the competition reflects the way nurse rosters are often made in reality: week by week, not all at once. A *weekly instance* is an instance covering one week; a *global instance* covers several weeks. A global instance is solved by solving a sequence of weekly instances for consecutive weeks. Each is hidden from the solver until it has solved the previous weekly instances.

XESTT has no representation of a sequence of instances connected by history. NRConv produces an XESTT representation of one weekly instance, based on files giving the general scenario, the week in question, and history. It could generate global instances, but one would have to trust the solve for each week to not look ahead. History can be handled by adjusting the limits of the constraints affected, using one constraint per resource. NRConv uses XESTT's history features to generate a single constraint with a history adjustment for each resource, which is clearer and less verbose.

One complication with history concerns the order in which the parts of the source instance are added to the NRC instance. It can be convenient to add constraints before nurses, when they precede nurses in the source model file. This is done by defining, say, a worker-set for all the nurses of one contract, passing that worker-set to that contract's constraints, and adding the nurses to the worker-set later. But that will not work for constraints affected by history, because they accept history information for individual nurses. Adding the constraints early, then adding their history later would work, but that is painful to organize.

Here is the order used by NRConv:

add days (one week's worth), and the one weekend
add shift types
add skills
add contracts
add nurses, including their skills and contracts
add nurse histories to nurses
add worker constraints (shift type, pattern, contract), including history
add cover constraints
add shift-off requests

The week file is used only at the end, for cover constraints and shift-off requests. But the scenario file is used out of order, partly to bring everything related to nurses together, but mainly to ensure that nurses are added before constraints, as explained above; and the history file is used in the middle of the scenario file. All this is easy to do because the converter reads all three files and stores them in memory as `KML_ELT` objects before starting the conversion. What is not so easy, however, is to recognize the dependencies and build the NRC instance in a correct order.

One would think that a Week 0 history file would have zero values for history, but in fact the supplied Week 0 files have many non-zero values. So NRConv fudges and assumes that a Week 0 history file has one week of history.

5.5. The Second International Nurse Rostering Competition static model

Some time after the second international timetabling competition ended, some papers appeared which tested a particular set of 'static' (multi-week) instances. So NRConv has been enhanced to convert these kinds of instances as well.

5.6. The Curtois-Qu 2014 model

Curtois and Qu have recently produced a new set of 24 plain text instances, documented at

```
http://www.cs.nott.ac.uk/~psztc/NRP/
```

These, and solutions posted by Curtois at the same place, have been converted using source models `cq14.txt` and `cq14-soln.txt`.

Again, much is familiar. Minimum limits on consecutive busy or free days do not apply to sequences that include the first or last day. This is modelled using XESTT's history mechanism in a somewhat artificial manner.

Many resources have a hard limit of 0 on the number of shifts they can take of a given shift type. Although this could be implemented like other workload limits, by a limit workload constraint, we choose instead to build, for each shift type `st`, the set of all workers with a non-zero workload limit for shifts of type `st`, and generate a prefer resources constraint for each shift of type `st` which has this set of workers for its preferred set.

The two approaches are formally equivalent, but the prefer resources constraints allow solvers to reduce the domains of shifts to just those workers who have a non-zero workload for that kind of shift, and so to avoid attempting assignments which are doomed to fail because of the zero workload limit. This has saved running time in the author's tests of his KHE18 solver.

The solutions to these instances available at the web site above are not the full set reported in Curtois' paper. Accordingly we requested and received a larger set from Curtois.

As in the Curtois original instances, more nurses may be assigned to a shift than the specified optimum, and NRConv creates extra event resources to allow for this. Some of the solutions received from Curtois, especially for the larger instances, overload shifts this way to an unreasonable degree. The worst cases occur in `Instance22.Solution.516686.roster` and `Instance22.Solution.516686_1.roster`, which assign 25 nurses to shift `d1` on day `140`, when the instance specifies an optimum cover of 1.

We have chosen to generate shifts with maximum cover $2c+5$, where c is the optimum cover, making solutions that overload shifts beyond that point invalid. Of the 372 solutions received from Curtois, 42 were rejected for this reason. The rest were classified using metadata in the solution files into four solution groups, one for each algorithm in Table 2 of Curtois' paper. The best solution for each instance in each solution group was included in archive `CQ14` (66 solutions altogether).