IBM ILOG Scheduler

Reference Manual

June 2009

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

About This Manual

This reference manual documents the classes and concepts of the IBM® ILOG® Scheduler library.

Group Summary	
optim.scheduler.modeling	The IBM® ILOG® Scheduler API.
optim.scheduler.solving	The IBM® ILOG® Scheduler API.

What Is Scheduler?

Scheduler is a C++ library for modeling scheduling problems. This library is not a new programing language: it lets you use data structures and control structures provided by C++. Thus, the Scheduler part of an application can be completely integrated with the rest of that application (for example, the graphic interface, connections to databases, etc.) because it can share the *same* objects.

What You Need to Know

This manual assumes that you are familiar with the operating system in which you are using Scheduler. Since Scheduler is written for C++ developers, this manual assumes that you can write C++ code and that you have a working knowledge of your C++ development environment.

Notation

Throughout this manual, the following typographic conventions apply:

- Samples of code are written in this `typeface`.
- The names of constructors and member functions appear in this `typeface` in the section where they are documented.
- Important ideas are emphasized like *this*.

Naming Conventions

The names of types, classes, and functions defined in the Concert Technology library begin with `Ilo`.

The names of classes are written as concatenated, capitalized words. For example:

IloActivity

A lower case letter begins the first word in names of arguments, instances, and member functions. Other words in such a name begin with a capital letter. For example,

```
aVar
IloActivity::getEndVar
```

There are no public data members in Scheduler, except in goals and demons. (This reference manual and the IBM ILOG Solver Reference Manual document goals and demons.)

Accessors begin with the keyword `get` followed by the name of the data member. Accessors for Boolean members begin with `is` followed by the name of the data member. Like other member functions, the first word in such a name begins with a lower case letter, and any other words in the name begin with a capital letter.

Modifiers begin with the keyword `set` followed by the name of the data member.

```
class Task  {
```

```
public:
  Task(char* name, IloInt duration);
  ~Task();
  IloInt  getDuration() const;
  void setDuration(IloInt duration);
  IloBool isCritical() const;
  void setCritical(IloBool critic);
};
```

Include Files

In this reference manual, the documentation of a class uses the caption "Include File" to indicate which header file you need to include in your application. The caption "Definition File" indicates the header file where the class is actually defined.

Concepts

Scheduler Overview

Overview

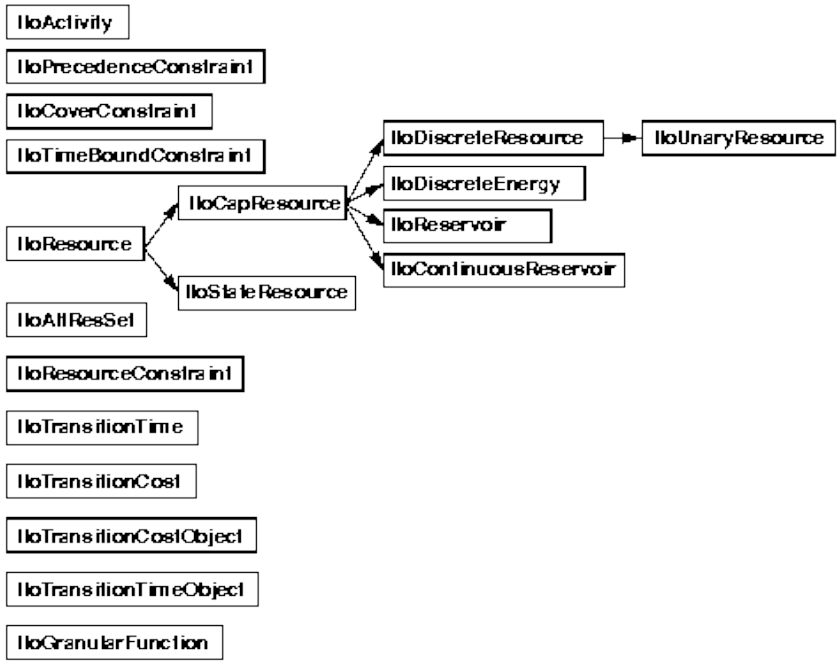
The Scheduler Model is composed of three main families of classes:

- Scheduling Object Classes
- Parameter Classes
- Model Parsing Object Classes

These class families are described in the following sections.

Scheduling Object Classes

Scheduling Object classes represent the core classes of the model. They allow representation of activities, temporal constraints, resources or resource requirements, and transition times and transition costs on resources. The following figure shows all the Scheduling Object classes.



IBM® ILOG® Scheduler Object Classes

Parameter Classes

Parameter classes represent characteristics or behaviors that can be attached to a scheduling object. For example, a scheduling object like a resource may have a maximal capacity that varies with time, or there may be breaks during which activities executing on that resource may be suspended. As another example, an activity may have different behaviors in response to resource capacity or breaks, or may require transition times if it follows certain other activities. Parameter classes represent these various characteristics.

Parameter classes have the following properties:

- The parameter characteristics can be modified using the API of the scheduling objects. For example, the API of the class representing a resource allows adding a new break in the break list of the resource.
- The characteristics can be shared between several scheduling objects. For example, several resource instances may share the same break list (such as, all these resources have breaks on weekends), or several activities may share the same behavior with respect to breaks. Sharing a characteristic saves memory in the model and allows easy modification of the value with only one function call for a group of objects.

Each parameter class has a default value, which can be directly modified. The parameters and scheduling objects are related in a specified pattern. See Parameter Design Pattern for more information.

Parameter classes can be classified according to the scheduling object classes to which they are attached -- either resources or activities. A third type of parameter, Generic Parameters, may represent different types of characteristics. For example, both the maximal and minimal capacity profiles of a capacity resource are represented by a parameter class describing a numeric step function. Generic parameters do not have any inherent semantics. They get particular semantics only when they are attached to a Scheduling Object.

The following is a complete list of Parameter classes.

Resource Parameter

Resource Basic Parameter: `IloResourceParam`

Activity Parameters

Activity Basic Parameter: `IloActivityBasicParam`

Activity Constraints Parameter: `IloActivityConstraintsParam`

Activity Break Parameter: `IloActivityBreakParam`

Activity Overlap Parameter: `IloActivityOverlapParam`

Generic Parameters

`IloNumToNumStepFunction`, `IloNumToNumSegmentFunction`, `IloNumToAnySetStepFunction` and `IloIntervalList` are documented in the extensions section of the *IBM ILOG Concert Technology Reference Manual*.

Time Interval List: `IloIntervalList`

Transition Cost: `IloTransitionParam`

Numeric Step Function: `IloNumToNumStepFunction`

Numeric Piecewise Linear Function: `IloNumToNumSegmentFunction`

Set Step Function: `IloNumToAnySetStepFunction`

Scheduler Overview Figure gives an overview of the links between Scheduling Objects and Parameters.

Parameters Organized by Function

Parameters can also be categorized according to their function. We distinguish three categories of parameters:

- *Description Parameters* are parameters that describe the scheduling objects. For example, a parameter can describe the maximal capacity of a resource.
- *Relaxation Parameters* are parameters that relax some features of the scheduling objects. For example, a relaxation parameter may specify that a due-date constraint on an activity does not need to be taken into account when solving the problem.

- *Enforcement Parameters* are parameters that specify how the constraints expressed on the scheduling objects will be enforced by the scheduler. The value of these parameters is taken from the enumeration `IloEnforcementLevel`. The exact semantics of each value must be defined by the scheduler. Those levels represent a scale of effort that the scheduler will spend at enforcing the corresponding type of constraints.

Following is a brief description of each parameter class.

IloResourceParam

Scheduling Object: `IloResource`

Category: Relaxation and Enforcement parameter

This parameter specifies if various resource characteristics must be enforced and how much effort must be spent by the scheduler to enforce them. It states, for example, if the break list, resource usage, precedence relations between activities requiring the resource, sequencing relations between activities requiring the resource, and/or transition times are to be enforced, and with how much effort.

IloActivityBasicParam

Scheduling Object: `IloActivity`

Category: Description parameter

This parameter describes:

- The maximal duration of the activity
- If the activity can be suspended by breaks or not

IloActivityConstraintsParam

Scheduling Object: `IloActivity`

Category: Relaxation parameter

This parameter allows relaxing the following types of constraints related with an activity:

- Break disjunctivity
- Precedence constraints
- Time-bound constraints
- Covering constraint
- Resource constraints.

IloActivityBreakParam

Scheduling Object: `IloActivity`

Category: Description parameter

This parameter describes the behavior of the activity with respect to breaks. In particular:

- Which breaks will be considered as disjunctive for the activity
- What is the minimal duration of processing time intervals of the activity
- Can the activity be suspended at its start and/or at its end

IloActivityOverlapParam

Scheduling Object: `IloActivity`

Category: Description parameter

This parameter describes if and how an activity may overlap a break.

IloIntervalList

Scheduling Object: Generic Parameter (used by the class `IloResource`)

Category: Description parameter

This parameter represents a list of time intervals associated with a numerical type. It is used to represent:

- A break list attached to a resource. In that case, the integer type corresponds to the type of break. (Identified as [brk] in Scheduler Overview Figure.)
- The time intervals on which the usage of a resource must be enforced. In that case, for all resources except for continuous reservoir, the integer type represents the time precision (time step) used to enforce this resource usage. (Identified as [usg] in Scheduler Overview Figure.)
- The time intervals on which the transition times on a resource must be enforced. (Identified as [tt] in Scheduler Overview Figure.)
- For a state resource, the time intervals during which the resource is constrained to be in use. (Identified as [max] in Scheduler Overview Figure.)

`IloIntervalList` is documented in the extensions section of the *IBM ILOG Concert Technology Reference Manual*.

IloTransitionParam

Scheduling Object: Generic Parameter (used by classes `IloTransitionCost` and `IloTransitionTime`)

Category: Description parameter

This parameter represents a numerical table. It is used to represent:

- The transition times on a resource. (Identified as [time] in Scheduler Overview Figure.)
- The transition costs on a unary resource. (Identified as [cost] in Scheduler Overview Figure.)

IloNumToNumStepFunction

Scheduling Object: Generic Parameter (used by class `IloCapResource)`

Category: Description parameter

This parameter represents the numeric step function $\mathfrak{f} : \mathfrak{N} \rightarrow \mathfrak{N}$

This parameter is used to represent the minimal [min] and maximal [max] capacity profile of a capacity resource and its initial occupation.

`IloNumToNumStepFunction` is documented in the extensions section of the *IBM ILOG Concert Technology Reference Manual*.

IloNumToNumSegmentFunction

Scheduling Object: Generic Parameter (used by class `IloContinuousReservoir`)

Category: Description parameter

This parameter represents the numeric piecewise linear function $\mathfrak{f} : \mathfrak{N} \rightarrow \mathfrak{N}$

This parameter is used to represent the minimal [min] and maximal [max] level profile of a continuous reservoir and its initial occupation.

`IloNumToNumSegmentFunction` is documented in the extensions section of the *IBM ILOG Concert Technology Reference Manual*.

IloNumToAnySetStepFunction

Scheduling Object: Generic Parameter (used by class `IloStateResource`)

Category: Description parameter

This parameter represents the set step function:

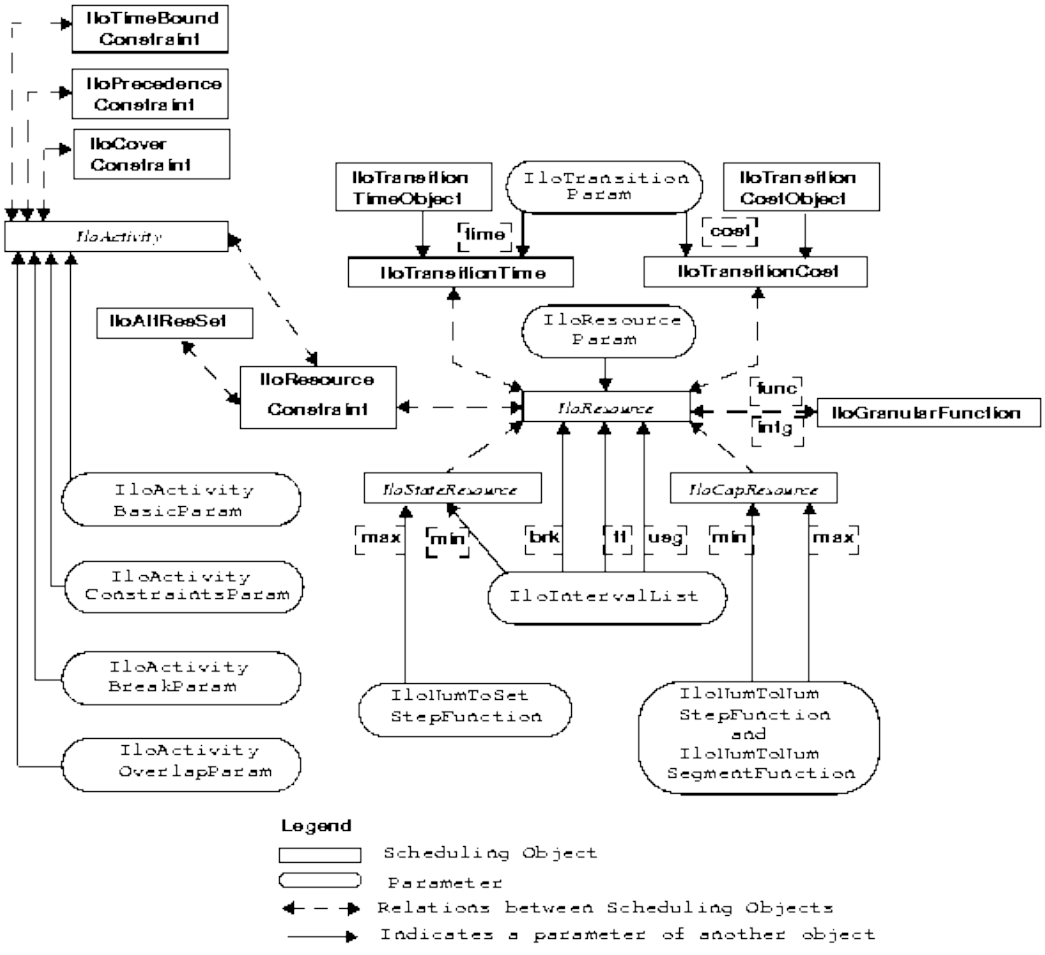
$f_{\Re \to \mathbf{S}}$ where $\mathbf{S} \subseteq \mathbf{A}$ and **A** is the set of all possible values of type `IloAny`.

This parameter is used to represent the possible states of the resource for each time interval. (Identified as [max] in Scheduler Overview Figure.)

`IloNumToAnySetStepFunction` is documented in the extensions section of the *IBM ILOG Concert Technology Reference Manual*.

Scheduler Overview Figure

This figure shows an overview of the links between scheduling objects and parameters in the Scheduler Model.



Model Parsing Object Classes

These classes are used to parse some objects or parameters of the model. The following is the exhaustive list of Model Parsing Object classes.

These classes are all documented in the extensions section of the *IBM ILOG Concert Technology Reference Manual*.

- Cursor on Interval Lists: `IloIntervalListCursor`
- Cursor on Numeric Step Functions: `IloNumToNumStepFunctionCursor`
- Cursor on Numeric Piecewise Linear Functions: `IloNumToNumSegmentFunctionCursor`
- Cursor on Set Step Functions: `IloNumToAnySetStepFunctionCursor`

Parameter Design Pattern

Let `schedclass` be a Scheduling Object class (for example, `IloResource`). Let `xxxParam` be a class that represents the characteristics `xxx` of the instances of the class `schedclass`. For example, let `xxxParam` be the break list of an instance of `IloResource`; in that case, `xxxParam` is an `IloIntervalList`.

For simplification, we suppose that `xxx` consists of one data member that can be accessed by the accessor `getxxxValue()`, and modified with the member function `setxxxValue(Value)`.

The handle class of parameters is defined as follows:

```
class xxxParam : {
public:
   xxxParam(const IloEnv&, ...);
   Value getxxxValue() const;
   void setxxxValue(Value) const;
   void reset();
};
```

The function `reset` allows resetting the parameter to its absolute default value. Instances of parameters `xxxParam` may be shared between several scheduling objects. (For example, several instances of `IloResource` may share the same break list represented as an instance of `IloIntervalList`). Thus, changing or resetting the value of an instance of a parameter *will affect all the scheduling objects that share this parameter*.

An instance of each class of parameter is stored in the scheduler environment (class `IloSchedulerEnv`). By default, all the Scheduling Objects built in that environment will share this parameter instance, unless the local API of those objects is used (see below). This instance can be accessed as follows:

```
class IloSchedulerEnv {
   public:
      xxxParam getxxxParam() const;
      void setxxxParam(const xxxParam& param);
 };
```

The functions `getxxxParam` and `setxxxParam` on the scheduler environment do *not* copy the parameter. The function `setxxxParam` is used to set a new `xxxParam` as default for all the scheduling objects that will be created later on the environment. This function will detach the previous instance of `xxxParam` from the environment but it will not detach this previous instance from the scheduling objects it was already attached to. The function `getxxxParam` can be used only to modify the value of the default parameter with `schedEnv.getxxxParam().setxxxValue(Value)`.

The class `schedclass` of scheduling objects provides an API to modify the characteristics `xxx` that are stored in an instance of `xxxParam`.

```
class SCHEDCLASS {
   public:
      SCHEDCLASS(const IloEnv&, ...);
      void setxxxValue(Value);
      Value getxxxValue() const;
      void setxxxParam(const xxxParam& param);
 };
```

The function `setxxxParam` is used to set a new `xxxParam` for the scheduling object. This function *does not copy the parameter*. When the function `setxxxValue(Value)` is called on the scheduling object `schedclass`, if the parameter `xxxParam` is shared between different objects, *a local copy* is created and the function is

applied to this local copy. The rationale is that the local API of a scheduling object must not modify the parameters that are set for other scheduling objects through shared parameters.

See Also

IloSchedulerEnv.

Calendars

Description

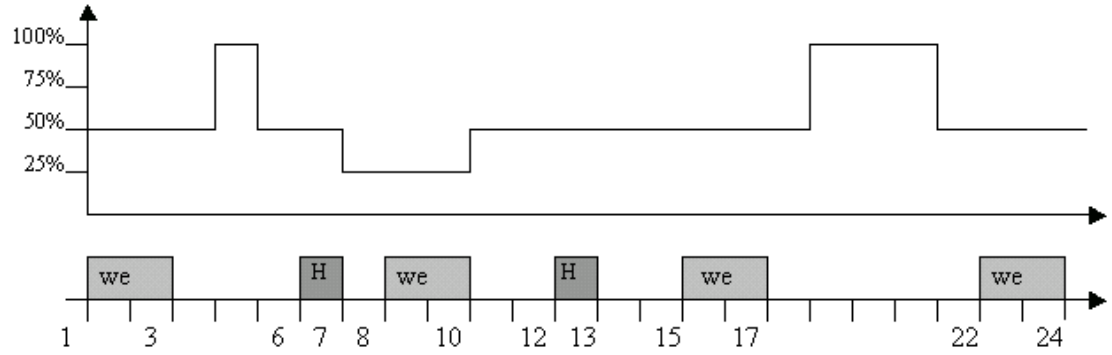
By default an activity is defined by three decision variables: start time (**S**), end time (**E**) and duration (**D**=**E**-**S**). In order to accurately model more complex dependencies between activities and their required resources, a fourth decision variable is defined: the processing time (**PT**), which corresponds to the amount of work the activity performs. The calendar object defines how those variables will interact.

A calendar object is composed by three complementary components:

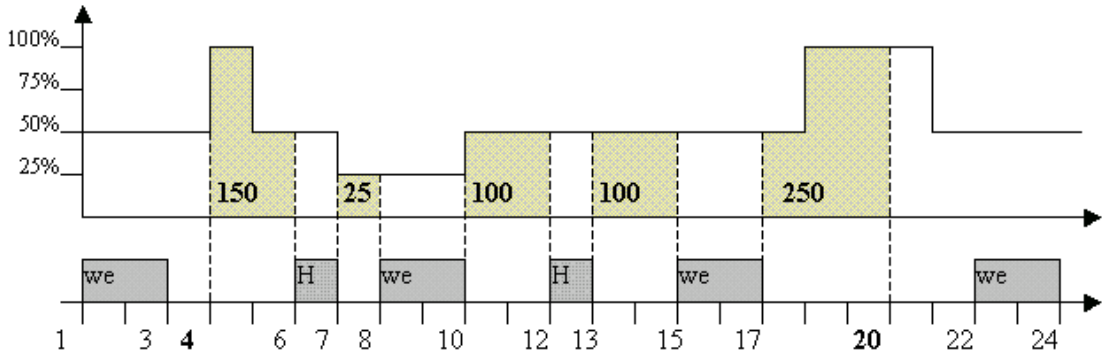
- A list of shift objects. A shift defines forbidden dates for the start, the end or the wall activity (for example, the activity cannot start during weekends).
- A list of breaks intervals. A break interval defines when the activity is suspended (for example, if breaks are defined by weekends, then an activity with a duration of four days starting on Friday finishes at the end of Wednesday).
- An efficiency curve. The efficiency curve defines the behavior of the duration regarding the start time of activity (for example, if the resource only works at 50% during weekends, then an activity with a nominal duration of four days starting on Friday finishes at the end of Tuesday).

Consider a resource **R** that is suspended every weekend and during holidays. In addition, **R** needs to be prepared the day before and the day after each suspension. This preparation will induce a 50% unavailability of **R**. Such a behavior can be modeled with a calendar constraint defined by an efficiency curve and break intervals as follows:

- An efficiency curve defined between 25% availability (when only one day of work) and 100%;
- A break interval for each weekend and each holiday.



In this example, the processing time of the activity **A** which starts in **S**=4 and ends in **E**=20 is then **PT**=150+25+100+100+250 = 625% and obviously its duration is **D**=20-4=16.



If the resource **R** needs to be switched off before weekends, then weekend break intervals can advantageously be replaced by shift intervals to model this disjunctive behavior. In this case, an activity has to be performed completely inside a week, so an activity with a processing time equal to 2 will be executed at least after time 17.

Decision Variables

By default an activity **A** is defined by three decision variables: start time (**S**), end time (**E**), and duration (**D=E-S**). A fourth variable, called processing Time (**PT**), is linked to the activity when using a calendar object. This processing time corresponds to the duration of the activity when neither break intervals nor efficiency need to be considered. This can also be called the nominal duration. The relation between start, end and processing time is defined by:

$$PT = \int_S^E efficiencyCurve(t) \cdot noSuspension(t) \cdot dt$$

with

$$noSuspension(t) = \begin{cases} 1 & when \quad t \notin Breaks \\ 0 & else \end{cases}$$

If there are no break intervals and no efficiency curve (by default equal to one on the horizon), **PT=E-S=D**. In addition to this relationship, activity variables can be overconstrained in the two following ways.

The activity is not breakable corresponds to:

$$D = \int_S^E noSuspension(t) \cdot dt$$

The activity does not use efficiency corresponds to:

$$PT = \int_S^E noSuspension(t) \cdot dt$$

Calendar Object Semantic

A calendar object is defined by a list of shift objects, a list of break intervals, and an efficiency curve. By default, lists are empty and the efficiency curve is equal to 1 on the scheduled horizon. The calendar object is associated with a resource or a resource constraint. When a calendar is defined both on a resource constraint and on the corresponding resource, the calendar of the resource is forgiven in order to only take into account the calendar of the resource constraint.

Consider the example with an activity **A**, a calendar object **C1** associated with the resource **R**, and a calendar

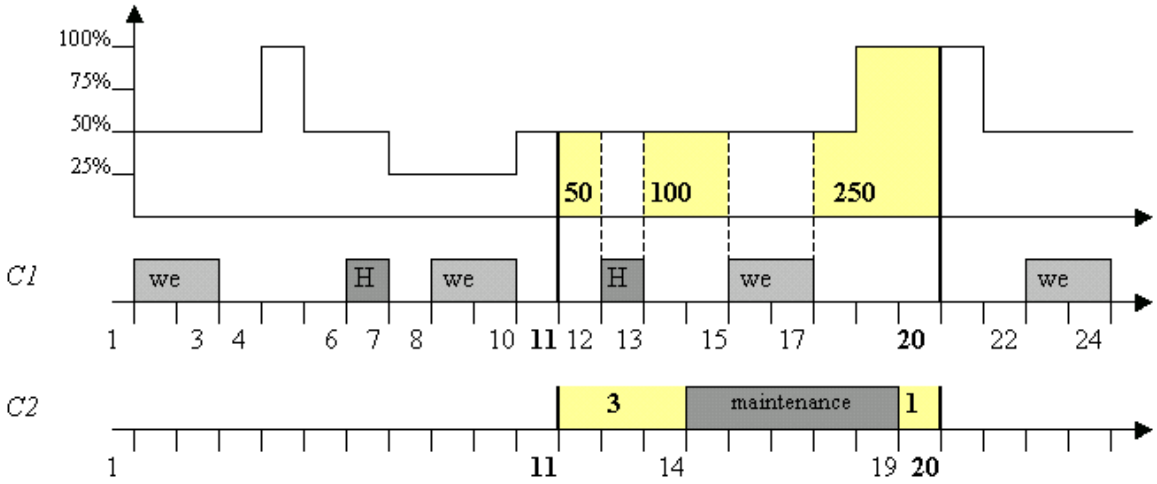
object **C2** associated with the resource constraint: **A.requires(R)**. **C1** is defined by start shift objects **S1** and **S2**. **C2** is defined by break interval **B1**. Then variables of activity **A** are constrained by the calendar object **C2**, and so the start of the activity does not have to respect **S1** neither **S2**.

Activities with Null Processing Time

Activities with null processing time are not affected by calendars. This feature provides an easy way to state that some activities must be "ignored" as far as calendars are concerned.

Activities Related to Several Calendars

Consider a breakable activity `act` that uses efficiency, with a processingTime equal to 4 and two unary resources `res1` and `res2`. There are two calendars **C1** (the one of the previous example) and **C2** (just a break in [14,19)) and two resource constraints `rct1=act.requires(res1)` with calendar **C1** and `rct2=act.requires(res2)` with calendar **C2**. As there is only one activity, there exists only one processingTime variable, and so the processingTime must be equal on both resource constraints (that is, with **C1** and **C2**). As the efficiency is most of the time smaller than 100%, the activity has to be suspended by the break of **C2** to be able to respect processingTime==4, as shown on the following figure. The only solution is to start at time 11 and to end at time 20.



So to avoid trouble and express that such an activity requires resource `res1` and that the "same" activity also requires resource `res2`, you must create another activity from the start until the end as defined by resource `res1` with the same start and end variables, and a new processing time variable (see IloActivity::shareStartWithStart and IloActivity::shareEndWithEnd).

Shift Object Semantic

A shift object is a set of forbidden dates:

$$Shift = \{t_i\}_i$$

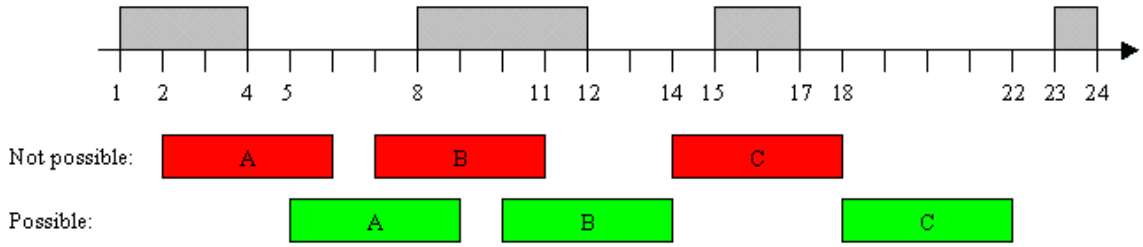
An associated type defines the shift behavior regarding the start and the end of the activity. The three types are as follows.

- OnStart: the activity cannot start in the shift, that is: $\forall i \quad S \neq t_i$
- OnEnd: the activity cannot end in the shift, that is: $\forall i \quad E \neq t_i$
- OnOverlap: the activity cannot overlap a forbidden date, that is: $\forall i \quad t_i < S \vee t_i \geq E$

The shift object can be defined by the user using the `MACROILOUSERSHIFTOBJECT`. However the predefined shift object `IloShiftListObject` may be enough in most of the cases as it deals with a list of shift intervals.

Consider the example with a given the calendar **C** associated with the resource **R**, and activities **A**, **B** and **C** all with durations of 4 and requiring **R**. **C** is defined by the shift interval list L=[1, 4), [8,12), [15, 17), [23, 24).

- If shift object is OnStart: **A** cannot start in 2, but it could start in 5;
- If shift object is OnEnd: **B** cannot end in 11, but it could end in 14;
- If shift object is OnOverlap: **C** cannot start in 14 and end in 18 because of the shift interval [15,17), but it could start in 18 and end in 22.

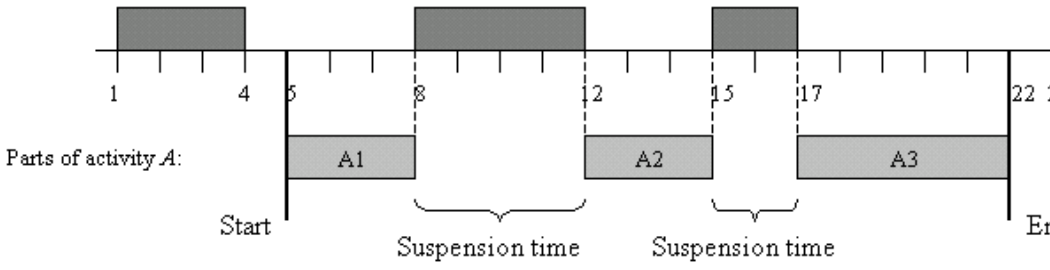


Break Intervals Semantic

Using breaks is the easiest way to model the suspension of an activity by a fixed time interval; so a suspended activity is divided in a set of convex time intervals:

$A = (A_i)_i$

For example:

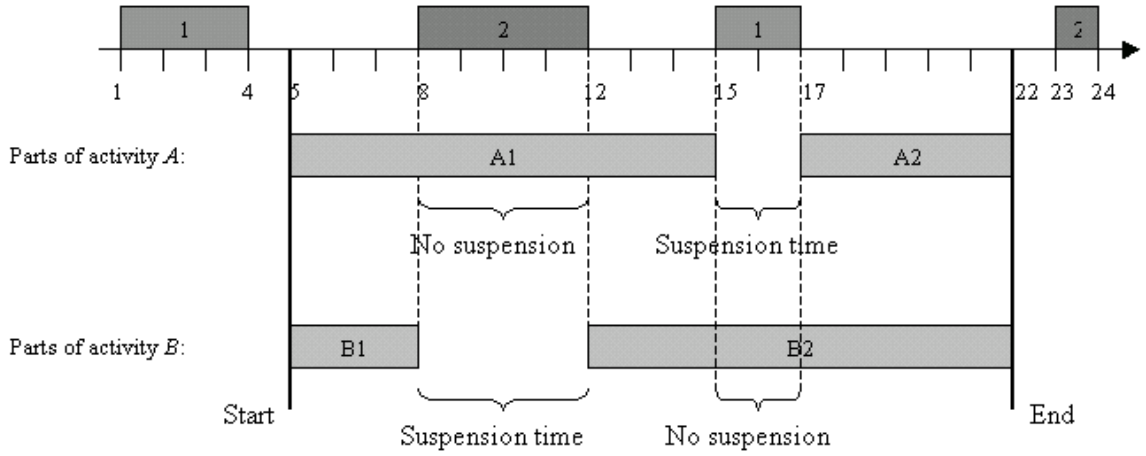


$A = [5,8) \cup [12,15) \cup [17,22)$

Ignored Breaks

Each time interval **I** is associated with a unique number called its type (see `IloIntervalList` in the extensions section of the *IBM ILOG Concert Technology Reference Manual*. Those types allow one to parameterize the behavior of activities with respect to certain breaks, to ignore breaks, or to express disjunctive behavior. See Disjunctive Break.

Each activity may be associated with a set of ignored break types. It means that every break **B** whose type belongs to this set will be ignored, as it does not exist. In the following figure, the activity **A** with ignored type set **{2}** is not suspended by breaks **2** whereas the activity **B** with ignored type set **{1}** is suspended by breaks **2**.



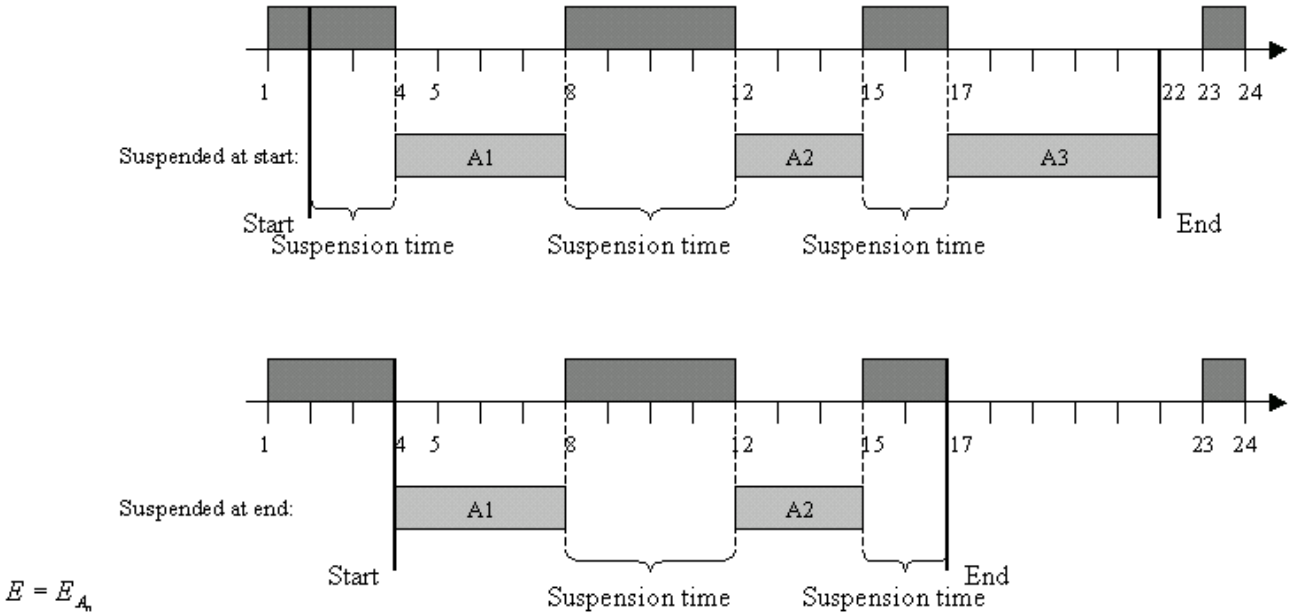
For more information, see IloActivity::addIgnoredBreakType and IloActivity::removeIgnoredBreakType.

Capability of an Activity to be Suspended at Start or at End

If an activity cannot be suspended at start, then there is no suspension time interval that starts at the start time of the activity.

$S = S_A$

In the same way, if an activity cannot be suspended at end, then there is no suspension time interval that finishes at the completion time of the activity.



$E = E_A$

For more information, see IloActivity::setCanBeSuspendedAtStart, IloActivity::setCanBeSuspendedAtEnd, IloActivity::canBeSuspendedAtStart, IloActivity::canBeSuspendedAtEnd.

Minimal Execution Duration

An activity can express the fact that the duration of each of its time intervals must be greater than a threshold duration **ExecD**.

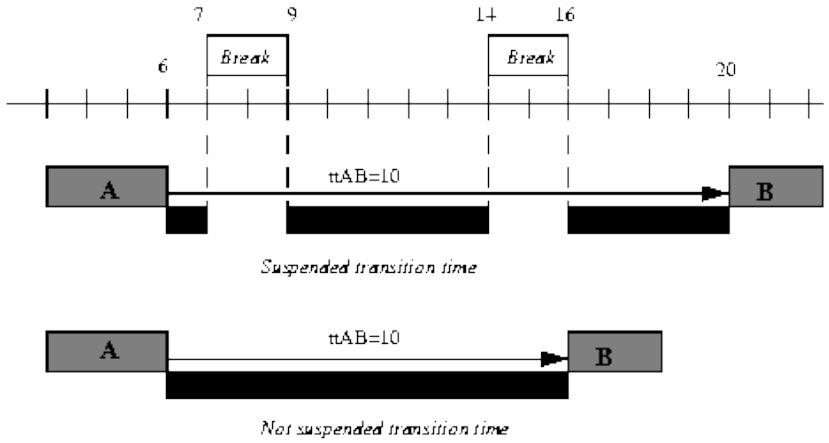
$$\forall i \quad |A_i| \geq ExecD.$$

On the preceding example, the minimal processing duration is 3 (the second processing interval). Thus, if the threshold minimal duration for processing intervals of this activity had been set to 4, the distribution of processing duration depicted in that figure could not be accepted in a solution. See IloActivity::setExecutionDurationMin.

Breaks and Transition Times

The transition time on a resource allows specifying a delay between two activities (see Transition Times). This delay can take into account the breaks defined on that resource. By default, the transition time on a resource is not suspended by the breaks. This can be illustrated by the following example: suppose a unary resource with a break list [0,2), [7,9), [14,16), and two activities *A* and *B* that require this resource. Activity *A* finishes at time *endA=6* and activity *B* is constrained to start after the end of activity *A*. If the transition time between *A* and *B* is *ttAB=10*, because of this transition time, the activity *B* cannot start before date *endA + ttAB =16*.

The member function `void IloTransitionTime::setSuspended(IloBool suspended=IloTrue)` allows specifying a transition time that is suspended by the breaks of the resource. In the example above, that would imply that activity *B* must start after date 20, as shown in the following figure.



Notice that the transition time on a resource with suspended time concerns all (but only) the breaks on that resource. That is, no calendar object must be defined on resource constraints attached to the resource.

Disjunctive Break

A break **B** is said to be disjunctive for an activity **act** if **act** behaves as a non-breakable activity with respect to this break **B**. That is, the activity **act** cannot be suspended by the break.

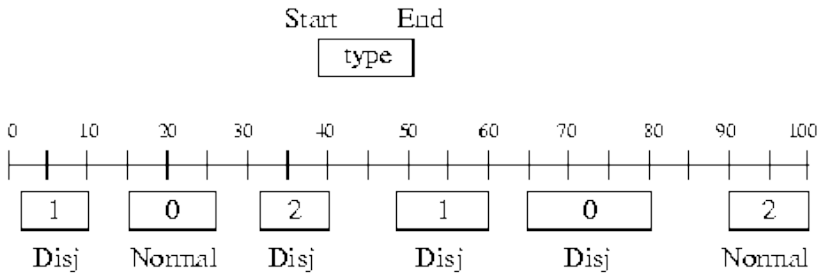
There are three different ways to implement disjunctive behavior with a calendar:

- Shift object typed as OnOverlap (see Shift Object Semantic). Notice that in most of the cases, using shifts is more efficient than breaks regarding disjunctive behavior.
- Disjunctive break type: Each activity may be associated with a set of disjunctive break types. It means that every break B whose type belongs to this set will be considered as disjunctive for the activity.
- Break size: Each activity may be associated with an interval [dmin, dmax] of possible duration for non-disjunctive breaks. Each break whose duration is outside this interval will be considered as

disjunctive for the activity.

A break that is not disjunctive is called a normal break. By default, all the breaks except for null duration breaks are considered as normal (non-disjunctive) breaks for an activity.

Suppose the break list described in the next figure and an activity whose set of disjunctive break types is 1 and interval of possible duration for normal breaks is [10,12]. The disjunctive status of breaks with respect to this activity is also given on the following figure. For instance, we see that the first break [2,10) will be considered as disjunctive for the activity as its type is 1. The break [15,26) will not be considered as disjunctive because its type does not belong to the set of disjunctive types and its duration (11) falls into the possible durations of normal breaks. The break [32,40) will be considered as disjunctive because its duration (8) does not fall into the possible durations of normal breaks.



For more information, see `IloActivity::addDisjunctiveBreakType`, `IloActivity::removeDisjunctiveBreakType`, `IloActivity::setDurationMaxNormalBreaks`, and `IloActivity::setDurationMinNormalBreaks`.

Possible Execution Break Overlap

The standard use of a break is to suspend the processing time of an activity. In some cases, however, you may want to relax this and allow some limited amount of processing to occur within a break. For example, you might want to specify that it is possible for the end of an activity to be processed inside a break, providing the length of processing time within a break is less than some value. In a factory with one 8-hour shift, where non-shift hours are represented as breaks, it might be better to finish a job as much as a half-hour after the end of the shift rather than to process the entire activity in the next shift.

Conceptually, a processing time overlap is a period of time during which the processing time of an activity overlaps a break. Two kinds of processing time overlaps are distinguished:

- Start Overlap
- End Overlap

No other way of overlapping breaks with processing time is allowed.

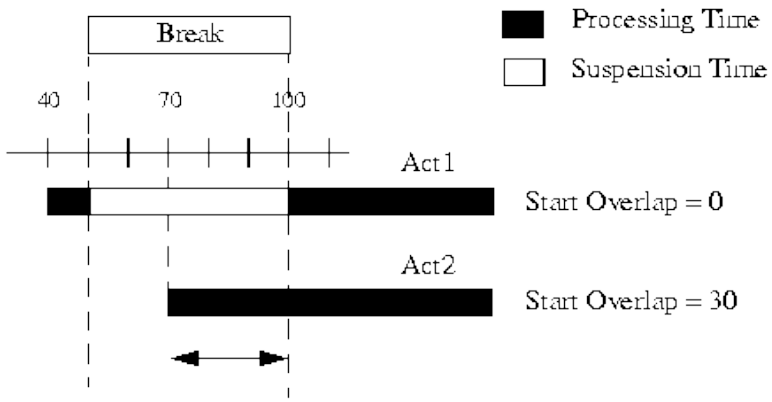
Note

It is important to understand the distinction between allowing an activity to be suspended at its start or end and allowing an activity to overlap a break at its start or end. When an activity is suspended at its start, its start time is inside a break but no processing takes place until that break is over. In contrast, when an activity overlaps a break at the start, the activity's start time is inside a break and some of its processing takes place inside that break. The same distinction holds between an activity that is suspended at end and an activity with a non-zero end overlap.

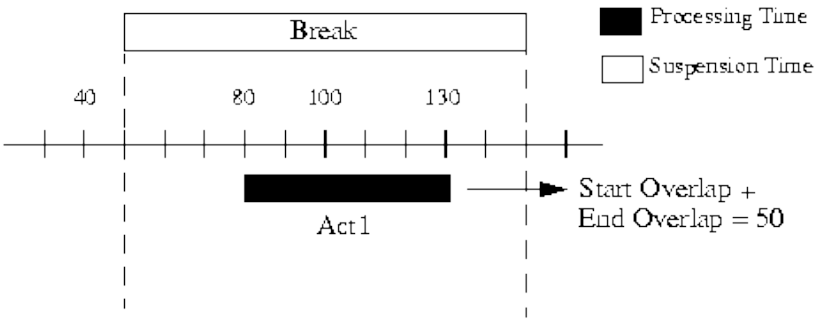
Start/End Overlap

If an activity **act** starts (resp. ends) processing in a break **B**, the duration of the execution overlap between the activity **act** and break **B** is called the start overlap (resp. end overlap) of the activity. If the activity does not start

in a break, its start overlap is 0. The following figure illustrates two situations together with the value of the start overlap.



In the special case where the activity starts and ends in the same break, no commitment is made to decide whether the overlap belongs to the start or end overlap of the activity. The following figure illustrates, for instance, the situation where an activity is completely processed inside a break.



For more information, see `IloActivity::getEndBreakOverlapMin`, `IloActivity::getEndBreakOverlapMax`, `IloActivity::getStartBreakOverlapMax`, `IloActivity::getStartBreakOverlapMin`, `IloActivity::setEndBreakOverlapMin`, `IloActivity::setEndBreakOverlapMax`, `IloActivity::setStartBreakOverlapMax`, `IloActivity::setStartBreakOverlapMin`.

Characterization of the Set of Possibly Overlapped Breaks

Each activity may be associated with a set of possibly overlapped at start (resp. end) break types. It means that every break *B* whose type belongs to this set will be possibly overlapped by the start (resp. end) of the activity.

Any break whose type does not belong to the set of possibly overlapped at start (resp. end) break types cannot be overlapped by the start (resp. end) of the activity. That is, the value of its start (resp. end) overlap is 0.

By default, these sets of possibly overlapped breaks are empty so that no break can be overlapped by the start or end of an activity.

As soon as an activity is associated with a non-empty set of breaks possibly overlapped at start (resp. end), and unless stated otherwise, the activity may overlap these breaks without limitation on the start (resp. end) overlap duration. The functions that allow you to constrain this start (resp. end) overlap duration are the ones listed in Start/End Overlap.

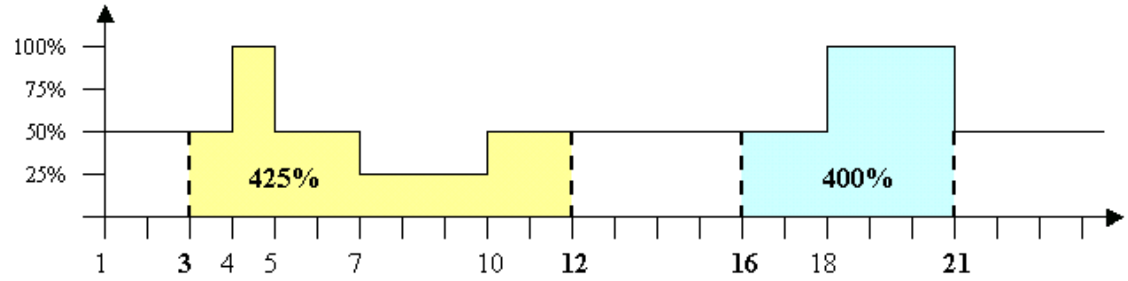
For more information, see `IloActivity::addStartBreakOverlapType`, `IloActivity::removeStartBreakOverlapType`, `IloActivity::addEndBreakOverlapType`, `IloActivity::removeEndBreakOverlapType`.

Efficiency Semantic

As described in the calendar object semantic, the efficiency part of the calendar allows you to model complex dependencies between the activity variables. The relation between start, end and processing time is defined by:

$$PT = \int_{s}^{e} efficiencyCurve(t) \cdot dt$$

Data regarding the description of the efficiencyCurve are stored in an instance of `IloGranularFunction`. This object primarily consists of a step-wise integer-to-integer function plus two additional parameters: the granularity and the rounding mode.



Consider for example, the preceding efficiency curve of the required resource **R** of activity **A** which has a processing time equal to 4. If the activity starts at time 3, then the activity ends at time 12 (note that in case of downward rounding mode the activity ends in 11 -- see Rounding, Inward & Outward). The duration is then 9. But when it starts in 16, the activity ends in 21 (even with downward rounding mode) and then its duration equals 5.

The efficiency part of the calendar constraint is a special case of the integral constraint (see Functional and Integral Constraints on Resources). Nevertheless this is the only way to safely constrain the processing time variable with an integral behavior when a calendar is used.

Functional and Integral Constraints on Resources

Description

Integral and functional constraints address the need for modeling complex dependencies between scheduling variables. Such dependencies makes the use of functions mandatory in order to accurately describe the fine relationships. These constraints concern all the activities processed by a resource. Here are some examples.

Productivity

A resource often needs to be considered to be unavailable (undergoing maintenance, for instance), which thereby changes the production of an activity as a function of its start time. Consider for example a manufacturing machine which produces half of its capability on Saturday and nothing on Sunday. If the usual capability is 6 pieces per hour, the machine will only produce 3 pieces per hour on Saturday and nothing on Sunday. Stated differently, we need to describe the resource as being 50% productive on Saturday, 0% on Sunday, and 100% for the rest of the week.

With Scheduler, you can model this behavior with the following code.

```
IloEnv env;
```

```
IloModel model(env);
IloGranularFunction week(env, 0, 7, 100);
week.setValue(5, 6, 50);
week.setValue(6, 7, 0);
IloReservoir res(env);
IloActivity act(env, 5);
Model.add(act.produces(res, IloNumVar(env, 1, 10, IloInt)));

IloConstraint productivity = IloResourceIntegralConstraint
                             (res, IloCapacityVariable, week);
model.add(productivity);
```

Cost function

A common use of functional relationships is for making a cost variable dependent on the duration or the end time of an activity.

For instance, the duration of an activity might trigger two different behaviors according to whether it lasts too long or not. Consider an activity that costs 10 units if it is executed fast enough (duration less than 5), but costs only 5 such units if there is enough room for a slow, cheaper, processing.

The cost function would be:

```
IloGranularFunction costFunction(env, 0, 10);
costFunction.setValue(0, 5, 10);
costFunction.setValue(5, 10, 5);
```

A resource taking the profile into account reads:

```
IloUnaryResource res(env);
IloConstraint cost = IloResourceFunctionalConstraint(res,
                                                     IloExternalVariable,
                                                     costFunction,
                                                     IloDurationVariable);
```

This constraint states that every activity executing on `res` will have a relation between its duration variable and its external variable. That relation is precisely that specified by `costFunction`.

Then we can set an activity's external variable to a given cost variable:

```
IloNumVar costVariable(env, 0, 10, IloInt);
IloActivity act(env, IloNumVar(env, 0, 10, ILOINT));
act.setExternalVariable(costVariable);
```

and subsequently use this cost in the model.

General case

Scheduler provides modeling tools for two types of relationships: functional and integral constraints.

- A functional relationship interprets the function as a response of the resource to a control parameter of the activity executing on it.
- An integral relationship sums up the function over an activity's duration (from `start` to `end`).

Data regarding the description of the constraints are stored in an instance of `IloGranularFunction`. This object primarily consists of a step-wise integer-to-integer function `func`, plus an additional parameter, the granularity `g`.

Given this data, constraints enforcing `Variable1 == f(Variable2)` and

$$\text{variable} == \{\int_{\text{start}}^{\text{end}} \texttt{func}\} / g$$

can be added to the model using:

```
IloConstraint IloResourceFunctionalConstraint(const IloResource resource,
                                              IloSchedVariable leftVar,
                                              const IloGranularFunction curve,
                                              IloSchedVariable rightVar =
                                                 IlcDurationVariable) const;
IloConstraint IloResourceIntegralConstraint(const IloResource resource,
                                            IloSchedVariable leftVar,
                                            const IloGranularFunction curve);
```

Note that the `leftVar` could be the processing time variable, but in such a case one has to be sure to not use a calendar object on the corresponding resource in order to avoid an over-constraint of the problem (see Calendars and `IloResource::ignoreCalendarConstraints`).

In Scheduler Engine, the following member functions are available:

```
IlcConstraint IlcResource::makeFunctionalConstraint
                              (IlcSchedVariable leftVar,
                               const IlcGranularFunction func,
                               IlcSchedVariable rightVar =
                                       IlcDurationVariable) const;
IlcConstraint IlcResource::makeIntegralConstraint(IlcSchedVariable leftVar,
                               const IlcGranularFunction func) const;
```

The following global functions are available:

```
IlcIntExp IlcFunctionalExp(const IlcGranularFunction func,
                           const IlcIntVar x);
IlcIntExp IlcActivityIntegralExp(IlcActivity act,
                                 IlcGranularFunction func);
```

There are several choices for the variable to be used on the left-side of these constraints, as well as the right-side variable of the functional constraint. These variables are designated by the `IloSchedVariable` enumeration in Scheduler Concert Technology and by the corresponding `IlcSchedVariable` enumeration in the Scheduler Engine.

When you post a constraint on a resource you enforce that for each resource constraint contributing to the resource, *y = = f(x)* where *x* and *y* are defined with the binding policy (the enumeration). The constraint posted by the user is *add(y==IlcFunctionalExp(func,x))*, where *x* and *y* are Solver variables.

Rounding Policy for Integral Constraints

When computing the integral of the function between the start and the end of an activity, the result is usually not a multiple of the granularity `g`. As the result of the division by `g` must be an integral, rounding occurs. The default rounding mode considers that any raw value of the integral:

$$I = \int_{start}^{end} func$$

in the range `[x*g, x*g+g)` gives an efficiency equal to `x` after being divided by `g`. That means that only the nearest lower integer is taken into account.

For example, let's take a granularity of 100, and a constant function of value 30. Suppose we want to enforce a processing time of 10 with such an integral constraint. The rounding will lead to several possible durations for the activity: d={34, 35, 36}. Indeed, with these three values for the duration, the integral of the function takes the valid values I={1020, 1050, 1080}, all contained in the required range [10*100, 10*100+100).

Several rounding modes can be associated to an instance of a granular function. The rounding mode will be taken into account when creating integral constraints. The enumerations `IloGranularFunctionRoundingMode` and `IlcGranularFunctionRoundingMode` are available, along with the member functions `IlcGranularFunction::setRoundingMode` and `IloGranularFunction::setRoundingMode`, to change the rounding mode.

Let `func` be a granular function with granularity `g` used within an integral constraint and `x` be a variable ranging in `[xmin, xmax]`.

Then, the raw integral value:

$$I = \int_{xmin}^{xmax} func$$

is considered to satisfy the integral constraint `x == I/g` according to the following rounding modes:

- `IloGranularFunctionRoundUpward`: `I` is in the range `[xmin*g, (xmax+1)*g)`.
- `IloGranularFunctionRoundDownward`: `I` is in the range `((xmin-1)*g, xmax*g]`.
- `IloGranularFunctionRoundOutward`: `I` is in the range `((xmin-1)*g, (xmax+1)*g)`.
- `IloGranularFunctionRoundInward`: `I` is in the range `[xmin*g, xmax*g]`.

The default rounding mode is `IloGranularFunctionRoundUpward`.

See Also

IloGranularFunction, IloSchedVariable, IloResource, IloActivity, IloResourceIntegralConstraint, IloResourceFunctionalConstraint, IlcGranularFunction, IlcSchedVariable, IlcResource, IlcGranularFunctionRoundingMode, IloGranularFunctionRoundingMode.

Global Constraints

Description

The global constraints are:

```
const IloInt IloRestoreNothing,
             IloRestoreActivityStart,
             IloRestoreActivityEnd,
             IloRestoreActivityDuration,
             IloRestoreActivityProcessingTime,
             IloRestoreActivityExternal,
             IloRestoreRCNext,
             IloRestoreRCPrev,
             IloRestoreRCDirectPredecessor,
             IloRestoreRCDirectSuccessor,
             IloRestoreRCSetup,
             IloRestoreRCTeardown,
             IloRestoreRCCapacity,
             IloRestoreRCSelected,
             IloRestoreAll;
```

Global constants are used to parameterize the restoration of `IloActivity` and `IloResourceConstraint` objects in the following four contexts.

- Local search
- An `IloRestoreSolution` goal
- `IloSolution::getConstraint` method
- `IloSolution::restore` method

In each of these contexts, the data that is restored (that is, inserted into the solver or, in the case of `IloSolution::getConstraint` added, via a constraint, to the model) depends on the value of the `restoreFields` flag that is associated with the extractable. The `restoreFields` are associated with an extractable in the `IloSchedulerSolution::add` method or by using the `IloSchedulerSolution::setRestorable` and `IloSchedulerSolution::setNonRestorable` methods.

Value	Field

`IloRestoreNothing`	none
`IloRestoreActivityStart`	start time variable of an `IloActivity`
`IloRestoreActivityEnd`	end time variable of an `IloActivity`
`IloRestoreActivityDuration`	duration variable of an `IloActivity`
`IloRestoreActivityProcessingTime`	processing time variable of an `IloActivity`
`IloRestoreActivityExternal`	external variable associated with an `IloActivity`
`IloRestoreRCNext`	next relation of an `IloResourceConstraint`
`IloRestoreRCPrev`	previous relation of an `IloResourceConstraint`
`IloRestoreRCDirectSuccessor`	direct successors of an `IloResourceConstraint`
`IloRestoreRCDirectPredecessor`	direct predecessors of an `IloResourceConstraint`
`IloRestoreRCSetup`	setup status of an `IloResourceConstraint`
`IloRestoreRCTeardown`	teardown status of an `IloResourceConstraint`
`IloRestoreRCCapacity`	capacity variable of an `IloResourceConstraint`
`IloRestoreRCSelected`	selected resource of an `IloResourceConstraint`
`IloRestoreAll`	all possible restorable fields of the associated extractable

See Also

IloActivity, IloSchedulerSolution, IloResourceConstraint.

Overflow

Description

Due to the limited size of data representation, IBM® ILOG® Scheduler requires several conditions on its input to avoid overflow or erroneous computations. Some of these conditions are model limitations and are checked when entering the search; they are listed in the following section, Model Limitations . Some other conditions can not be checked systematically during the search, for computational efficiency reasons. They are listed in the section Scheduler Engine Limitations.

Model Limitations

Not all scheduling model instances (class `IloModel`) can be handled by IBM ILOG Scheduler. In particular, only integer dates are considered by the engine. Attempts to extract a model that contains time and date data that are not `IloIntVar` instances will raise the following exception: `integer variable expected during extraction`.

This limitation applies to:

- start, end, duration, processing time, and external variables of activities,
- delay variables of precedence constraints,
- capacity variables of resource constraints (alternative included),
- cost, setup cost, teardown cost and cost sum variables of transition costs, and
- time bound variables of time bound constraints.

Moreover, if necessary, every computation instance of such variables will be clamped to the default range specified in the corresponding `IloSchedulerEnv`. See the description of the method

`IloSchedulerEnv::setIntMaxAtExtraction` for more information.

Other data, expected to be in integer form, is rounded if it has been modeled using an instance of `IloNumVar` or with an `IloNum` data type. If applicable, the bounds of `IloNumVar` variable instances will be rounded toward minus infinity (*floor* rounding) or toward plus infinity (*ceiling* rounding, as in `ceil()`) depending on the semantics of the data.

Some numeric data is required to be in integer form (*must-be-integer*), as no rounding is meaningful. Otherwise, an `IloIntegerExpectedException` exception will be raised.

IBM ILOG Scheduler objects concerned by rounding policies and integer requirements are listed in the following table.

Scheduler Rounding Policies and Integer Requirements

Object	Floor Rounding	Ceiling Rounding	Must-be-integer
`IloSchedulerEnv`	horizon	origin	
`IloActivityBreakParam`	duration max normal breaks, start break overlap max, end break overlap max	duration min normal breaks, minimal execution duration, start break overlap min, end beak overlap min	disjunctive break type, start break overlap type, end break overlap type
`IloActivityBasicParam`	duration max		external variable value
`IloPrecedenceConstraint`		delay value for types `IloStartsAfterStart` `IloStartsAfterEnd` `IloEndsAfterStart` `IloEndsAfterEnd`	delay value for types `IloStartsAtStart` `IloStartsAtEnd` `IloEndsAtStart` `IloEndsAtEnd`
`IloTimeBoundConstraint`	time bound value for types `IloStartsBefore`, `IloEndsBefore`	time bound value for types `IloStartsAfter`, `IloEndsAfter`	time bound value for types `IloStartsAt`, `IloEndsAt`
`IloResource` capacity enforcement intervals	capacity max	capacity min	date min/max
`IloReservoir`	capacity		initial level
`IloContinuousReservoir`	capacity		
`IloDiscreteResource`	capacity		
`IloDiscreteEnergy`	capacity		
Resource Constraint, Alternative Resource Constraint		capacity	
`IloGranularFunction`			granularity, segments
`IloTransitionParam`		setup and teardown values, transition values	
`IloTransitionTime`		setup and teardown values, transition values	
`IloTransitionCost`	setup or teardown cost max	setup and teardown values, transition values	
`IloIntervalList`			interval start, end, type

Scheduler Engine Limitations

Several built-in constants are available in IBM ILOG Scheduler Engine in order to validate the range of the input data: `IlcIntMin`, `IlcIntMax`, `IlcFloatMin` and `IlcFloatMax`. The first two apply to integer data (dates, capacities, and so forth), whereas the latter correspond to floating-point data (num-to-num and granular functions, reservoir capacity, texture measurement, and so forth). Their actual value depends on the architecture, either 32-bit or 64-bit. Note that Concert Technology also has the corresponding `IloIntMax`, `IloIntMin` limiting constants defined, which are smaller in range than `IlcIntMin`, `IlcIntMax` on 64-bit architecture. On 32-bit architecture, these constraints are equal.

During the search, computations are subject to overflows resulting from invalid input. No systematic check is performed during critical parts of computations, as it would result in a dramatic performance slow down. It is thus necessary to validate the potentially out-of-range input before entering the search. Potential overflows are described below, grouped by functionality class.

General Schedule

In IBM ILOG Scheduler, start and end dates are represented by integer. It is required that the maximum start time plus the maximum duration, or processing time, does not exceed the `IlcIntMax` value. Similarly, the minimum end date minus the maximum duration should not be less than `IlcIntMin`. Note that the default upper bound of the start, end and duration variables of a model activity (`IloActivity`) are equal to `IloIntMax`/2.

If transition times are involved (either by tables or through transition time objects) between a sequence of two resource constraints, there are restrictions on the permissible transition time values. The minimum end time of the preceding activity plus the transition time should not exceed `IlcIntMax`. Similarly, the maximum start time of the second activity minus the transition time should be greater than `IlcIntMin`.

The same kind of limitations apply with the optional delay available with the precedence constraint. Depending on the type of the precedence constraint (`IloEndsAfterEnd`, `IloStartsAtStart`, and so forth), two variables (the preceding and following) are constrained with a delay. To avoid overflows and underflows, the sum of preceding variable and the delay should always remain within the range [`IlcIntMin`, `IlcIntMax`]. Similarly, the difference between the following variable and the delay should be within the same range.

Edge Finder Constraint

The edge finder constraint may be applied on discrete and unary resources, depending on the capacity enforcement level. See the table in Interpretation of Capacity Enforcement Levels for details on when the edge finder is applied.

The edge finder constraint performs propagation based on the grouping of resource constraints as a whole. As a consequence, the sum of all the minimum processing times of activities pertaining to a unary resource constraints should not exceed `IlcIntMax`. Similarly, for discrete resource constraints the sum of all minimum processing times multiplied by the minimum capacity should not be greater than `IlcIntMax`.

Balance Constraint

This constraint is available on a discrete resource with a capacity enforcement of `IloExtended`, or on a reservoir with capacity enforcement level higher than `IloHigh`. On a discrete resource, the balance constraint may overflow if the sum of the maximal capacity required by all resource constraints exceeds `IlcIntMax`.

On a reservoir, the balance constraint may overflow if the sum of the maximal capacity of all the producing resource constraints exceeds `IlcIntMax`, or if the sum of the maximal capacity of all the consuming resource constraints exceeds `IlcIntMax`. The balance constraint also overflows if the maximal capacity of the reservoir exceeds `IlcIntMax/2`.

Moreover, during extra propagation of the precedence graph (available on a discrete resource with precedence enforcement level higher than `IloHigh`), overflow can occur if the sum of the minimal energy (minimal duration * minimal capacity) over all resource constraints on the resource exceeds `IlcIntMax`.

Capacity Resource and Associated Timetable Constraint

In a capacity resource, the maximum capacity should be less than `IlcIntMax`/2, either by construction or when using a member function to set the capacity level (`IlcDiscreteEnergy::setEnergyMax`, `IlcReservoir::setLevelMax`, `IlcDiscreteResource::setCapacityMax`). The same limitation applies on the timetable constructed from a `IlcCapResource`. This includes, for instance, the fourth argument of `IlcCapResource::makeTimetableConstraint`.

Energy Resource and Associated Timetable Constraint

In a discrete energy resource, the maximum energy should be less than `IlcIntMax`/2, either by construction or when using `IlcDiscreteEnergy::setEnergyMax`. In IlOG Scheduler Engine, the same limitation applies on the timetable constructed from an `IlcDiscreteEnergy`. This includes, for instance, the fourth argument of `IlcCapResource::makeTimetableConstraint`.

Functional and Integral Constraint

During propagation of an integral constraint, the values of the integral of a granular function (class `IloGranularFunction`) are internally represented with `IlcNum` variables. It is required that the integral between the minimum start time and the maximum end time do not overflow `IlcFloatMax`.

Practical Advice

Whenever unexpected results occur during the search, it might be profitable to use the trace mechanism (instance of `IlcSchedulerTrace`) and carefully examine the range of input data and internal state of variables to detect whether an overflow occurred.

See Also

IloSchedulerEnv, IloEnforcementLevel, IloGranularFunction, IloResource. Also `IloIntervalList` in the extensions section of the *IBM ILOG Concert Technology Reference Manual*.

Resource Usage Profiles

Description

In Scheduler, each resource can be associated with a minimal and a maximal usage profile.

For capacity resources, the maximal usage profile describes the maximal available quantity of the resource over time whereas the minimal usage profile describes the minimal quantity of the resource that needs to be consumed by activities over time. These usage profiles are represented by instances of the class `IloNumToNumStepFunction`, that is, a step function from real numbers to real numbers. On a discrete resource, the usage profiles represent the instantaneous capacity of the resource over time. On a reservoir, it represents the level of the reservoir over time. On a discrete energy resource, it represents the energy of the resource available per time bucket (see Resource Usage Enforcement).

For continuous reservoirs, the maximal usage profile describes the maximal possible level of the reservoir over time, whereas the minimal usage profile describes the minimal possible level over time. These usage profiles are represented by instances of the class `IloNumToNumSegmentFunction`, that is a piecewise linear function.

For state resources, the maximal usage profile describes the possible states of the resource over time. This usage profile is represented by an instance of the class `IloNumToAnySetStepFunction`, that is, a step function from real numbers to a set of states. The minimal usage profile of the state resource describes a set of time intervals during which the state resource must be used by at least one activity of the environment. This usage profile is represented by an instance of the class `IloIntervalList`.

`IloNumToNumStepFunction`, `IloNumToNumSegmentFunction`, `IloNumToAnySetStepFunction` and `IloIntervalList` are documented in the extensions section of the *IBM ILOG Concert Technology Reference Manual*.

Resource Usage Enforcement

A parameter associated with each resource allows changing the enforcement level of the resource usage profiles (member functions `IloResourceParam::getCapacityEnforcement` and `IloResourceParam::setCapacityEnforcement`). The semantics of these levels depends on the scheduler.

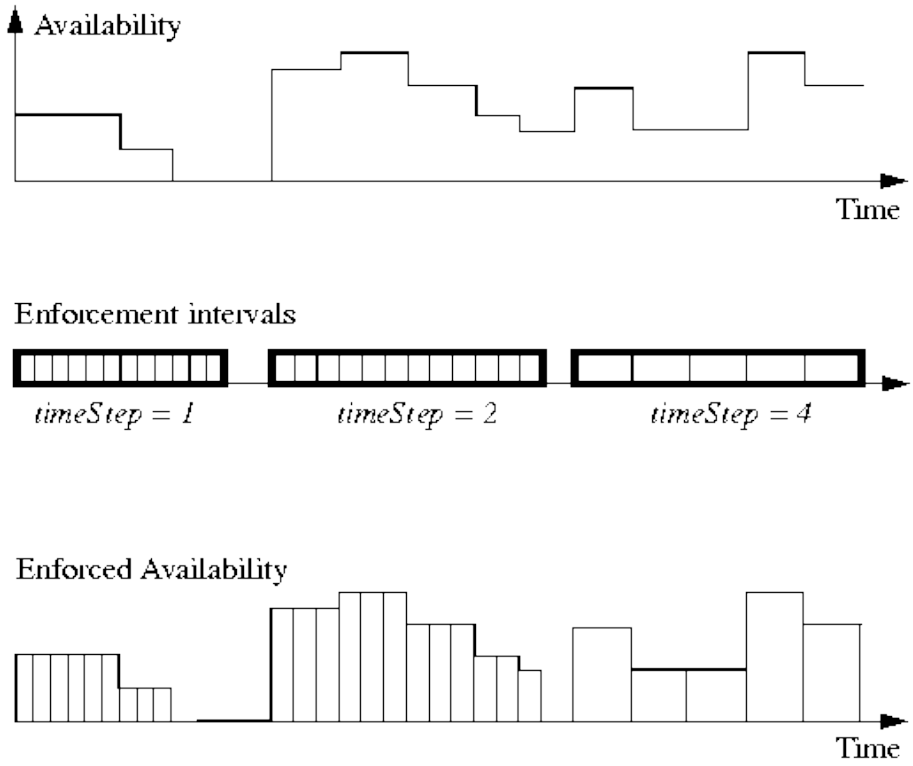
Resource Usage Enforcement Intervals

Each resource can be associated with a list of temporal intervals during which the resource usage profiles (maximal and/or minimal) need to be enforced. Each time interval [timeMin, timeMax) in the list contains a time step that defines the quantum of time used when enforcing the resource usage (see the following figure). By default, when nothing is specified, the resource usage profiles are enforced over the entire scheduling horizon with a time step equal to one.

Note that for a discrete energy resource (instance of `IloDiscreteEnergy`), the maximal and minimal usage profiles define the energy of the resource *per time step*.

Note that for a continuous reservoir, there is no time step. The value associated with each interval is ignored by Scheduler engine.

Resource Usage Profiles Figure



See Also

`IloNumToNumStepFunction`, `IloNumToNumSegmentFunction`, `IloNumToAnySetStepFunction` and `IloIntervalList` are documented in the extensions section of the *IBM ILOG Concert Technology Reference Manual*.

IloResource and IloResourceParam.

Temporal Relations

Description

Scheduler allows the posting of two kinds of temporal relations in the scheduling environment: temporal constraints (`IloPrecedenceConstraint`, `IloTimeBoundConstraint`) between activities of the environment, and temporal relations (precedence, sequence) between two resource constraints on the same resource.

Temporal Constraints Between Activities

Temporal constraints between activities are either instances of the class `IloPrecedenceConstraint` or instances of the class `IloTimeBoundConstraint`. Both classes are a subclass of the IBM® ILOG® Concert Technology class `IloConstraint`.

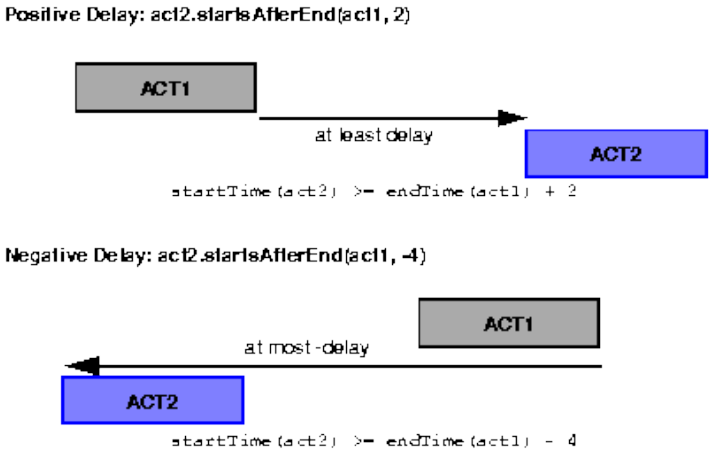
Precedence Constraints

Precedence constraints are instances of the class `IloPrecedenceConstraint`. They restrict the order of activities. They constrain an activity to start or end before, at, or after the start or end time of another activity.

Precedence constraints are created through member functions of the class `IloActivity`. Like all subclasses of `IloConstraint`, they must be added to the model to be considered in the search for solutions.

Precedence constraints involve the concept of delay. Delay is an amount of time (constant or variable) that must elapse between the two variables involved in the precedence constraint. If the delay is negative, it indicates the *inverse* of the maximal duration allowed to elapse between the two variables. In other words, `endpoint2` can occur before `endpoint1`, but the difference between them cannot exceed -`delay`. See the following figure.

Positve and Negative Delays Figure



Precedence constraints between activities can be enforced in different ways by the scheduling algorithm. The member function `IloResource::setPrecedenceEnforcement` can be used to specify how much effort the scheduling algorithm must spend at enforcing precedence constraints between activities.

Time-Bound Constraints

Time-bound constraints are instances of the class `IloTimeBoundConstraint`. They constrain an activity to start or end before, at, or after a given time. They are created by a member function of `IloActivity`. Like all subclasses of `IloConstraint`, they must be added to the model to be considered in the search for solutions.

Temporal relations between resource constraints

Scheduler allows representation of two types of temporal relations between resource constraints on the same resource: precedence relations and sequencing relations. Note that unlike temporal constraints between activities, temporal relations between resource constraints are not objects, but relations expressed with member functions of the class `IloResourceConstraint`.

Before we describe in detail the different temporal relations between resource constraints, we need to introduce the concept of a contributing resource constraint.

In a model, a resource constraint `rct` is said to *surely contribute* if and only if it affects the availability of the resource. That is:

- `rct` has been directly added to the model (not through a metaconstraint), and
- the minimal processing time of the activity of `rct` is strictly greater than zero if the time extent of `rct` is `IloFromStartToEnd`, and
- the minimal capacity required by `rct` is strictly greater than zero if `rct` is a capacity resource constraint.

In a model, a resource constraint `rct` is said to *not possibly contribute* if and only if it is sure that `rct` will not affect the availability of the resource. That is:

- the opposite of `rct` has been directly added to the model (not through a metaconstraint), or
- the processing time of the activity is equal to zero and the time extent of `rct` is `IloFromStartToEnd`, or
- `rct` is a capacity resource constraint and the capacity required by `rct` is equal to zero.

Precedence Relations

A precedence relation is defined between two resource constraints with the member function `setSuccessor()`.

`rct1.setSuccessor(rct2)` states that if `rct1` and `rct2` surely contribute to the model, then the activity of `rct2` is constrained to execute after the activity of `rct1` on the resource (that is, the start time of the activity of `rct2` is greater than or equal to the end time of the activity of `rct1`).

Precedence relations between resource constraints can be enforced in different ways by the scheduling algorithm. The member function `IloResource::setPrecedenceEnforcement` can be used to specify how much effort the scheduling algorithm must spend at enforcing precedence relations between resource constraints.

Sequencing Relations

Sequencing relations are expressed with the member functions `IloResourceConstraint::setNext`, `IloResourceConstraint::setSetup`, `IloResourceConstraint::setNotSetup`, `IloResourceConstraint::setNotNext`, `IloResourceConstraint::setTeardown` and `IloResourceConstraint::setNotTeardown`.

`rct1.setNext(rct2)` states that if `rct1` and `rct2` surely contribute to the model, then no other resource constraint `rct` that surely contributes in the model can be executed between `rct1` and `rct2`.

`rct1.setNotNext(rct2)` states that if `rct1` and `rct2` surely contribute to the model, then another resource constraint `rct` that surely contributes to the model must execute between `rct1` and `rct2`.

`rct.setSetup()` states that if `rct` surely contributes to the model, then `rct` is the first resource constraint to execute on the resource.

`rct.setNotSetup()` states that if `rct` surely contributes to the model, then `rct` is not the first resource constraint to execute on the resource.

`rct.setTeardown()` states that if `rct` surely contributes to the model, then `rct` is the last resource constraint to execute on the resource.

`rct.setNotTeardown()` states that if `rct` surely contributes to the model, then `rct` is not the last resource constraint to execute on the resource.

Sequencing relations between resource constraints can be enforced in different ways by the scheduling algorithm. The member function `IloResourceParam::setSequenceEnforcement` can be used to specify how much effort the scheduling algorithm must spend at enforcing sequencing relations between resource constraints.

Precedence Constraints On Activities vs. Precedence Relations On Resource Constraints

This section answers the question: Should one use precedence constraints between activities, or precedence relations between resource constraints? This question arises because at first glance, a precedence constraint `startsAfterEnd` between two activities and a precedence relation `setSuccessor` between two resource constraints look very similar.

There are in fact three essential differences.

1. Precedence relations between resource constraints have an effect on the start and end time of the corresponding activities only if the resource constraints surely contribute to the model, that is, if they affect for certain the availability of the resource. For example, a precedence relation between two resource constraints that requires a null quantity of the same discrete resource will not constrain the start and end time of the activities. From this point of view, a precedence relation between resource constraints `rct1` and `rct2` can be seen as a metaconstraint (`rct1` does not contribute or `rct2` does not contribute or `act2.startsAfterEnd(act1)`). Precedence constraints between activities constrain the start and end time of activities regardless of which resource the activities require.
2. A precedence constraint between two activities is a C++ object, whereas a precedence relation between two resource constraints is expressed simply by enforcing that relation on the resource. From this point of view, precedence relations between resource constraints are lighter than precedence constraints between activities. On the other hand, since precedence constraints between two activities are instances of `IloConstraint`, the constraints can be used in a metaconstraint. This is not possible for precedence relations between resource constraints.
3. The formalism of precedence constraints between activities allows representation of some (constant or variable) delays between activities. This is not possible with precedence relations between resource constraints.

See Also

IloActivity, IloActivityConstraintsParam, IloPrecedenceConstraint, IloTimeBoundConstraint, IloResourceConstraint, IloResource, IloResourceParam.

Transition Times

Description

Given two activities `A1` and `A2`, the transition time between `A1` and `A2` on a resource `R` is the amount of time that must elapse between the end of `A1` and the beginning of `A2`, when `A1` precedes `A2` on resource `R`.

A transition time (instance of `IloTransitionTime`) can be created on a resource.

By default, the transition time on a resource is not suspended by the breaks defined on that resource. The member function `IloTransitionTime::setSuspended` allows specifying a transition time that is suspended by the breaks of the resource. See Breaks and Transition Times for an illustration of suspended transition times.

A parameter associated with each resource allows one to change the level of enforcement of transition times (member functions `IloResourceParam::getTransitionTimeEnforcement` and `IloResourceParam::setTransitionTimeEnforcement`). The semantics of these levels depends on the scheduler.

The list of time intervals over which the transition times must be enforced is given as a parameter of the resource, and can be modified with the member functions `IloResource::addTransitionTimeEnforcementInterval`, and `IloResource::removeTransitionTimeEnforcementInterval`.

An instance of `IloTransitionTime` associates a resource with either a transition parameter (instance of `IloTransitionParam`) or a user-defined transition time object (subclass of `IloTransitionTimeObject`.)

Transition Time Defined with a Transition Parameter

A transition parameter (instance of `IloTransitionParam`) is a square table describing the transition between activities, with two arrays describing the activity setup and teardown values. An integer transition type is associated with each activity. The accessors are the functions `IloActivity::getTransitionType` and `IloActivity::setTransitionType`. They allow you to define instances of `IloTransitionTime` from `IloTransitionParam` where the arrays are indexed by the transition types of activities.

Note that when a transition time defined with an `IloTransitionParam` is associated with a resource, activities of different types cannot overlap on that resource.

Transition Time Defined with a Transition Time Object

A transition time object is a subclass of `IloTransitionTimeObject` defined with the macro `ILOTRANSITIONTIMEOBJECT0`.

This macro makes it easy to define your own transition time together with its extraction function, in case a simple square table as the one provided by `IloTransitionParam` is not sufficient for your purposes.

See Also

IloActivity, IloTransitionTime, IloTransitionParam, IloResource, IloTransitionTimeObject, ILOTRANSITIONTIMEOBJECT0, IloResourceParam.

Transition Costs

Description

A unary resource can only process one activity at a time, so all activities requiring the same unary resource must be chronologically ordered to find a solution. As a result, in any solution to a problem that includes a unary resource, each unary resource defines a directed path through all the activities requiring it.

Between each pair of consecutive activities, some cost may be incurred to switch the resource from processing the first activity to processing the second. These costs may be related to modifications to the resource that require manpower, material, and energy, such as adjusting or purging a machine.

In Scheduler, transition cost is defined as the cost between an activity and the activity that will execute next to it on a unary resource. In addition, Scheduler lets you define a setup cost for the activity that starts the usage of the resource, and a teardown cost for the activity that ends the usage of the resource.

Several transition costs (instances of `IloTransitionCost`) can be created on a unary resource.

An instance of `IloTransitionCost` associates a unary resource with either a transition parameter (instance of `IloTransitionParam`) or a user defined transition cost object (subclass of `IloTransitionCostObject`).

Once a transition cost has been defined on a unary resource, the sum of the costs, the setup and teardown costs on the resource, and the cost of the next and previous transitions of a given activity can be accessed as instances of `IloNumVar` variables (see Cost Types Figure).

Transition Cost Defined with a Transition Parameter

A transition parameter (instance of `IloTransitionParam`) is a square table describing the transition between activities, with two arrays describing the activity setup and teardown values. Scheduler associates an integer transition type with each activity. The accessors are the functions `IloActivity::getTransitionType` and `IloActivity::setTransitionType`. They allow you to define instances of `IloTransitionCost` from `IloTransitionParam`, where the arrays are indexed by the transition types of activities.

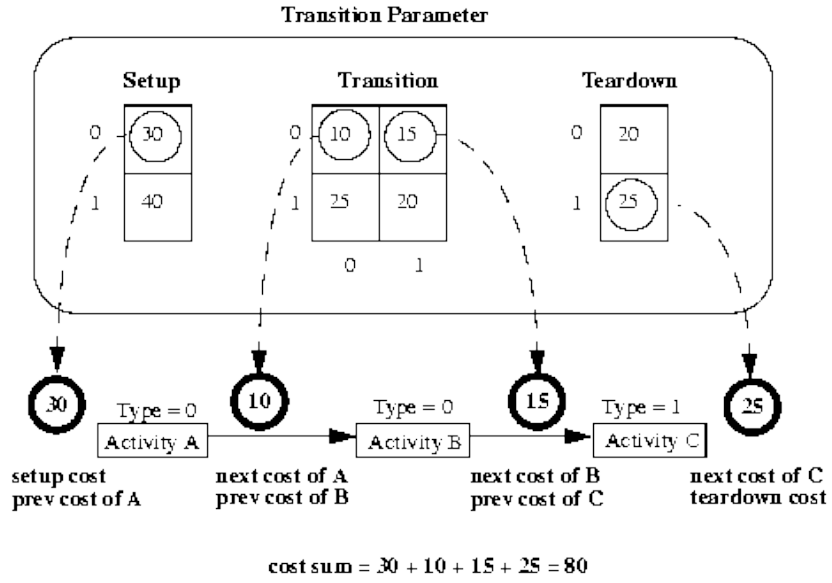
Transition Cost Defined with a Transition Cost Object

A transition cost object is a subclass of `IloTransitionCostObject` defined with the macro `ILOTRANSITIONCOSTOBJECT0`.

This macro makes it easy to define your own transition cost together with its extraction function in case a simple square table as the one provided by `IloTransitionParam` is not sufficient.

Cost Types Figure

This figure shows an example of setup, transition, and teardown costs.



See Also

IloTransitionCost, IloTransitionParam, IloTransitionCostObject, ILOTRANSITIONCOSTOBJECT0, IloUnaryResource.

A Look at Scheduler Modeling and Solver

Description

The Scheduler API allows you to model scheduling problems without any assumptions on the way a solver will produce a solution. Scheduler is used to build a problem definition; modeling objects are extracted from the problem definition to the solver; and the solver uses the extracted objects to solve all or part of the problem. When used with Solver, the extraction of the modeling classes first creates corresponding instances of classes in another layer; this layer is Scheduler Engine. Scheduler Engine is specific to Solver solution techniques. The

Scheduler extractor is the object that interprets the problem definition objects and creates the Schedule Engine objects for solving the problem. As with the Scheduler Engine, the Scheduler extractor is specific to Solver solution techniques.

At extraction time, the Scheduler Engine instances are created and initialized given the data in the parameters of the Scheduler instances. If a model element is changed, and if a Scheduler Engine object is concerned by this change, a re-extraction will occur before the next Solver search phase. The incorporation of changes to the model, during search, follows the same protocol as Solver: unless otherwise stated, no model changes will be taken into account until the next Solver search phase. See the *IBM ILOG Solver Reference Manual* for more details.

A benefit of the modeling concept and parameter framework of Scheduler is that it allows users to manage large scheduling problems with light memory usage. Scheduler classes have a lighter memory usage than the corresponding classes in Scheduler Engine.

Correspondence between Scheduler and Scheduler Engine Classes

The following table indicates the correspondence between Scheduler classes and Scheduler Engine classes. The Scheduler classes listed are extracted to the corresponding Scheduler Engine classes.

Class Correspondence Between Scheduler and Scheduler Engine

Scheduler Class	Scheduler Engine Class
`IloActivity`	`IlcActivity`
`IloResource`	`IlcResource`
`IloCapResource`	`IlcCapResource`
`IloDiscreteResource`	`IlcDiscreteResource`
`IloUnaryResource`	`IlcUnaryResource`
`IloDiscreteEnergy`	`IlcDiscreteEnergy`
`IloReservoir`	`IlcReservoir`
`IloContinuousReservoir`	`IlcContinuousReservoir`
`IloStateResource`	`IlcStateResource`
`IloTimeBoundConstraint`	`IlcTimeBoundConstraint`
`IloPrecedenceConstraint`	`IlcPrecedenceConstraint`
`IloResourceConstraint`	`IlcResourceConstraint and IlcAltResConstraint`

Modeling Scheduler Engine Classes that are not in Scheduler

There are some Scheduler Engine classes that do not have counterparts in Scheduler because the Scheduler Engine API is very specific to Solver techniques. Examples include the break lists (instances of the class `IlcBreakList`), and the timetable (instance of subclass of `IlcIntTimetable` and `IlcAnyTimetable`). These classes are actually Solver objects involved in the global constraints and whose contents represent functions defined over the time axis.

The time axis is represented in Scheduler with a numerical integer type, a data of type `IloNum` or instances of class `IloNumVar`. The break list and timetables are modeled in Scheduler with a set of parameters on the resources whose content is a function on the integer time axis.

The following Scheduler classes are used in modeling break lists and timetables. Note that `IloNumToAnySetStepFunction`, `IloNumToNumStepFunction`, `IloNumToNumSegmentFunction`, and

`IloIntervalList` are documented in the extensions section of the *IBM ILOG Concert Technology Reference Manual*.

`IloIntervalList` : Breaks list, definition interval of the set of timetables on a resource, must-be-in-use constraint on a state resource.

`IloNumToAnySetStepFunction` : Maximal set of states of the timetables of an instance of `IlcStateResource`.

`IloNumToNumStepFunction` : Maximal and minimal capacity, initial content of a capacity resource.

`IloNumToNumSegmentFunction` : Maximal and minimal capacity, initial content of a continuous reservoir.

Scheduler Engine Classes

The function type definitions, enumerations, and some classes of Scheduler Engine are designed only for writing search goals or constraint propagation functions. Examples include the following:

- `IlcResourceIterator`, `IlcActivityDeltaIterator` and any type of iterators on the Solver object involved in the problem.
- `IlcActivityIteratorFilter` and all other enumerations used by iterators.
- `IlcResourceSelectorObject` and any selector, evaluator, or predicate object used in the Solver search procedure.
- Any instance of a subclass of `IlcDemon`.

Since these classes are not related to modeling, they do not have any counterpart in Scheduler.

Pure Scheduler Classes

Pure Scheduler classes are classes or enumerations designed for the parameterization of the basic model objects. These classes are only taken into consideration at extraction time in order to create the related constrained data structure. Changing this object will lead to a re-extraction if required.

Examples include `IloActivityBreakParam`, `IloNumToNumStepFunction` (see *IBM ILOG Concert Technology Reference Manual*), and `IloEnforcementLevel`.

Implicitly Created Solver Variables

In order to implement the propagation of global constraints, to write heuristic selectors and goals, and to let the user define constraints, the Scheduler Engine object may automatically create Solver variables or data structures. As these constrained objects are useless in the initial model, they are not created in the Scheduler model, but only at the beginning of the search.

The Solver variables created, and their corresponding Scheduler Engine classes, are shown in the following table.

Solver Variables and their Corresponding Scheduler Engine Class

Solver Variables	Corresponding Scheduler Engine Class
Duration and overlap variables.	`IlcActivity`
Next and prev variables of the sequence constraint on a unary resource (the setup and teardown variables are also created).	`IlcResourceConstraint`
Index variable of the alternative of the resource.	`IlcAltResConstraint`

The data structures used in the global constraint of the resource are the timetables and the precedence graphs.

Global Constraints

Global constraints are algorithms that enforce the conditions on the set of resource constraints declared on a resource. Different global constraints can be selected to enforce the same constraint. For example, the maximum capacity limitation on a unary resource can be enforced by the timetable constraint, the disjunctive constraint, or the edge finder algorithm.

This notion is very specific to constraint programing techniques, therefore there is no direct way to explicitly declare which Solver global constraint to use at the Scheduler model level.

There are two ways for the Scheduler Engine extractor to determine which global constraint to add to the algorithm. These two methods are described in Adding Global Resource Constraints from the Scheduler Model.

API

The Scheduler Engine and Scheduler APIs are very similar. For example, the read accessors on objects are the same, but the argument types and return values are different. Compare the following:

```
IloNumVar IloActivity::getStartVariable() const;
IlcIntVar IlcActivity::getStartVariable() const;
IloBool IloActivity::isBreakable() const;
IlcBool IlcActivity::isBreakable() const;
```

The main difference lies in the fact that Scheduler is a pure modeling API, with no restrictions on editing capabilities; whereas Scheduler Engine must respect the properties of Solver. That is the reason that only monotonic accessors are allowed during the search.

```
void IloActivity::setBreakable(IloBool breakable) const;
void IloActivityBasicParam::setBreakable
                           (IloBool breakable) const;
void IlcActivity::setBreakable(IlcBool breakable);
```

The last function always applies before entering the search. During Solver search it is forbidden to set an activity to be breakable. The same type of arguments holds for the following:

```
void IloResourceConstraint::setNext
                    (const IloResourceConstraint& next) const;
void IloResourceConstraint::unsetNext() const;
void IlcResourceConstraint::setNext
                    (IlcResourceConstraint next);
void IlcResourceConstraint::unsetNext() const;
```

The last function is not valid during the search because it is non-monotonic.

Some API elements may have the same purpose, and apply to corresponding classes, but have a slightly different semantics in Scheduler and Scheduler Engine. For example:

```
void IloResource::keepOpen(IloBool open);
void IlcResource::close();
```

The first function (with an argument of `IloTrue`) specifies that not all the resource constraints are declared in the model and that therefore, the solver should allow the addition of new resource constraints during search. The second function specifies that the `IlcResource` is closed; that is, that it is not kept open during the search.

Nevertheless, the second function is such that the instance of `IlcResource` cannot be reopened, even when outside the search. When the function `IloResource::keepOpen` is called on the model resource, a re-extraction of the resource is required and the resource will not be closed during the next search.

A parameter object in Scheduler can be used to model data for multiple Scheduler Engine classes. For example, `IloIntervalList` is used to store data for both `IlcBreakList` and `IlcIntTimetable`. The `IloIntervalList` API has a natural correspondence to the API of the Scheduler Engine objects. For example, if the instance of `IloIntervalList` is used to define the break list of a resource, the following API:

```
void IloIntervalList::addInterval(IloNum start, IloNum end,
```

```
                                    IloNum type = 0) const;
void IloIntervalList::removeInterval
                               (IloNum start, IloNum end) const;
```

corresponds to:

```
void IlcBreakList::addBreak(IlcInt start, IlcInt end,
                            IlcInt type = 0);
void IlcBreakList::removeBreak(IlcInt start, IlcInt end);
```

on the corresponding instance of `IlcBreakList`.

If the instance of `IloIntervalList` is used to define the set of timetables on a resource, these accessors will set up the definition of the timetable constraints. That determines the time interval definition of the instance of `IlcIntTimetable` managed by the global capacity resource constraint.

`IloIntervalList` is documented in the extensions section of the *IBM ILOG Concert Technology Reference Manual*.

Solver Extensions

As Solver is based on constraint programing, the interpretation of the Scheduler model is quite natural. However, some functions of Solver cannot be easily accessed by Scheduler. This is mainly related to writing code dependent upon the current state of the data in the Solver. So, to access these functions of Solver, the user must use Scheduler Engine to implement the code. Examples of this include extending or creating constraints and demons, or extending or creating search goals. Examples of some of the Scheduler Engine functions offered for this purpose are shown in the following table.

Accessing Certain Solver Functions

To Access the Solver Function	Use Scheduler Engine Class (examples)
For search goals and constraints:	
Iterators	`IlcResourceConstraintIterator`
Constrained data class	`IlcIntTimetable`
Members of classes	`IlcBool IlcUnaryResource::isRanked;void IlcResourceConstraint::rankFirst; IlcIntVar IlcAltResConstraint::getIndexVariable;`
For constraints and demons:	
Delta Domain Management	`IlcInt IlcIntTimetable::getRangeTimeMin() const;IlcResourceConstraintDeltaIterator`
Constraint events	`void IlcIntTimetable::whenRangeInterval (const IlcDemon g);void IlcResource::whenPredecessors (const IlcDemon g);`
For global resource constraints:	
Creation and inspection of global resource constraints	`void IlcResource::setBreaks(IlcBreakList bl);IlcBool IlcResource::hasTimetableConstraint() const;`
Parameterization of global resource constraints	`void IlcDiscreteResource::setEdgeFinder (IlcInt level);`
For search goals:	

Selector object	`IlcActivitySelectorObject`
Choice points	`IlcTryRankFirst(IlcResourceConstraint rct);`
Member functions	`void IlcActivity::postpone();`

Adding Global Resource Constraints from the Scheduler Model

Description

There are two ways for the extractor to determine which global resource constraint is implicitly added to the Solver from Scheduler: the content of the model or the enforcement level parameters.

The content of the model is used when the Solver has only one way to enforce the constraint. For example, the transition cost on a resource can only be handled by the Solver with a sequence constraint posted on the resource. Therefore, a sequence constraint is automatically posted if the unary resource is defined with a transition cost. The goal `IloRankForward` needs the disjunctive constraint on the unary and state resources. Therefore, when the `IloRankForward` is used, the disjunctive constraint is automatically posted.

The enforcement level parameter is used along with categories of global constraints and resource classes to select one or several global resource constraints. The higher the enforcement level, the higher the computational time allocated by the Solver for the constraint propagation on the resource. For example, a capacity enforcement on a unary resource will add the disjunctive constraint at `IloBasic` level and set the edge finder at a higher level (such as `IloMediumHigh`). Details of the enforcement level parameter are discussed in Resource Enforcement as Global Constraint Declaration.

A global resource constraint is therefore added to the Solver if either a component of the model needs it, or if an interpretation of an enforcement level exists. However, if the parameter is ignored in the Scheduler model (for example, with `IloActivity::ignoreResourceConstraints`), the global constraint is not posted, regardless of any other modeling data.

Goals

`IloRankForward` and `IloRankBackward` add the disjunctive constraint on the related unary and state resources.

`IloSequenceForward` and `IloSequenceBackward` add the sequence constraint on the related unary resources.

Parameters

`IloIntervalList` and `IloNumToNumStepFunction` are documented in the extensions section of the *IBM ILOG Concert Technology Reference Manual*.

A non-empty instance of `IloIntervalList` as break list parameter adds the breaks constraint on the related resources.

A non-empty instance of `IloIntervalList` as capacity enforcement parameter adds the timetable constraint on the related resources.

A non-empty instance of `IloIntervalList` as transition time enforcement parameter adds the timetable constraint on the related resources.

A non-empty instance of `IloIntervalList` as must-be-in-use enforcement parameter adds the timetable constraint on the related state resources.

A non-empty instance of `IloNumToNumStepFunction` as any of capacity max, capacity min, or initial level parameter adds the timetable constraint on the related capacity resources.

A non-empty instance of `IloNumToNumSegmentFunction` as any of capacity max, capacity min, or initial level parameter adds the timetable constraint on the related continuous reservoir.

A non-empty instance of `IloNumToAnySetStepFunction` as initial possible states adds the timetable constraint on the related state resources.

Closure

If a resource is *not* declared to be kept open, then the global constraints that manage a closed resource are added.

Precedence Graph

The usage of any of the following member functions of adds the precedence graph global constraint on the resource(s) of the invoking resource constraint.

- IloResourceConstraint::setNext
- IloResourceConstraint::setNotNext
- IloResourceConstraint::setNotSetup
- IloResourceConstraint::setNotTeardown
- IloResourceConstraint::setSetup
- IloResourceConstraint::setSuccessor
- IloResourceConstraint::setTeardown

Transition Time and Cost

The usage of the following constructor adds the disjunctive constraint or the type timetable constraint.

```
IloTransitionTime(IloResource resource,
                  IloTransitionParam param,
                  const char* name = 0);
```

Which constraint is added depends upon the type of the argument `resource` and its capacity enforcement parameter. The disjunctive constraint of the unary and state resource subsumes the timetable constraint.

The usage of the following constructors will add the sequence constraint on the argument `resource`.

```
IloTransitionCost(IloUnaryResource resource,
                  IloTransitionParam param,
                  const char* name = 0);
IloTransitionCost(IloUnaryResource resource,
                  IloTransitionParam param,
                  IloBool isNext,
                  const char* name = 0);
```

Balance Constraint

Description

The balance constraint is a global constraint that can be used to enforce the capacity of a discrete resource or a discrete reservoir.

The balance constraint alone is sufficient to ensure that the minimal and maximal capacity of the resource are enforced, provided that the minimal or maximal capacity of the resource do not vary over time.

Be aware that if the minimal or maximal capacity of the resource does vary over time, then the balance constraint alone is not sufficient and it must be used together with the timetable constraint in order to ensure the soundness of the search.

The balance constraint internally maintains a precedence graph between the time-points (start, end) of the activities on the resource. It propagates by analyzing this graph in order to discover new time bounds for activities as well as new precedence relations between activity time-points.

In general, the balance constraint is useful as soon as there is a significant number of precedence relations between the time-points of the activities on the resource. Note that precedence constraints (for example, `IlcActivity::startsAfterEnd`) impose such precedence relations.

It is important to note that the memory requirement of the balance constraint is higher than the one of the disjunctive or the timetable constraint because of the internal precedence graph. Furthermore, if *n* denotes the number of activities on the resource, the worst-case complexity of the balance constraint is in *O(n3)* and the average complexity is in *O(n2)*. Therefore, the balance constraint should be used for small or medium-size problems, where a strong pruning that exploits precedence relations is required.

Principle

The balance constraint internally maintains a precedence graph between the time-points (start, end) of the activities on the resource. More precisely:

- The vertices of this graph are the start and end time-points of the activities that uses the discrete resource or the reservoir; and
- The directed edges are precedence relations between those time-points: strict or non-strict precedence. These precedence relations may come from the initial scheduling problem, from search decisions or from the propagation of the balance constraint or other global constraints.

The balance constraint algorithm computes the resource level balance just before and just after each vertex in the precedence graph, taking into account the quantity of resource produced or consumed by the vertices that surely and/or possibly execute before the current vertex. It allows one to restrict the domain of the time variables of activities as well as to discover new precedence relations between time-points.

As an example of propagation, consider a very simple scheduling problem with a reservoir with maximal level 2, initial level 0, two temporally ordered producing activities *p1* and *p2* that produce 2 units of reservoir at their end time-point, and four temporally ordered consuming activities *c1*, *c2*, *c3* and *c4* that consume 1 unit of reservoir at their start time-point. The balance constraint will automatically order all the activities as follows:

$end(p1) \leq start(c1) \leq start(c2) \leq end(p2) \leq start(c3) \leq start(c4)$

Balance Constraint and Timetable Constraint

The main difference between the balance constraint and the timetable constraint is that the timetable constraint relies on the absolute position of activities in time whereas the balance constraint relies on their relative position. The timetable constraint estimates the resource level at absolute dates, whereas the balance constraint estimates the resource level at nodes of the internal precedence graph, given their relative positioning. In general, these nodes correspond to time-points that are still not instantiated.

One can show that neither of the two global constraints is dominated by the other: both can perform some propagation that the other one would not detect.

Both global constraints require a complex internal structure (the timetable and the precedence graph). For the balance constraint, this internal structure is heavier as it must store the relative position of activities but it allows, in many cases, a stronger propagation than the timetable constraint.

As stated earlier, the balance constraint is sufficient to ensure that any solution on a discrete resource or a reservoir satisfies the resource maximal and minimal capacity constraint (provided that the maximal or minimal capacity of the resource does not vary over time). Thus, it is entirely possible to use the balance constraint alone to enforce the capacity of a discrete resource or a reservoir. Nevertheless, as the balance constraint is more

complex than the timetable constraint, it is a good idea to systematically post a timetable constraint whenever a balance constraint is posted. It may allow some additional propagation as well as a global speed up of the propagation.

Balance Constraint and Disjunctive Constraint

The balance constraint can be considered as a generalization of the disjunctive constraint for discrete resources and reservoirs.

On a discrete resource, the balance constraint subsumes the disjunctive constraint: everything that is found by the disjunctive constraint will also be found by the balance constraint, but some adjustments are found by the balance constraint that cannot be found by the disjunctive constraint.

Consider an example of a discrete resource of maximal capacity 10, and three activities `act1`, `act2` and `act3` of duration 10 that require four units of this discrete resource. The three activities cannot overlap. Suppose that we have the following precedence constraints:

```
act1.endsAfterStart(act3, 1)
act3.startsAfterStart(act2)
act3.endsAfterEnd(act2)
act2.endsAfterStart(act3, 1)
```

The balance constraint will discover that activity `act1` cannot start to be processed before the end of activity `act2`. A consequence is that the start time of `act1` must be greater than 10. Note that neither the disjunctive constraint nor the timetable constraint on the discrete resource would deduce this adjustment.

Using a Balance Constraint

To create a balance constraint, use the member function `IlcCapResource::makeBalanceConstraint`.

Remember that the balance constraint created by one of these member functions must be posted in order to be taken into account.

See Also

IlcDiscreteResource, IlcReservoir.

Disjunctive Constraint

Description

The standard way of dealing with *resource constraints* in Scheduler Engine is to create a Timetable representing available capacity over time. In the case of discrete, unary, and state resources an alternative representation of capacity constraints is available. This alternative representation consists of posting a unique, global, *disjunctive constraint* that states that if the resource is used by two activities throughout two time intervals *[ti1 ti2)* and *[tj1 tj2)* and the activities are incompatible, (that is, they cannot be scheduled simultaneously), then either *ti2* is less than or equal to *tj1*, or *tj2* is less than or equal to *ti1*.

Creating a Disjunctive Constraint

To create such a unique, global, disjunctive constraint, use the member functions `IlcDiscreteResource::makeDisjunctiveConstraint` or `IlcStateResource::makeDisjunctiveConstraint`.

Timetables Differ from Disjunctive Constraints

Here are highlights of the differences between those two representations.

- The disjunctive representation deals only with requiring activities: to specify that the resource is not available over a given time interval, the user must create a “fake” activity that requires the resource over that interval. The use of the disjunctive representation may therefore prove costly (in CPU-time and memory) if a large collection of “fake” activities is created.
- The disjunctive representation is more CPU-time consuming, but the propagation of global disjunctive constraints may result in more precise time bounds than the propagation of the corresponding timetable constraints. In the context of a particular scheduling application, more CPU-time may be spent propagating the disjunctive constraints, but this extra propagation may result in a better exploration of the search space and, consequently, in a drastic improvement of the overall CPU-time.
- When no "fake" activities are created, the disjunctive representation requires less memory than the timetable representation.

Using Timetables with Disjunctive Constraints: Performance Considerations

It is possible to use both of the two representations in the same application and for the same resource. In such a case, redundant constraint propagation is performed. A priori, the CPU-time increases, but the combined effect of:

- The more effective propagation of the disjunctive constraint and
- The removal of "fake" activities,

may, in the end, make the combined use of both representations a more viable approach than the use of either of them separately.

Implementation

Mathematically, the disjunctive constraint associated with a unary resource `R` implements the following relation: for any two activities `A` and `B` that require `capacity(A)` and `capacity(B)` units of `R` (`capacity(A)` and `capacity(B)` can be Boolean variables) over two time intervals `[start(A) end(A))` and `[start(B) end(B))`,

```
((end(A) + getTransitionTime(A, B) <= start(B))
  || (end(B) + getTransitionTime(B, A) <= start(A))
  || (end(A) - start(A) == 0)
  || (end(B) - start(B) == 0)
  || (capacity(A) == 0)
  || (capacity(B) == 0))
```

That disjunctive constraint is called *global* because the same constraint applies to every `A` and `B`.

See Also

Edge Finder, IlcDiscreteResource, IlcResource, IlcStateResource, IlcUnaryResource, Ranking , Timetable, Transition Time in Scheduler Engine.

Durability

Description

Scheduler Engine offers an easy way to build applications where the same resources are used successively in different scheduling problems and information about resource availability is kept from one problem to another.

Such resources are called *durable resources*. The information about resource availability that is kept from one problem to another is stored in the timetable constraints of the resources.

The typical steps that may be encountered by an application using durability are:

- Create durable resources in a model.
- Extract these resources on a durable scheduler that is an instance of `IlcSchedule`. The durable scheduler can be seen as an object that stores the information that is kept from one problem to another (timetables of durable resources).
- Different schedules (instances of `IlcSchedule`), called computation schedules, declare an interest in using some durable resources.
- Once a computation schedule has access to the resources it needs (we say that the resources are locked by the computation schedule), a complete scheduling problem can be defined and solved.
- At any point, the computation schedule can unlock any of the locked resources; the unlocked resources keep their timetable information.
- After a resource is unlocked, further computation done on the computation schedule does not interfere with the unlocked resource, except the backtracking: that is, backtracking a decision that modified the timetable of a resource will undo the modification, even if the resource has been unlocked.

Example

The main elements of the API that allow the use of durable resources across different scheduling problems are illustrated in the following example.

The following code creates a durable schedule that gets its durable resources extracted from a Scheduler model.

```
IloEnv env0;              // Allocation environment
IloModel model0(env0);
IloUnaryResource resource(env0);
model0.add(resource);
IloSolver solver0(env0);
IlcScheduler durSched(solver0);
durSched.setDurable();   // A durable schedule is created
solver0.extract(model0); // Extract durable resources
durSched.close();        // All durable resources were defined
```

Now we define a goal `SolveSubProblem`, which is in charge of scheduling one activity on the durable unary resource. In the goal “wrapper,” the computation resource is obtained from the durable scheduler.

```
ILCGOAL1(SolveSubProblemIlc,
         IlcUnaryResource, compResource) {
  IloSolver solver = getSolver();
  IlcSchedule compSched(solver, 0, 24); // computation schedule
  IlcActivity act(compSched, 8);
  compSched.lock(1, compResource);   // Locking before using
  solver.add(act.requires(compResource));
  solver.startsNewSearch(IlcSetTimes(compSched));
  solver.next();
  compSched.unlock(1, compResource); // Resource no longer needed
  solver.out() << act << endl;      // Display scheduled activity
  solver.endSearch();
  return 0;
}
ILOCPGOALWRAPPER2(SolveSubProblem, solver,
                  IlcScheduler, durSched,
                  IloUnaryResource, resource) {
  return SolveSubProblem(solver, durSched.getUnaryResource(resource));
}
```

The following code creates a computation solver to solve a first instance of `sub-problem` on the durable resource.

```
IloEnv env1; // Computation environment
IloSolver solver1(env1);
solver1.solve
       (SolveSubProblem(env1, solver1, compResource));
env1.end();
```

It outputs: `[0 -- 8 --> 8]`. The activity of this first sub-problem is scheduled to start at time 0. From now on, the durable resource remains occupied on the time interval [0, 8). Thus a second computation solver trying to place another activity on the resource will be aware of the resource availability:

```
IloEnv env2; // Computation environment
IloSolver solver2(env2);
solver2.solve
       (SolveSubProblem(env2, solver2, compResource));
env2.end();
```

It outputs: `[8 -- 8 --> 16]`. The activity of the second `sub-problem` is scheduled to start at time 8.

Writing Multi-Threaded Applications

It is possible to use durable resources in multi-threaded applications. Scheduler is multi-thread safe if each thread uses a different solver for creating scheduler objects and those objects are not accessed across different threads.

Durable resources may be concurrently used by different threads through the methods `lock` and `unlock`.

- The `lock` method may block while waiting for all arguments to be unlocked by other threads.
- The `unlock` ethod allows other threads to gain access to its arguments.

Limitations

Currently, the only resources that can be managed as durable resources are discrete and unary resources, and energy resources.

The durable resources are shared by several independent computational solvers. The whole set of requirements on a durable resource is not known. Also, the following limitations exist:

- The minimal capacity/energy of a durable resource is undefined: `setCapacityMin`, `getCapacityMin`, `setEnergyMin`, `getEnergyMin` will not work reliably.
- The maximal capacity/energy of a durable resource should only be changed on the durable environment and protected with a lock. It is not reversible.
- A durable resource cannot be closed.

Managing the Initial Requirement Amount on a Durable Resource

When entering a new session using durable resources, or when a commitment from a previous search phase must be removed (to figure out a rollback on a database), one needs specific functions for having non-monotonic, non- reversible behavior. Those functions are:

```
void IlcCapResource::incrDurableRequirement
                    (IlcInt t1, IlcInt t2, IlcInt capacity,
IlcInt outward = IlcTrue, IlcInt breaksDuration = 0);
void IlcCapResource::incrDurableRequirement
                    (IlcIntToIntStepFunction func);
```

These functions modify the requirement amount that corresponds to an activity starting a `t1`, ending at `t2` and requiring the capacity of the argument `capacity`. The signature of this function, using an instance of `IlcIntToIntStepFunction`, nearly behaves like the iteration of the basic signature on each step of the function.

If the argument `capacity` is greater than 0, the effect of the function is to add `capacity` to the requirement amount to the resource; that is, to actually decrease the available capacity in the resource.

If the argument `capacity` is less than 0, the effect of the function is to remove `capacity` from the requirement amount of the resource; that is, to increase the available capacity in the resource.

The coherency of the requirement amount with respect to the resource capacity is under the responsibility of the user. For example, one should be cautious that the requirements that are undone do not exceed the activity requirements that are committed on the resource when a search using the durable resource is launched.

For a multi-threaded durable resource, these functions are enclosed in a critical section. That is, these functions are MT-hot.

In addition to the limitations described in the previous paragraph, the durable schedule be must closed and the resource must not be involved in a computational solver.

See Also

IlcResource, IlcSchedule, IlcWorkServer, IlcCapResource.

Edge Finder

Description

Scheduler Engine provides the *edge finder* as a way to increase propagation for discrete and unary resources.

You can influence the level of constraint propagation when using disjunctive constraints. Rather than considering only pairs of activities *{A1 A2}* to prove that *A1* must precede *A2* or vice-versa, the constraint propagation process can consider all pertinent tuples *{A1 ... An}* of activities to prove that some activity *Ai* must execute first (or must execute last) among *{A1 ... An}*. You can switch this extra propagation on or off with the member function `IlcDiscreteResource::setEdgeFinder`.

The edge finder takes the break list on the resource into account, that is, it uses the fact that by default, activities cannot be processed inside breaks. The edge finder also uses the information that some activities can have an overlap with breaks. The edge finder does not use the eventually available timetable of the resource at hand.

See Also

Disjunctive Constraint, IlcDiscreteResource

Metaconstraints

Description

A metaconstraint is a constraint on one or more constraints.

The *IBM ILOG Solver User's Manual* explains how to create and use metaconstraints. The specific constraint classes of Scheduler Engine are fully compatible with the metaconstraint protocol of Solver. The following points also apply:

- The metaconstraint protocol cannot be used with the following Scheduler global constraints: disjunctive, timetable, precedence graph, balance, sequence, integral, or functional constraints. These are the constraints created by the following member functions on resource classes: `makeDisjunctiveConstraint`, `makeTimetableConstraint`, `makePrecedenceGraphConstraint`, `makeBalanceConstraint`, `makeSequenceConstraint`, `makeFunctionalConstraint`, and `makeIntegralConstraint`.
- Instances of `IlcResourceConstraint` defined for capacity resources (that is, for instances of `IlcDiscreteResource`, `IlcDiscreteEnergy`, `IlcReservoir`, `IlcUnaryResource`) and for continuous reservoir (that is, for instances of `IlcContinuousReservoir`) must require or provide non-zero capacity.
- Scheduler Engine supports a *constructive exclusive disjunction* between an activity and a set of capacity resources. See the classes `IlcAltResSet` and `IlcAltResConstraint` for more information about that possibility.

There are no restrictions with respect to the metaconstraint protocol for instances of `IlcTimeBoundConstraint`.

There are no restrictions with respect to the metaconstraint protocol for instances of `IlcPrecedenceConstraint`.

There are no restrictions with respect to the metaconstraint protocol for instances of `IlcResourceConstraint` when the constraint is posted on instances of `IlcStateResource`.

See Also

IlcAltResConstraint, IlcAltResSet, IlcPrecedenceConstraint, IlcResource, IlcResourceConstraint, IlcTimeBoundConstraint, Disjunctive Constraint, Timetable.

Precedence Graph Constraints

Description

Resource Precedence Graph Constraints

A *resource precedence graph* is a directed graph that can be associated with a resource and whose nodes are the resource constraints posted on the resource.

On a resource precedence graph, an edge between two resource constraints *(rct1, rct2)* means that the activity of *rct1* is constrained to execute before the activity of *rct2*, provided that both resource constraints *rct1* and *rct2* definitely affect the availability of the resource (processing time and required capacity strictly greater than zero).

A *resource precedence graph constraint* is a constraint that creates a precedence graph structure for the resource with which it is associated and allows propagation of the precedence information contained in this graph on the variables of the resource constraints (start and end time, processing time, required capacity).

Creating a Resource Precedence Graph Constraint

A resource precedence graph constraint can be created on a resource with the member function `IlcResource::makePrecedenceGraphConstraint`.

A resource precedence graph constraint is automatically created for each unary resource with a sequence constraint.

Overview of Functions Related to Precedence Graphs

When a resource is associated with a precedence graph constraint, new precedence relations between resource constraints are automatically discovered and added to the graph by the *disjunctive constraint*, the *edge finder*, and the *sequence constraint*. New edges on the resource precedence graph are also discovered when the following precedence constraints are posted on the solver between two activities that require the resource:

- `IlcPrecedenceConstraintType::IlcStartsAfterEnd` with positive or null delay
- `IlcPrecedenceConstraintType::IlcStartsAtEnd` with positive or null delay
- `IlcPrecedenceConstraintType::IlcEndsAtStart` with negative or null delay

Some member functions of the class `IlcResourceConstraint` allow direct addition of new relations on the graph (`IlcResourceConstraint::setNext`, `IlcResourceConstraint::setSetup`, `IlcResourceConstraint::setSuccessor`, `IlcResourceConstraint::setTeardown`). Before entering the search, it is also possible to remove relations already added on the graph (`IlcResourceConstraint::unsetNext`, `IlcResourceConstraint::unsetSetup`, `IlcResourceConstraint::unsetSuccessor`, `IlcResourceConstraint::unsetTeardown`).

Transitive closure of the graph is automatically computed and maintained during the search.

The information stored in the precedence graph can be accessed with member functions of the class `IlcResourceConstraint` --such as `IlcResourceConstraint::hasAsNext`, `IlcResourceConstraint::isDirectlySucceededBy`, `IlcResourceConstraint::isSucceededBy` --and with the resource constraint iterator `IlcResourceConstraintIterator`. The availability of these accessors and iterators is restricted before entering the search, as the transitive closure of the graph has not been computed.

Status of Resource Constraints in a Precedence Graph

In a resource precedence graph, a resource constraint *rct* is said to surely contribute if and only if:

- *rct* has been posted, and
- the time extent of *rct* is not `IlcNever`, and
- the minimal processing time of the activity of *rct* is strictly greater than zero if the time extent of *rct* is `IlcTimeExtent::IlcFromStartToEnd`, and
- the minimal capacity required by *rct* is strictly greater than zero if *rct* is a capacity resource constraint.

In a resource precedence graph, a resource constraint *rct* is said to not possibly contribute if and only if it is sure that *rct* will not affect the availability of the resource. That is:

- the opposite of *rct* is posted, or
- the time extent of *rct* is `IlcNever`, or
- the processing time of the activity is equal to zero and the time extent of *rct* is `IlcTimeExtent::IlcFromStartToEnd`, or
- *rct* is a capacity resource constraint and the capacity required by *rct* is equal to zero.

Relative Positions of Resource Constraints in a Precedence Graph

Let *rct1* and *rct2* be two resource constraints on the same resource precedence graph.

- *rct2* is said to be a successor of *rct1* on the graph if the fact that both *rct1* and *rct2* definitely affect the availability of the resource implies that the activity of *rct2* is constrained to execute after the activity of *rct1* (that is, the start time of the activity of *rct2* is greater than or equal to the end time of the activity of *rct1*).
- The pair *(rct1,rct2)* is said to be unranked on the graph if and only if neither *rct1* is a successor of *rct2* nor *rct2* is a successor of *rct1*.
- *rct1* is said to be ranked on the graph if and only if there is no resource constraint *rct* such that the pair *(rct1,rct)* is unranked.
- *rct2* is said to be a direct successor of *rct1* on the graph if and only if *rct2* is a successor of *rct1* and there is no resource constraint *rct* that definitely affects the availability of the resource such that *rct* is a successor of *rct1* and *rct2* is a successor of *rct* on the current state of the resource precedence graph.
- *rct2* is said to be possibly next to *rct1* on the graph if and only if either the pair *(rct1,rct2)* is unranked or *rct2* is a direct successor of *rct1*.
- If the resource is closed, *rct2* is said to be next to *rct1* on the graph if and only if both *rct1* and *rct2* are ranked and *rct2* is a direct successor of *rct1*. If *rct2* is next to *rct1*, no other resource constraint can be scheduled between *rct1* and *rct2*. In a resource precedence graph, a resource constraint can have several direct successors whereas it has at most one next resource constraint.
- If the resource is not closed, no next relation can be deduced because other resource constraints could still be added.
- *rct1* is said to be a setup resource constraint if and only if no resource constraint *rct* exists such that *rct1* is next to *rct*.
- *rct1* is said to be a teardown resource constraint if and only if no resource constraint *rct* can be next to it.

Example

Consider a closed resource with five resource constraints *rct1*, *rct2*, *rct3*, *rct4*, and *rct5* that definitely affect the availability of the resource. Suppose that the following relations have been added:

- *rct1* is succeeded by *rct2* and *rct3*,
- *rct2* is succeeded by *rct4* and *rct5*,

- *rct3* is succeeded by *rct4*, and
- *rct4* is succeeded by *rct5*.

After the transitive closure of the graph has been computed during search, we find that:

- *rct1* is succeeded by *rct5*
- *rct2* is not directly succeeded by *rct5* because *rct4* is necessarily between *rct2* and *rct5*
- the pair (rct2, *rct3*) is not ranked
- the pair (*rct3*, *rct5*) is ranked because *rct3* is succeeded by *rct5*
- *rct4* is ranked
- *rct4* has *rct5* as next resource constraint
- *rct3* has no next resource constraint as there are still two possible candidates: *rct2* and *rct4*
- *rct4* is not a next resource constraint of *rct3*
- *rct1* is a setup resource constraint
- *rct5* is a teardown resource constraint

This list is not exhaustive and, of course, the initial relations are still valid in the transitive closure.

Precedence Graph Events and Delta Sets of Resource Constraints

Several events can be defined on a resource constraint with a precedence graph (IlcResource::whenDirectPredecessors, IlcResource::whenDirectSuccessors, IlcResource::whenPredecessors, IlcResource::whenSuccessors, IlcResource::whenNext, IlcResource::whenPrevious). These events are triggered as soon as the corresponding set of resource constraints is modified because the structure of the precedence graph has changed.

The modifications of these sets of resource constraints are stored in special sets called delta sets. These delta sets can be accessed during the execution of the goals associated with the event (see `IlcResourceConstraintDeltaIterator`). When all the graph events associated with a resource constraint have been processed, these delta sets are cleared.

For example, the following code displays the set of new direct successor links in the graph as soon as they are discovered.

```
class PrintCtI :public IlcConstraintI {
      public:
      PrintCtI(IloSolver solver) :IlcConstraintI(solver) {}
      void printNewDirectSuccessors(IlcResourceConstraint rct) {
        getSolver().out() << "New direct successors of " << rct << ":" << endl;
        for (IlcResourceConstraintDeltaIterator ite(rct, IlcDirectSuccessors); ite.ok(); ++ite)
             getSolver().out() << "\t" << *ite << endl;
         }
      };

      ILCRESOURCEDEMON(PrintDirectSucc, PrintCtI, printNewDirectSuccessors);


      PrintCtI* printer = new PrintCtI(solver);
      IlcResource resource ...;

      resource.whenDirectSuccessors(PrintDirectSucc(printer));
```

Light Resource Precedence Graph Constraint

The light precedence graph is a light version of the precedence graph constraint on unary resources. It is a global constraint that maintains the sequence of activities that have been ranked first, the sequence of activities that have been ranked last and a partition of the other (unranked) activities according to their status (Not)PossibleFirst/Last. The light precedence graph constraint is sufficient to enforce the unit capacity of the resource and the successor links between resource constraints. The main interest of the light precedence graph relies on the fact that its average complexity is linear with the number of activities.

The functionality consists of two member functions:

- `IlcResource::makeLightPrecedenceGraphConstraint` allows the creation and return of a light precedence graph constraint.
- `IlcResource::hasLightPrecedenceGraphConstraint` returns `IlcTrue` if and only if a light precedence graph constraint has been created on the resource.

When the light precedence graph is created and posted on a unary resource, the member function `IlcResource::hasRankInfo` returns `IlcTrue`, and it allows the use of the rank functionalities listed below:

- On the methods `IlcResourceConstraint::rankFirst`, `IlcResourceConstraint::rankNotFirst`, `IlcResourceConstraint::rankLast`, `IlcResourceConstraint::rankNotLast`, `IlcResourceConstraint::setSuccessor`, `IlcResourceConstraint::isPossibleFirst`, `IlcResourceConstraint::isPossibleLast`, `IlcResourceConstraint::isRanked`, `IlcResourceConstraint::isRankedFirst`, and `IlcResourceConstraint::isRankedLast`.
- Also on the methods `IlcUnaryResource::isRanked`, `IlcResource::whenRankedFirstRC`, `IlcResource::whenRankedLastRC`, `IlcResource::getLastRankedFirstRC`, `IlcResource::getLastRankedLastRC`, `IlcResource::getOldLastRankedFirstRC`, and `IlcResource::getOldLastRankedLastRC`.
- On the nested iterators `IlcResource::ResourceConstraintIterator` and `IlcResource::ResourceConstraintDeltaIterator`.
- And on the goals `IlcRank` and `IlcRankBackward`.

Note that both the disjunctive constraint (`makeDisjunctiveConstraint`) and the (classical) precedence graph constraint (`makePrecedenceGraphConstraint`) automatically create a light precedence graph constraint. The light precedence graph constraint should be used on unary resources with a large number of activities when the quadratic complexity of the disjunctive constraint is too expensive.

Schedule Precedence Graph Constraints

Whereas the nodes of a resource precedence graph represent resource constraints (see above), we describe in this section the concept of *schedule precedence graph* whose nodes are the activities created on a schedule.

On a schedule precedence graph, an edge between two activities *(act1, act2)* means that activity *act1* is constrained to execute before activity *act2*.

A *schedule precedence graph constraint* is a constraint that creates a precedence graph structure for the schedule with which it is associated and allows propagation of the precedence information contained in this graph on the variables of the activities (start and end time).

Creating a Schedule Precedence Graph Constraint

A schedule precedence graph constraint can be created on a schedule with the member function `IlcSchedule::makePrecedenceGraphConstraint`.

Overview of Functions Related to Schedule Precedence Graphs

When a schedule is associated a precedence graph constraint, the coherence between this global graph and the resource precedence graphs is ensured: when new precedence relations are discovered on a resource with a precedence graph, these relations are also added on the schedule precedence graph and, reciprocally, when new precedence relations are added on the schedule precedence graph, the resource precedence graphs are updated accordingly.

New edges on the schedule precedence graph are automatically discovered when the following precedence constraints are posted on the solver:

- `IlcPrecedenceConstraintType::IlcStartsAfterEnd` with positive or null delay
- `IlcPrecedenceConstraintType::IlcStartsAtEnd` with positive or null delay
- `IlcPrecedenceConstraintType::IlcEndsAtStart` with negative or null delay

Some member functions of the class `IlcActivity` allow direct addition of new precedence relations on the graph (`IlcActivity::setSuccessor`). Before entering the search, it is also possible to remove relations

already added on the graph (`IlcActivity::unsetSuccessor`).

Note that adding a new precedence relation on the graph leads exactly to the same propagation as posting a `IlcPrecedenceConstraintType::IlcStartsAfterEnd` constraint with null delay.

Transitive closure of the graph is automatically computed and maintained during the search.

The information stored in the precedence graph can be accessed with member functions of the class `IlcActivity` (such as `IlcActivity::isDirectlySucceededBy`, and `IlcActivity::isSucceededBy`), and with the activity iterator `IlcActivityIterator`. The availability of these accessors and iterators is limited before entering the search, as the transitive closure of the graph has not been computed.

Relative Positions of Activities in a Precedence Graph

Let *act1* and *act2* be two activities of a schedule with precedence graph.

- Activity *act2* is said to be a successor of *act1* on the graph if it is constrained to execute after *act1*.
- The pair *(act1,act2)* is said to be unranked on the graph if and only if neither *act1* is a successor of *act2* nor *act2* is a successor of *act1*.
- Activity *act1* is said to be ranked on the graph if and only if there is no activity *act* such that the pair *(act1,act)* is unranked.
- Activity *act2* is said to be a direct successor of *act1* on the graph if and only if *act2* is a successor of *act1* and there is no activity *act* of the schedule such that *act* is a successor of *act1* and *act2* is a successor of *act* on the current state of the schedule precedence graph.

Precedence Graph Events and Delta Sets of Activities

Several events can be defined on an activity with precedence graph (`IlcSchedule::whenDirectPredecessors`, `IlcSchedule::whenDirectSuccessors`, `IlcSchedule::whenPredecessors`, `IlcSchedule::whenSuccessors`.) These events are triggered as soon as the corresponding set of activities is modified because the structure of the precedence graph has changed.

The modifications of these sets of activities are stored in special sets called delta sets. These delta sets can be accessed during the execution of the goals associated with the event (see `IlcActivityDeltaIterator`). When all the graph events associated with an activity have been processed, then these delta sets are cleared.

Example

For example, the following code displays the set of new direct successors of the activity `act` as soon as new direct successors of `act` are discovered on the precedence graph.

```
class PrintCtI :public IlcConstraintI {
  public:
     PrintCtI(IloSolver solver) :IlcConstraintI(solver) {}
     void printNewDirectSuccessors(IlcActivity act) {
          getSolver().out() << "New direct successors of " << act << ":" <<
  endl;
          for (IlcActivityDeltaIterator ite(act, IlcDirectSuccessors);
  ite.ok(); ++ite)
          getSolver().out() << "\t" << *ite << endl;
     }
  };

ILCRESOURCEDEMON(PrintDirectSucc, PrintCtI, printNewDirectSuccessors);

PrintCtI* printer = new PrintCtI(solver);
IlcSchedule sched ...;

sched.whenDirectSuccessors(PrintDirectSucc(printer));
```

See Also

IlcActivity, IlcActivityIterator, IlcActivityIteratorFilter, IlcActivityDeltaIterator, IlcResource, IlcResourceConstraint, IlcResourceConstraintIterator, IlcResourceConstraintDeltaIterator, IlcResourceConstraintIteratorFilter

Ranking

Description

Activities may be ranked automatically as part of the constraint propagation process.

Ranking a resource constraint first means that the activity corresponding to the resource constraint will be executed before every activity on the resource that has not already been ranked first. Specifying that a resource constraint is not ranked first (that is, `IlcResourceConstraint::rankNotFirst`) means that the corresponding activity will execute after at least one of the activities that is currently not ranked. A similar (inverse) rule applies to ranking last.

More precisely, the propagation of global constraints (light precedence graph, disjunctive constraint) allows Scheduler Engine to deduce that some activities cannot be first or cannot be last: internal versions of the member functions `IlcResourceConstraint::rankNotFirst` and `IlcResourceConstraint::rankNotLast` are called as part of the constraint propagation process.

Ranking facilities are defined only for unary resources with a posted light precedence graph constraint, or state resources with a posted disjunctive constraint. Use the member function `IlcResource::hasRankInfo` to find out whether or not ranking facilities are available on a given resource

A resource constraint can be ranked if and only if its time extent is `IlcTimeExtent::IlcFromStartToEnd`.

In addition, if the resource is *closed* (as defined for the parent class `IlcResource`) and only one activity can be first, then this activity is necessarily first: an internal version of the member function `IlcResourceConstraint::rankFirst` is called as part of the constraint propagation process.

Similarly, if the resource is *closed* and only one activity can be last, then this activity is necessarily last: an internal version of the member function `rankLast` is called as part of the constraint propagation process.

If the resource is *not* closed, then ranking an activity to be not first or not last on a resource has no influence on its earliest start time.

The member functions `IlcResourceConstraint::isPossibleFirst`, `IlcResourceConstraint::isPossibleLast`, `IlcResourceConstraint::isRanked`, `IlcResourceConstraint::isRankedFirst`, and `IlcResourceConstraint::isRankedLast` let you know whether the activity that corresponds to a given resource constraint can be first, can be last, is already ranked, ranked first or ranked last.

Non-contributing resource constraints (those not posted, with null processing time or with null capacity requirement) are automatically considered as being ranked; therfore, they are not possible first or possible last anymore.

Furthermore, some member functions are available on the class `IlcResource` to access the last resource constraint that has been ranked first or last (`IlcResource::getLastRankedFirstRC`, `IlcResource::getLastRankedLastRC`) as well as to post demons on the detection of new ranked first or last activities on the resource (`IlcResource::whenRankedFirstRC`, `IlcResource::whenRankedLastRC`).

Some iterators are available for a chronological traversal of all the activities that have been ranked first or last or for iterating over the resource constraints that are possible first or last (`IlcResource::ResourceConstraintIterator`, `IlcResource::ResourceConstraintDeltaIterator`).

See Also

IlcResource, IlcResourceConstraint, IlcTimeExtent, Precedence Graph Constraints, Disjunctive Constraint

Large Neighborhoods

Description

The main idea behind *Large Neighborhood Search* is to focus the search on a sub-part of the decision variables of the problem. In this approach, we start from a known solution (current solution) and try to improve it by restoring (freezing) the values subset of some decision variables stored in this solution, while leaving other decision variables free. A goal is then used to try fix these remaining decision variables.

The set of decision variables is partitioned into two sets: the set of *selected* decision variables, and the other variables. Different restore policies are then applied, depending if the variable belongs or not to the set of selected decision variables.

To define the set of selected decision variables, use the pure virtual member function `defineSelected` of the class `IloLargeNHoodI`. Its signature is:

```
IloSolution IloLargeNHoodI::defineSelected(IloSolver solver, IloInt index)
```

The instance of class `IloSolution` returned must contain the set of selected decision variables corresponding to the neighbor with index `index`.

Local Search versus Large Neighborhood Search

At first glance, a large neighborhood and a neighborhood look very similar. But there are also some differences:

- In a local search approach, the neighborhoods are enumerated explicitly (method `define`). The size of the neighborhood is the number of neighbors. A goal may be used to validate the neighbor: to check for an instance where all the constraints are fulfilled.
- In a large neighborhood approach, the size of the neighborhood is the number of sub-problems to solve. Each sub-problem may contain a potentially a large number of neighbors, and thus is a combinatorial problem itself. To exhibit a neighbor a search algorithm must be used.

Suppose there is a scheduling problem made of three activities requiring a unary resource. Further, we have found a first solution and wish to iteratively select an activity and try to *relocate* it on the resource. In a local search approach, the size of the neighborhood is nine: each activity can be either first, last or in the middle on the resource. A sub-goal may be used to ensure that all other constraints are fulfilled (side constraints). In a large neighborhood search approach, the number of sub-problems to solve is the number of activities (three in this case). For each sub-problem, we can use, for instance, the goal `IloRankForward` to find a neighbor.

Scheduler Large Neighborhood

Large neighborhoods dedicated to scheduling problems derive from the class `IloSchedulerLargeNHoodI`.

Two predefined neighborhoods dedicated to scheduling problems are available:

- `IloRelocateActivityNHood` which iteratively selects an activity and tries to reschedule it somewhere else.
- `IloTimeWindowNHood` which iteratively selects a time window and tries to reschedule the activities within the time window.

The class `IloSchedulerLargeNHoodI` provides specialization of functionalities available in class `IloLargeNHoodI` for activities and resource constraints. For instance, in class `IloLargeNHood`, there is a member function `isSelected` used to check if an extractable is selected or not:

```
IloBool isSelected(IloExtractable extr) const;
```

A similar member function exists on class `IloLargeNHood` to check if activities and resource constraints are selected:

```
IloBool isSelected(IloActivity activity) const;

IloBool isSelected(IloResourceConstraint rc) const;
```

In addition to the above functionalities, the main difference between class `IloSchedulerLargeNHoodI` and class `IloLargeNHoodI` is that class `IloSchedulerLargeNHoodI` overloads the method `finalizeDelta`. To ease the definition of neighborhoods dedicated to scheduling problems, this method removes from any resource constraint that is not selected the next, the previous, the direct successors and the direct predecessors, in case these are selected. These are then replaced by new appropriate direct predecessors and direct successors.

As an example, suppose a set of activities are scheduled on a unary resource, and a time window neighborhood is applied. In the following equation, **R** is the resource, **[Ai]** represents an activity and | represents the time window boundary.

R: [A0][A1]..[Ai-1] | A[i]..[Aj] | [Aj+1]..[Ak]

Here, the time window contains all activities from Ai to Aj. In the current solution Ai is the next (and also the direct successor) of Ai-1. Similarly in the current solution, activity Aj is the previous (and also the direct predecessors) of activity Aj+1. Both the next of Ai-1, the successor of Ai-1, the previous of Aj+1 and the direct predecessors of Aj+1 are removed. Also, Aj+1 is set as the successor of Ai-1, and Ai-1 is set as the predecessor of Aj+1.

Extending the Library of Large Neighborhoods

There are two ways to extend the library with new large neighborhoods:

- by combining existing large neighborhoods.
- by deriving from class `IloLargeNHoodI`.

Combining Large Neighborhoods

There are two ways to combine large neighborhoods:

- Union of two neighborhoods
- Intersection of two neighborhoods

Union of Neighborhoods

The size of the resulting neighborhood is the product of the sizes of the two neighborhoods: *size(n1)*size(n2)*.

The set of selected extractables of the union for index *i* is the union of the sets of selected extractables for index *i1* for neighborhood *n1*, and for index *i2* for neighborhood *n2* such that *i = i1*size(n2) + i2*.

For any extractable, if it belongs to the union of the selected sets, it will be restored only if both neighborhoods specify that it must be restored. In case it does not belong to the union of the selected sets, then it is restored if at least one of the two neighborhoods specifies that it must be restored.

For example, a neighborhood that relocates two activities at once can be defined as the union of two relocate activity neighborhoods:

```
IloSchedulerLargeNHood n1 = IloRelocateActivityNHood(env);
IloSchedulerLargeNHood n2 = IloRelocateActivityNHood(env);
IloLargeNHood nhood = IloUnionNHood(env, n1, n2);
```

Intersection of Neighborhoods

As for the union, the size of the resulting neighborhood is the product of the sizes of the two neighborhoods: *size(n1)*size(n2)*.

The set of selected extractables of the intersection for index *i* is the intersection of the sets of selected extractables for index *i1* for neighborhood *n1* and for index *i2* for neighborhood *n2* such that *i = i1*size(n2) + i2*.

For any extractable, if it belongs to the intersection of the selected sets, it will be restored only if both neighborhoods specify that it must be restored. In case it does not belong to the intersection of the selected sets, then it is restored if at least one of the two neighborhoods specify that it must be restored.

For instance, a neighborhood that relocates activities within a time window with a window size 20 and a window step 10 can be defined as:

```
IloSchedulerLargeNHood n1 = IloRelocateActivityNHood(env);
IloSchedulerLargeNHood n2 = IloTimeWindowNHood(env, 20, 10);
IloLargeNHood nhood = IloIntersectNHood(env, n1, n2);
```

Implementing a New Large Neighborhood

When deriving from class `IloLargeNHoodI` (or `IloSchedulerLargeNHoodI`), you have to overload at least the three following virtual member functions:

```
IloSolution defineSelected(IloSolver solver, IloInt index);
void start(IloSolver solver, IloSolution);
IlcInt getSize(IloSolver solver, IloInt index);
```

The method `defineSelected` is a pure virtual member function and must be overloaded to define the selected extractables corresponding to a neighbor with *index*.

Methods `start` and `getSize` are pure virtual member functions of class `IloNHoodI` and must be overloaded as for any neighborhood.

Other virtual member functions on class `IloLargeNHoodI` are:

```
void defineRestoreInfo(IloSolver solver, IloSolution delta);
void finalizeDelta(IloSolver solver, IloSolution delta);
```

Method `defineRestoreInfo` is used to specify for each extractable the restore information; that is, either the restore fields for activities and resource constraints, or the restore status for other extractables. If this information differs from the current solution, then this information is added to the *delta* solution provided as argument. Default implementation uses the predicates.

Method `finalizeDelta` can be used to complete the definition *delta*. By default it doesn't do anything.

Note that class `IloLargeHNoodI` derives from class `IloNHoodI` and redefines the virtual method `IloNHoodI::define` . It calls method `defineSelected`, and then if not empty, calls method `defineRestoreInfo` and `finalizeDelta`. So when deriving class `IloLargeHNoodI`, you do not need to overload virtual method `define`.

Resource Enforcement as Global Constraint Declaration

Description

There are two ways for the Scheduler extractor to determine which global resource constraint is implicitly added to the Solver: the content of the model, or the enforcement level parameter. This section discusses details of the enforcement level by parameters. For related information, see Adding Global Resource Constraints from the Scheduler Model.

An enforcement level is related to some semantic capability of the resource such as having a finite capacity or a finite set of states, or enforcing a total order on the activities.

Scheduler offers a set of enforcement levels in the parameter class `IloResourceParam` . Refer to the enumeration `IloEnforcementLevel` for more information.

```
IloEnforcementLevel IloResourceParam::getBreaksEnforcement() const;
IloEnforcementLevel IloResourceParam::getCapacityEnforcement() const;
IloEnforcementLevel
             IloResourceParam::getPrecedenceEnforcement() const;
IloEnforcementLevel IloResourceParam::getSequenceEnforcement() const;
IloEnforcementLevel
             IloResourceParam::getTransitionTimeEnforcement() const;
```

The notion of resource enforcement is interpreted in the extraction as the addition of one or several global constraints.

Each type of enforcement level can be ignored (for example, `IloResourceParam::ignoreCapacityConstraints`). Ignoring an enforcement level is interpreted by the Scheduler extractor as a relaxation of the related global constraint, which is then not posted.

Each class of resource has its own default (value `IloBasic`) for each type of enforcement level. This default is fixed in Scheduler and corresponds to the most common usage of the resource.

This parametrization is very important because the pruning algorithms have very different performance levels. The most effective algorithms are also the most time consuming. For example, the timetable constraint algorithm is, on average, linear in the number of activities, the edge finder level `1` is quadratic on average, and, at worst, the disjunctive constraint is somewhere in-between.

Breaks Enforcement

The breaks enforcement is interpreted with the break constraint by the Scheduler extractor. There is only one pruning algorithm available in the Scheduler Engine, therefore, the enforcement levels have no effect (except when the break constraints are ignored, in which case no enforcement is done). The break enforcement corresponds to a call to the function:

```
void IlcResource::setBreakList(IlcBreakList bl);
```

All levels for all resources are extracted as posting the breaks constraint if the interval list of the Scheduler resource is not empty and the `ignoreBreaksConstraint` is `IloFalse`.

Capacity Enforcement

The capacity enforcement is related to the main global constraint for a resource class. It corresponds to the maximal and minimal capacity constraints for a discrete resource, the maximal and minimal energy for an energy resource, the maximal and minimal content for a reservoir, and the possible set of states and must-be-in-use status for a state resource.

The Scheduler extractor has five sets of algorithms with which to interpret the capacity enforcement level: the timetable constraint, the light precedence graph constraint, the disjunctive constraint, the edge finder, and the balance constraint. These constraints are expressed with the corresponding functions:

```
IlcConstraint IlcCapResource::makeTimetableConstraint();
IlcConstraint IlcUnaryResource::makeLightPrecedenceGraphConstraint();
IlcConstraint IlcCapResource::makeDisjunctiveConstraint();
void IlcDiscreteResource::setEdgeFinder(IlcInt level)
IlcConstraint IlcCapResource::makeBalanceConstraint();
```

Interpretation of Capacity Enforcement Levels

The following table lists the enforcement levels for the various resources and their corresponding capacity enforcement. Default values are in **bold**.

Capacity Enforcement Level	Constraint Algorithm
For Discrete Resources:	
`IloLow`, `IloMediumLow`, **IloBasic**	**Timetable**
`IloMediumHigh`	Timetable +Disjunctive
`IloHigh`	Timetable + Disjunctive + Edge finder (1)
`IloExtended`	Timetable + Disjunctive + Edge finder (1) + Balance
For Unary Resources:	
`IloLow`	Timetable
`IloMediumLow`	Light Precedence Graph
`IloBasic`	Light Precedence Graph + Disjunctive
`IloMediumHigh`	Light Precedence Graph + Disjunctive + Edge finder (1)
`IloHigh`	Light Precedence Graph + Disjunctive + Edge finder (2)
`IloExtended`	Light Precedence Graph + Disjunctive + Edge finder (2) + Balance
For Discrete Energy and Continuous Reservoir:	
All levels	**Timetable**
For Reservoir:	
`IloLow`, `IloMediumLow`, **IloBasic**, `IloMediumHigh`	**Timetable**
`IloHigh`, `IloExtended`	Timetable + Balance
For State Resources:	
`IloLow`, `IloMediumLow`, **IloBasic**	**Timetable**
`IloMediumHigh`, `IloHigh`, `IloExtended`	Timetable + Disjunctive

Precedence Enforcement

The precedence enforcement is related to the effort of the solver in analyzing the precedence relations between resource constraints on a resource. It is interpreted by the Scheduler extractor as the precedence graph constraint. For some classes of resources, namely the discrete and unary resources, specific algorithms can be triggered dealing with both the capacity limitations and the precedence graph knowledge. The precedence enforcement functions are:

```
IlcConstraint IlcResource::makePrecedenceGraphConstraint();
void IlcDiscreteResource::setPrecedencePropagation(IlcInt level = 1);
```

Note that when the capacity constraint is set to be ignored, the precedence propagation levels are ineffectual.

Interpretation of Precedence Enforcement Levels

Precedence Enforcement Level	Constraint Algorithm
For Discrete and Unary Resources:	
`IloLow`, `IloMediumLow`, **IloBasic**	**No global constraint**
`IloMediumHigh`	Precedence Graph
`IloHigh`	Precedence Graph + Level 1
`IloExtended`	Precedence Graph + Level 2
For Discrete Energy, Discrete and Continuous Reservoirs, and State Resource:	
`IloLow`, `IloMediumLow`, **IloBasic**	**No global constraint**
`IloMediumHigh`, `IloHigh`, `IloExtended`	Precedence Graph

A global schedule precedence graph (see the member function `IlcSchedule::makePrecedenceGraphConstraint`) is created when the enforcement level of the scheduler environment is set to a value `IloMediumHigh` or higher. See the member function `IloSchedulerEnv::setPrecedenceEnforcement`.

Sequence Enforcement

Sequence enforcement is the way to model the fact that the execution of some activities must be synchronized, and that a cost may arise when transitioning between activities. This enforcement only applies to unary resources, and requires either the precedence graph constraint or the following sequence constraint. These constraints are expressed with the corresponding functions:

```
IlcConstraint IlcUnaryResource::makeSequenceConstraint();
IlcConstraint IlcUnaryResource::makePrecedenceGraph();
```

Interpretation of Sequence Enforcement Levels

Sequence Enforcement Level	Constraint Algorithm
For Unary Resources:	
`IloLow`, `IloMediumLow`, **IloBasic**	**No global constraint**
`IloMediumHigh`	Precedence Graph
`IloHigh`, `IloExtended`	Sequence Constraint
For all other resource classes:	
Does not apply.	

Transition Time Enforcement

The transition time enforcement is related to the effort of the solver in enforcing the transition times between activities processed on a given resource.

The Scheduler extractor has three sets of algorithms with which to interpret the transition time enforcement level: the type timetable constraint, the disjunctive constraint and the precedence graph constraint.

These constraints are expressed with the corresponding functions:

```
IlcConstraint IlcResource::makeTypeTimetableConstraint();
IlcConstraint IlcDiscreteResource::makeDisjunctiveConstraint();
IlcConstraint IlcResource::makePrecedenceConstraint();
```

The following table lists the enforcement levels for the various resources and their corresponding transition time enforcement. Default values are in **bold**.

Interpretation of Transition Time Enforcement Levels

Transition Time Enforcement Level	Constraint Algorithm
For Unary Resources:	
`IloLow`, `IloMediumLow`, **IloBasic**	**Light Precedence Graph (1)**
`IloMediumHigh`, `IloHigh`, `IloExtended`	Disjunctive Constraint (4)
For State Resources:	
`IloLow`, `IloMediumLow`, **IloBasic**	**Type Timetable (2) (3)**
`IloMediumHigh`, `IloHigh`, `IloExtended`	Disjunctive Constraint (4)
For all other resource classes:	
`IloLow`, `IloMediumLow`, **IloBasic**, `IloMediumHigh`, `IloHigh`, `IloExtended`	**Type Timetable (5)**

Note the following additions or exceptions to the interpretation described in the table.

1. The *Light Precedence Graph* is a lighter version of the precedence graph constraint restricted to the management of rank information (ranked first or last, possible first or last).
2. When the transition time on the state resource is a user defined transition time built with the macro `ILOTRANSITIONTIMEOBJECT0`, the Disjunctive Constraint is used instead of the Type Timetable Constraint, as the Type Timetable Constraint is not able to enforce such a transition time object
3. When the capacity, the precedence or the sequence enforcement levels are so that a Disjunctive, a Precedence Graph or a Sequence Constraint is already defined on the state resource (see the previous sections Capacity Enforcement, Precedence Enforcement, and Sequence Enforcement), then the Type Timetable is not used as the Disjunctive or the Precedence Graph Constraint is sufficient to enforce transition times.
4. When the capacity constraints are to be ignored on the resource (see the member function `IloResource::ignoreCapacityConstraints`), then the Precedence Graph Constraint is used instead of the Disjunctive Constraint, as posting the Disjunctive Constraint would also enforce capacity constraints.
5. On discrete resources, discrete energy resources and reservoirs, the Type Timetable is used to enforce transition times except if the transition time on the resource is a user-defined transition time built with the macro `ILOTRANSITIONTIMEOBJECT0`. In that case, the Precedence Graph Constraint is used instead.

Duration Enforcement

The duration enforcement is related to the effort of the solver in enforcing propagation on the duration variable of activities. The Scheduler extractor has one way to enforce a stronger propagation on those variables than the default propagation thanks to the functions:

1. void IlcDiscreteResource::setTimetablePropagation(IlcInt level=1L);
2. void IlcDiscreteEnergy::setTimetablePropagation(IlcInt level=1L);

The following table lists the enforcement levels for the various resources and their corresponding duration enforcement. Default values are in bold.

Duration Enforcement Level	Constraint Algorithm
For Discrete and Discrete Energy Resources:	
`IloLow`, `IloMediumLow`, **IloBasic**	**No global constraint**
`IloMediumHigh`, `IloHigh`, `IloExtended`	Extra timetable propagation
For all other resource classes:	
`IloLow`, `IloMediumLow`, **IloBasic**, `IloMediumHigh`, `IloHigh`, `IloExtended`	**No global constraint**

Rounding, Inward & Outward

Description

The member functions:

`IlcActivity::requires`

`IlcActivity::provides`

`IlcActivity::requiresNot`

all accept an argument, `outward`. This argument is important *only* when one of the timetables of the resource that is required or provided by the invoking activity has a time step greater than one.

The time step of a timetable is defined by the argument `timeStep` of the member function `makeTimetableConstraint` (`IlcCapResource` and `IlcStateResource`).

The meaning of the argument `outward` is best illustrated by an example. Let's say we have a unary resource; its timetable starts at time 0, its time step is 5, and an activity of duration 5 requires the resource with time extent `IlcTimeExtent::IlcFromStartToEnd`. Let's assume further that it starts at time 1 (one). We represent those ideas like this:

```
/* Must be during search (e.g., inside a goal) */

IloSolver solver = getSolver();
IlcScheduler schedule(solver, 0, 100);
IlcUnaryResource resource(schedule, IlcFalse);
solver.add(resource.makeTimetableConstraint(5));
IlcActivity act(schedule, 5);
solver.add(act.requires(resource, 1,
                        IlcFromStartToEnd,
                        outward));
act.setStartTime(1);
```

If `outward` is `IlcTrue`, the activity uses the resource from time 0 (zero) to time 10. That is, the occupancy of the activity is rounded *outward* toward the nearest valid times that correspond to time steps.

In contrast, if `outward` is `IlcFalse`, the activity does *not* use the resource at all. That is, the occupancy of the activity is rounded *inward* toward the nearest valid times that correspond to time steps.

Outward rounding is useful when you want to express the idea that even if an activity requires a resource only part of a time period, the resource is still considered in use for the entire time period. In contrast, inward rounding corresponds to a situation where an activity requires a resource *only* when the activity uses the resource throughout the entire time period.

Rounding arguments are not used for instances of `IlcDiscreteEnergy`, for which only the energy consumed in time buckets is relevant, nor for instances of `IlcContinuousReservoir`, for which the time step of the timetable is always 1.

See Also

IlcActivity, IlcResourceConstraint, Timetable

Sequence Constraint

Description

Since a unary resource can only process one activity at a time, all activities requiring the same unary resource must be chronologically ordered to find a solution. As a result, in any solution to a problem that includes a unary resource, each unary resource defines a directed path through all the activities requiring it.

The nodes of such a path correspond to resource constraints of the time extent `IlcTimeExtent::IlcFromStartToEnd`. The links between the resource constraints can hold transition costs. See `IlcTransitionCostObject` for more details.

The path has, for its first node, a virtual node before any activities. The link between this first node and the first activity on the resource holds the setup cost. This first activity is called the setup activity.

The path also has, for its last node, a virtual node after all activities. The link between this last node and the last activity on the resource holds the teardown cost. This last activity is called the teardown activity.

Scheduler Engine maintains the relationships between the variables defining a path (next of an activity, previous of an activity, cost of the links between activities) and the scheduler variables (start and end times of the activities and required capacity). The sequence constraint uses the transition time function associated with the resource. When a disjunctive constraint is posted on the resource, the sequence constraint will use the ranking information to improve propagation.

The sequence constraint allows the next and previous variables to be used with any Solver generic constraint algorithms dealing with paths and forests.

Creating a Sequence Constraint

To create such a sequence constraint, use the member function `IlcUnaryResource::makeSequenceConstraint`

The valid resource constraints are implicitly indexed when the resource is closed. Then the next, previous, and costs variables are created. In other words, the variables used by the path representation of the resource are created when the sequence constraint is created and the resource closed.

Indexing of the Path

Let `Nb` be the number of resource constraints of time extent `IlcTimeExtent::IlcFromStartToEnd` on a closed unary resource. The first node before any resource constraint is indexed by zero. The last node after any resource constraint is indexed by `Nb + 1`. A valid resource constraint is indexed by a unique number between `1` and `Nb`.

Not Visited Nodes

A node may or may not be visited in the path. A node is visited if it has a next and previous node visited by the path. If a node is not visited, its next and previous variables are bound to a dummy value. Such a node does not have any link on the path and so it does not contribute to the sum of costs of the links in the path.

With a unary resource, an activity is visited only if its required resource constraint contributes to the resource. That is, if the product of the processing time of the activity multiplied by the required capacity is not null. In the sequence constraint on the unary resource, the value of the next and previous variable of a not visited resource constraint is -1.

See Also

IlcActivity, IlcTimeExtent, IlcTransitionTimeObject, IlcUnaryResource, IlcResourceConstraint, Transition Cost (Setup and Teardown Costs) In Scheduler Engine

Texture Measurements

Description

From a general perspective, a texture measurement is a measurement of some aspect of a search state. The actual measured values can then be used for any purpose. A typical use is to guide search by helping to identify characteristics of a search state that can be exploited to aid heuristic search. It is important to distinguish between the texture measurements themselves, and the uses to which they may be put; a single texture measurement may be used in multiple ways for multiple purposes.

For example, the first-fail heuristic in constraint programing chooses the variable with the smallest domain size to be assigned a value in the current search state. The domain size can be seen as a texture measurement: it is an aspect of the search state that can be efficiently evaluated and then used for some purpose (in this case, as a basis for a heuristic decision). Note that nothing limits the uses to which the domain sizes can be put. While assigning the variable with the smallest domain is a useful heuristic, there may be other uses, perhaps unrelated to heuristic search, for which the domain size information can be used.

Scheduler Engine implements a specific type of texture, `IlcResourceTexture`, which is a measure of the criticality of a resource over the scheduling horizon. Criticality is a floating point value with minimal predefined semantics. All that is required is that higher levels of criticality are represented by larger floating point values.

The criticality curve can be used to drive heuristic search. For example, in a resource allocation application, it may be useful to assign an activity to the resource with the lowest criticality over the activity's execution time window. In an application more directed towards scheduling, identifying the set of activities that demand the most critical time point across all resources may help heuristic search. In this latter case, a heuristic might sequence a pair of activities from the set.

To define a resource texture measurement it is necessary to define the impact of a resource constraint on a resource and to define how the criticality is computed given the aggregated demand and variance for all the resource constraints on a resource.

This impact that a resource constraint has on a resource is represented by an `IlcRCTexture` object which represents two curves: the demand curve of a resource constraint for a resource and the variance of this demand. Representation of the variance curve is optional. Users can define the demand and variance curves by subclassing the implementation class `IlcRCTextureI` and by subclassing the RC Texture factory class

IlcRCTextureFactoryI. The factory class simply returns an instance of the user-defined IlcRCTextureI subclass. The factory is used internally to create a new RC texture object whenever a new resource constraint is added to a resource. A number of RC texture and RC texture factory classes are predefined in the Scheduler Engine (such as IlcRCTextureProbabilisticI, and IlcRCTextureProbabilisticFactoryI).

The criticality calculation is similarly defined by subclassing IlcTextureCriticalityCalculatorI. This class has two pure virtual functions that are used to calculate the criticality for a maximum and minimum resource capacity constraint given the aggregate demand and variance of all possible resource constraints of a resource. A number of criticality calculators are predefined in the Scheduler Engine (such as IlcRelativeDemandCriticalityCalculatorI and IlcProbabilisticCriticalityCalculatorI).

Internally, the texture maintenance algorithm uses an RC factory to create an instance of a subclass of IlcRCTextureI for each resource constraint on a resource. The demand and variance curve corresponding to each resource constraint are aggregated across the scheduling horizon and, then, for each relevant time point, the IlcTextureCriticalityCalculatorI subclass is called to compute the criticality given the aggregate demand and variance.

At the modeling level, there are a number of predefined classes corresponding to the predefined Scheduler Engine classes (such as IloTextureCriticalityCalculatorI and IloRCTextureFactory). These classes allow the specification of the texture measurements in an instance of IloTextureParam. We also provide a number of utilities (such as the macros ILOTEXTURECRITICALITYCALCULATOR0 and ILORCTEXTUREFACTORY0) that create user-defined modeling objects that correspond to user-defined Scheduler Engine objects. For example, with the former macro you can create a model object MyIloTextureCalculator that corresponds to and is extracted as your Scheduler Engine object MyIlcTextureCalculator.

See Also

IlcResourceTexture, IlcTextureSuccessorGoal, IlcTextureAltSuccessorGoal, IlcResourceTextureIterator, IlcRCTexture, IlcRCTextureIterator, IlcRCTextureI, IlcRCTextureESTI, IlcRCTextureProbabilisticI, IlcRCTextureTargetI, IlcRCTextureFactory, IlcRCTextureFactoryI, IlcRCTextureFactoryI, IlcRCTextureProbabilisticFactoryI, IlcRCTextureTargetFactoryI, IlcTextureCriticalityCalculator, IlcTextureCriticalityCalculatorI, IlcRelativeDemandCriticalityCalculatorI, IlcProbabilisticCriticalityCalculatorI, IloTextureSuccessorGoal, IloTextureAltSuccessorGoal, IloTextureParam, IloRCTextureFactory, IloRCTextureFactoryI, ILORCTEXTUREFACTORY0, IloTextureCriticalityCalculator, IloTextureCriticalityCalculatorI, ILOTEXTURECRITICALITYCALCULATOR0.

Timetable

Description

An important concept embodied in the implementation of resources is the concept of a *timetable constraint*. Semantically, a table can be seen as a variable, the value of which is a function associating a value v(t) with each point in time t. In the Scheduler Engine, we have two types of timetables, corresponding to the classes IlcIntTimetable and IlcAnyTimetable.

In the case of IlcIntTimetable, the values v(t) are *integers*. An instance of the class IlcIntTimetable manages the minimal and maximal possible values for v(t) for each point in time t. It also allows you to define constraints on these minimal and maximal values as a function of time. The class IlcIntTimetable is used for *capacity* resources, that is, instances of the classes IlcDiscreteEnergy, IlcDiscreteResource, or IlcReservoir.

In the case of IlcAnyTimetable, the values v(t) are pointers to arbitrary objects. An instance of the class IlcAnyTimetable manages all possible values for v(t) for each point in time t. It also allows you to define constraints on these possible values as a function of time. The class IlcAnyTimetable is used for *state* resources, that is, instances of the class IlcStateResource.

Initial Occupation

When associated with a resource by a timetable constraint, a timetable represents the occupation of the resource by activities. Scheduler Engine provides a way to setup an initial occupation of capacity resources without having to declare the corresponding activities.

The initial occupation is defined by an instance of the `IlcIntToIntStepFunction` class. The value of the function at a time point is considered to be the sum of the requirements of fictitious activities.

This facility is intended to help in solving problems by iteratively adding a new set of activities to schedule or in improving solutions by rescheduling a subset of the activities.

Implementation Considerations

The timetables in Scheduler Engine store only the instants in time at which the status of the timetable changes. Information in the timetable is accessed and modified in time proportional to the number of status changes.

See Also

IlcAnyTimetable, IlcCapResource, IlcIntTimetable, IlcResource, IlcStateResource, IBM ILOG Solver Reference Manual:IlcIntToIntStepFunction

Transition Cost (Setup and Teardown Costs) In Scheduler Engine

Description

Since a unary resource can only process one activity at a time, all activities requiring the same unary resource must be totally ordered in a solution. As a result, in any solution to a problem that includes a unary resource, each unary resource defines a directed path through all the activities requiring it.

Between each pair of consecutive activities, some cost may be incurred to switch the resource from processing the first activity to processing the second. These costs may be related to modifications to the resource that require manpower, material, and energy, such as adjusting or purging a machine.

In Scheduler Engine, transition cost is defined as the cost between two immediately successive activities in the sequence constraint of a unary resource. In addition, Scheduler Engine lets you define a setup cost for the activity that starts the usage of the resource and a teardown cost for the activity that ends the usage of the resource.

The transition cost object instance of `IlcTransitionCostObject` can be added to a unary resource with a sequence constraint using the function `IlcUnaryResource::addNextTransitionCost` or `IlcUnaryResource::addPrevTransitionCost`. The transition cost will be taken into account by the propagation.

The transition cost can be variable or constant. By constant we mean that it only depends upon the precedence relationship between two activities. By variable we mean that the evaluation of the transition cost depends upon current knowledge about the other variables and constraints involved. If it is variable, the transition cost object must define its minimal and maximal value given the current knowledge about the sequence.

Scheduler Engine associates an integer transition type to each activity. The accessors are the functions `IlcActivity::getTransitionType` and `IlcActivity::setTransitionType`. They allow you to define instances of `IlcTransitionCostObject` from integer tables where the rows and columns are indexed by the transition types.

See Also

IlcMakeTransitionCost, IlcTransitionCostObject, IlcTransitionTable, IlcUnaryResource, Transition Cost (Setup and Teardown Costs) In Scheduler Engine

Transition Time in Scheduler Engine

Description

For discrete resources, discrete energy, and state resources, you can define *transition times*. Given two activities *A1* and *A2*, the transition time between *A1* and *A2* is an amount of time that must elapse between the end of *A1* and the beginning of *A2* when *A1* precedes *A2*.

Transition times are taken into account when the disjunctive constraint, the precedence graph constraint, or the type timetable constraint on a resource are posted. Note however, that the disjunctive constraint does not exist on reservoirs and discrete energy resources.

The precedence graph constraint propagates transition time based on the following precedence relation: If *A2* is a successor of *A1* on the precedence graph, then the transition time between *A1* and *A2* is propagated.

Propagation of transition times is not handled the same way for a disjunctive constraint as for a type timetable constraint.

The first difference is the definition of which activities are incompatible. Although both the disjunctive constraint and the type timetable constraint propagate transition times only between activities that are incompatible, the disjunctive constraint defines incompatibility based on the resource demand, and the type timetable constraint defines incompatibility based on the transition types of activities. Therefore care must be taken, especially with discrete and state resources, when choosing which constraint to use. We advise using the type timetable to propagate transition times on discrete resources, because it is very possible that no pair of activities will be incompatible based on their resource demands.

A second difference is that the disjunctive constraint propagates transition times more than the type timetable constraint. Expressing transition times with a type timetable constraint is therefore less CPU-time consuming, especially when the number of activities is large.

Here is an example to illustrate the extra propagation performed by a disjunctive constraint. Suppose two activities *A1* and *A2* have a duration of 4, *A1* is to be executed between 0 and 14 and has type 0, *A2* executes between 4 and 21 and has type 1, and they both require the same unary resource. The transition time between these activities is symmetric and is equal to 3. The disjunctive constraint will deduce that *A2* must start from at least time 7 since *A2* cannot be scheduled before *A1*. The type timetable constraint will not make that deduction.

If we look at activity *A1*, we could in principle remove type 1from the interval *[lst - 3, ect + 3)*, where *lst* stands for latest start time of *A1*, *ect* for earliest completion time of *A1*, and 3 is the transition time. With *lst* for *A1* equal to 10 and *ect* equal to 4, we see this rule will not result in the actual removal of type 1 from a time interval. Similar reasoning holds for activity *A2*.

You can define the transition time function of a resource by passing an instance of the class `IlcTransitionTimeObject` to the constructor of the resource, or by calling the member function `IlcResource::setTransitionTimeObject`. To have the type timetable constraint propagate the transition times, the instance of `IlcTransitionTimeObject` must have been built with an instance of the class `IlcTransitionTable`. When you build the transition time object that way, the propagation of the disjunctive constraint or the type timetable constraint automatically takes transition times into consideration. However, activities with a type not represented in the instance of `IlcTransitionTable` are not affected by the transition times of other activities.

With the member function `IlcResource::getTransitionTime` you can inspect the transition time between two activities that require the invoking resource. This member function accepts two resource constraints as arguments and returns the transition time between the two corresponding activities. By default (that is, when transition times are not defined), the member function returns zero (0).

Scheduler Engine associates an integer transition type with each activity. The accessors are the functions `IlcActivity::getTransitionType` and `IlcActivity::setTransitionType`. They allow you to define instances of `IlcTransitionTimeObject` from integer tables where the rows and columns are indexed by the transition types.

There are some restrictions when the transition times are taken into account by the type timetable constraint:

- The type timetable constraint alone does not propagate suspended transition times (See Calendars). Suspended transition times can be propagated by creating a precedence graph constraint or (on a unary or state resource) a disjunctive constraint.
- The instance of `IlcTransitionTimeObject` that defines the transition time for the resource must have been built with an instance of `IlcTransitionTable`.
- Only resource constraints with a time extent equal to `IlcTimeExtent::IlcFromStartToEnd` are taken into account. Resource constraints having a time extent different from `IlcTimeExtent::IlcFromStartToEnd` are ignored.
- The resource cannot be durable.

See Also

IlcDiscreteEnergy, IlcDiscreteResource, IlcMakeTransitionTime, IlcStateResource, IlcTransitionTable, IlcTransitionTime, IlcTransitionTimeObject, IlcUnaryResource, Disjunctive Constraint, Type Timetable Constraint, Precedence Graph Constraints

Type Timetable Constraint

Description

The type timetable constraint performs propagation based on the transition types of the activities. These types are set with the function `IlcActivity::setTransitionType`.

The main purpose of the type timetable constraint is to offer an alternative way of propagating transition times instead of through a disjunctive constraint. For more details on the differences between handling transition times with a disjunctive constraint and a type timetable constraint see Transition Time in Scheduler Engine.

The type timetable constraint states that two activities that have a different type are incompatible and it propagates the following constraints.

Let *act1* and *act2* be two activities that have types *type1* and *type2* respectively, and let *tt(typeX, typeY)* be the transition time from *typeX* to *typeY*. If both *act1* and *act2* actually require the resource and have a minimal duration greater than 0 and if *type1* is different from *type2*, the type timetable constraint assures that:

endTime(act1) + tt(type1, type2) ≤ startTime(act2) or

endTime(act2) + tt(type2, type1) ≤ startTime(act1)

If *type1* is equal to *type2* and *tt(type1, type2) != 0*, the type timetable constraint assures that:

endTime(act1) + tt(type1, type2) ≤ startTime(act2) or

endTime(act2) + tt(type2, type1) ≤ startTime(act1) or

(startTime(act1) = startTime(act2) and endTime(act1) = endTime(act2))

If *type1* is equal to *type2* and *tt(type1, type2) = 0*, one of the following two cases will occur:

- No constraint on the start and end times of *act1* and *act2* is propagated. That is, activities of *type1* can partially overlap.
- The previous constraint occurs. That is, if two activities overlap, they start and end at the same time.

The type timetable constraint enforces batching activities. A boolean argument in the function that makes the type timetable constraint allows you to declare the batching options.

Note that the type timetable constraint is available only for discrete resources and discrete energy resources.

Some restrictions exist when transition times are taken into account by the type timetable constraint:

- The type timetable constraint alone does not propagate suspended transition times (See Calendars). Suspended transition times can be propagated by creating a precedence graph constraint or (on a unary or state resource) a disjunctive constraint.
- The resource cannot be durable.
- The instance of `IlcTransitionTimeObject` that defines the transition time for the resource must have been built with an instance of `IlcTransitionTable`.
- Only resource constraints with a time extent equal to `IlcTimeExtent::IlcFromStartToEnd` are taken into account. Resource constraints having a time extent different from `IlcTimeExtent::IlcFromStartToEnd` are ignored.

See Also

IlcActivity, IlcTransitionTable, Transition Time in Scheduler Engine.

Moved or Obsolete Functions and Classes

A number of classes and functions have either moved to another library, or have been made obsolete.

Moved Functions and Classes

This is a list of functions and classes that no longer exist in the IBM® ILOG® Scheduler library. The function of selectors, predicates, evaluators, and comparators has moved from Scheduler to Solver for 6.1. For more information on these items, please see the concept *Selectors* and individual classes in the *IBM ILOG Solver Reference Manual*.

- `IlcActivitySelectorObject`
- `IlcAltRCSelectorObject`
- `IlcRCSelectorObject`
- `IlcResourceSelectorObject`
- `IlcRCComparator`
- `IlcRCEvaluator`
- `IlcRCEvaluatorI`
- `IlcRCPredicate`
- `IlcRCPredicateI`
- `IlcRCSequenceNextSelectorObject`
- `IlcRCSequencePrevSelectorObject`
- `IlcRCSequenceSelectorObject`
- `IlcSchedulerComparator`
- `IlcSchedulerEvaluatorI`
- `IlcSchedulerPredicate`
- `IlcSchedulerPredicateI`
- `IlcSchedulerEvaluator`
- `IlcActivityComparator`
- `IlcActivityEvaluator`
- `IlcActivityPredicate`
- `IlcAltRCComparator`
- `IlcAltRCEvaluator`
- `IlcAltRCPredicate`
- `IlcResourceComparator`
- `IlcResourceEvaluator`
- `IlcResourcePredicate`
- `IlcActivityEvaluatorI`
- `IlcActivityPredicateI`
- `IlcAltRCEvaluatorI`
- `IlcAltRCPredicateI`
- `IlcResourceEvaluatorI`
- `IlcResourcePredicateI`

This is a list of functions and classes that moved from Scheduler to Concert. This move occurred in Scheduler 6.0. For more information on these classes, please see individual classes in the extensions section of the *IBM ILOG Concert Technology Reference Manual*.

- `IloIntervalList`
- `IloIntervalListCursor`
- `IloNumToNumStepFunction`
- `IloNumToNumStepFunctionCursor`
- `IloNumToAnySetStepFunction`
- `IloNumToAnySetStepFunctionCursor`
- `IloNumToNumSegmentFunction`
- `IloNumToNumSegmentFunctionCursor`

Obsolete Functions and Classes

This is a list of functions from Scheduler 6.0 that are now obsolete.

Obsolete Function or Class	Replac
`IloSchedulerSolution::getSolution`	
`IloSchedulerSolution::hasPrecedenceInformation`	
`IloSchedulerSolution::hasSetupRC`	`IloSchedulerSolution::getSetupRC`
`IloSchedulerSolution::hasTeardownRC`	`IloSchedulerSolution::getTeardownRC`
`IloSchedulerSolution::hasNextRC`	`IloSchedulerSolution::getNextRC`
`IloSchedulerSolution::hasPrevRC`	`IloSchedulerSolution::getPrevRC`
`IlcActDurationMaxEvaluator`	`IlcActivityDurationMaxEvaluator`
`IlcActDurationMinEvaluator`	`IlcActivityDurationMinEvaluator`
`IlcActEndMaxEvaluator`	`IlcActivityEndMaxEvaluator`
`IlcActEndMinEvaluator`	`IlcActivityEndMinEvaluator`
`IlcActEndVarBoundPredicate`	`IlcActivityEndVarBoundPredicate`
`IlcActIsBreakablePredicate`	`IlcActivityIsBreakablePredicate`
`IlcActIsRankedPredicate`	`IlcActivityIsRankedPredicate`
`IlcActPostponedBackwardPredicate`	`IlcActivityPostponedBackwardPredicate`
`IlcActPostponedPredicate`	`IlcActivityPostponedPredicate`
`IlcActProcessingTimeMaxEvaluator`	`IlcActivityProcessingTimeMaxEvaluator`
`IlcActProcessingTimeMinEvaluator`	`IlcActivityProcessingTimeMinEvaluator`
`IlcActProcessingTimeVarBoundPredicate`	`IlcActivityProcessingTimeVarBoundPredicate`
`IlcActRandomEvaluator`	`IlcActivityRandomEvaluator`
`IlcActStartMaxEvaluator`	`IlcActivityStartMaxEvaluator`
`IlcActStartMinEvaluator`	`IlcActivityStartMinEvaluator`
`IlcActStartVarBoundPredicate`	`IlcActivityStartVarBoundPredicate`
`IlcActTransitionTypeEvaluator`	`IlcActivityTransitionTypeEvaluator`
`IlcAltRCCapacityEvaluator`	`IlcAltResConstraintCapacityEvaluator`

IlcAltRCPossibleEvaluator	IlcAltResConstraintNbPossibleEvaluator
IlcAltRCResourceSelectedPredicate	IlcAltResConstraintResourceSelectedPredicate
IlcAltRCVariableConstraintPredicate	IlcAltResConstraintVariableConstraintPredicate
IlcCapacityResourcePredicate	IlcResourceIsCapacityResourcePredicate
IlcContinuousReservoirPredicate	IlcResourceIsContinuousReservoirPredicate
IlcDiscreteEnergyPredicate	IlcResourceIsDiscreteEnergyPredicate
IlcDiscreteResourcePredicate	IlcResourceIsDiscreteResourcePredicate
IlcIsReservoirEvaluator	IlcResourceIsReservoirPredicate
IlcRCCapacityConstraintPredicate	IlcResourceConstraintCapacityConstraintPredicate
IlcRCCapacityMaxEvaluator	IlcResourceConstraintCapacityMaxEvaluator
IlcRCCapacityMinEvaluator	IlcResourceConstraintCapacityMinEvaluator
IlcRCDurationMaxEvaluator	
IlcRCDurationMinEvaluator	
IlcRCEndMaxEvaluator	
IlcRCEndMinEvaluator	
IlcRCHasNextPredicate	IlcResourceConstraintHasNextPredicate
IlcRCHasPrevPredicate	IlcResourceConstraintHasPrevPredicate
IlcRCInwardConstraintPredicate	IlcResourceConstraintInwardConstraintPredicate
IlcRCNegativeConstraintPredicate	IlcResourceConstraintNegativeConstraintPredicate
IlcRCPossibleFirstPredicate	IlcResourceConstraintPossibleFirstPredicate
IlcRCPossibleLastPredicate	IlcResourceConstraintPossibleLastPredicate
IlcRCPossibleSetupPredicate	IlcResourceConstraintPossibleSetupPredicate
IlcRCPossibleTeardownPredicate	IlcResourceConstraintPossibleTeardownPredicate
IlcRCPossiblyContributesPredicate	IlcResourceConstraintPossiblyContributesPredicate
IlcRCProcessingTimeMaxEvaluator	
IlcRCProcessingTimeMinEvaluator	
IlcRCProvidingConstraintPredicate	IlcResourceConstraintProvidingConstraintPredicate
IlcRCRandomEvaluator	IlcResourceConstraintRandomEvaluator
IlcRCSetupPredicate	IlcResourceConstraintSetupPredicate
IlcRCSlopeConstraintPredicate	IlcResourceConstraintSlopeConstraintPredicate
IlcRCSlopeEvaluator	IlcResourceConstraintSlopeEvaluator
IlcRCStartMaxEvaluator	
IlcRCStartMinEvaluator	
IlcRCStateConstraintPredicate	IlcResourceConstraintStateConstraintPredicate
IlcRCStateSetConstraintPredicate	IlcResourceConstraintStateSetConstraintPredicate
IlcRCSurelyContributesPredicate	IlcResourceConstraintSurelyContributesPredicate

IlcRCTeardownPredicate	IlcResourceConstraintTeardownPredicate
IlcRCTransitionCostEvaluator	IlcResourceConstraintNextTransitionCostEvaluator
IlcRCTransitionTypeEvaluator	
IlcRCVariableConstraintPredicate	IlcResourceConstraintVariableConstraintPredicate
IlcRCVirtualSinkNodePredicate	
IlcRCVirtualSourceNodePredicate	
IlcResCapacityEvaluator	IlcResourceCapacityEvaluator
IlcResClosedPredicate	IlcResourceClosedPredicate
IlcResEnergyEvaluator	IlcResourceEnergyEvaluator
IlcReservoirPredicate	IlcResourceIsReservoirPredicate
IlcResGlobalSlackEvaluator	IlcResourceGlobalSlackEvaluator
IlcResHasAltResConstraintPredicate	IlcResourceHasAltResConstraintPredicate
IlcResHasBreaksPredicate	IlcResourceHasBreaksPredicate
IlcResHasTexturePredicate	IlcResourceHasTexturePredicate
IlcResLocalSlackEvaluator	IlcResourceLocalSlackEvaluator
IlcResRandomEvaluator	IlcResourceRandomEvaluator
IlcResRankedPredicate	IlcResourceRankedPredicate
IlcResSequencedPredicate	IlcResourceSequencedPredicate
IlcResTextureEvaluator	IlcResourceTextureEvaluator
IlcStateResourcePredicate	IlcResourceIsStateResourcePredicate
IlcUnaryResourcePredicate	IlcResourceIsUnaryResourcePredicate

Group optim.scheduler.modeling

The IBM® ILOG® Scheduler API.

Class Summary
IloActivity
IloActivityBasicParam
IloActivityBreakParam
IloActivityConstraintsParam
IloActivityOverlapParam
IloActivityShiftParam
IloAltResConstraintIterator
IloAltResSet
IloAltResSet::Iterator
IloCalendar
IloCalendar::ShiftObjectIterator
IloCapResource
IloContinuousReservoir
IloCoverConstraint
IloDiscreteEnergy
IloDiscreteResource
IloGranularFunction
IloGranularFunction::Cursor
IloPrecedenceConstraint
IloRCTextureFactory
IloRCTextureFactoryI
IloRelocateActivityNHoodI
IloReservoir
IloResource
IloResourceConstraint
IloResourceConstraintIterator
IloResourceParam
IloResourceValue
IloSchedulerEnv
IloSchedulerLargeNHood
IloSchedulerLargeNHoodI
IloSchedulerSolution
IloSchedulerSolution::ActivityIterator
IloSchedulerSolution::ResourceConstraintIterator
IloSchedulerSolution::ResourceIterator
IloShape
IloShiftListObject

IloShiftObject
IloStateResource
IloTextureCriticalityCalculator
IloTextureCriticalityCalculatorI
IloTextureParam
IloTimeBoundConstraint
IloTimeWindowNHood
IloTimeWindowNHoodI
IloTimeWindowNHoodI::IloTimeWindow
IloTransitionCost
IloTransitionCostObject
IloTransitionCostObjectI
IloTransitionParam
IloTransitionTime
IloTransitionTimeObject
IloTransitionTimeObjectI
IloUnaryResource
IloVariableSlopeShape

Macro Summary
ILORCTEXTUREFACTORY0
ILOTEXTURECRITICALITYCALCULATOR0
ILOTRANSITIONCOSTOBJECT0
ILOTRANSITIONTIMEOBJECT0

Enumeration Summary
IloActivitySelector
IloEnforcementLevel
IloGranularFunctionRoundingMode
IloPrecedenceConstraintType
IloResourceConstraintSelector
IloResourceSelector
IloSchedVariable
IloSchedulerSolution::IloResourceConstraintIteratorFilter
IloSequenceIndexSelector
IloTimeBoundConstraintType
IloTimeExtent

Function Summary
IloAssignAlternative
IloIntersectNHood
IloRankBackward

IloRankForward
IloRelocateActivityNHood
IloResourceFunctionalConstraint
IloResourceIntegralConstraint
IloSequenceBackward
IloSequenceForward
IloSetTimesBackward
IloSetTimesForward
IloShapeLowerThan
IloTextureAltSuccessorGoal
IloTextureSuccessorGoal
IloTimeWindowBackwardChronologicalComparator
IloTimeWindowBackwardChronologicalComparator
IloTimeWindowForwardChronologicalComparator
IloTimeWindowForwardChronologicalComparator
IloUnionNHood

The IBM® ILOG® Scheduler API.

Group optim.scheduler.solving

The IBM® ILOG® Scheduler API.

Class Summary
IlcActivity
IlcActivityDeltaIterator
IlcActivityIterator
IlcAltRCDemon
IlcAltResConstraint
IlcAltResConstraintIterator
IlcAltResSet
IlcAltResSetIterator
IlcAnyTimetable
IlcAnyTimetableCursor
IlcAnyTimetableIterator
IlcCalendar
IlcCalendar::ShiftObjectIterator
IlcCapResource
IlcContinuousReservoir
IlcContinuousReservoirIterator
IlcDiscreteEnergy
IlcDiscreteEnergyIterator
IlcDiscreteResource
IlcDiscreteResourceIterator
IlcFollowingActivityIterator
IlcGranularFunction
IlcGranularFunctionCursor
IlcIntTimetable
IlcIntTimetableCursor
IlcIntTimetableIterator
IlcIntToFloatSegmentFunction
IlcIntToFloatSegmentFunctionCursor
IlcIntervalList
IlcIntervalListCursor
IlcPossibleAltResIterator
IlcPrecedenceConstraint
IlcPrecedingActivityIterator
IlcProbabilisticCriticalityCalculatorI
IlcRCTexture
IlcRCTextureESTFactoryI
IlcRCTextureESTI

IlcRCTextureFactory
IlcRCTextureFactoryI
IlcRCTextureI
IlcRCTextureIterator
IlcRCTextureProbabilisticFactoryI
IlcRCTextureProbabilisticI
IlcRCTextureTargetFactoryI
IlcRCTextureTargetI
IlcRelativeDemandCriticalityCalculatorI
IlcReservoir
IlcReservoirIterator
IlcResource
IlcResourceConstraint
IlcResourceConstraintDeltaIterator
IlcResourceConstraintIterator
IlcResourceDemon
IlcResourceIterator
IlcResource::ResourceConstraintDeltaIterator
IlcResource::ResourceConstraintIterator
IlcResourceTexture
IlcResourceTextureIterator
IlcSchedule
IlcScheduleDemon
IlcScheduler
IlcSchedulerPrintTrace
IlcSchedulerTrace
IlcSchedulerTraceI
IlcShape
IlcShiftListObject
IlcShiftObject
IlcStateResource
IlcStateResourceIterator
IlcStateResourceIterator
IlcTextureCriticalityCalculator
IlcTextureCriticalityCalculatorI
IlcTimeBoundConstraint
IlcTransitionCostObject
IlcTransitionCostObjectI
IlcTransitionTable
IlcTransitionTimeObject
IlcTransitionTimeObjectI

IlcUnaryResource
IlcUnaryResourceIterator
IlcVariableSlopeShape
IlcWorkServer

Typedef Summary
IlcSchedulerTraceFilter

Macro Summary
ILCALTRCDEMON
ILCRESOURCEDEMON
ILCSCHEDULEDEMON
IlcTransitionCost
IlcTransitionTime
ILCUSERSHIFTOBJECT

Enumeration Summary
IlcActivityIteratorFilter
IlcFailReason
IlcGranularFunctionRoundingMode
IlcPrecedenceConstraintType
IlcResourceConstraintIteratorFilter
IlcResource::RankFilter
IlcSchedVariable
IlcSchedulerChange
IlcSlopeConstraintMode
IlcSolverChange
IlcTimeBoundConstraintType
IlcTimeExtent

Function Summary
IlcActivityAltResConstraintTranslator
IlcActivityDurationMaxEvaluator
IlcActivityDurationMinEvaluator
IlcActivityEndMaxEvaluator
IlcActivityEndMinEvaluator
IlcActivityEndVarBoundPredicate
IlcActivityIntegralExp
IlcActivityIsBreakablePredicate
IlcActivityIsRankedPredicate
IlcActivityPostponedBackwardPredicate
IlcActivityPostponedPredicate
IlcActivityProcessingTimeMaxEvaluator

IlcActivityProcessingTimeMinEvaluator
IlcActivityProcessingTimeVarBoundPredicate
IlcActivityRandomEvaluator
IlcActivityResourceConstraintTranslator
IlcActivityStartMaxEvaluator
IlcActivityStartMinEvaluator
IlcActivityStartVarBoundPredicate
IlcActivityTransitionTypeEvaluator
IlcAltResConstraintCapacityEvaluator
IlcAltResConstraintNbPossibleEvaluator
IlcAltResConstraintResourceSelectedPredicate
IlcAltResConstraintVariableConstraintPredicate
IlcAssign
IlcAssignAlternative
IlcFunctionalExp
IlcGetThreadId
IlcMakeTransitionCost
IlcMakeTransitionTime
IlcProbabilisticCriticalityCalculator
IlcRank
IlcRankBackward
IlcRCTextureESTFactory
IlcRCTextureProbabilisticFactory
IlcRCTextureTargetFactory
IlcRelativeDemandCriticalityCalculator
IlcResourceCapacityEvaluator
IlcResourceClosedPredicate
IlcResourceConstraintCapacityConstraintPredicate
IlcResourceConstraintCapacityMaxEvaluator
IlcResourceConstraintCapacityMinEvaluator
IlcResourceConstraintHasNextPredicate
IlcResourceConstraintHasPrevPredicate
IlcResourceConstraintInwardConstraintPredicate
IlcResourceConstraintNegativeConstraintPredicate
IlcResourceConstraintNextTransitionCostEvaluator
IlcResourceConstraintPossibleFirstPredicate
IlcResourceConstraintPossibleLastPredicate
IlcResourceConstraintPossibleNextVisitor
IlcResourceConstraintPossiblePrevVisitor
IlcResourceConstraintPossibleSetupPredicate
IlcResourceConstraintPossibleTeardownPredicate

IlcResourceConstraintPossiblyContributesPredicate
IlcResourceConstraintPrevTransitionCostEvaluator
IlcResourceConstraintProvidingConstraintPredicate
IlcResourceConstraintRandomEvaluator
IlcResourceConstraintSetupPredicate
IlcResourceConstraintSlopeConstraintPredicate
IlcResourceConstraintSlopeEvaluator
IlcResourceConstraintStateConstraintPredicate
IlcResourceConstraintStateSetConstraintPredicate
IlcResourceConstraintSurelyContributesPredicate
IlcResourceConstraintTeardownPredicate
IlcResourceConstraintVariableConstraintPredicate
IlcResourceConstraintVirtualNodePredicate
IlcResourceEnergyEvaluator
IlcResourceGlobalSlackEvaluator
IlcResourceHasAltResConstraintPredicate
IlcResourceHasBreaksPredicate
IlcResourceHasTexturePredicate
IlcResourceIsCapacityResourcePredicate
IlcResourceIsContinuousReservoirPredicate
IlcResourceIsDiscreteEnergyPredicate
IlcResourceIsDiscreteResourcePredicate
IlcResourceIsReservoirPredicate
IlcResourceIsStateResourcePredicate
IlcResourceIsUnaryResourcePredicate
IlcResourceLocalSlackEvaluator
IlcResourceRandomEvaluator
IlcResourceRankedPredicate
IlcResourceResourceConstraintTranslator
IlcResourceResourceConstraintTranslator
IlcResourceSequencedPredicate
IlcResourceTextureEvaluator
IlcScheduleOrPostpone
IlcScheduleOrPostponeBackward
IlcSequence
IlcSequenceBackward
IlcSetTimes
IlcSetTimesBackward
IlcShapeLowerThan
IlcTestSequencedResource
IlcTextureAltSuccessorGoal

IlcTextureSuccessorGoal
IlcTryAssign
IlcTryRankFirst
IlcTryRankLast
IlcTrySetSuccessor
operator<<
operator<=

The IBM® ILOG® Scheduler API.

Class IloSchedulerSolution::ActivityIterator

Definition file: ilsched/ilosolution.h
Include file: <ilsched/iloscheduler.h>

An instance of this class traverses the list of `IloActivity` instances that have been stored in an `IloSchedulerSolution`.

See Also: IloActivity, IloSchedulerSolution, IloSchedulerSolution::ResourceConstraintIterator, IloSchedulerSolution::ResourceIterator

Constructor Summary	
`public`	`ActivityIterator(IloSchedulerSolution sol)`

Method Summary	
`public IloBool`	`ok() const`
`public IloActivity`	`operator*()`
`public ActivityIterator &`	`operator++()`

Constructors

`public` **`ActivityIterator`**`(IloSchedulerSolution sol)`

This constructor creates an iterator to traverse all the activities that are stored in the given scheduler solution.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all the activities have been scanned by the iterator.

`public IloActivity` **`operator*`**`()`

This operator returns the current instance of `IloActivity`, the one to which the invoking iterator points. If the iterator is set past the end position, then this operator returns an empty handle.

`public ActivityIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator to the next instance of `IloActivity`.

Class IloGranularFunction::Cursor

Definition file: ilsched/ilogfbase.h
Include file: <ilsched/iloscheduler.h>

An instance of `IloGranularFunction::Cursor` traverses the segments of a granular function.

See Also: IloGranularFunction

Constructor Summary	
public	Cursor(const IloGranularFunction func, IloNum x)

Method Summary	
public IloNum	getSegmentMax() const
public IloNum	getSegmentMin() const
public IloNum	getValue() const
public IloBool	ok() const
public void	operator++()
public void	operator--()

Constructors

`public` **`Cursor`**`(const IloGranularFunction func, IloNum x)`

This constructor creates a cursor to traverse the segments of the granular function `func`. It is initialized at the segment containing the position `x`. If this position is invalid, an error will be raised.

Methods

`public IloNum` **`getSegmentMax`**`() const`

This member function returns the right-most valid position pertaining to the current segment.

`public IloNum` **`getSegmentMin`**`() const`

This member function returns the left-most valid position pertaining to the current segment.

`public IloNum` **`getValue`**`() const`

This member function returns the value taken by the function on the current segment.

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the cursor is a valid one. Otherwise, it returns `IloFalse`.

`public void` **`operator++`**`()`

This left-increment operator shifts the current position of the cursor to the next segment of the function.

`public void` **`operator--`**`()`

This left-decrement operator shifts the current position of the cursor to the previous segment of the function.

Class IlcActivity

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>

The class `IlcActivity` is the class for managing activities.

Activities can be linked together by *precedence* constraints (instances of the class `IlcPrecedenceConstraint`). Activities can be linked to resources by *resource* constraints (instances of the class `IlcResourceConstraint`). Activities can also be constrained temporally by instances of the class `IlcTimeBoundConstraint`.

Calendars, Processing Time, and Duration

In most of the cases, the four variables defining an activity are linked through the following relation: *start+duration==end*. Also, as the processing time corresponds to the nominal duration of the activity, *duration==processingTime*. Nevertheless, when calendars are defined, the activity duration can be greater than the nominal one due to break suspensions or efficiency. So the duration of an activity depends on the nominal duration and its ability to deal with breaks, *isBreakable*, and with efficiency, *useEfficiency*. Such abilities are defined by the `IlcActivity` constructor arguments `breakable` and `useEfficiency`.

Notice that a breakable activity can only be suspended by breaks; it cannot be suspended to execute another activity.

For a given instance of a breakable activity, some breaks may be allowed to suspend the activity while others may not. A break is said to be disjunctive with respect to a breakable activity if the activity cannot be suspended by that break—that is, the activity must be processed either before the break or after it.

By default, only the breaks with a duration equal to zero are considered as disjunctive.

The notion of a start (end) break overlap variable allows expression of the fact that an activity can start (finish) the processing inside some special breaks—called *possibly overlapped breaks*—and allows posting of constraints on possible overlap duration.

Restrictions with the use of break overlap variables on activities:

- No end break overlap variable can be defined on a breakable activity that has been created as `mayBeSuspendedAtEnd`.
- No start break overlap variable can be defined on a breakable activity that has been created as `mayBeSuspendedAtStart`.

In order to model more efficiently disjuntive behaviors and forbidden dates, it is possible to use shift objects `IlcShiftObject` attached to an `IlcCalendar`. Shifts can be represented directly in intention by the user or in extension using an `IlcIntervalList`. Using interval types, activity can ignore some breaks or shifts.

Detecting Inconsistencies

When changing the domain of a start time variable, an end time variable, a duration variable, or a processing time variable, Scheduler Engine will detect an inconsistency if a changed domain directly conflicts with temporal constraints and with the resource constraints of activities with fixed (instantiated) start and end times and processing time.

Scheduler Engine *may or may not* detect the inconsistency if the changed domain conflicts with the temporal and resource constraints associated with *unscheduled* activities, that is, activities for which the start and end times and/or the processing time are not instantiated yet.

Consuming Resources

An activity *consumes* a resource if some amount of the resource capacity must be made available for the execution of the activity and the capacity is *non-recoverable*, that is, the capacity is required from the beginning of the activity up to the end of time.

Producing and Consuming Reservoirs

An activity *produces* if some amount of the reservoir capacity is made available through the execution of the activity.

Time Extents

By default, an activity uses a resource from the activity's start time to its end time. However, it is possible to specify that a resource is used during a time range different from the default "start to end" range. The enumeration `IlcTimeExtent` is defined for this purpose.

Alternative Resource Set

When the member functions `IlcActivity::consumes` and `IlcActivity::requires` take the argument `IlcAltResSet`, all the resources in the set must be capacity resources. When the member functions `IlcActivity::produces` and `IlcActivity::provides` take the argument `IlcAltResSet`, all the resources in the set must be reservoirs.

Functional and Integral Constraints and External Variables

Functional and integral constraints are constraints of the form `y_rct=f(x_rct)` or `y_rct=sum{start_rct->end_rct}f(t).dt` that hold for every resource constraints `rct` on a given resource (see Functional and Integral Constraints on Resources.

External variables are useful in case some variable `x_rct` or `y_rct` above is not a variable already associated with the resource constraint (start, end, processing time of the activity, capacity demand, etc.). In that case, member functions on `IlcActivity` allow setting a given Solver variable as as external variable of the activity and use it in the functional/integral constraint (see the enumeration `IlcSchedVariable`).

Printing or Displaying Activities

The printed representation of an instance of the class `IlcActivity` consists of two parts: its name, followed by information about the start time, end time, duration, and if appropriate, the processing time of the activity.

This information is enclosed in brackets and for each of the four variables, consists of either a single value (if the start time, end time, or duration is precisely known) or a minimal value and a maximal value separated by two dots. The following examples represent activities that have not been suspended by breaks:

`[2 -- 3 --> 5]` represents an activity which starts at 2 and ends at 5. The duration of this activity is 3.

`[2 -- 3..5 --> 5..7]` represents an activity which starts at 2 and ends sometime between 5 and 7. The duration of this activity is between 3 and 5.

`[2..4 -- 3 --> 5..7]` represents an activity of duration 3 which starts sometime between 2 and 4 and ends sometime between 5 and 7.

The following examples represent activities that have been suspended by breaks:

`[2 -- (2) 3 --> 5]` represents an activity which starts at 2 and ends at 5. The processing time of this activity is 2. Its duration is 3. Such situation can appears with calendars when breaks or efficiency ar used. For instance, a break of duration 1 is situated between 2 and 5, or an efficiency function with value 2/3 between 2 and 5.

`[5 -- (3..5) 3..6 --> 8..11]` represents an activity of processing time between 3 and 5 and duration between 3 and 6; it starts at 5 and ends sometime between 8 and 11.

`[10..30 -- (5) (5..15) --> 15..45]` represents an activity of processing time 5 and duration between 5 and 15; it starts sometime between 10 and 30 and ends sometime between 15 and 45.

For more information, see Calendars, Precedence Graph Constraints, and Functional and Integral Constraints on Resources.

See Also: IlcActivityIterator, IlcPrecedenceConstraint, IlcResourceConstraint, IlcSchedule, IlcScheduleOrPostpone, IlcSchedVariable, IlcTimeBoundConstraint

Constructor Summary	
public	IlcActivity()
public	IlcActivity(IlcActivityI * impl)
public	IlcActivity(const IlcSchedule schedule, IlcInt processingTime, IlcBool breakable=IlcFalse, IlcBool mayBeSuspendedAtStart=IlcFalse, IlcBool mayBeSuspendedAtEnd=IlcFalse, IlcBool useEfficiency=IlcFalse)
public	IlcActivity(const IlcSchedule schedule, IlcIntVar processingTimeVariable, IlcBool breakable=IlcFalse, IlcBool mayBeSuspendedAtStart=IlcFalse, IlcBool mayBeSuspendedAtEnd=IlcFalse, IlcBool useEfficiency=IlcFalse)
public	IlcActivity(const IlcSchedule schedule, IlcIntVar startVariable, IlcIntVar endVariable, IlcIntVar processingTimeVariable, IlcBool breakable=IlcFalse, IlcBool mayBeSuspendedAtStart=IlcFalse, IlcBool mayBeSuspendedAtEnd=IlcFalse, IlcBool useEfficiency=IlcFalse)
public	IlcActivity(const IlcSchedule schedule, IlcIntVar startVariable, IlcIntVar endVariable, IlcIntVar processingTimeVariable, IlcIntVar durationVariable, IlcBool mayBeSuspendedAtStart=IlcFalse, IlcBool mayBeSuspendedAtEnd=IlcFalse, IlcBool useEfficiency=IlcFalse)

Method Summary	
public void	addDisjunctiveBreakType(IlcIntSet setOfTypes)
public void	addDisjunctiveBreakType(IlcInt type)
public void	addEndBreakOverlapType(IlcInt type)
public void	addEndBreakOverlapType(IlcIntSet typeSet)
public void	addIgnoredBreakType(IlcIntSet setOfTypes)
public void	addIgnoredBreakType(IlcInt type)
public void	addIgnoredShiftType(IlcIntSet setOfTypes)
public void	addIgnoredShiftType(IlcInt type)
public void	addStartBreakOverlapType(IlcInt type)
public void	addStartBreakOverlapType(IlcIntSet typeSet)
public IlcBool	canBeSuspendedAtEnd() const
public IlcBool	canBeSuspendedAtStart() const
public IlcAltResConstraint	consumes(IlcAltResSet set, IlcIntVar capacity)
public IlcAltResConstraint	consumes(IlcAltResSet set, IlcInt capacity=1)
public IlcResourceConstraint	consumes(IlcCapResource resource, IlcIntVar capacity)
public IlcResourceConstraint	consumes(IlcCapResource resource, IlcInt capacity=1)
public IlcConstraint	covers(IlcActivityArray actAr)
public IlcTimeBoundConstraint	endsAfter(IlcIntVar time)
public IlcTimeBoundConstraint	endsAfter(IlcInt time)
public IlcPrecedenceConstraint	endsAfterEnd(IlcActivity act, IlcIntVar)

public IlcPrecedenceConstraint	endsAfterEnd(IlcActivity act, IlcInt delay=0)
public IlcPrecedenceConstraint	endsAfterStart(IlcActivity act, IlcIntVar)
public IlcPrecedenceConstraint	endsAfterStart(IlcActivity act, IlcInt delay=0)
public IlcTimeBoundConstraint	endsAt(IlcIntVar time)
public IlcTimeBoundConstraint	endsAt(IlcInt time)
public IlcPrecedenceConstraint	endsAtEnd(IlcActivity act, IlcIntVar)
public IlcPrecedenceConstraint	endsAtEnd(IlcActivity act, IlcInt delay=0)
public IlcPrecedenceConstraint	endsAtStart(IlcActivity act, IlcIntVar)
public IlcPrecedenceConstraint	endsAtStart(IlcActivity act, IlcInt delay=0)
public IlcTimeBoundConstraint	endsBefore(IlcIntVar time)
public IlcTimeBoundConstraint	endsBefore(IlcInt time)
public IlcInt	getDurationMax() const
public IlcInt	getDurationMaxNormalBreaks() const
public IlcInt	getDurationMin() const
public IlcInt	getDurationMinNormalBreaks() const
public IlcIntVar	getDurationVariable() const
public IlcInt	getEndBreakOverlapMax() const
public IlcInt	getEndBreakOverlapMin() const
public IlcIntVar	getEndBreakOverlapVariable() const
public IlcInt	getEndMax() const
public IlcInt	getEndMin() const
public IlcIntVar	getEndVariable() const
public IlcInt	getExecutionDurationMin() const
public IlcIntVar	getExternalVar() const
public IlcInt	getExternalVarMax() const
public IlcInt	getExternalVarMin() const
public IlcActivityI *	getImpl() const
public const char *	getName() const
public IlcAny	getObject() const
public IlcInt	getProcessingTimeMax() const
public IlcInt	getProcessingTimeMin() const
public IlcIntVar	getProcessingTimeVariable() const
public IlcSchedule	getSchedule() const
public IloSolver	getSolver() const
public IloSolverI *	getSolverI() const
public IlcInt	getStartBreakOverlapMax() const
public IlcInt	getStartBreakOverlapMin() const
public IlcIntVar	getStartBreakOverlapVariable() const
public IlcInt	getStartMax() const
public IlcInt	getStartMin() const
public IlcIntVar	getStartVariable() const

public IlcInt	getTransitionType() const
public IlcBool	hasEndBreakOverlapVariable() const
public IlcBool	hasStartBreakOverlapVariable() const
public IlcBool	isBreakable() const
public IlcBool	isDirectlySucceededBy(IlcActivity) const
public IlcBool	isDisjunctiveBreakType(IlcInt type) const
public IlcBool	isExternalVarBound() const
public IlcBool	isIgnoredBreakType(IlcInt type) const
public IlcBool	isIgnoredShiftType(IlcInt type) const
public IlcBool	isPostponed() const
public IlcBool	isPostponedBackward() const
public IlcBool	isRanked() const
public IlcBool	isSucceededBy(IlcActivity) const
public IlcBool	operator!=(const IlcActivity & activity) const
public void	operator=(const IlcActivity & h)
public IlcBool	operator==(const IlcActivity & activity) const
public void	postpone()
public void	postponeBackward()
public IlcAltResConstraint	produces(IlcAltResSet set, IlcIntVar capacity)
public IlcAltResConstraint	produces(IlcAltResSet set, IlcInt capacity=1)
public IlcResourceConstraint	produces(IlcContinuousReservoir resource, IlcIntVar capacity)
public IlcResourceConstraint	produces(IlcContinuousReservoir resource, IlcInt capacity=1)
public IlcResourceConstraint	produces(IlcReservoir resource, IlcIntVar capacity)
public IlcResourceConstraint	produces(IlcReservoir resource, IlcInt capacity=1)
public IlcAltResConstraint	provides(IlcAltResSet set, IlcIntVar capacity, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcFalse)
public IlcAltResConstraint	provides(IlcAltResSet set, IlcInt capacity=1, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcFalse)
public IlcResourceConstraint	provides(IlcReservoir resource, IlcIntVar capacity, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcFalse)
public IlcResourceConstraint	provides(IlcReservoir resource, IlcInt capacity=1, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcFalse)
public void	removeDisjunctiveBreakType(IlcIntSet setOfTypes)
public void	removeDisjunctiveBreakType(IlcInt type)
public void	removeEndBreakOverlapType(IlcInt type)
public void	removeEndBreakOverlapType(IlcIntSet typeSet)
public void	removeIgnoredBreakType(IlcIntSet setOfTypes)
public void	removeIgnoredBreakType(IlcInt type)

public void	removeIgnoredShiftType(IlcIntSet setOfTypes)
public void	removeIgnoredShiftType(IlcInt type)
public void	removeStartBreakOverlapType(IlcInt type)
public void	removeStartBreakOverlapType(IlcIntSet typeSet)
public IlcResourceConstraint	requires(IlcStateResource resource, IlcAnySetVar states, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcResourceConstraint	requires(IlcStateResource resource, IlcAnySet states, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcResourceConstraint	requires(IlcStateResource resource, IlcAnyVar state, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcResourceConstraint	requires(IlcStateResource resource, IlcAny state, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcAltResConstraint	requires(IlcAltResSet set, IlcIntVar capacity, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcAltResConstraint	requires(IlcAltResSet set, IlcInt capacity=1, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcResourceConstraint	requires(IlcCapResource resource, IlcIntVar capacity, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcResourceConstraint	requires(IlcCapResource resource, IlcInt capacity=1, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcResourceConstraint	requiresNot(IlcStateResource resource, IlcAnySetVar states, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcResourceConstraint	requiresNot(IlcStateResource resource, IlcAnySet states, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcResourceConstraint	requiresNot(IlcStateResource resource, IlcAnyVar state, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcResourceConstraint	requiresNot(IlcStateResource resource, IlcAny state, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public void	setDuration(IlcInt duration)
public void	setDurationMax(IlcInt durationMax)
public void	setDurationMaxNormalBreaks(IlcInt duration)
public void	setDurationMin(IlcInt durationMin)
public void	setDurationMinNormalBreaks(IlcInt duration)
public void	setEndBreakOverlap(IlcInt overlap)
public void	setEndBreakOverlapMax(IlcInt overlapMax)
public void	setEndBreakOverlapMin(IlcInt overlapMin)
public void	setEndBreakOverlapVariable(IlcIntVar overlapVariable)

public void	setEndMax(IlcInt endMax)
public void	setEndMin(IlcInt endMin)
public void	setEndTime(IlcInt endTime)
public void	setExecutionDurationMin(IlcInt)
public void	setExternalValue(IlcInt)
public void	setExternalVar(IlcIntVar)
public void	setExternalVarMax(IlcInt)
public void	setExternalVarMin(IlcInt)
public void	setName(const char * name) const
public void	setObject(IlcAny object) const
public void	setProcessingTime(IlcInt processingTime)
public void	setProcessingTimeMax(IlcInt processingTimeMax)
public void	setProcessingTimeMin(IlcInt processingTimeMin)
public void	setStartBreakOverlap(IlcInt overlap)
public void	setStartBreakOverlapMax(IlcInt overlapMax)
public void	setStartBreakOverlapMin(IlcInt overlapMin)
public void	setStartBreakOverlapVariable(IlcIntVar overlapVariable)
public void	setStartMax(IlcInt startMax)
public void	setStartMin(IlcInt startMin)
public void	setStartTime(IlcInt startTime)
public void	setSuccessor(IlcActivity ct)
public void	setTransitionType(IlcInt value)
public IlcTimeBoundConstraint	startsAfter(IlcIntVar time)
public IlcTimeBoundConstraint	startsAfter(IlcInt time)
public IlcPrecedenceConstraint	startsAfterEnd(IlcActivity act, IlcIntVar)
public IlcPrecedenceConstraint	startsAfterEnd(IlcActivity act, IlcInt delay=0)
public IlcPrecedenceConstraint	startsAfterStart(IlcActivity act, IlcIntVar)
public IlcPrecedenceConstraint	startsAfterStart(IlcActivity act, IlcInt delay=0)
public IlcTimeBoundConstraint	startsAt(IlcIntVar time)
public IlcTimeBoundConstraint	startsAt(IlcInt time)
public IlcPrecedenceConstraint	startsAtEnd(IlcActivity act, IlcIntVar)
public IlcPrecedenceConstraint	startsAtEnd(IlcActivity act, IlcInt delay=0)
public IlcPrecedenceConstraint	startsAtStart(IlcActivity act, IlcIntVar)
public IlcPrecedenceConstraint	startsAtStart(IlcActivity act, IlcInt delay=0)
public IlcTimeBoundConstraint	startsBefore(IlcIntVar time)
public IlcTimeBoundConstraint	startsBefore(IlcInt time)
public void	unsetSuccessor(IlcActivity ct)
public IlcBool	useEfficiency() const

Constructors

```
public IlcActivity()
```

This constructor creates an empty handle. You must initialize it before you use it.

```
public IlcActivity(IlcActivityI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IlcActivity(const IlcSchedule schedule, IlcInt processingTime, IlcBool
breakable=IlcFalse, IlcBool mayBeSuspendedAtStart=IlcFalse, IlcBool
mayBeSuspendedAtEnd=IlcFalse, IlcBool useEfficiency=IlcFalse)
```

This constructor creates a new instance of `IlcActivity` and adds it to the activities managed in the given `schedule`. The new activity is constrained to execute sometime between the time origin and the time horizon of the schedule. Its processing time is set to the given `processingTime`. If the value of the argument `breakable` is equal to `IlcTrue`, then the created activity is a breakable activity; that is, an activity that can be suspended by breaks. Otherwise and by default, the activity is a non-breakable activity. The arguments `mayBeSuspendedAtStart` and `mayBeSuspendedAtEnd` are relevant for breakable activities only. They tell whether the activity can be suspended at its start or end time. If the value of the argument `useEfficiency` is equal to `IlcTrue`, then the created activity is able to deal with efficiency functions defined on required resources. See Calendars for more information.

```
public IlcActivity(const IlcSchedule schedule, IlcIntVar processingTimeVariable,
IlcBool breakable=IlcFalse, IlcBool mayBeSuspendedAtStart=IlcFalse, IlcBool
mayBeSuspendedAtEnd=IlcFalse, IlcBool useEfficiency=IlcFalse)
```

This constructor creates a new instance of `IlcActivity` and adds it to the activities managed in the given `schedule`. The new activity is constrained to execute sometime between the time origin and the time horizon of the schedule. Its processing time variable is given by the argument `processingTimeVariable`. If the value of the argument `breakable` is equal to `IlcTrue`, then the created activity is a breakable activity, that is, an activity that can be suspended by breaks. Otherwise and by default, the activity is a non-breakable activity. The arguments `mayBeSuspendedAtStart` and `mayBeSuspendedAtEnd` are relevant for breakable activities only. They tell whether the activity can be suspended at its start or end time. If the value of the argument `useEfficiency` is equal to `IlcTrue`, then the created activity is able to deal with efficiency functions defined on required resources. See Calendars for more information.

```
public IlcActivity(const IlcSchedule schedule, IlcIntVar startVariable, IlcIntVar
endVariable, IlcIntVar processingTimeVariable, IlcBool breakable=IlcFalse, IlcBool
mayBeSuspendedAtStart=IlcFalse, IlcBool mayBeSuspendedAtEnd=IlcFalse, IlcBool
useEfficiency=IlcFalse)
```

This constructor creates a new instance of `IlcActivity` and adds it to the activities managed in the given `schedule`. The new activity is created with the start, end and processing time variables given as arguments. If the value of the argument `breakable` is equal to `IlcTrue`, then the created activity is a breakable activity, that is, an activity that can be suspended by breaks. Otherwise and by default, the activity is a non-breakable activity. The arguments `mayBeSuspendedAtStart` and `mayBeSuspendedAtEnd` are relevant for breakable activities only. They tell whether the activity can be suspended at its start or end time. If the value of the argument `useEfficiency` is equal to `IlcTrue`, then the created activity is able to deal with efficiency functions defined on required resources. See Calendars for more information.

```
public IlcActivity(const IlcSchedule schedule, IlcIntVar startVariable, IlcIntVar
endVariable, IlcIntVar processingTimeVariable, IlcIntVar durationVariable, IlcBool
mayBeSuspendedAtStart=IlcFalse, IlcBool mayBeSuspendedAtEnd=IlcFalse, IlcBool
useEfficiency=IlcFalse)
```

This constructor creates a new instance of `IlcActivity` and adds it to the activities managed in the given `schedule`. The new activity is breakable. Its start, end, processing time and duration variables are given as arguments. The arguments `mayBeSuspendedAtStart` and `mayBeSuspendedAtEnd` tell whether the activity can be suspended at its start or end time. If the value of the argument `useEfficiency` is equal to `IlcTrue`, then the created activity is able to deal with efficiency functions defined on required resources. See Calendars for more information.

Methods

`public void` **`addDisjunctiveBreakType`**`(IlcIntSet setOfTypes)`

This member function adds the set of types `setOfTypes` to the set of disjunctive break types of the invoking activity. If a break type belongs to the set of disjunctive break types of a breakable activity, the activity must be completely processed either before or after that type of break.

Initially, a breakable activity is created with an empty set of disjunctive break types.

`public void` **`addDisjunctiveBreakType`**`(IlcInt type)`

This member function adds the type `type` to the set of disjunctive break types of the invoking activity. If a break type belongs to the set of disjunctive break types of a breakable activity, the activity must be completely processed either before or after that type of break.

Initially, a breakable activity is created with an empty set of disjunctive break types.

`public void` **`addEndBreakOverlapType`**`(IlcInt type)`

For each activity, a set of break types is given that defines which break types can possibly be overlapped by the end of the activity. This member function adds `type` to this set of break types on the invoking activity. By default, the set is empty.

This member function is available only outside the search. In particular, it cannot be called within a goal.

`public void` **`addEndBreakOverlapType`**`(IlcIntSet typeSet)`

For each activity, a set of break types is given that defines which break types can possibly be overlapped by the end of the activity. This member function adds the types in `typeSet` to this set of break types on the invoking activity. By default, the set is empty.

This member function is available only outside the search. In particular, it cannot be called within a goal.

`public void` **`addIgnoredBreakType`**`(IlcIntSet setOfTypes)`

This member function adds the set of types `setOfTypes` to the set of ignored break types of the invoking activity. That is, the invoking activity behaves as if breaks of type included in `setOfTypes` never exist.

Initially, an activity is created with an empty set of ignored break types.

`public void` **`addIgnoredBreakType`**`(IlcInt type)`

This member function adds the type `type` to the set of ignored break types of the invoking activity. That is, the invoking activity behaves as if breaks of type `type` never exist.

Initially, an activity is created with an empty set of ignored break types.

```
public void addIgnoredShiftType(IlcIntSet setOfTypes)
```

This member function adds the set of types `setOfTypes` to the set of ignored shift types of the invoking activity. That is, the invoking activity behaves as if shifts of type included in `setOfTypes` never exist.

Initially, an activity is created with an empty set of ignored shift types.

```
public void addIgnoredShiftType(IlcInt type)
```

This member function adds the type `type` to the set of ignored shift types of the invoking activity. That is, the invoking activity behaves as if shifts of type `type` never exist.

Initially, an activity is created with an empty set of ignored shift types.

```
public void addStartBreakOverlapType(IlcInt type)
```

For each activity, a set of break types is given that defines which break types can possibly be overlapped by the start of the activity. This member function adds `type` to this set of break types on the invoking activity. By default, the set is empty.

This member function is available only outside the search. In particular, it cannot be called within a goal.

```
public void addStartBreakOverlapType(IlcIntSet typeSet)
```

For each activity, a set of break types is given that defines which break types can possibly be overlapped by the start of the activity. This member function adds the types in `typeSet` to this set of break types on the invoking activity. By default, the set is empty.

This member function is available only outside the search. In particular, it cannot be called within a goal.

```
public IlcBool canBeSuspendedAtEnd() const
```

This member function returns `IlcTrue` if the invoking activity can be suspended at the end of the activity. Otherwise, it returns `IlcFalse`.

```
public IlcBool canBeSuspendedAtStart() const
```

This member function returns `IlcTrue` if the invoking activity can be suspended at the start of the activity. Otherwise, it returns `IlcFalse`.

```
public IlcResourceConstraint consumes(IlcCapResource resource, IlcInt capacity=1)
public IlcAltResConstraint consumes(IlcAltResSet set, IlcIntVar capacity)
```

```
public IlcAltResConstraint consumes(IlcAltResSet set, IlcInt capacity=1)
public IlcResourceConstraint consumes(IlcCapResource resource, IlcIntVar capacity)
```

An activity consumes a resource if some amount of the resource capacity must be made available for the execution of the activity and the capacity is non-recoverable after the end of the activity. For example, an activity might consume a raw material in manufacturing a product. If the resource is discrete (class `IlcDiscreteResource`, `IlcReservoir`, or `IlcDiscreteEnergy`), the activity requires the capacity at all times after the activity's start time. The corresponding member function implies that the occupancy of the resource by the activity is rounded inward toward the nearest valid time that corresponds to a time step.

These two expressions are equivalent: `activity.consumes(reservoir, capacity);` and `activity.requires(reservoir, capacity, IlcAfterStart);`

If the resource is a continuous reservoir (class `IlcContinuousReservoir`), the consumption is continuous and linear from the start time to the end time of the invoking activity. Since the time step of a timetable for a continuous reservoir is 1, the returned resource constraint has no inward/outward rounding mode. Its time extent is not defined as it does not match any case of the enumeration `IlcTimeExtent`.

If the invoking activity consumes a resource in `set`, the consumption will be discrete if the selected resource is an instance of `IlcDiscreteResource`, `IlcReservoir`, or `IlcDiscreteEnergy`. It will be continuous if the selected resource is an instance of `IlcContinuousReservoir`. An instance of `IloSolver::SolverErrorException` is thrown if either the capacity is a strictly negative integer or if the capacity is a constrained integer variable with a strictly negative minimal value.

```
public IlcConstraint covers(IlcActivityArray actAr)
```

This member function creates a cover constraint. This constraint states that the start time of the invoking activity is equal to the earliest of the start times of the activities in the array given as argument, and that the end time of the invoking activity is equal to the latest of the end times of the activities in the array given as argument. In other words, the invoking activity exactly covers the activities in the array given as argument.

```
public IlcTimeBoundConstraint endsAfter(IlcInt time)
public IlcTimeBoundConstraint endsAfter(IlcIntVar time)
```

This member function states that the invoking activity must end after or at `time`. More formally, `act.endsAfter(time)` means `end(act) >= time`.

```
public IlcPrecedenceConstraint endsAfterEnd(IlcActivity act, IlcInt delay=0)
public IlcPrecedenceConstraint endsAfterEnd(IlcActivity act, IlcIntVar)
```

This member function states that the invoking activity ends after the end of `act`. In addition, at least the given delay must elapse between the end of `act` and the end of the invoking activity.

The member function can be invoked with a negative `delay`, which means that the invoking activity can end before the end of `act`, but the difference between the end time of `act` and the end time of the invoking activity cannot exceed `-delay`.

More formally, `act1.endsAfterEnd(act, delay)` means `end(act1) >= end(act) + delay`.

```
public IlcPrecedenceConstraint endsAfterStart(IlcActivity act, IlcInt delay=0)
public IlcPrecedenceConstraint endsAfterStart(IlcActivity act, IlcIntVar)
```

This member function states that the invoking activity ends after the beginning of `act`. In addition, at least the given `delay` must elapse between the beginning of `act` and the end of the invoking activity.

The member function can be invoked with a negative `delay`, which means that the invoking activity can end before the beginning of `act`, but the difference between the start time of `act` and the end time of the invoking activity cannot exceed `-delay`.

More formally, `act1.endsAfterStart(act, delay)` means `end(act1) >= start(act) + delay`.

```
public IlcTimeBoundConstraint endsAt(IlcInt time)
public IlcTimeBoundConstraint endsAt(IlcIntVar time)
```

This member function states that the invoking activity must end at `time`. More formally, `act.endsAt(time)` means `end(act) == time`.

```
public IlcPrecedenceConstraint endsAtEnd(IlcActivity act, IlcInt delay=0)
public IlcPrecedenceConstraint endsAtEnd(IlcActivity act, IlcIntVar)
```

This member function states that exactly the given delay must elapse between the end of `act` and the end of the invoking activity. More formally, `act1.endsAtEnd(act, delay)` means `end(act1) == end(act) + delay`.

```
public IlcPrecedenceConstraint endsAtStart(IlcActivity act, IlcInt delay=0)
public IlcPrecedenceConstraint endsAtStart(IlcActivity act, IlcIntVar)
```

This member function states that exactly the given delay must elapse between the beginning of `act` and the end of the invoking activity. More formally, `act1.endsAtStart(act, delay)` means `end(act1) == start(act) + delay`.

```
public IlcTimeBoundConstraint endsBefore(IlcInt time)
public IlcTimeBoundConstraint endsBefore(IlcIntVar time)
```

This member function states that the invoking activity must end before or at `time`. More formally, `act.endsBefore(time)` means `end(act) <= time`.

```
public IlcInt getDurationMax() const
```

This member function returns the longest possible duration of the invoking activity.

```
public IlcInt getDurationMaxNormalBreaks() const
```

This member function returns the threshold duration above which all breaks are, by default, considered as disjunctive. The default value is `IlcIntMax`.

```
public IlcInt getDurationMin() const
```

This member function returns the shortest possible duration of the invoking activity.

```
public IlcInt getDurationMinNormalBreaks() const
```

This member function returns the threshold duration under which all breaks are considered as disjunctive. By default, the value of this minimal duration is 1 so that only the breaks with null duration are considered as disjunctive.

```
public IlcIntVar getDurationVariable() const
```

This member function returns the Solver variable that represents the duration of the invoking activity.

```
public IlcInt getEndBreakOverlapMax() const
```

This member function returns the maximal value of the end break overlap variable of the invoking activity.

```
public IlcInt getEndBreakOverlapMin() const
```

This member function returns the minimal value of the end break overlap variable of the invoking activity.

```
public IlcIntVar getEndBreakOverlapVariable() const
```

This member function returns the end break overlap variable of the invoking activity.

```
public IlcInt getEndMax() const
```

This member function returns the latest possible end time of the invoking activity.

```
public IlcInt getEndMin() const
```

This member function returns the earliest possible end time of the invoking activity.

```
public IlcIntVar getEndVariable() const
```

This member function returns the Solver variable that represents the end time of the invoking activity.

```
public IlcInt getExecutionDurationMin() const
```

A breakable activity executes during a set of disjoint temporal intervals. These execution intervals are separated by intervals that correspond to the breaks that suspend the activity.

This member function returns the minimal duration for the execution intervals of the invoking activity.

The default minimal duration is 1. It can be redefined by calling the member function `IlcActivity::setExecutionDurationMin`.

```
public IlcIntVar getExternalVar() const
```

This member function returns the external variable of the invoking activity. Note that by default, the external variable of an activity is a variable with a domain `[IlcIntMin, IlcIntMax]`.

```
public IlcInt getExternalVarMax() const
```

This member function returns the maximal value of the external variable of the invoking activity.

```
public IlcInt getExternalVarMin() const
```

This member function returns the minimal value of the external variable of the invoking activity.

```
public IlcActivityI * getImpl() const
```

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

```
public const char * getName() const
```

This member function returns the name of the invoking object.

```
public IlcAny getObject() const
```

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

```
public IlcInt getProcessingTimeMax() const
```

This member function returns the longest possible processing time for the invoking activity.

```
public IlcInt getProcessingTimeMin() const
```

This member function returns the shortest possible processing time for the invoking activity.

```
public IlcIntVar getProcessingTimeVariable() const
```

This member function returns the Solver variable that represents the processing time for the invoking activity.

```
public IlcSchedule getSchedule() const
```

This member function returns the schedule to which the invoking activity belongs. Each activity belongs to a unique schedule, an instance of `IlcSchedule`.

```
public IloSolver getSolver() const
```

This member function returns an instance of `IloSolver` associated with the invoking object.

```
public IloSolverI * getSolverI() const
```

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

```
public IlcInt getStartBreakOverlapMax() const
```

This member function returns the maximal value of the start break overlap variable of the invoking activity.

```
public IlcInt getStartBreakOverlapMin() const
```

This member function returns the minimal value of the start break overlap variable of the invoking activity.

```
public IlcIntVar getStartBreakOverlapVariable() const
```

This member function returns the start break overlap variable of the invoking activity.

```
public IlcInt getStartMax() const
```

This member function returns the latest possible start time of the invoking activity.

```
public IlcInt getStartMin() const
```

This member function returns the earliest possible start time of the invoking activity.

```
public IlcIntVar getStartVariable() const
```

This member function returns the Solver variable that represents the start time of the invoking activity.

```
public IlcInt getTransitionType() const
```

The transition type of an activity is an integer intended to define transition time and cost from an indexed classification of activities. It is used by transition tables (instances of the class `IlcTransitionTable`).

This member function returns the transition type of the invoking activity. By default, the transition type is set to zero.

```
public IlcBool hasEndBreakOverlapVariable() const
```

This member function returns `IlcTrue` if the invoking activity has an end break overlap variable. Otherwise, it returns `IlcFalse`.

An end break overlap variable is created on an activity as soon as one of the following member functions has been called: `IlcActivity::addEndBreakOverlapType`, `IlcActivity::setEndBreakOverlapVariable`, `IlcActivity::setEndBreakOverlapMax`, `IlcActivity::setEndBreakOverlapMin`, or `IlcActivity::setEndBreakOverlap`.

```
public IlcBool hasStartBreakOverlapVariable() const
```

This member function returns `IlcTrue` if the invoking activity has a start break overlap variable. Otherwise, it returns `IlcFalse`.

A start break overlap variable is created on an activity as soon as one of the following member functions has been called: `IlcActivity::addStartBreakOverlapType`, `IlcActivity::setStartBreakOverlapVariable`, `IlcActivity::setStartBreakOverlapMax`, `IlcActivity::setStartBreakOverlapMin`, or `IlcActivity::setStartBreakOverlap`.

```
public IlcBool isBreakable() const
```

This member function returns `IlcTrue` if the invoking activity is a breakable activity.Otherwise, it returns `IlcFalse`.

```
public IlcBool isDirectlySucceededBy(IlcActivity) const
```

This member function returns `IlcTrue` if the invoking activity is directly succeeded by the activity `act`. Otherwise, it returns `IlcFalse`.

This member function should be used only in search and only when a precedence graph constraint has been created on the schedule.

```
public IlcBool isDisjunctiveBreakType(IlcInt type) const
```

This member function returns `IlcTrue` if `type` is a disjunctive break type of the invoking activity.

```
public IlcBool isExternalVarBound() const
```

This member function returns `IlcTrue` if and only if the external variable of the invoking activity is bound.

```
public IlcBool isIgnoredBreakType(IlcInt type) const
```

This member function returns `IloTrue` if the type `type` belongs to the set of ignored break types of the invoking activity. Otherwise, it returns `IloFalse`.

```
public IlcBool isIgnoredShiftType(IlcInt type) const
```

This member function returns `IloTrue` if the type `type` belongs to the set of ignored shift types of the invoking activity. Otherwise, it returns `IloFalse`.

```
public IlcBool isPostponed() const
```

This member function returns `IlcTrue` if the earliest start time of the invoking activity is equal to the earliest start time at the moment of the most recent call to the member function `postpone`. Otherwise, it returns `IlcFalse`.

```
public IlcBool isPostponedBackward() const
```

This member function returns `IlcTrue` if the latest end time of the invoking activity is equal to the latest end time at the moment of the most recent call to the member function `IlcActivity::postponeBackward`. Otherwise, it returns `IlcFalse`.

```
public IlcBool isRanked() const
```

This member function returns `IlcTrue` if all the other activities of the schedule are constrained to execute either before or after the invoking activity. Otherwise, it returns `IlcFalse`.

This member function should be used only in search and only when a precedence graph constraint has been created on the schedule.

```
public IlcBool isSucceededBy(IlcActivity) const
```

Before entering the search, this function returns `IlcTrue` if and only if a successor relation has been added with the member function `IlcActivity::setSuccessor`.

In search, this member function returns `IlcTrue` if the invoking activity is succeeded by the activity `act`. Otherwise, it returns `IlcFalse`.

This member function should be used only when a precedence graph constraint has been created on the schedule.

```
public IlcBool operator!=(const IlcActivity & activity) const
```

This operator returns `IlcTrue` if and only if `activity` does *not* refer to the same implementation object as the invoking activity.

```
public void operator=(const IlcActivity & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public IlcBool operator==(const IlcActivity & activity) const
```

This operator returns `IlcTrue` if and only if `activity` refers to the same implementation object as the invoking activity.

```
public void postpone()
```

This function insures that the invoking activity is treated as a postponed activity until the earliest start time changes.

```
public void postponeBackward()
```

This function insures that the invoking activity is treated as a backward postponed activity until the latest end time changes.

```
public IlcResourceConstraint produces(IlcReservoir resource, IlcInt capacity=1)
public IlcAltResConstraint produces(IlcAltResSet set, IlcIntVar capacity)
public IlcAltResConstraint produces(IlcAltResSet set, IlcInt capacity=1)
public IlcResourceConstraint produces(IlcContinuousReservoir resource, IlcIntVar
capacity)
public IlcResourceConstraint produces(IlcContinuousReservoir resource, IlcInt
capacity=1)
public IlcResourceConstraint produces(IlcReservoir resource, IlcIntVar capacity)
```

An activity *produces* if some amount of the reservoir capacity is made available through the execution of the activity. This member function states that the invoking activity produces the given `capacity` of the given reservoir.

If the reservoir is discrete (class `IlcReservoir`), this member function implies that the occupancy of the reservoir by the activity is rounded inward toward the nearest valid time that corresponds to a time step.

These two expressions are equivalent:

```
activity.produces(reservoir, capacity);
activity.provides(reservoir, capacity, IlcAfterEnd);
```

If the reservoir is continuous (class `IlcContinuousReservoir`), the production process is continuous and linear from the start time to the end time of the invoking activity. Since the time step of a timetable for a continuous reservoir is 1, the returned resource constraint has no inward/outward rounding mode. Its time extent is not defined, since it does not match any case of the enumeration `IlcTimeExtent`.

If the invoking activity produces for a reservoir in `set`, the production will be discrete if the selected reservoir is an instance of `IlcReservoir`. It will be continuous if the selected reservoir is an instance of `IlcContinuousReservoir`).

An instance of `IloSolver::SolverErrorException` is thrown if either of the following conditions occurs:

- if `capacity` is a strictly negative integer, or
- if `capacity` is a constrained integer variable with a strictly negative minimal value.

```
public IlcResourceConstraint provides(IlcReservoir resource, IlcInt capacity=1,
IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcFalse)
public IlcAltResConstraint provides(IlcAltResSet set, IlcIntVar capacity,
IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcFalse)
public IlcAltResConstraint provides(IlcAltResSet set, IlcInt capacity=1,
IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcFalse)
public IlcResourceConstraint provides(IlcReservoir resource, IlcIntVar capacity,
IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcFalse)
```

This member function states that the invoking activity provides the given `capacity` of the given reservoir. By default, the activity provides the reservoir from the beginning to the end of its execution. However, the optional argument `extent` is available to represent cases in which the activity provides the reservoir over a different time extent, as explained in `IlcTimeExtent`.

The argument `outward` is important only when one of the timetables of the reservoir has a time step greater than 1 (one). In that case, `outward` defines whether the occupancy of the reservoir by the activity should be

rounded outward or inward towards the nearest valid time that corresponds to a step (apply on discrete or unary resources and reservoirs).

An instance of `IloSolver::SolverErrorException` is thrown if any of the following conditions occurs.

- if `capacity` is a strictly negative integer
- if `capacity` is a constrained integer variable with a strictly negative minimal value

The set of alternative resources can contain a continuous reservoir if `extent` is `IlcNever`, `IlcAlways`, `IlcAfterStart` or `IlcAfterEnd`. If the time extent is `IlcNever`, the activity does not provide any capacity. If the time extent is `IlcAlways`, the capacity is provided at any time. If the time extent is `IlcAfterStart` or `IlcAfterEnd`, the capacity is not provided before the start of the activity, is totally provided after its end and linearly provided between its start and its end. For any other time extent, an instance of `IloSolver::SolverErrorException` is thrown.

```
public void removeDisjunctiveBreakType(IlcIntSet setOfTypes)
```

This member function removes the set of types `setOfTypes` from the set of disjunctive break types of the invoking activity. If a break type belongs to the set of disjunctive break types of a breakable activity, the activity must be completely processed either before or after that type of break.

This member function is available only outside the search. In particular, it cannot be called from within a goal.

```
public void removeDisjunctiveBreakType(IlcInt type)
```

This member function removes the type `type` from the set of disjunctive break types of the invoking activity. If a break type belongs to the set of disjunctive break types of a breakable activity, the activity must be completely processed either before or after that type of break.

This member function is available only outside the search. In particular, it cannot be called from within a goal.

```
public void removeEndBreakOverlapType(IlcInt type)
```

For each activity, a set of break types is given that defines which break types can possibly be overlapped by the end of the activity. This member function removes `type` from this set of break types on the invoking activity.

This member function is available only outside the search. In particular, it cannot be called from within a goal.

```
public void removeEndBreakOverlapType(IlcIntSet typeSet)
```

For each activity, a set of break types is given that defines which break types can possibly be overlapped by the end of the activity. This member function removes all the types of `typeSet` from this set of break types on the invoking activity.

This member function is available only outside the search. In particular, it cannot be called from within a goal.

```
public void removeIgnoredBreakType(IlcIntSet setOfTypes)
```

This member function removes the set of types `setOfTypes` from the set of ignored break types of the invoking activity.

```
public void removeIgnoredBreakType(IlcInt type)
```

This member function removes the type `type` from the set of ignored break types of the invoking activity.

```
public void removeIgnoredShiftType(IlcIntSet setOfTypes)
```

This member function removes the set of types `setOfTypes` from the set of ignored shift types of the invoking activity.

```
public void removeIgnoredShiftType(IlcInt type)
```

This member function removes the type `type` from the set of ignored shift types of the invoking activity.

```
public void removeStartBreakOverlapType(IlcInt type)
```

For each activity, a set of break types is given that defines which break types can possibly be overlapped by the start of the activity. This member function removes `type` from this set of break types on the invoking activity.

This member function is available only outside the search. In particular, it cannot be called from within a goal.

```
public void removeStartBreakOverlapType(IlcIntSet typeSet)
```

For each activity, a set of break types is given that defines which break types can possibly be overlapped by the start of the activity. This member function removes all the types of `typeSet` from this set of break types on the invoking activity.

This member function is available only outside the search. In particular, it cannot be called from within a goal.

```
public IlcResourceConstraint requires(IlcStateResource resource, IlcAny state,
IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcResourceConstraint requires(IlcStateResource resource, IlcAnySetVar
states, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcResourceConstraint requires(IlcStateResource resource, IlcAnySet states,
IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcResourceConstraint requires(IlcStateResource resource, IlcAnyVar state,
IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
```

This member function states that the invoking activity requires the given resource in the given state or states. By default, the activity requires `resource` from the beginning to the end of its execution. However, the optional argument `extent` is available to represent cases in which the activity requires the resource over a different time extent, as explained in `IlcTimeExtent`.

The argument `outward` is important only when one of the timetables of resource has a time step greater than 1 (one). In that case, `outward` defines whether the occupancy of the resource by the activity should be rounded outward or inward towards the nearest valid time that corresponds to a step.

```
public IlcResourceConstraint requires(IlcCapResource resource, IlcInt capacity=1,
IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcAltResConstraint requires(IlcAltResSet set, IlcIntVar capacity,
```

```
IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcAltResConstraint requires(IlcAltResSet set, IlcInt capacity=1,
IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcResourceConstraint requires(IlcCapResource resource, IlcIntVar capacity,
IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
```

This member function states that the invoking activity requires the given `capacity` of the given `resource`. For example, an activity might require the presence of a worker on a shift. By default, the `activity` requires the `resource` from the beginning to the end of its execution. However, the optional argument `extent` is available to represent cases in which the `activity` requires the `resource` over a different time extent, as explained in `IlcTimeExtent`.

The argument `outward` is important only when one of the timetables of resource has a time step greater than 1 (one). In that case, `outward` defines whether the occupancy of the resource by the activity should be rounded outward or inward towards the nearest valid time that corresponds to a step.

When the given resource is an instance of `IlcDiscreteEnergy`, it means that the given capacity is required for each unit of time in the given time extent.

An instance of `IloSolver::SolverErrorException` is thrown when any of the following conditions occurs:

1. if `capacity` is a strictly negative integer
2. if `capacity` is a constrained integer variable with a strictly negative minimal value

The member function must not be called if `resource` is a continuous reservoir. However, the set of alternative resources can contain a continuous reservoir if the `IlcTimeExtent` extent is `IlcNever`, `IlcAlways`, `IlcAfterStart` or `IlcAfterEnd`. If the time extent is `IlcNever`, the activity does not require any capacity. If the time extent is `IlcAlways`, the capacity is required at any time. If the time extent is `IlcAfterStart` or `IlcAfterEnd`, the capacity is not required before the start of the activity, is totally required after its end and linearly required between its start and its end. For any other time extent, an instance of `IloSolver::SolverErrorException` is thrown.

Example

In the following example, `activity` requires 2 units of the discrete resource `resource`. As 0 is considered a possible duration for the activity, nothing occurs when the maximal capacity of the resource is set to 1 over the interval [0 10). When the earliest end time of the activity becomes 5, Scheduler Engine automatically deduces (by constraint propagation) that the activity cannot start before 5. Indeed, if the activity starts before 5 (say, at t with t < 5), then the activity requires the resource at least from t to 5. However, this situation is impossible since the resource is not available in sufficient quantity. Hence, the activity cannot start before 5. Similarly, when the minimal duration of the activity is set to 1, the earliest start time of the activity automatically becomes 10.

```
/// Must be during search (e.g., inside a goal) ///
IloSolver solver = getSolver();
IlcScheduler schedule(solver, 0, 24);
IlcDiscreteResource resource(schedule, 3);
IlcIntVar processingTime(solver,0, 24);
IlcActivity activity(schedule, processingTime);
solver.add(activity.requires(resource, 2));
solver.out() << activity << endl;
resource.setCapacityMax(0, 10, 1);
solver.out() << activity << endl;
activity.setEndMin(5);
solver.out() << activity << endl;
activity.setProcessingTimeMin(1);
solver.out() << activity << endl;
```

The output of that program looks like the following.

```
[0..24 -- 0..24 --> 0..24]
[0..24 -- 0..24 --> 0..24]
[5..24 -- 0..19 --> 5..24]
[10..23 -- 1..14 --> 11..24]
```

```
public IlcResourceConstraint requiresNot(IlcStateResource resource, IlcAny state,
IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcResourceConstraint requiresNot(IlcStateResource resource, IlcAnySetVar
states, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcResourceConstraint requiresNot(IlcStateResource resource, IlcAnySet
states, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
public IlcResourceConstraint requiresNot(IlcStateResource resource, IlcAnyVar
state, IlcTimeExtent extent=IlcFromStartToEnd, IlcBool outward=IlcTrue)
```

This member function states that the invoking activity requires the given `resource` in any state that does not belong to the given `state` or set of `states`. The state or states may change during execution, but must never belong to the given `state` or set of `states`. By default, the activity requires the `resource` from the beginning to the end of its execution. However, the optional argument `extent` is available to represent cases in which the activity requires the resource over a different time extent, as explained in `IlcTimeExtent`.

The argument `outward` is important only when one of the timetables of `resource` has a time step greater than 1 (one). In that case, `outward` defines whether the occupancy of the `resource` by the activity should be rounded outward or inward towards the nearest valid time that corresponds to a step.

```
public void setDuration(IlcInt duration)
```

This member function sets the duration of the invoking activity to `duration`.

Detecting Inconsistencies

When changing the domain of a start time variable, an end time variable, a duration variable, or a processing time variable, Scheduler Engine will detect an inconsistency if a changed domain directly conflicts with temporal constraints and with the resource constraints of activities with fixed (instantiated) start and end times and processing time.

Scheduler Engine *may or may not* detect the inconsistency if the changed domain conflicts with the temporal and resource constraints associated with *unscheduled* activities, that is, activities for which the start and end times and/or the processing time are not instantiated yet.

```
public void setDurationMax(IlcInt durationMax)
```

This member function states that the duration of the invoking activity can be at most `durationMax`. An inconsistency may be detected as documented in IlcActivity::setDuration.

```
public void setDurationMaxNormalBreaks(IlcInt duration)
```

This member function states that the invoking activity must be completely processed either before or after any break whose duration is strictly greater than `duration`.

Increasing this maximal duration has no effect in search.

```
public void setDurationMin(IlcInt durationMin)
```

This member function states that the duration of the invoking activity must be at least `durationMin`. An inconsistency may be detected as documented in IlcActivity::setDuration.

```
public void setDurationMinNormalBreaks(IlcInt duration)
```

This member function states that the invoking activity must be completely processed either before or after any break whose duration is strictly lower than `duration`.

By default, the value of this minimal duration for a breakable activity is 1 so that only the breaks with null duration are considered as disjunctive.

Decreasing this minimal duration has no effect in search.

```
public void setEndBreakOverlap(IlcInt overlap)
```

This member function sets `overlap` as the value of the end break overlap variable.

```
public void setEndBreakOverlapMax(IlcInt overlapMax)
```

This member function sets `overlapMax` as the new maximal value of the end break overlap variable.

```
public void setEndBreakOverlapMin(IlcInt overlapMin)
```

This member function sets `overlapMin` as the new minimal value of the end break overlap variable.

```
public void setEndBreakOverlapVariable(IlcIntVar overlapVariable)
```

This member function sets `overlapVariable` as the end break overlap variable of the invoking activity.

For an activity that finishes in a possibly overlapped break:

- if the activity starts before this break, the value of the end break overlap variable is defined as the duration between the start of the break and the completion time of the activity;
- if the activity starts in this break (that is, the activity is processed completely inside the break), the only constraint is that the sum of the start break overlap variable and the end break overlap variable be equal to the processing time of the activity.

The value is defined as 0 for an activity that does not finish in a possibly overlapped break. By default, the end break overlap variable of an activity with no possibly overlapped break is considered to be bound to 0.

This member function is available only outside the search. In particular, it cannot be called from within a goal.

```
public void setEndMax(IlcInt endMax)
```

This member function states that the invoking activity must not end after `endMax`. An inconsistency may be detected as documented in IlcActivity::setDuration.

```
public void setEndMin(IlcInt endMin)
```

This member function states that the invoking activity must not end before `endMin`. An inconsistency may be detected as documented in IlcActivity::setDuration.

```
public void setEndTime(IlcInt endTime)
```

This member function sets the end time of the invoking activity to `endTime`. An inconsistency may be detected as documented in IlcActivity::setDuration.

```
public void setExecutionDurationMin(IlcInt)
```

A breakable activity executes during a set of disjoint temporal intervals. These execution intervals are separated by intervals that correspond to the breaks that suspend the activity.

This member function states that the duration of the temporal intervals during which the invoking breakable activity executes must all be greater or equal to the value provided. Note that this value must be a strictly positive integer. By default, breakable activities are created with a minimal duration for execution intervals of 1.

In search, trying to decrease the current minimal duration for execution intervals has no effect.

```
public void setExternalValue(IlcInt)
```

This member function sets the value of the external variable of the invoking activity.

```
public void setExternalVar(IlcIntVar)
```

This member function sets the external variable of the invoking activity. Before entering the search, this member function will override any previous specification of external variable. This function may be called only one time during the search if no external variable has been specified. Any attempt to call this member function during the search in a situation where some external variable has already been specified on the activity will raise an error.

```
public void setExternalVarMax(IlcInt)
```

This member function sets the maximal value of the external variable of the invoking activity.

```
public void setExternalVarMin(IlcInt)
```

This member function sets the minimal value of the external variable of the invoking activity.

```
public void setName(const char * name) const
```

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

```
public void setObject(IlcAny object) const
```

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

```
public void setProcessingTime(IlcInt processingTime)
```

This member function assigns `processingTime` as the processing time for the invoking activity. An inconsistency may be detected as documented in IlcActivity::setDuration.

```
public void setProcessingTimeMax(IlcInt processingTimeMax)
```

This member function states that the processing time of the invoking activity can be at most `processingTimeMax`. An inconsistency may be detected as documented in IlcActivity::setDuration.

```
public void setProcessingTimeMin(IlcInt processingTimeMin)
```

This member function states that the processing time of the invoking activity must be at least `processingTimeMin`. An inconsistency may be detected as documented in IlcActivity::setDuration.

```
public void setStartBreakOverlap(IlcInt overlap)
```

This member function sets `overlap` as the value of the start break overlap variable.

```
public void setStartBreakOverlapMax(IlcInt overlapMax)
```

This member function sets `overlapMax` as the new maximal value of the start break overlap variable.

```
public void setStartBreakOverlapMin(IlcInt overlapMin)
```

This member function sets `overlapMin` as the new minimal value of the start break overlap variable.

```
public void setStartBreakOverlapVariable(IlcIntVar overlapVariable)
```

This member function sets `overlapVar` as the start break overlap variable of the invoking activity.

For an activity that starts in a possibly overlapped break:

- if the activity finishes after this break, the value of the start break overlap variable is defined as the duration between the start of the activity and the end time of the break;
- if the activity finishes in this break (that is, the activity is completely processed inside the break), the only constraint is that the sum of the start break overlap variable and the end break overlap variable be equal to the processing time of the activity.

The value is defined as 0 for an activity that does not start in a possibly overlapped break. By default, the start break overlap variable of an activity with no possibly overlapped break is considered to be bound to 0.

This member function is available only outside the search. In particular, it cannot be called from within a goal.

```
public void setStartMax(IlcInt startMax)
```

This member function states that the invoking activity must not start after `startMax`. An inconsistency may be detected as documented in IlcActivity::setDuration.

```
public void setStartMin(IlcInt startMin)
```

This member function states that the invoking activity must not start before `startMin`. An inconsistency may be detected as documented in IlcActivity::setDuration.

```
public void setStartTime(IlcInt startTime)
```

This member function sets the start time of the invoking activity to `startTime`. An inconsistency may be detected as documented by in IlcActivity::setDuration.

```
public void setSuccessor(IlcActivity ct)
```

This member function states that the invoking activity has the activity `ct` as successor on the precedence graph of the schedule. That is, this member function adds an edge on the precedence graph.

This member function should be used only when a precedence graph constraint has been created on the schedule.

```
public void setTransitionType(IlcInt value)
```

The transition type of an activity is an integer intended to define transition time and cost from an indexed classification of activities. It is used by transition tables (instances of the class `IlcTransitionTable`).

This member function sets the transition type of the invoking activity to `value`.

```
public IlcTimeBoundConstraint startsAfter(IlcInt time)
public IlcTimeBoundConstraint startsAfter(IlcIntVar time)
```

This member function states that the invoking activity must start after or at `time`. More formally, `act.startsAfter(time)` means `start(act) >= time`.

```
public IlcPrecedenceConstraint startsAfterEnd(IlcActivity act, IlcInt delay=0)
public IlcPrecedenceConstraint startsAfterEnd(IlcActivity act, IlcIntVar)
```

This member function states that the invoking activity starts after the end of `act`. (In other words, `act` precedes the invoking activity.) In addition, at least the given `delay` must elapse between the end of `act` and the beginning of the invoking activity.

The member function can be invoked with a negative `delay`, which means that the invoking activity can start before the end of `act`, but the difference between the end time of `act` and the start time of the invoking activity cannot exceed `-delay`.

More formally, `act1.startsAfterEnd(act, delay)` means `start(act1) >= end(act) + delay`.

```
public IlcPrecedenceConstraint startsAfterStart(IlcActivity act, IlcInt delay=0)
public IlcPrecedenceConstraint startsAfterStart(IlcActivity act, IlcIntVar)
```

This member function states that the invoking activity starts after the beginning of `act`. In addition, at least the given `delay` must elapse between the beginning of `act` and the beginning of the invoking activity.

The member function can be invoked with a negative `delay`, which means that the invoking activity can start before the beginning of `act`, but the difference between the start time of `act` and the start time of the invoking activity cannot exceed `-delay`.

More formally, `act1.startsAfterStart(act, delay)` means `start(act1) >= start(act) + delay`.

```
public IlcTimeBoundConstraint startsAt(IlcInt time)
public IlcTimeBoundConstraint startsAt(IlcIntVar time)
```

This member function states that the invoking activity must start at `time`. More formally, `act.startsAt(time)` means `start(act) == time`.

```
public IlcPrecedenceConstraint startsAtEnd(IlcActivity act, IlcInt delay=0)
public IlcPrecedenceConstraint startsAtEnd(IlcActivity act, IlcIntVar)
```

This member function states that exactly the given delay must elapse between the end of `act` and the beginning of the invoking activity.

More formally, `act1.startsAtEnd(act, delay)` means `start(act1) == end(act) + delay`.

```
public IlcPrecedenceConstraint startsAtStart(IlcActivity act, IlcInt delay=0)
public IlcPrecedenceConstraint startsAtStart(IlcActivity act, IlcIntVar)
```

This member function states that exactly the given delay must elapse between the beginning of `act` and the beginning of the invoking activity.

More formally, `act1.startsAtStart(act, delay)` means `start(act1) == start(act) + delay`.

```
public IlcTimeBoundConstraint startsBefore(IlcInt time)
public IlcTimeBoundConstraint startsBefore(IlcIntVar time)
```

This member function states that the invoking activity must start before or at `time`. More formally, `act.startsBefore(time)` means `start(act) <= time`.

```
public void unsetSuccessor(IlcActivity ct)
```

This member function removes a successor relation that had previously been added on the graph with `IlcActivity::setSuccessor`.

This member function should be used only before entering the search and only when a precedence graph constraint has been created on the schedule.

```
public IlcBool useEfficiency() const
```

This member function returns `IloTrue` if the processing time of the invoking activity is computed using the efficiency function of resource calendars. Otherwise, it returns `IloFalse`.

Class IlcActivityDeltaIterator

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>

An instance of this class traverses a delta set of activities, for example, the new direct predecessors of a given activity. This iterator is only available during the search.

See Also: IlcActivity, IlcActivityIteratorFilter, IlcSchedule

Constructor and Destructor Summary	
public	`IlcActivityDeltaIterator(IlcActivity act, IlcActivityIteratorFilter filter)`

Method Summary	
public IlcBool	`ok() const`
public IlcActivity	`operator*() const`
public IlcActivityDeltaIterator &	`operator++()`

Constructors and Destructors

`public` **`IlcActivityDeltaIterator`**`(IlcActivity act, IlcActivityIteratorFilter filter)`

This constructor creates a delta iterator to traverse the elements of the subset of activities specified by `filter` whose status has changed with respect to `act`.

The possible filters are `IlcDirectPredecessors`, `IlcDirectSuccessors`, `IlcPredecessors`, and `IlcSuccessors`.

The possible statuses of an activity with respect to `act` are: unranked, direct predecessor, direct successor, indirect predecessor, indirect successor.

Thus, with the filters `IlcDirectSuccessors` or `IlcDirectPredecessors`, the delta iterator traverses the set of new direct successors or new direct predecessors of `act`.

With the filters `IlcSuccessors` or `IlcPredecessors`, the delta iterator traverses the union of the set of new direct successors and the set of new indirect successors of `act` or the union of the set of new direct predecessors and the set of new indirect predecessors of `act`. Note that this delta set is a superset of the set of new successors or new predecessors of `act`.

The delta sets of activities are emptied once all the demons attached to the precedence graph events of an activity have been executed. Thus, any attempt to traverse a delta set of activities outside the execution of such a demon may lead to unexpected behavior.

This constructor can be used only if a schedule precedence graph is associated with the schedule of activity `act`. If the schedule is not associated with a precedence graph, this constructor raises an error.

Methods

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if the current position of the iterator is a valid one. It returns `IlcFalse` if all the activities have been scanned by the iterator.

```
public IlcActivity operator*() const
```

This operator returns the current instance of `IlcActivity`, the one to which the invoking iterator points. If the iterator is set past the end position, then this operator returns an empty handle.

```
public IlcActivityDeltaIterator & operator++()
```

This left-increment operator shifts the current position of the iterator to the next delta instance of `IlcActivity`.

Class IlcActivityIterator

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>

An instance of this class traverses the set of activities.

For more information, see Precedence Graph Constraints.

See Also: IlcActivity, IlcActivityIteratorFilter, IlcSchedule

Constructor and Destructor Summary	
public	`IlcActivityIterator(const IlcSchedule schedule)`
public	`IlcActivityIterator(IlcActivity, IlcActivityIteratorFilter filter)`

Method Summary	
public IlcBool	`ok() const`
public IlcActivity	`operator*() const`
public IlcActivityIterator &	`operator++()`

Constructors and Destructors

`public` **`IlcActivityIterator`**`(const IlcSchedule schedule)`

This constructor creates an iterator to traverse all the activities of `schedule` during the search. If used before entering the search, this iterator will traverse an empty list of activities.

`public` **`IlcActivityIterator`**`(IlcActivity, IlcActivityIteratorFilter filter)`

When a *schedule precedence graph* is associated with the schedule of an activity, this constructor creates an iterator to traverse the subset of activities specified by the filter `filter`. If the schedule is not associated with a precedence graph, this constructor raises an error.

Before entering the search, only the filter `IlcSuccessors` is permitted. It allows the definition of an iterator that traverses the subset of activities `act0` for which the successor relation `(act,act0)` has been added with the member function `act.setSuccessor(act0)`. Any attempt to use another filter before entering the search will raise an error. In search, all the filters are allowed.

For example, the following loop, during the search, displays the set of activities that are direct successors of a given activity `act` in the precedence graph of the schedule:

```
for (IlcActivityIterator ite(act, IlcDirectSuccessors);
     ite.ok();
     ++ite) {
  solver.out() << *ite << endl;
}
```

Methods

```
public IlcBool ok() const
```

This member function returns `IlcTrue` if the current position of the iterator is a valid one. It returns `IlcFalse` if all the activities have been scanned by the iterator.

```
public IlcActivity operator*() const
```

This operator returns the current instance of `IlcActivity`, the one to which the invoking iterator points. If the iterator is set past the end position, then this operator returns an empty handle.

```
public IlcActivityIterator & operator++()
```

This left-increment operator shifts the current position of the iterator to the next instance of `IlcActivity`.

Class IlcAltRCDemon

Definition file: ilsched/altresh.h
Include file: <ilsched/ilsched.h>

An instance of `IlcAltRCDemon` represents a demon that is associated with all the possible resources on an alternative resource constraint. To attach the demon on the instance of `IlcAltResConstraint`, use the member function `IlcAltResConstraint::whenRange`. The demon is trigerred each time a change in the ranges of a possible alternative happens: that is, a change in the start, end, duration, processing time, or the capacity range. An instance of this class can be created with the macro `ILCALTRCDEMON`.

See Also: ILCALTRCDEMON, IlcAltResConstraint

Constructor Summary	
public	IlcAltRCDemon()
public	IlcAltRCDemon(IlcAltRCDemonI * impl)

Method Summary	
public IlcAltRCDemonI *	getImpl() const
public void	operator=(const IlcAltRCDemon & h)

Constructors

`public` **`IlcAltRCDemon`**`()`

This constructor creates an instance which is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

`public` **`IlcAltRCDemon`**`(IlcAltRCDemonI * impl)`

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

Methods

`public IlcAltRCDemonI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public void` **`operator=`**`(const IlcAltRCDemon & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument. After execution of this operator, the invoking object and the provided argument point to the same implementation object.

Class IlcAltResConstraint

Definition file: ilsched/altresh.h
Include file: <ilsched/ilsched.h>

An instance of the class `IlcAltResConstraint` expresses the fact that an activity uses exactly one resource from a set of alternatives that make up an instance of `IlcAltResSet`.

Instances of the class `IlcAltResConstraint` are created by member functions of the class `IlcActivity`:

- `IlcActivity::consumes`
- `IlcActivity::produces`
- `IlcActivity::provides`
- `IlcActivity::requires`

An Exclusive Constructive Disjunctive Constraint

We call an instance of `IlcAltResConstraint` a *disjunctive* constraint because it defines a boolean disjunction so that only one alternative among a set of alternative resources is used to execute the activity.

We say that it is a *constructive* disjunction because the Scheduler Engine makes deductions on the basis of the constraint as it goes along to propagate further and more efficiently than modeling with the logical or Solver metaconstraints. Scheduler Engine maintains the temporal and demand bounds of the activity for each possible resource.

We also say that an instance of `IlcAltResConstraint` is *exclusive* because one and only one resource among the alternatives will be used by an activity.

Index Variable

A Solver variable indexing the resources of the set of alternative resources is provided. It is intended to add constraints between the selected resource. Let *N* be the number of resources in the set, an instance of class `IlcAltResSet`. The index variable is created with a domain *<0, N>*.

The values in *<0, N-1>* index the resources. The value *N* is an escape value in the case of no resource selection. This last value is used in the metaconstraint protocol to express the fact that the instance of `IlcAltResConstraint` is false. When the alternative resource constraint is posted (and therefore must be true), this escape value is automatically removed from the domain of the index variable.

Selection and Solution Search

An alternative resource constraint behaves very much like an ordinary resource constraint as far as an activity is concerned. The alternative resource constraint manages the set of possible resources for the activity, treating them so that they behave as one resource from the point of view of the activity. The domains of the start, end, and duration variables of the activity contain any value that is allowed for at least one of the possible resources of the activity.

When one resource is selected, the associated resource constraint is added. Once a resource has been selected, the other resources are no longer possible. If, for some of them, resource constraints had already been created, the capacity of these resource constraints is reduced to zero.

To make the search for a solution efficient, it is a good idea to select a resource for an activity for which a set of alternative resources is defined when you schedule it.

Metaconstraint Protocol

`IlcAltResConstraint` is fully compatible with the metaconstraint protocol of Solver. Stating that an instance of `IlcAltResConstraint` is false is conventionally the same as stating that the set of resources is not used by

the activity.

Resource Constraints

The resource constraint of a *possible* resource is automatically created:

- when the resource is *closed* so that the minimal capacity of any timetable of the resource will be propagated;
- when the resource is an instance of the class `IlcUnaryResource` for which a *light precedence graph constraint* has been added to the solver so that its ranking properties will be respected and the transition time taken into account.
- When the calendar constraint is posted on the resource.

If need be, you can create the resource constraint of a possible resource. To do so, you use the member function `IlcAltResConstraint::getResourceConstraint`.

For more information, see Metaconstraints.

See Also: IlcActivity, IlcAltResConstraintIterator, IlcAltResSet, IlcAssign, IlcAssignAlternative, IlcPossibleAltResIterator, IlcResource, IlcTimeExtent

Constructor Summary	
`public`	`IlcAltResConstraint()`
`public`	`IlcAltResConstraint(IlcAltResConstraintI * impl)`

Method Summary	
`public IlcActivity`	`getActivity() const`
`public IlcAltResSet`	`getAltResSet() const`
`public IlcInt`	`getCapacity() const`
`public IlcInt`	`getCapacityMax(const IlcResource resource) const`
`public IlcInt`	`getCapacityMin(const IlcResource resource) const`
`public IlcIntVar`	`getCapacityVariable() const`
`public IlcInt`	`getDurationMax(const IlcResource resource) const`
`public IlcInt`	`getDurationMin(const IlcResource resource) const`
`public IlcInt`	`getEndMax(const IlcResource resource) const`
`public IlcInt`	`getEndMin(const IlcResource resource) const`
`public IlcAltResConstraintI *`	`getImpl() const`
`public IlcIntVar`	`getIndexVariable() const`
`public IlcInt`	`getNumberOfPossible() const`
`public IlcInt`	`getProcessingTimeMax(const IlcResource resource) const`
`public IlcInt`	`getProcessingTimeMin(const IlcResource resource) const`
`public IlcResourceConstraint`	`getResourceConstraint(const IlcResource resource) const`
`public IlcCapResource`	`getSelected() const`
`public IlcInt`	`getStartMax(const IlcResource resource) const`
`public IlcInt`	`getStartMin(const IlcResource resource) const`
`public IlcTimeExtent`	`getTimeExtent() const`

public IlcBool	isInwardConstraint() const
public IlcBool	isPossible(const IlcResource resource) const
public IlcBool	isProvidingConstraint() const
public IlcBool	isResourceSelected() const
public IlcBool	isSelected(const IlcResource resource) const
public IlcBool	isVariableResourceConstraint() const
public void	operator=(const IlcAltResConstraint & h)
public void	setCapacityMax(IlcResource resource, IlcInt max)
public void	setCapacityMin(IlcResource resource, IlcInt min)
public void	setDurationMax(IlcResource resource, IlcInt max)
public void	setDurationMin(IlcResource resource, IlcInt min)
public void	setEndMax(IlcResource resource, IlcInt max)
public void	setEndMin(IlcResource resource, IlcInt min)
public void	setNotPossible(IlcResource resource)
public void	setProcessingTimeMax(IlcResource resource, IlcInt max)
public void	setProcessingTimeMin(IlcResource resource, IlcInt min)
public void	setSelected(IlcResource resource)
public void	setStartMax(IlcResource resource, IlcInt max)
public void	setStartMin(IlcResource resource, IlcInt min)
public void	whenRange(const IlcAltRCDemon g) const

Constructors

public **IlcAltResConstraint**()

This constructor creates an instance which is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

public **IlcAltResConstraint**(IlcAltResConstraintI * impl)

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

Methods

public IlcActivity **getActivity**() const

This member function returns the activity associated with the invoking constraint.

public IlcAltResSet **getAltResSet**() const

This member function returns the instance of `IlcAltResSet` associated with the invoking constraint.

public IlcInt **getCapacity**() const

This member function returns the quantity required or provided by the activity associated with the invoking constraint.

```
public IlcInt getCapacityMax(const IlcResource resource) const
```

This member function returns the maximal capacity of the invoking alternative resource constraint assuming `resource` is the selected resource.

```
public IlcInt getCapacityMin(const IlcResource resource) const
```

This member function returns the minimal capacity of the invoking alternative resource constraint assuming `resource` is the selected resource.

```
public IlcIntVar getCapacityVariable() const
```

This member function returns the constrained variable representing the quantity required or provided by the activity associated with the invoking constraint.

```
public IlcInt getDurationMax(const IlcResource resource) const
```

This member function returns the longest duration of the activity of the invoking alternative resource constraint assuming `resource` is the selected resource.

```
public IlcInt getDurationMin(const IlcResource resource) const
```

This member function returns the shortest duration of the activity of the invoking alternative resource constraint assuming `resource` is the selected resource.

```
public IlcInt getEndMax(const IlcResource resource) const
```

This member function returns the latest end time of the activity of the invoking alternative resource constraint assuming `resource` is the selected resource.

```
public IlcInt getEndMin(const IlcResource resource) const
```

This member function returns the earliest end time of the activity of the invoking alternative resource constraint assuming `resource` is the selected resource.

```
public IlcAltResConstraintI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcIntVar getIndexVariable() const
```

This member function returns the variable that contains the indices of all the possible resources for the invoking constraint. The indices correspond to the indices of the resources in the instance of the class `IlcAltResSet` for which the invoking constraint was defined.

```
public IlcInt getNumberOfPossible() const
```

This member function returns the number of possible resources that could be selected for the activity associated with the invoking constraint. It returns 1 (one) if a resource has already been selected.

```
public IlcInt getProcessingTimeMax(const IlcResource resource) const
```

This member function returns the longest processing time for the activity of the invoking alternative resource constraint assuming `resource` is the selected resource.

```
public IlcInt getProcessingTimeMin(const IlcResource resource) const
```

This member function returns the shortest processing time for the activity of the invoking alternative resource constraint assuming `resource` is the selected resource.

```
public IlcResourceConstraint getResourceConstraint(const IlcResource resource)
const
```

This member function returns the resource constraint associated with `resource` in the invoking alternative resource constraint. If `resource` is possible but not yet selected, the minimal capacity of the resource constraint is set to 0 (zero). If resource is not possible, the maximal capacity of the resource constraint is set to 0 (zero), implying that the resource constraint does not use `resource` at all. The resource constraint is automatically added to the solver if `resource` is possible.

```
public IlcCapResource getSelected() const
```

This member function returns the resource that has been selected for the activity associated with the invoking constraint. An instance of `IloSolver::SolverErrorException` is thrown if no resource has been selected.

```
public IlcInt getStartMax(const IlcResource resource) const
```

This member function returns the latest start time of the activity of the invoking alternative resource constraint assuming `resource` is the selected resource.

```
public IlcInt getStartMin(const IlcResource resource) const
```

This member function returns the earliest start time of the activity of the invoking alternative resource constraint assuming `resource` is the selected resource.

```
public IlcTimeExtent getTimeExtent() const
```

This member function returns the time extent of the activity associated with the invoking constraint.

```
public IlcBool isInwardConstraint() const
```

This member function returns `IlcTrue` if and only if the occupancy of the selected resource by the invoking constraint is to be rounded inward toward the nearest valid time that corresponds to a time step. This rounding is important only when one of the timetables of the selected resource has a time step greater than 1 (one).

```
public IlcBool isPossible(const IlcResource resource) const
```

This member function returns `IlcTrue` if `resource` can be selected for the activity associated with the invoking constraint. Otherwise, it returns `IlcFalse`.

```
public IlcBool isProvidingConstraint() const
```

This member function returns `IlcTrue` if and only if the invoking constraint was constructed by one of these member functions:`IlcActivity::provides` or `IlcActivity::produces`.

```
public IlcBool isResourceSelected() const
```

This member function returns `IlcTrue` if the activity associated with the invoking constraint has selected a resource as the only possible one. Otherwise, it returns `IlcFalse`.

```
public IlcBool isSelected(const IlcResource resource) const
```

This member function returns `IlcTrue` if `resource` has been selected by the invoking constraint. Otherwise, it returns `IlcFalse`.

```
public IlcBool isVariableResourceConstraint() const
```

This member function returns `IlcTrue` if and only if the activity associated with the invoking constraint has a variable representing the required or provided quantity.

```
public void operator=(const IlcAltResConstraint & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument. After execution of this operator, the invoking object and the provided argument point to the same implementation object.

```
public void setCapacityMax(IlcResource resource, IlcInt max)
```

This member function states that `max` is the maximal capacity of the invoking alternative resource constraint if `resource` is the selected resource.

```
public void setCapacityMin(IlcResource resource, IlcInt min)
```

This member function states that `min` is the minimal capacity of the invoking alternative resource constraint if `resource` is the selected resource.

```
public void setDurationMax(IlcResource resource, IlcInt max)
```

This member function states that `max` is the longest duration of the activity of the invoking alternative resource constraint if `resource` is the selected resource.

```
public void setDurationMin(IlcResource resource, IlcInt min)
```

This member function states that `min` is the shortest duration of the activity of the invoking alternative resource constraint if `resource` is the selected resource.

```
public void setEndMax(IlcResource resource, IlcInt max)
```

This member function states that `max` is the latest end time of the activity of the invoking alternative resource constraint if `resource` is the selected resource.

```
public void setEndMin(IlcResource resource, IlcInt min)
```

This member function states that `min` is the earliest end time of the activity of the invoking alternative resource constraint if `resource` is the selected resource.

```
public void setNotPossible(IlcResource resource)
```

This member function states that it is not possible for `resource` to be selected.

```
public void setProcessingTimeMax(IlcResource resource, IlcInt max)
```

This member function states that `max` is the longest processing time for the activity of the invoking alternative resource constraint if `resource` is the selected resource.

```
public void setProcessingTimeMin(IlcResource resource, IlcInt min)
```

This member function states that `min` is the shortest processing time for the activity of the invoking alternative resource constraint if `resource` is the selected resource.

```
public void setSelected(IlcResource resource)
```

This member function states that `resource` has been selected for the activity associated with the invoking constraint. More precisely, it sets (if possible) the value of the index variable to:

- the index of `resource` if `resource` is in the alternative resource set (so that the invoking constraint becomes or stays *true*).
- *N*, where *N* is the number of resources in this set, otherwise (so that the invoking constraint becomes or stays *false*).

```
public void setStartMax(IlcResource resource, IlcInt max)
```

This member function states that `max` is the latest start time of the activity of the invoking alternative resource constraint if `resource` is the selected resource.

```
public void setStartMin(IlcResource resource, IlcInt min)
```

This member function states that `min` is the earliest start time of the activity of the invoking alternative resource constraint if `resource` is the selected resource.

```
public void whenRange(const IlcAltRCDemon g) const
```

This member function associates the demon `g` with the *By Resource Domain* propagation event of an alternative resource constraint. The demon is an instance of the class `IlcAltRCDemon` designed for this purpose. Whenever the propagation of the start, end, duration, processing time, and capacity range with respect to a possible resource occurs, the demon `g` is executed. Please refer to the Macro `ILCALTRCDEMON` for full information and an example.

Class IlcAltResConstraintIterator

Definition file: ilsched/altresh.h
Include file: <ilsched/ilsched.h>

An instance of the class `IlcAltResConstraintIterator` is an iterator that traverses, during the search, the constraints defined between an instance of `IlcActivity` and an instance of `IlcAltResSet`. If used before entering the search, this iterator will traverse an empty list of alternative resource constraints.

If you iterate over all the constraints that require or provide the resources in that set, you are actually traversing all the resources that *might* be required or provided. If you want to traverse *only* the constraints that surely provide or require the resources in the set of an instance of `IlcAltResSet`, then you have to use the member function `IlcConstraint::isPosted` (documented in the *IBM ILOG Solver Reference Manual*) to distinguish among them. See the example in `IlcResourceConstraintIterator` for a program that makes this distinction correctly.

See Also: IlcAltResConstraint, IlcAltResSet, IlcTimeExtent

Constructor and Destructor Summary	
public	IlcAltResConstraintIterator(IlcAltResSet set)
public	IlcAltResConstraintIterator(IlcActivity act)
public	IlcAltResConstraintIterator(IlcAltResSet set, IlcTimeExtent extent)
public	IlcAltResConstraintIterator(IlcActivity act, IlcTimeExtent extent)

Method Summary	
public IlcBool	ok() const
public IlcAltResConstraint	operator*() const
public IlcAltResConstraintIterator &	operator++()

Constructors and Destructors

public **IlcAltResConstraintIterator**(IlcAltResSet set)

This constructor creates a new instance of `IlcAltResConstraintIterator` that traverses the constraints that require or provide the capacity resources in `set`.

public **IlcAltResConstraintIterator**(IlcActivity act)

This constructor creates a new instance of `IlcAltResConstraintIterator` that traverses the alternative resource constraints defined on `act`.

public **IlcAltResConstraintIterator**(IlcAltResSet set, IlcTimeExtent extent)

This constructor creates a new instance of `IlcAltResConstraintIterator` that traverses the constraints that require or provide the capacity resources in `set` that have the time extent indicated by `extent`.

```
public IlcAltResConstraintIterator(IlcActivity act, IlcTimeExtent extent)
```

This constructor creates a new instance of `IlcAltResConstraintIterator` that traverses the alternative resource constraints defined on `act` that have the time extent indicated by `extent`.

Methods

```
public IlcBool ok() const
```

This member function returns `IlcTrue` if there is a current constraint and the invoking iterator points to it. Otherwise, it returns `IlcFalse`.

```
public IlcAltResConstraint operator*() const
```

This operator returns the current constraint, the one to which the invoking iterator points. If the iterator is set past the end position, then this operator returns an empty handle.

```
public IlcAltResConstraintIterator & operator++()
```

This operator shifts the iterator to point to the next element.

Class IlcAltResSet

Definition file: ilsched/altresh.h
Include file: <ilsched/ilsched.h>

An instance of the class `IlcAltResSet` represents a special *set* of resources to which activities can be assigned. When an activity requires an instance of this class, the activity requires exactly one of the resources represented in that set. For convenience, an instance of `IlcAltResSet` behaves like a standard resource. To that end, the class includes member functions that reproduce the properties and behavior of a standard resource.

The set of resources (the alternatives) must consist of *capacity* resources (that is, instances of `IlcDiscreteEnergy`, `IlcDiscreteResource`, `IlcUnaryResource`, or `IlcReservoir`).

Each resource in the set must be unique.

In case the elements of the set of alternative resources do not conform to these rules, an error will be raised when the set is constructed.

If you change the resources in the set during its use, you will produce unpredictable behavior; in fact, such changes may lead to erroneous behavior.

Redundant Resources

It is possible to consider the set of resources as a resource whose theoretical capacity is the sum of the capacities of the resources of the set. Let us call this resource the redundant resource of the set.

An alternative resource constraint on the set is a resource constraint on this redundant resource. Therefore, posting the timetable constraint or the edge finder constraint on the redundant resource may lead to more efficient propagation. Global and resource constraints on the redundant resource have exactly the same purpose as a redundant constraint in a Solver model.

Notice that if the resources of the set are instances of `IlcUnaryResource` and if they have strictly identical properties (the activities have the same processing time, do not require any other resources, and there is no transition time), the redundant resource gives exactly the same solution as the alternative resource. In such a case, it is more efficient to represent the set of unary resources with a single discrete resource rather than with a alternative resource set.

Printing or Displaying Sets of Resources

The printed representation of an instance of the class `IlcAltResSet` consists of its name, followed by the list of resources. If there are more than 10 resources, only the number of resources is displayed. For example:

`[r1, r2, r3]` represents a set of resources containing the three resources `r1`, `r2`, and `r3`.

`[size = 14]` represents a set of resources containing 14 resources.

If the Solver trace is active and the resource is not named, the string "`IlcAltResSet`" is followed by the address of the implementation object. The address will be enclosed in parentheses.

See Also: IlcAltResConstraint, IlcAltResSetIterator, IlcAssignAlternative, IlcDiscreteResource, IlcReservoir, IlcResource, IlcSchedule, IlcUnaryResource

Constructor Summary	
`public`	`IlcAltResSet()`
`public`	`IlcAltResSet(IlcAltResSetI * impl)`

public	IlcAltResSet(IlcSchedule schedule, IlcInt size)
public	IlcAltResSet(IlcSchedule schedule, IlcResourceArray array)

Method Summary	
public void	close()
public IlcBool	contains(const IlcResource resource) const
public IlcAltResSetI *	getImpl() const
public IlcInt	getIndex(const IlcResource resource) const
public const char *	getName() const
public IlcAny	getObject() const
public IlcCapResource	getRedundantResource() const
public IlcSchedule	getSchedule() const
public IlcInt	getSize() const
public IloSolver	getSolver() const
public IloSolverI *	getSolverI() const
public IlcBool	hasRedundantResource() const
public IlcBool	isClosed() const
public void	makeRedundantResource(IlcBool timetable=IlcFalse)
public IlcBool	operator!=(const IlcAltResSet & resource) const
public void	operator=(const IlcAltResSet & h)
public IlcBool	operator==(const IlcAltResSet & resource) const
public IlcResource &	operator[](IlcInt index)
public void	setFilled()
public void	setName(const char * name) const
public void	setObject(IlcAny object) const

Constructors

public **IlcAltResSet**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcAltResSet**(IlcAltResSetI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IlcAltResSet**(IlcSchedule schedule, IlcInt size)

This constructor creates a new instance of `IlcAltResSet` and adds it to those managed by `schedule`. The size of the set (that is, the number of alternative capacity resources) is indicated by `size`. Before you use the new instance, you must initialize its set with instances of capacity resources.

public **IlcAltResSet**(IlcSchedule schedule, IlcResourceArray array)

This constructor creates a new instance of `IlcAltResSet` and adds it to those managed by `schedule`. The constructor initializes the set with the resources of the resource array passed as argument.

Methods

```
public void close()
```

This member function closes the invoking object; that is, it closes *all* the resources present in the invoking instance of `IlcAltResSet`.

```
public IlcBool contains(const IlcResource resource) const
```

This member function returns `IlcTrue` if `resource` belongs to the invoking instance of `IlcAltResSet`. Otherwise, it returns `IlcFalse`.

```
public IlcAltResSetI * getImpl() const
```

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcInt getIndex(const IlcResource resource) const
```

This member function returns the index of `resource` in the invoking alternative capacity resource set.

```
public const char * getName() const
```

This member function returns the name of the invoking object.

```
public IlcAny getObject() const
```

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

```
public IlcCapResource getRedundantResource() const
```

This member function returns the redundant resource of the invoking set, if it has been previously created.

```
public IlcSchedule getSchedule() const
```

This member function returns the schedule to which the invoking instance of `IlcAltResSet` belongs. Each alternative resource in the set belongs to the same schedule, an instance of `IlcSchedule`.

```
public IlcInt getSize() const
```

This member function returns the number of alternative capacity resources in the invoking instance of `IlcAltResSet`.

```
public IloSolver getSolver() const
```

This member function returns an instance of `IloSolver` associated with the invoking object.

```
public IloSolverI * getSolverI() const
```

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

```
public IlcBool hasRedundantResource() const
```

This member function returns `IlcTrue` if the redundant resource of the invoking set has been created. Otherwise, it returns `IlcFalse`.

```
public IlcBool isClosed() const
```

This member function returns `IlcTrue` if *all* of the capacity resources in the set of the invoking instance of `IlcAltResSet` have been closed. The member function returns `IlcFalse` otherwise.

```
public void makeRedundantResource(IlcBool timetable=IlcFalse)
```

This member function builds the redundant resource associated with the invoking alternative resources set. When the redundant resource is created, the resource constraints on the resources of the set and the alternative resource constraints of the set are automatically added to the redundant resource. If the argument `timetable` is set to `IlcTrue`, the timetable constraint is added to the redundant resource.

```
public IlcBool operator!=(const IlcAltResSet & resource) const
```

This operator returns `IlcTrue` if the invoking instance and the argument resource are not identical; that is, they are handles of different implementation objects. Otherwise, it returns `IlcFalse`.

```
public void operator=(const IlcAltResSet & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public IlcBool operator==(const IlcAltResSet & resource) const
```

This operator returns `IlcTrue` if the invoking instance and the argument resource are identical; that is, they are both handles with the same implementation object. Otherwise, it returns `IlcFalse`.

```
public IlcResource & operator[](IlcInt index)
```

This operator returns a reference to the resource located at `index` in the invoking alternative resource set.

```
public void setFilled()
```

The call to this member function confirms that all alternative resources were added to the invoking alternative resource set. After calling this member function, no resources can be added to the set.

The use of this member function is mandatory only for alternative resource sets constructed on durable schedules.

```
public void setName(const char * name) const
```

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

```
public void setObject(IlcAny object) const
```

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

Class IlcAltResSetIterator

Definition file: ilsched/altresh.h
Include file: <ilsched/ilsched.h>

An instance of the class `IlcAltResSetIterator` is an iterator that traverses a *set of sets* of alternative resources. For example, you might need an iterator to traverse all the instances of `IlcAltResSet` managed by a particular schedule; or you might be interested in traversing all the instances of `IlcAltResSet` to which a given resource belongs.

See Also: IlcAltResSet, IlcResource, IlcSchedule

Constructor and Destructor Summary	
public	`IlcAltResSetIterator(IlcResource resource)`
public	`IlcAltResSetIterator(const IlcSchedule schedule)`

Method Summary	
public IlcBool	`ok() const`
public IlcAltResSet	`operator*() const`
public IlcAltResSetIterator &	`operator++()`

Constructors and Destructors

`public` **`IlcAltResSetIterator`**`(IlcResource resource)`

This constructor creates a new instance of `IlcAltResSetIterator` that traverses all the alternative resource sets that have `resource` as an element.

`public` **`IlcAltResSetIterator`**`(const IlcSchedule schedule)`

This constructor creates a new instance of `IlcAltResSetIterator` that traverses all the instances of `IlcAltResSet` that are managed by `schedule`.

Methods

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if there is a current instance of `IlcAltResSet` and the invoking iterator points to it. Otherwise, it returns `IlcFalse`.

`public IlcAltResSet` **`operator*`**`() const`

This operator returns the current instance of `IlcAltResSet`, the one to which the invoking iterator points. If the iterator is set past the end position, then this operator returns an empty handle.

```
public IlcAltResSetIterator & operator++()
```

This operator shifts the iterator to the next instance of `IlcAltResSet`.

Class IlcAnyTimetable

Definition file: ilsched/timetabh.h
Include file: <ilsched/ilsched.h>

An instance of the handle class `IlcAnyTimetable` represents a *timetable* in which the values are pointers to arbitrary objects (rather than integers). In the Scheduler Engine, these timetables are used to manage the *states* of resources.

A timetable is defined over an *interval*, `[timeMin timeMax)`, where `timeMin` is the origin of the timetable and `timeMax` is its horizon. In addition to the origin and horizon, you may optionally indicate the *period* of the timetable. The period must be a positive integer, and furthermore, the size of the interval (that is,`timeMax - timeMin`) must be an integer multiple of the period. If a period is specified, then the values managed by the timetable can change only at times indicated by `timeMin + i * period`.

Two types of propagation events can be triggered when this kind of timetable is modified. An event of type `domainInterval` indicates that there are some times at which some modification of the domain occurred. An event of type `valueInterval` indicates that there are some times at which the value became bound. In order to perform propagation, member functions allow you to associate demons with each type of event.

The information stored into a timetable is reversible. In particular, when modifiers are called, the state before their call will be saved by Solver.

For more information, see Timetable.

See Also: IlcAnyTimetableCursor, IlcAnyTimetableIterator, IlcIntTimetable, IlcStateResource

Constructor and Destructor Summary	
public	`IlcAnyTimetable(IlcSchedule, const IlcAnySet states, IlcInt timeMin, IlcInt timeMax, IlcInt period=1)`

Method Summary	
public IlcInt	`getDomainTimeMax() const`
public IlcInt	`getDomainTimeMin() const`
public IlcManager	`getManager() const`
public const char *	`getName() const`
public IlcAny	`getObject() const`
public IlcInt	`getPeriod() const`
public IlcInt	`getTimeMax() const`
public IlcInt	`getTimeMin() const`
public IlcAny	`getValue(IlcInt time) const`
public IlcInt	`getValueTimeMax() const`
public IlcInt	`getValueTimeMin() const`
public IlcBool	`isAlwaysPossible(IlcAny state, IlcInt timeMin, IlcInt timeMax) const`
public IlcBool	`isAlwaysRequired(IlcAny state, IlcInt timeMin, IlcInt timeMax) const`
public IlcBool	`isBound(IlcInt time) const`

public IlcBool	isEverPossible(IlcAny state, IlcInt timeMin, IlcInt timeMax) const
public IlcBool	isEverRequired(IlcAny state, IlcInt timeMin, IlcInt timeMax) const
public IlcBool	isPossible(IlcAny state, IlcInt time) const
public IlcBool	isRequired(IlcAny state, IlcInt time) const
public void	removePossibleStates(IlcInt timeMin, IlcInt timeMax, const IlcAnySet states)
public void	removeState(IlcInt timeMin, IlcInt timeMax, IlcAny state)
public void	setName(const char * name)
public void	setObject(IlcAny object)
public void	setPossibleStates(IlcInt timeMin, IlcInt timeMax, const IlcAnySet states)
public void	setState(IlcInt timeMin, IlcInt timeMax, IlcAny state)
public void	whenDomainInterval(const IlcDemon c) const
public void	whenValueInterval(const IlcDemon c) const

Constructors and Destructors

public **IlcAnyTimetable**(IlcSchedule, const IlcAnySet states, IlcInt timeMin, IlcInt timeMax, IlcInt period=1)

This constructor creates a timetable to manage a set of states over the interval [timeMin, timeMax). The constructor adds that timetable to those managed by schedule. The timetable starts at timeMin and extends to timeMax, divided into equal periods of size period.

An instance of IloSolver::SolverErrorException is thrown if any of the following conditions occur:

- timeMax - timeMin is not strictly positive;
- period is not strictly positive;
- timeMax - timeMin is not an integer multiple of period.

Methods

public IlcInt **getDomainTimeMax**() const

When it is called during the execution of a demon associated with the invoking timetable by the member function IlcAnyTimetable::whenDomainInterval, this member function returns the time domainTimeMax, that is, the maximum of the interval [domainTimeMin, domainTimeMax) containing all the times at which the domain has changed. The return value of this member function is not meaningful outside the execution of such a demon.

public IlcInt **getDomainTimeMin**() const

When it is called during the execution of a demon associated with the invoking timetable by the member function IlcAnyTimetable::whenDomainInterval, this member function returns the time domainTimeMin, that is, the minimum of the interval [domainTimeMin, domainTimeMax) containing all the times at which the domain has changed. The return value of this member function is not meaningful outside the execution of such a demon.

```
public IlcManager getManager() const
```

This member function returns the manager (a handle) of the invoking timetable.

```
public const char * getName() const
```

This member function returns a pointer to the name of the invoking timetable. If the invoking timetable has no name, then this function returns the empty string.

```
public IlcAny getObject() const
```

An instance of the class `IlcAnyTimetable` may be a data member of another "external" object. In such a case, it may be useful to find the external object from the instance of `IlcAnyTimetable`. The member function `getObject` accesses such an inverse link.

In fact, this member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns the null pointer otherwise.

```
public IlcInt getPeriod() const
```

This member function returns the size of the periods of the invoking timetable. The meaning of this size is that the timetable may change only at times representing the beginning of periods, that is, times of the form `(getTimeMin() + i * getPeriod())`.

```
public IlcInt getTimeMax() const
```

This member function returns the time horizon of the invoking timetable.

```
public IlcInt getTimeMin() const
```

This member function returns the time origin of the invoking timetable.

```
public IlcAny getValue(IlcInt time) const
```

This member function returns the value at `time` of the invoking timetable. An instance of `IloSolver::SolverErrorException` is thrown if the timetable is not bound at `time`.

```
public IlcInt getValueTimeMax() const
```

When it is called during the execution of a demon associated with a timetable by the member function `IlcAnyTimetable::whenValueInterval`, this member function returns the time `valueTimeMax`, that is, the maximum of the interval `[valueTimeMin, valueTimeMax)` containing all the times at which the value has been bound. The return value of this member function is not meaningful outside the execution of such a demon.

```
public IlcInt getValueTimeMin() const
```

When it is called during the execution of a demon associated with a timetable by the member function `IlcAnyTimetable::whenValueInterval`, this member function returns the time `valueTimeMin`, that is, the minimum of the interval `[valueTimeMin, valueTimeMax)` containing all the times at which the value has been bound. The return value of this member function is not meaningful outside the execution of such a demon.

```
public IlcBool isAlwaysPossible(IlcAny state, IlcInt timeMin, IlcInt timeMax) const
```

This member function returns `IlcTrue` if it is possible for the invoking timetable to be in the given `state` *over the entire* interval `[timeMin, timeMax)`. Otherwise, it returns `IlcFalse`.

```
public IlcBool isAlwaysRequired(IlcAny state, IlcInt timeMin, IlcInt timeMax) const
```

This member function returns `IlcTrue` if it is certain that the invoking timetable is in the given `state` *over the entire* interval `[timeMin, timeMax)`. Otherwise, it returns `IlcFalse`.

```
public IlcBool isBound(IlcInt time) const
```

This member function returns `IlcTrue` if the invoking timetable is bound to a value at `time`. Otherwise, it returns `IlcFalse`.

```
public IlcBool isEverPossible(IlcAny state, IlcInt timeMin, IlcInt timeMax) const
```

This member function returns `IlcTrue` if it is possible that the invoking timetable is in the given `state` *at some point* in the interval `[timeMin, timeMax)`. Otherwise, it returns `IlcFalse`.

```
public IlcBool isEverRequired(IlcAny state, IlcInt timeMin, IlcInt timeMax) const
```

This member function returns `IlcTrue` if it is certain that the invoking timetable is in the given `state` *at some point* in the interval `[timeMin, timeMax)`. Otherwise, it returns `IlcFalse`.

```
public IlcBool isPossible(IlcAny state, IlcInt time) const
```

This member function returns `IlcTrue` if it is possible that the invoking timetable is in the given `state` at the given `time`. Otherwise, it returns `IlcFalse`.

```
public IlcBool isRequired(IlcAny state, IlcInt time) const
```

This member function returns `IlcTrue` if it is certain that the invoking timetable is in the given `state` the time indicated by `time`. Otherwise, it returns `IlcFalse`.

```
public void removePossibleStates(IlcInt timeMin, IlcInt timeMax, const IlcAnySet
states)
```

This member function states that the invoking timetable must not be in any of the given `states` at any time in the interval `[timeMin, timeMax)`. The set of "impossible" states must be provided as an instance of `IlcAnySet`. That class is documented in the *Solver Reference Manual*.

```
public void removeState(IlcInt timeMin, IlcInt timeMax, IlcAny state)
```

This member function states that the invoking timetable must not be in the given `state` at any time in the interval `[timeMin timeMax)`.

```
public void setName(const char * name)
```

This member function sets the name of the invoking timetable to a copy of the given `name`.

```
public void setObject(IlcAny object)
```

It is possible to associate an object (other than the implementation object) with a handle by means of this member function. If the invoking handle has no associated object, then `object` becomes the associated object. If the invoking handle already has an associated object, an instance of `IloSolver::SolverErrorException` is thrown. The argument `object` must not be the null pointer; otherwise, an instance of `IloSolver::SolverErrorException` is thrown.

```
public void setPossibleStates(IlcInt timeMin, IlcInt timeMax, const IlcAnySet
states)
```

This member function states that the invoking timetable must be in any of the given `states` at all times in the interval `[timeMin, timeMax)`. The set of possible states must be provided as an instance of `IlcAnySet`. That class is documented in the *Solver Reference Manual*.

```
public void setState(IlcInt timeMin, IlcInt timeMax, IlcAny state)
```

This member function states that the invoking timetable must be in the given `state` at all times in the interval `[timeMin, timeMax)`.

```
public void whenDomainInterval(const IlcDemon c) const
```

This member function associates the demon `c` with the `domainInterval` propagation event of the invoking timetable. Whenever a `domainInterval` propagation event occurs, the demon is executed.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever a domain propagation event or a series of such events occurs, the constraint is propagated.

A call to the demon signifies that there are *some* times at which the domain has changed. (The domain changes whenever a state is removed from the possible set of states.) The interval `[domainTimeMin, domainTimeMax)` is the least interval containing all these times.

```
public void whenValueInterval(const IlcDemon c) const
```

This member function associates the demon `c` with the `valueInterval` propagation event of the invoking timetable. Whenever a `valueInterval` propagation event occurs, the demon is executed.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever a `valueInterval` propagation event occurs, the constraint is propagated.

A call to the demon signifies that there are *some* times at which the value has been bound. (A value is bound when its set of possible states is reduced to one state.) The interval `[valueTimeMin, valueTimeMax)` is the least interval containing all these times.

Class IlcAnyTimetableCursor

Definition file: ilsched/timetabh.h
Include file: <ilsched/ilsched.h>

Objects of the class `IlcAnyTimetableCursor` allow you to inspect the contents of timetables of type `IlcAnyTimetable`. A *region* of a timetable is a subinterval `[timeMin, timeMax)` of the interval where it is defined such that all the times in the region share the same information and any two adjacent regions store different information. *Cursors* are intended to iterate forward or backward over the regions of a timetable.

Note

The structure of a timetable cannot be changed while a cursor is being used to inspect the timetable. Therefore, functions that change the structure of the timetable should not be called while the cursor is being used; for example, `IlcAnyTimetable::removePossibleStates`.

See Also: IlcAnyTimetable

Constructor and Destructor Summary	
public	IlcAnyTimetableCursor(const IlcAnyTimetable table, IlcInt time)

Method Summary	
public IlcInt	getTimeMax() const
public IlcInt	getTimeMin() const
public IlcAny	getValue() const
public IlcBool	isBound() const
public IlcBool	isPossible(IlcAny state) const
public IlcBool	ok() const
public void	operator++()
public void	operator--()

Constructors and Destructors

`public` **`IlcAnyTimetableCursor`**`(const IlcAnyTimetable table, IlcInt time)`

This constructor creates a cursor to inspect the information stored in the timetable `table`. This cursor lets you iterate forward or backward over the regions composing the timetable. The cursor initially indicates the region containing `time`.

Methods

`public IlcInt` **`getTimeMax`**`() const`

This member function returns the time ending the region currently indicated by the cursor.

`public IlcInt` **`getTimeMin`**`() const`

This member function returns the time beginning the region currently indicated by the cursor.

```
public IlcAny getValue() const
```

This member function returns the value of the region indicated by the invoking timetable cursor. An instance of `IloSolver::SolverErrorException` is thrown if the timetable is not bound at the cursor position.

```
public IlcBool isBound() const
```

This member function returns `IlcTrue` if the set of possible states indicated by the invoking timetable cursor has been bound; that is, the set of possible states has been reduced to a single state. Otherwise, it returns `IlcFalse`.

```
public IlcBool isPossible(IlcAny state) const
```

This member function returns `IlcTrue` if the given state is a member of the set of possible states corresponding to the region currently indicated by the invoking timetable cursor. Otherwise, it returns `IlcFalse`.

```
public IlcBool ok() const
```

This member function returns `IlcFalse` if the cursor does not currently indicate a region included in the interval of the timetable. Otherwise, it returns `IlcTrue`. Any attempt to use the cursor after `ok()` returns `IlcFalse` could lead to undefined behavior.

```
public void operator++()
```

This operator moves the cursor to the region adjacent "on the right" to the current region (forward iteration).

```
public void operator--()
```

This operator moves the cursor to the region adjacent "on the left" to the current region (backward iteration).

Class IlcAnyTimetableIterator

Definition file: ilsched/state.h
Include file: <ilsched/ilsched.h>

An instance of this class traverses the set of timetables associated with a state resource.

See Also: IlcAnyTimetable, IlcSchedule

Constructor Summary	
public	IlcAnyTimetableIterator(IlcStateResource res)

Method Summary	
public IlcAnyTimetable	operator*()
public IlcAnyTimetableIterator &	operator++()

Constructors

`public` **`IlcAnyTimetableIterator`**`(IlcStateResource res)`

This constructor creates an iterator to traverse all the timetables of a state resource.

Methods

`public IlcAnyTimetable` **`operator*`**`()`

This operator accesses the instance of `IlcAnyTimetable` located at the current position of the iterator. If the iterator is set past the end position, this operator returns an empty handle.

`public IlcAnyTimetableIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator.

Class IlcCalendar

Definition file: ilsched/sbecprop.h

An instance of `IloCalendar` allows modeling complex behavior for activity variables (start, end, duration and processing time) within a resource. This behaviour could represent, for example, holidays, resource performances, and so forth. For more information see Calendars.

A calendar object is defined by three components:

- A set of breaks which basically can suspend the execution of the concerned activity (see Calendars)
- A set of shifts which can, for example, forbid some start dates (see Shift Object Semantic)
- A granular step-wise function to define the efficiency of the resource along the schedule (see Functional and Integral Constraints on Resources)

Constructor and Destructor Summary	
public	`IlcCalendar()`
public	`IlcCalendar(IlcCalendarI * impl)`
public	`IlcCalendar(const IlcSchedule schedule)`

Method Summary	
public void	`addShiftObject(IlcShiftObject shiftObj)`
public IlcIntervalList	`getBreakList() const`
public IlcGranularFunction	`getEfficiency() const`
public IlcCalendarI *	`getImpl() const`
public const char *	`getName() const`
public IlcAny	`getObject() const`
public IlcSchedule	`getSchedule() const`
public IloSolver	`getSolver() const`
public IloSolverI *	`getSolverI() const`
public void	`operator=(const IlcCalendar & h)`
public void	`removeShiftObject(IlcShiftObject shiftObj)`
public void	`setBreakList(IlcIntervalList list)`
public void	`setEfficiency(IlcGranularFunction f)`
public void	`setName(const char * name) const`
public void	`setObject(IlcAny object) const`

Inner Class
IlcCalendar::ShiftObjectIterator

Constructors and Destructors

`public` **`IlcCalendar`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

```
public IlcCalendar(IlcCalendarI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

```
public IlcCalendar(const IlcSchedule schedule)
```

This constructor creates a new instance of `IlcCalendar`. Its name is set to `name`

Methods

```
public void addShiftObject(IlcShiftObject shiftObj)
```

This member function adds the shift object `shiftObj` to the invoking calendar.

```
public IlcIntervalList getBreakList() const
```

This member function returns the list of breaks attached to the invoking calendar, if such a list exists.

```
public IlcGranularFunction getEfficiency() const
```

This member function returns the efficiency function attached to the invoking calendar, if such a function exists.

```
public IlcCalendarI * getImpl() const
```

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

```
public const char * getName() const
```

This member function returns the name of the invoking object.

```
public IlcAny getObject() const
```

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

```
public IlcSchedule getSchedule() const
```

This member function returns the schedule, an instance of `IlcSchedule`, to which the invoking calendar belongs.

```
public IloSolver getSolver() const
```

This member function returns an instance of `IloSolver` associated with the invoking object.

```
public IloSolverI * getSolverI() const
```

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

```
public void operator=(const IlcCalendar & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public void removeShiftObject(IlcShiftObject shiftObj)
```

This member function removes the shift object `shiftObj` from the invoking calendar.

```
public void setBreakList(IlcIntervalList list)
```

This member function sets `list` as the new break list of the calendar.

```
public void setEfficiency(IlcGranularFunction f)
```

This member function sets `f` as the granular step-wise function that models the efficiency of the calendar within the schedule.

```
public void setName(const char * name) const
```

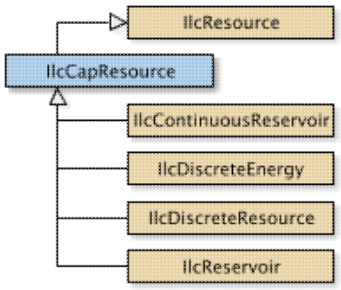
This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

```
public void setObject(IlcAny object) const
```

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

Class IlcCapResource

Definition file: ilsched/capacity.h
Include file: <ilsched/ilsched.h>



`IlcCapResource` is the root class for *capacity* resources, that is, resources that are defined to have a limited integer capacity over time. In the Scheduler Engine, there are the following classes of capacity resources:

- `IlcDiscreteEnergy`
- `IlcDiscreteResource`
- `IlcReservoir`
- `IlcContinuousReservoir`
- `IlcUnaryResource`

Closing a Resource

The inherited member function `IlcResource::close` specifies that all the activities requiring or providing the invoking resource are known; that is, they have been linked to the resource. This information allows additional constraint propagation to take place, for example, the propagation of minimal capacity constraints. Propagating minimal capacity constraints is particularly useful in the case of resource allocation problems for which some minimal amount of provided capacity must be reached. With such information, indeed, the system can eliminate situations in which minimal capacity amounts cannot be reached using only the activities already defined.

An instance of `IloSolver::SolverErrorException` is thrown if you attempt to add a new requiring or providing activity to a capacity resource that has been closed.

Initial Occupation

The timetable of the resource represents the occupation of the resource by activities. Scheduler Engine offers a way to set up the initial occupation without having to declare the corresponding activities.

That facility is intended to help in solving a problem by iteratively adding a new set of activities to schedule or in improving a solution by rescheduling a subset of the activities.

For discrete resources, that is for instances of `IlcDiscreteResource`, `IlcDiscreteEnergy`, and `IlcReservoir`, the initial occupation is defined with an instance of the `IlcIntToIntStepFunction` class. For continuous reservoirs, that is instances of `IlcContinuousReservoir`, the initial occupation is defined with an instance of the `IlcIntToFloatSegmentFunction` class. The value of the stepwise or piecewise linear function at an integer point in time is considered as the sum of the requirements of fictitious or previously constructed activities.

For instances of `IlcDiscreteResource` and `IlcDiscreteEnergy`, the initial level is zero outside the definition domain of the function.

For instances of `IlcReservoir` and `IlcContinuousReservoir`, if the definition domain of the function intersects the temporal interval of the time table, the initial level is given by the function on its definition domain, and by zero elsewhere. That is, Scheduler Engine ignores the initial level of the reservoir. If the definition domain of the function does not intersect the temporal interval of the time table, the initial level of the reservoir is used as usual.

Note

The content of the function is stored by copy in the resource. Any modifications to the function after it is copied will not be seen by the timetables of the resource.

For more information, see Timetable, Disjunctive Constraint, and in the *IBM ILOG Solver Reference Manual*, `IlcIntToIntStepFunction`.

See Also: IlcIntervalList, IlcResource, IlcResourceConstraint, IlcResourceIterator, IlcSchedule, IlcRCTextureFactory, IlcIntToFloatSegmentFunction, IlcTextureCriticalityCalculator, IlcAltResSet

Constructor Summary	
public	IlcCapResource()
public	IlcCapResource(IlcCapResourceI * impl)

Method Summary	
public IlcCapResourceI *	getImpl() const
public IlcResourceTexture	getMaxTextureMeasurement() const
public IlcResourceTexture	getMinTextureMeasurement() const
public IlcIntTimetable	getTimetable() const
public IlcIntTimetable	getTimetable(IlcInt time) const
public IlcBool	hasInitialOccupation() const
public IlcBool	hasMaxTextureMeasurement() const
public IlcBool	hasMinTextureMeasurement() const
public void	incrDurableRequirement(IlcIntToIntStepFunction func)
public void	incrDurableRequirement(IlcInt t1, IlcInt t2, IlcInt cap, IlcBool inward, IlcInt db)
public IlcBool	isRedundantResource() const
public IlcConstraint	makeBalanceConstraint()
public IlcResourceTexture	makeMaxTextureMeasurement(IlcRCTextureFactory=0, IlcTextureCriticalityCalculator=0)
public IlcResourceTexture	makeMinTextureMeasurement(IlcRCTextureFactory=0, IlcTextureCriticalityCalculator=0)
public IlcConstraint	makeTimetableConstraint(IlcInt timeMin, IlcInt timeMax, IlcInt timeStep, IlcInt capacity)
public IlcConstraint	makeTimetableConstraint(IlcInt timeMin, IlcInt timeMax, IlcInt timeStep)
public IlcConstraint	makeTimetableConstraint(IlcInt timeStep=1)
public void	operator=(const IlcCapResource & h)
public void	setInitialOccupation(IlcIntToFloatSegmentFunction func)
public void	setInitialOccupation(IlcIntToIntStepFunction func)
public void	unsetInitialOccupation()

Inherited Methods from `IlcResource`
close, getCalendar, getDisjunctiveConstraint, getDurableSchedule, getImpl, getLastRankedFirstRC, getLastRankedLastRC, getLastSurelyContributingRankedFirstRC, getLastSurelyContributingRankedLastRC, getName, getObject,

getOldLastRankedFirstRC, getOldLastRankedLastRC, getPrecedenceGraphConstraint, getSchedule, getSolver, getSolverI, getTimetableConstraint, getTransitionTime, hasCalendar, hasDisjunctiveConstraint, hasLightPrecedenceGraphConstraint, hasPrecedenceGraphConstraint, hasPrecedenceInfo, hasRankInfo, hasTimetableConstraint, isCapacityResource, isClosed, isContinuousReservoir, isDiscreteEnergy, isDiscreteResource, isDurable, isReservoir, isStateResource, isTransitionTimeSuspended, isUnaryResource, makeFunctionalConstraint, makeIntegralConstraint, makeLightPrecedenceGraphConstraint, makePrecedenceGraphConstraint, operator!=, operator=, operator==, setCalendar, setName, setObject, setTransitionTimeObject, setTransitionTimeSuspended, whenContribution, whenDirectPredecessors, whenDirectSuccessors, whenNext, whenPossibleNext, whenPossiblePrevious, whenPredecessors, whenPrevious, whenRankedFirstRC, whenRankedLastRC, whenSuccessors

Constructors

```
public IlcCapResource()
```

This constructor creates an instance which is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

```
public IlcCapResource(IlcCapResourceI * impl)
```

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

Methods

```
public IlcCapResourceI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcResourceTexture getMaxTextureMeasurement() const
```

This member function returns the texture measurement associated with the maximum constraint of the invoking resource. If no texture measurement has been associated with the maximum constraint of the invoking resource, an empty handle is returned.

```
public IlcResourceTexture getMinTextureMeasurement() const
```

This member function returns the texture measurement associated with the minimum constraint of the invoking resource. If no texture measurement has been associated with the minimum constraint of the invoking resource, an empty handle is returned.

```
public IlcIntTimetable getTimetable() const
```

This member function returns the first timetable of the invoking resource (first in chronological order). An instance of `IloSolver::SolverErrorException` is thrown is no timetable exists for the invoking resource. The invoking resource must not be a continuous reservoir.

```
public IlcIntTimetable getTimetable(IlcInt time) const
```

This member function returns the timetable that includes `time`. An instance of `IloSolver::SolverErrorException` is thrown if no timetable is defined at `time`. The invoking resource must not be a continuous reservoir.

```
public IlcBool hasInitialOccupation() const
```

This member function returns `IlcTrue` if an initial occupation has been set up on the invoking resource. Otherwise, it returns `IlcFalse`.

Initial Occupation

The timetable of the resource represents the occupation of the resource by activities. Scheduler Engine offers a way to set up the initial occupation without having to declare the corresponding activities.

That facility is intended to help in solving a problem by iteratively adding a new set of activities to schedule or in improving a solution by rescheduling a subset of the activities.

For discrete resources, that is instances of `IlcDiscreteResource`, `IlcDiscreteEnergy`, and `IlcReservoir`, the initial occupation is defined with an instance of the `IlcIntToIntStepFunction` class. For continuous reservoirs, that is instances of `IlcContinuousReservoir`, the initial occupation is defined with an instance of the `IlcIntToFloatSegmentFunction` class. The value of the stepwise or piecewise linear function at an integer point in time is considered as the sum of the requirements of fictitious or previously constructed activities.

For instances of `IlcDiscreteResource` and `IlcDiscreteEnergy`, the initial level is zero outside the definition domain of the function.

For instances of `IlcReservoir` and `IlcContinuousReservoir`, if the definition domain of the function intersects the temporal interval of the time table, the initial level is given by the function on its definition domain, and by zero elsewhere. That is, Scheduler Engine ignores the initial level of the reservoir. If the definition domain of the function does not intersect the temporal interval of the time table, the initial level of the reservoir is used as usual.

> **Note**
>
> The content of the function is stored by copy in the resource. Any modifications to the function after it is copied will not be seen by the timetables of the resource.

```
public IlcBool hasMaxTextureMeasurement() const
```

This member function returns `IlcTrue` if a texture measurement has been created on the maximum constraint of the invoking resource. Otherwise, it returns `IlcFalse`.

```
public IlcBool hasMinTextureMeasurement() const
```

This member function returns `IlcTrue` if a texture measurement has been created on the minimum constraint of the invoking resource. Otherwise, it returns `IlcFalse`.

```
public void incrDurableRequirement(IlcIntToIntStepFunction func)
```

The purpose of this function is to provide a non-reversible, non-monotonic edition of the requirement amount of a durable resource. Refer to Durability for complete information on that subject.

This function modifies the requirement amount that corresponds to an activity starting a `t1`, ending at `t2` and requiring the capacity of the argument `capacity`. This function is similar to having an activity on each step starting at `t1`, ending at `t2` and of value `capacity` of the argument `func`.

If the argument `capacity` is greater than 0, the effect of the function is to add `capacity` to the amount of requirement to the resource on the interval `[t1, t2)`; that is, to actually decrease the available capacity in the resource.

If the argument `capacity` is less than 0, the effect of the function is to remove `capacity` from the requirement amount of the resource on the interval `[t1, t2)`; that is, to increase the available capacity in the resource.

In the case of a discrete resource timetable of timestep different from one, an instance of `IloSolver::SolverErrorException` is thrown if the function steps do not fit the timestep of the resource. In the case of energy with a break timetable, the equivalent activities are considered as being not breakable.

The coherency of the requirement amount with respect to the resource capacity is under the responsibility of the user. For example, one should be cautious that the requirements that are undone do not exceed the requirement of activities committed on the resource when a search using the durable resource is launched.

For a multi-threaded durable resource, these functions are enclosed in a critical section. That is, these functions are MT-hot.

An instance of `IloSolver::SolverErrorException` is thrown if the schedule is not durable, if the durable schedule is not closed, or if the resource is in used by a computational manager.

```
public void incrDurableRequirement(IlcInt t1, IlcInt t2, IlcInt cap, IlcBool
inward, IlcInt db)
```

The purpose of this function is to provide a non-reversible, non-monotonic edition of the requirement amount of a durable resource. Refer to Durability for complete information on that subject.

This function modifies the requirement amount that corresponds to an activity starting a `t1`, ending at `t2` and requiring the capacity of the argument `cap`.

If the argument `cap` is greater than 0, the effect of the function is to add `cap` to the amount of requirement to the resource on the interval `[t1, t2)`; that is, to actually decrease the available capacity in the resource.

If the argument `cap` is less than 0, the effect of the function is to remove `cap` from the requirement amount of the resource on the interval `[t1, t2)`; that is, to increase the available capacity in the resource.

The optional argument `inward` is used by discrete resource timetable constraints of a time step greater than one as the rounding policy for the equivalent activity. The optional argument `db` is used by energy with break timetable constraints as the duration of the breaks for the equivalent breakable activity (that is, end time - start time - processing time).

The coherency of the requirement amount with respect to the resource capacity is under the responsibility of the user. For example, one should be cautious that the requirements that are undone do not exceed the requirement of activities committed on the resource when a search using the durable resource is launched.

For a multi-threaded durable resources, these functions are enclosed in a critical section. That is, these functions are MT-hot.

An instance of `IloSolver::SolverErrorException` is thrown if the tuple `[t1, t2, db)` is invalid for defining an activity. An instance of `IloSolver::SolverErrorException` is thrown if the schedule is not durable, if the durable schedule is not closed, or if the resource is in used by a computational manager.

```
public IlcBool isRedundantResource() const
```

This method returns `IlcTrue` if the invoking `IlcCapResource` object was created as a redundant resource, using the method `IlcAltResSet::makeRedundantResource`. It returns `IlcFalse` otherwise.

```
public IlcConstraint makeBalanceConstraint()
```

This member function creates a balance constraint on the invoking discrete resource or reservoir. This constraint allows a stronger propagation of the discrete resource capacity. See Balance Constraint for more information. That constraint must be posted in order to be taken into account.

```
public IlcResourceTexture makeMaxTextureMeasurement(IlcRCTextureFactory=0,
IlcTextureCriticalityCalculator=0)
```

This member function creates an instance of `IlcResourceTexture` on the maximum constraint of the invoking resource. By default, that is, if not otherwise specified, the `IlcRCTextureFactory` used is an instance of `IlcRCTextureProbabilisticFactoryI` and the `IlcTextureCriticalityCalculator` used is an instance of `IlcProbabilisticCriticalityCalculatorI`.

```
public IlcResourceTexture makeMinTextureMeasurement(IlcRCTextureFactory=0,
IlcTextureCriticalityCalculator=0)
```

This member function creates an instance of `IlcResourceTexture` on the minimum constraint of the invoking resource. By default, that is, if not otherwise specified, the `IlcRCTextureFactory` used is an instance of `IlcRCTextureProbabilisticFactoryI` and the `IlcTextureCriticalityCalculator` used is an instance of `IlcProbabilisticCriticalityCalculatorI`.

```
public IlcConstraint makeTimetableConstraint(IlcInt timeMin, IlcInt timeMax, IlcInt
timeStep, IlcInt capacity)
```

This member function creates and returns a timetable constraint for the invoking resource. This timetable constraint implies that the capacity of the resource is limited to `capacity` from `timeMin` to `timeMax` and allowed to change only at times `timeMin + i * timeStep`. If the invoking resource is an instance of `IlcDiscreteResource` or `IlcUnaryResource`, `capacity` represents the maximal theoretical capacity. If the invoking resource is an instance of `IlcDiscreteEnergy`, `capacity` represents the maximal available energy.

An instance of `IloSolver::SolverErrorException` is thrown if any of the following conditions occur:

- if `timeStep` is not strictly positive;
- if `timeMin` is not strictly less than `timeMax`;
- if `timeMax` minus `timeMin` is not a multiple of `timeStep`;
- if the new timetable overlaps another timetable that has already been created for the invoking resource.

```
public IlcConstraint makeTimetableConstraint(IlcInt timeMin, IlcInt timeMax, IlcInt
timeStep)
```

This member function creates and returns a timetable constraint for the invoking resource. This timetable constraint implies that the capacity of the resource is limited to the theoretical capacity of the resource from `timeMin` to `timeMax` and allowed to change only at times `timeMin + i * timeStep`.

An instance of `IloSolver::SolverErrorException` is thrown if any of the following conditions occur:

- if `timeStep` is not strictly positive;
- if `timeMin` is not strictly less than `timeMax`;

- if `timeMax` minus `timeMin` is not a multiple of `timeStep`;
- if the new timetable overlaps another timetable that has already been created for the invoking resource.

```
public IlcConstraint makeTimetableConstraint(IlcInt timeStep=1)
```

This member function creates and returns a timetable constraint for the invoking resource. This timetable constraint implies that the capacity of the resource is limited to the theoretical capacity of the resource from the time origin `timeMin` to the time horizon, and allowed to change only at times `timeMin + i * timeStep`.

An instance of `IloSolver::SolverErrorException` is thrown if any of the following conditions occur:

- if `timeStep` is not strictly positive;
- if the time horizon minus the time origin is not a multiple of `timeStep`;
- if the new timetable overlaps another timetable that has already been created for the invoking resource.

```
public void operator=(const IlcCapResource & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument. After execution of this operator, the invoking object and the provided argument point to the same implementation object.

```
public void setInitialOccupation(IlcIntToFloatSegmentFunction func)
```

This member function sets the argument `func` as the initial level of the timetables of the invoking capacity resource. The invoking resource must be a continuous reservoir. The argument `func` is copied. The initial occupation is considered at post time of the timetable constraint. That is, a call to `setInitialOccupation` after the adding of the timetable constraint in the solver and the entering of the solver in search mode has no effect.

Refer to `IlcCapResource::hasInitialOccupation` for more information on initial occupation.

```
public void setInitialOccupation(IlcIntToIntStepFunction func)
```

This member function sets the argument `func` as the initial level of the timetables of the invoking resource. The argument `func` is copied. The initial occupation is considered at post time of the timetable constraint. That is, a call to `setInitialOccupation` after the adding of the timetable constraint in the solver and the entering of the solver in search mode has no effect.

Refer to `IlcCapResource::hasInitialOccupation` for more information on initial occupation.

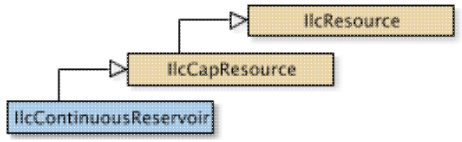
```
public void unsetInitialOccupation()
```

This member function unsets the initial level of the timetables of the invoking resource. That is, the resource no longer has an initial occupation.

Refer to `IlcCapResource::hasInitialOccupation` for more information on initial occupation.

Class IlcContinuousReservoir

Definition file: ilsched/contires.h
Include file: <ilsched/ilsched.h>



An instance of the class `IlcContinuousReservoir` represents a resource which activities can either fill or empty in a continuous process. For example, if an activity that starts at time *st* and ends at time *et* can fill the reservoir by a capacity *C*, the quantity put in the continuous reservoir by the activity at time *t* is the following.

- 0 if *t* < *st*;
- *(t-st)C/(et-st)* if *st* <= *t* < *et*;
- *C* if *et* <= *t*.

If the duration of the activity is null, the filling (or emptying) process is not continuous since the quantity *C* is instantaneously put in (or removed from) the reservoir at time *et* (which is equal to *st*).

Scheduler Engine ensures no attempt is made to further empty an already-empty continuous reservoir. Furthermore, if you define a maximal level of the continuous reservoir, then this maximal level will never be exceeded.

When the problem model represents an ongoing process, the continuous reservoir may already have some non-zero level present. To avoid this situation, simply pass an initial level to the constructor of `IlcContinuousReservoir`.

The capacity of a continuous reservoir can vary over time. You can define temporary maximal and minimal levels by using member functions of `IlcContinuousReservoir`.

Closing a Continuous Reservoir

As for the class `IlcReservoir`, the member function `IlcResource::close` is crucial for propagation affecting the class `IlcContinuousReservoir`. If `close` is not called, new activities filling or emptying the reservoir can still be added; thus preventing propagation. The continuous reservoir must be closed before propagation can take place.

Disjunctive Constraints

As with reservoirs, there are no disjunctive constraints with continuous reservoirs.

Printing or Displaying Continuous Reservoirs

The printed representation of an instance of the class `IlcContinuousReservoir` consists of its name, if it exists, and its theoretical capacity followed by its initial level. The two values are enclosed in brackets and separated by a dash (-).

For example:

`r1[100 - 10]` represents the continuous reservoir named `r1` which has a capacity equal to 100 and an initial level equal to 10.

If the Solver trace is active and the resource is not named, the string "`IlcContinuousReservoir`" is followed by the address of the implementation object. The address will be enclosed in parentheses.

If the theoretical capacity of the reservoir is equal to its maximal value (that is, `IlcIntMax/2`), the string "`Maximum Capacity`" is displayed instead of its numerical value.

See Also: IlcCapResource, IlcContinuousReservoirIterator, IlcResource, IlcResourceConstraint, IlcSchedule

Constructor Summary	
public	IlcContinuousReservoir()
public	IlcContinuousReservoir(IlcContinuousReservoirI * impl)
public	IlcContinuousReservoir(IlcSchedule schedule, IlcInt capacity=IlcMaxCapacityReservoir, IlcFloat initialLevel=0, IlcBool timetable=IlcTrue)

Method Summary	
public IlcContinuousReservoirI *	getImpl() const
public IlcFloat	getInitialLevel() const
public IlcFloat	getLevelMax(IlcInt time) const
public IlcFloat	getLevelMaxMax(IlcInt t1, IlcInt t2) const
public IlcFloat	getLevelMaxMin(IlcInt t1, IlcInt t2) const
public IlcFloat	getLevelMin(IlcInt time) const
public IlcFloat	getLevelMinMax(IlcInt t1, IlcInt t2) const
public IlcFloat	getLevelMinMin(IlcInt t1, IlcInt t2) const
public IlcFloatTimetable	getTimetable(IlcInt time) const
public IlcFloatTimetable	getTimetable() const
public IlcConstraint	makeTimetableConstraint(IlcInt timeMin, IlcInt timeMax, IlcFloat precision=ILC_CONTINUOUS_RESERVOIR_PRECISION)
public void	operator=(const IlcContinuousReservoir & h)
public void	setLevelMax(IlcInt timeMin, IlcInt timeMax, IlcFloat levelMax)
public void	setLevelMin(IlcInt timeMin, IlcInt timeMax, IlcFloat levelMin)

Inherited Methods from `IlcCapResource`
getImpl, getMaxTextureMeasurement, getMinTextureMeasurement, getTimetable, getTimetable, hasInitialOccupation, hasMaxTextureMeasurement, hasMinTextureMeasurement, incrDurableRequirement, incrDurableRequirement, isRedundantResource, makeBalanceConstraint, makeMaxTextureMeasurement, makeMinTextureMeasurement, makeTimetableConstraint, makeTimetableConstraint, makeTimetableConstraint, operator=, setInitialOccupation, setInitialOccupation, unsetInitialOccupation

Inherited Methods from `IlcResource`
close, getCalendar, getDisjunctiveConstraint, getDurableSchedule, getImpl, getLastRankedFirstRC, getLastRankedLastRC, getLastSurelyContributingRankedFirstRC, getLastSurelyContributingRankedLastRC, getName, getObject, getOldLastRankedFirstRC, getOldLastRankedLastRC, getPrecedenceGraphConstraint, getSchedule, getSolver, getSolverI, getTimetableConstraint, getTransitionTime, hasCalendar, hasDisjunctiveConstraint, hasLightPrecedenceGraphConstraint, hasPrecedenceGraphConstraint, hasPrecedenceInfo, hasRankInfo, hasTimetableConstraint, isCapacityResource, isClosed, isContinuousReservoir, isDiscreteEnergy, isDiscreteResource, isDurable, isReservoir, isStateResource, isTransitionTimeSuspended, isUnaryResource, makeFunctionalConstraint, makeIntegralConstraint, makeLightPrecedenceGraphConstraint,

makePrecedenceGraphConstraint, operator!=, operator=, operator==, setCalendar, setName, setObject, setTransitionTimeObject, setTransitionTimeSuspended, whenContribution, whenDirectPredecessors, whenDirectSuccessors, whenNext, whenPossibleNext, whenPossiblePrevious, whenPredecessors, whenPrevious, whenRankedFirstRC, whenRankedLastRC, whenSuccessors

Constructors

`public` **`IlcContinuousReservoir`**`()`

This constructor creates an instance which is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

`public` **`IlcContinuousReservoir`**`(IlcContinuousReservoirI * impl)`

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

`public` **`IlcContinuousReservoir`**`(IlcSchedule schedule, IlcInt capacity=IlcMaxCapacityReservoir, IlcFloat initialLevel=0, IlcBool timetable=IlcTrue)`

This constructor creates a new instance of `IlcContinuousReservoir` and adds it to the set of resources managed in the given `schedule`. The `capacity` expresses the capacity of the new continuous reservoir. The capacity may be consumed by certain activities and produced by others. The argument `initialLevel` defines an initial amount in the continuous reservoir at the time origin of the schedule. By default, the reservoir is assumed to be empty at the time origin; that is, the initial level is 0 (zero). The default value of `capacity` is `IlcIntMax/2`; that is the maximal theoretical capacity that is allowed. Any capacity greater than `IlcIntMax/2` will be treated as if it were equal to `IlcIntMax/2`.

If `timetable` is `IlcTrue`, then a timetable constraint is posted, defining the level of the reservoir to be between 0 (zero) with theoretical capacity over the interval `[timeMin, timeMax)`, where `timeMin` is the origin and `timeMax` is the horizon of the schedule. An instance of `IloSolver::SolverErrorException` is thrown if `capacity` is strictly negative.

Methods

`public IlcContinuousReservoirI * ` **`getImpl`**`() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public IlcFloat` **`getInitialLevel`**`() const`

This member function returns the initial level of the continuous reservoir; that is, the initial level that was passed to the continuous reservoir constructor.

`public IlcFloat` **`getLevelMax`**`(IlcInt time) const`

This member function returns the maximal level that is present at the given time. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking continuous reservoir do not cover the given time.

`public IlcFloat` **`getLevelMaxMax`**`(IlcInt t1, IlcInt t2) const`

This member function returns the maximal consumable level throughout the integer time points of the interval `[timeMin, timeMax)` (that is, the maximal value over the time points `timeMin, ... timeMax-1` of the maximal resource level). An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking continuous reservoir do not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public IlcFloat getLevelMaxMin(IlcInt t1, IlcInt t2) const
```

This member function returns the maximal consumable level, over the integer time points of the interval `[timeMin, timeMax)`, of the minimal resource level. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking continuous reservoir do not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public IlcFloat getLevelMin(IlcInt time) const
```

This member function returns the minimal level that is present at the given time. An instance of `IloSolver::SolverErrorException` is thrown if the timetable of the invoking continuous reservoir does not cover the given time.

```
public IlcFloat getLevelMinMax(IlcInt t1, IlcInt t2) const
```

This member function returns the minimal consumable level, over the integer time points of the interval `[timeMin, timeMax)`, of the maximal resource level. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking continuous reservoir do not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public IlcFloat getLevelMinMin(IlcInt t1, IlcInt t2) const
```

This member function returns the minimal consumable level throughout the integer time points of the interval `[timeMin, timeMax)` (that is, the minimal value over the time points `timeMin, ..., timeMax-1` of the minimal resource level). An instance of `IloSolver::SolverErrorException` is thrown if the timetable of the invoking continuous reservoir does not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public IlcFloatTimetable getTimetable(IlcInt time) const
```

This member function returns the timetable that includes `time`. An instance of `IloSolver::SolverErrorException` is thrown if no timetable is defined at `time`. The invoking resource must not be a continuous reservoir.

```
public IlcFloatTimetable getTimetable() const
```

This member function returns the first timetable of the invoking resource (first in chronological order). An instance of `IloSolver::SolverErrorException` is thrown is no timetable exists for the invoking resource. The invoking resource must not be a continuous reservoir.

```
public IlcConstraint makeTimetableConstraint(IlcInt timeMin, IlcInt timeMax,
IlcFloat precision=ILC_CONTINUOUS_RESERVOIR_PRECISION)
```

This member function creates and returns a timetable constraint for the invoking continuous reservoir. This timetable constraint implies that the capacity of the continuous reservoir is limited to the theoretical capacity of the reservoir from `timeMin` to `timeMax`. The time step of the timetable is 1. An instance of `IloSolver::SolverErrorException` is thrown if `timeMin` is not strictly less than `timeMax` or if the new timetable overlaps another timetable that has already been created for the invoking continuous reservoir.

```
public void operator=(const IlcContinuousReservoir & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument. After execution of this operator, the invoking object and the provided argument point to the same implementation object.

```
public void setLevelMax(IlcInt timeMin, IlcInt timeMax, IlcFloat levelMax)
```

This member function states that the level of the continuous reservoir can be at most `levelMax` at each integer time point of the interval `[timeMin, timeMax)`. An instance of `IloSolver::SolverErrorException` is thrown if the timetable of the invoking continuous reservoir does not cover the complete interval indicated by `[timeMin, timeMax)`. The continuous reservoir must be closed in order to propagate constraints.

```
public void setLevelMin(IlcInt timeMin, IlcInt timeMax, IlcFloat levelMin)
```

This member function states that the level of the continuous reservoir must be at least `levelMin` at each integer time point of the interval `[timeMin, timeMax)`. An instance of `IloSolver::SolverErrorException` is thrown if the timetable of the invoking continuous reservoir does not cover the complete interval indicated by `[timeMin, timeMax)`. The continuous reservoir must be closed in order to propagate constraints.

Class IlcContinuousReservoirIterator

Definition file: ilsched/contires.h
Include file: <ilsched/ilsched.h>

An instance of this class traverses the set of continuous reservoirs.

See Also: IlcContinuousReservoir, IlcSchedule

Constructor and Destructor Summary	
public	IlcContinuousReservoirIterator(const IlcSchedule schedule)

Method Summary	
public IlcBool	ok() const
public IlcContinuousReservoir	operator*() const
public IlcContinuousReservoirIterator &	operator++()

Constructors and Destructors

public **IlcContinuousReservoirIterator**(const IlcSchedule schedule)

This constructor creates an iterator to traverse all the continuous reservoirs of `schedule`.

Methods

public IlcBool **ok**() const

This member function returns `IlcTrue` if the current position of the iterator is a valid one. It returns `IlcFalse` if all the continuous reservoirs have been scanned by the iterator.

public IlcContinuousReservoir **operator***() const

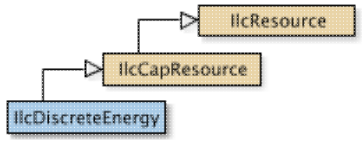
This operator accesses the instance of `IlcContinuousReservoir` located at the current position of the iterator. If the iterator is set past the end position, this operator returns an empty handle.

public IlcContinuousReservoirIterator & **operator++**()

This left-increment operator shifts the current position of the iterator.

Class IlcDiscreteEnergy

Definition file: ilsched/energy.h
Include file: <ilsched/ilsched.h>



An instance of the class `IlcDiscreteEnergy` represents a resource available as a certain amount of energy (for example, in watt-hours or human-months) over certain time buckets (for example, minutes, hours, months, or years). The available energy of a time bucket is used by the activities that are executed within that time bucket. As a consequence, constraints exist on the energy capacity of the discrete energy resource.

For example, let's assume that each unit of time corresponds to an hour, and that we have defined a discrete energy resource that has a time step of 24 (corresponding to a day), and energy 10 (machine-hours). Then if we have an activity of duration 3 (hours) that requires the resource with capacity 2 (machines), it uses energy of 6 (machine-hours). Thus, if this activity is scheduled on the first day, the remaining energy for that first day is 4 (machine-hours).

An instance of the class `IlcDiscreteEnergy` differs from an instance of the class `IlcDiscreteResource` in that it uses the concept of energy, whereas `IlcDiscreteResource` uses the concept of instantaneous capacity. However, when the time step of the timetables is 1, the energy over an interval corresponds to the instantaneous capacity, and so in that case, there is no difference between the two classes. The same is true in the case of a time table with breaks which, in the case of a time bucket of duration 1, either a break fills the bucket, or the duration of breaks in the bucket is zero.

Energy Relaxation

A common use of discrete energy is to relax the capacity constraint of a discrete resource. Instead of enforcing an instantaneous available capacity, the energy relaxation enforces that the average capacity over a time bucket cannot be exceeded. The corresponding energy is defined as the product of this average instantaneous capacity times the duration of the time bucket. An advantage is that if several activities with different start and end times share the same time bucket, the average complexity of the propagation decreases.

In a more formal way, let `act` be an activity, and `svar`, `evar` and `dvar` be the start, end, and duration variables of `act`. So we have `evar = svar + dvar`. Now let `R` be an energy resource, `bkt` a time bucket of duration `bktdur`, `bktmin` the time bucket start time, and `bktmax` the end time. Let `cvar` be the capacity required by `act` on `R`.

The duration of the execution of `act` in the time bucket `bkt` is:

```
d(act, bkt) =  min(bktdur, dvar, evar - bktmin, bktmax - svar).
```

The energy required by `act` on `bkt` is `E(act, bkt) = d(act, bkt)*cvar`.

The discrete energy resource constraint enforces that, for each time bucket `bkt`, the sum over all activities of `E(act, bkt)` is less than or equal to the maximum available energy in `bkt`.

One notices that if the bucket encompasses the activity execution, then `d(act, bkt)` is reduced to `dvar`, and the required energy is `dvar*cvar`. When the activity execution overlaps the time bucket, `d(act, bkt) = bktdur = bktmax - bktmin`, and the required energy is `(bktmax - bktmin)*cvar`.

The last remark reveals that the energy relaxation has a drawback in the case of breakable activities. The effective required energy should only depend upon the processing time, and not the entire duration. For example, in the case of a time bucket that contains the entire activity execution, `dvar*cvar` overestimates the energy that is effectively required, by failing to subtract the unused capacity during break times. Likewise, in the case of an activity execution that overlaps the time bucket, `(bktmax - bktmin)*cvar` overestimates the energy required, because breaks can take place between `bktmin` and `bktmax`.

This energy relaxation also has a drawback in the case of unary resources. For example, suppose a bucket of duration 6, and a break at the end of the bucket of duration 2. There are two ways to determine the energy. One way is to state that the net bucket energy is 6 - 2 = 4. However, a breakable activity using 1 unit of energy per unit duration, with a duration greater than 6, could not overlap the bucket, since only 4 units of energy are available over 6 time units. This is incorrect as such an activity should be able to overlap the bucket and suspend processing during the break. Alternatively, we could state that the break does not use any energy, thus allowing 6 energy units in the first 4 time units of the bucket. However, this would allow two activities of duration 3 to exist within the bucket. Recall, however, that this energy resource is a relaxation of a unary resource. Having two activities of duration 3 plus a break of 2 within an interval of size 6 clearly violates the unary resource. If possible, we would like to have a relaxation that would not allow such a "solution."

To avoid these drawbacks, Scheduler Engine offers a relaxation in the case of an energy resource with breaks and breakable activities. The main idea is to describe the bucket as containing a break duration and an energy available outside the break to process the activities. A relaxation of the processing time in the bucket for an activity is then used.

With the previous notations, let `ptvar` and `dbvar` be the processing time variable and the duration of breaks variable for an activity. We have `dvar = ptvar + dbvar`. Let `db` be the duration of the breaks in the bucket `bkt`. The relaxation of the processing time effectively executed in a bucket is given by:

```
pt(act, bkt) = min(bktdur, dvar, evar - bktmin,

bktmax - svar) - min(db, dbvar)
```

If the bucket contains the activity execution, then `dbvar < db` and `pt(act, bkt)` is reduced to `dvar - dbvar = ptvar`, and the required energy is `ptvar*cvar`. If the activity execution overlaps the time bucket, then `pt(act, bkt) = bktdur - db = bktmax - bktmin - db`, and the required energy is `(bktmax - bktmin - db)*cvar`, and this is the expected result from the energy with breaks relaxation.

Notice that in case of a non-breakable activity, `dbvar = 0` and we have the regular formula for the energy calculation. In the same way, this relaxation allows us to take into account that activities can overlap breaks. Similarly, if no break overlap happens, and if the duration of breaks is equal to the length of the bucket, `pt(act, bkt) = 0`.

The duration of breaks in a time bucket must be given before the propagation. Otherwise one can always suppose that the time bucket can be filled by breaks, making the relaxation inefficient.

This relaxation may or may not be used in conjunction with break constraints. In the last case, the user must bound the duration of the breakable activities.

The propagation of energy constraints is always based on timetables.

Printing or DisplayingDiscrete Energy Resources

The printed representation of an instance of the class `IlcDiscreteEnergy` consists of its name, followed by information about its capacity and the time step of its timetable enclosed in parentheses. If no timetable has been created, the empty parenthesis are displayed. For example:

`[10 (5)]` represents a discrete energy resource with a capacity equal to 10 and a timetable time step equal to 5.

`[3 ()]` represents a discrete energy resource with a capacity equal to 3 and no timetable.

If the Solver trace is active and the resource is not named, the string `"IlcDiscreteEnergy"` is followed by the address of the implementation object. The address will be enclosed in parentheses.

For more information, see Calendars, Timetable, Transition Time in Scheduler Engine, and Type Timetable Constraint.

See Also: IlcCapResource, IlcDiscreteEnergyIterator, IlcIntTimetable, IlcResource, IlcResourceConstraint

Constructor Summary	
public	IlcDiscreteEnergy()
public	IlcDiscreteEnergy(IlcDiscreteEnergyI * impl)
public	IlcDiscreteEnergy(IlcSchedule schedule, IlcInt timeStep, IlcInt energy, IlcBool timetable=IlcTrue)
public	IlcDiscreteEnergy(IlcSchedule schedule, IlcInt timeStep, IlcInt energy, IlcTransitionTimeObject ttobj, IlcBool timetable=IlcTrue)

Method Summary	
public IlcBool	areBreaksClosed() const
public void	closeBreaks()
public IlcInt	getBreaksDuration(IlcInt time) const
public IlcInt	getEnergy() const
public IlcInt	getEnergyMax(IlcInt time) const
public IlcInt	getEnergyMaxMax(IlcInt timeMin, IlcInt timeMax) const
public IlcInt	getEnergyMaxMin(IlcInt timeMin, IlcInt timeMax) const
public IlcInt	getEnergyMin(IlcInt time) const
public IlcInt	getEnergyMinMax(IlcInt timeMin, IlcInt timeMax) const
public IlcInt	getEnergyMinMin(IlcInt timeMin, IlcInt timeMax) const
public IlcDiscreteEnergyI *	getImpl() const
public IlcConstraint	getTypeTimetableConstraint() const
public IlcBool	hasTypeTimetableConstraint() const
public IlcConstraint	makeTimetableConstraintWB(IlcInt timeMin, IlcInt timeMax, IlcInt timeStep, IlcInt energy, IlcInt db)
public IlcConstraint	makeTypeTimetableConstraint(IlcBool useBatch=IlcFalse)
public void	operator=(const IlcDiscreteEnergy & h)
public void	setBreaksDuration(IlcInt timeMin, IlcInt timeMax, IlcInt value)
public void	setEnergyMax(IlcInt timeMin, IlcInt timeMax, IlcInt energyMax)
public void	setEnergyMin(IlcInt timeMin, IlcInt timeMax, IlcInt energyMin)
public void	setTimetablePropagation(IlcInt level=1L)

Inherited Methods from `IlcCapResource`
getImpl, getMaxTextureMeasurement, getMinTextureMeasurement, getTimetable, getTimetable, hasInitialOccupation, hasMaxTextureMeasurement, hasMinTextureMeasurement, incrDurableRequirement, incrDurableRequirement, isRedundantResource, makeBalanceConstraint, makeMaxTextureMeasurement, makeMinTextureMeasurement, makeTimetableConstraint, makeTimetableConstraint, makeTimetableConstraint, operator=, setInitialOccupation, setInitialOccupation, unsetInitialOccupation

Inherited Methods from `IlcResource`
close, getCalendar, getDisjunctiveConstraint, getDurableSchedule, getImpl,

```
getLastRankedFirstRC, getLastRankedLastRC, getLastSurelyContributingRankedFirstRC,
getLastSurelyContributingRankedLastRC, getName, getObject,
getOldLastRankedFirstRC, getOldLastRankedLastRC, getPrecedenceGraphConstraint,
getSchedule, getSolver, getSolverI, getTimetableConstraint, getTransitionTime,
hasCalendar, hasDisjunctiveConstraint, hasLightPrecedenceGraphConstraint,
hasPrecedenceGraphConstraint, hasPrecedenceInfo, hasRankInfo,
hasTimetableConstraint, isCapacityResource, isClosed, isContinuousReservoir,
isDiscreteEnergy, isDiscreteResource, isDurable, isReservoir, isStateResource,
isTransitionTimeSuspended, isUnaryResource, makeFunctionalConstraint,
makeIntegralConstraint, makeLightPrecedenceGraphConstraint,
makePrecedenceGraphConstraint, operator!=, operator=, operator==, setCalendar,
setName, setObject, setTransitionTimeObject, setTransitionTimeSuspended,
whenContribution, whenDirectPredecessors, whenDirectSuccessors, whenNext,
whenPossibleNext, whenPossiblePrevious, whenPredecessors, whenPrevious,
whenRankedFirstRC, whenRankedLastRC, whenSuccessors
```

Constructors

`public` **`IlcDiscreteEnergy`**`()`

This constructor creates an instance which is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

`public` **`IlcDiscreteEnergy`**`(IlcDiscreteEnergyI * impl)`

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

`public` **`IlcDiscreteEnergy`**`(IlcSchedule schedule, IlcInt timeStep, IlcInt energy, IlcBool timetable=IlcTrue)`

This constructor creates a new instance of `IlcDiscreteEnergy` and adds it to the set of resources managed in the given `schedule`. If `timetable` is `IlcTrue`, then a timetable constraint is posted; that constraint states that the energy of the resource is limited to `energy` for each interval `[t (t + timeStep))` where `t = timeMin + i * timeStep`, `timeMin` is the origin, and `t` is less than the horizon.

An instance of `IloSolver::SolverErrorException` is thrown if any of the following conditions occur:

- if `timeStep` is not strictly positive;
- if the time horizon minus the time origin is not a multiple of `timeStep`;
- if `energy` is negative.

If timetable is `IlcFalse`, then `energy` is managed as the energy of the resource, and `timeStep` is ignored in that case.

`public` **`IlcDiscreteEnergy`**`(IlcSchedule schedule, IlcInt timeStep, IlcInt energy, IlcTransitionTimeObject ttobj, IlcBool timetable=IlcTrue)`

This constructor creates a new instance of `IlcDiscreteEnergy` and adds it to the set of resources managed in the given `schedule`. If `timetable` is `IlcTrue`, then a timetable constraint is posted; that constraint states that the energy of the resource is limited to `energy` for each interval `[t (t + timeStep))` where `t = timeMin + i * timeStep`, `timeMin` is the origin, and `t` is less than the horizon.

An instance of `IloSolver::SolverErrorException` is thrown if any of the following conditions occur:

- if `timeStep` is not strictly positive;
- if the time horizon minus the time origin is not a multiple of `timeStep`;
- if `energy` is negative.

If timetable is `IlcFalse`, then `energy` is managed as the energy of the resource, and `timeStep` is ignored in that case.

The argument `ttobj` indicates which transition time function will be used for the invoking resource. The argument `ttobj` must have been built with an instance of `IlcTransitionTable`. Transition times are taken into account only when the type timetable constraint is posted. Please see Transition Time in Scheduler Engine and Type Timetable Constraint for more information.

Methods

```
public IlcBool areBreaksClosed() const
```

This member function returns `IlcTrue` if all the timetables with breaks declared on the invoking energy resource are closed. If a timetable with breaks is closed, the durations of breaks is completely specified for each bucket of the timetable.

At post time of the timetable constraints, the declaration of the break duration is automatically closed.

```
public void closeBreaks()
```

This member function closes the declaration of the duration of breaks for all the timetables with breaks declared on the invoking energy. If a timetable with breaks is closed, the durations of breaks is completely specified for each bucket of the timetable.

At post time of the timetable constraints, the declaration of the break duration is automatically closed.

```
public IlcInt getBreaksDuration(IlcInt time) const
```

This member function returns the duration of breaks declared in the bucket of the timetable containing the given time. If the timetable does not take into account the duration of the breaks per bucket, it returns 0. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the given time.

```
public IlcInt getEnergy() const
```

This member function returns the theoretical energy of the invoking resource, that is, the energy that was passed to the resource constructor. If the theoretical energy is unlimited, this member function returns `IlcIntMax`.

```
public IlcInt getEnergyMax(IlcInt time) const
```

This member function returns the maximal energy that can be used at the given `time`. An instance of `IloSolver::SolverErrorException` is thrown if a timetable of the invoking resource do not cover the given `time`.

```
public IlcInt getEnergyMaxMax(IlcInt timeMin, IlcInt timeMax) const
```

This member function returns the maximal energy that can be used throughout the interval `[timeMin, timeMax)` (that is, the maximal value over the interval `[timeMin, timeMax)` of the maximal resource energy). An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public IlcInt getEnergyMaxMin(IlcInt timeMin, IlcInt timeMax) const
```

This member function returns the maximal value, over the interval `[timeMin, timeMax)`, of the minimal resource energy. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public IlcInt getEnergyMin(IlcInt time) const
```

This member function returns the minimal energy that must be used at the given `time`. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the given `time`.

```
public IlcInt getEnergyMinMax(IlcInt timeMin, IlcInt timeMax) const
```

This member function returns the minimal value, over the interval `[timeMin, timeMax)`, of the maximal resource energy. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public IlcInt getEnergyMinMin(IlcInt timeMin, IlcInt timeMax) const
```

This member function returns the minimal energy that must be used throughout the interval `[timeMin, timeMax)` (that is, the minimal value over the interval `[timeMin, timeMax)` of the minimal resource energy). An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public IlcDiscreteEnergyI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcConstraint getTypeTimetableConstraint() const
```

This member function returns the type timetable constraint of the invoking resource.

```
public IlcBool hasTypeTimetableConstraint() const
```

This member function returns `IlcTrue` if the invoking resource has a type timetable constraint. Otherwise, it returns `IlcFalse`.

```
public IlcConstraint makeTimetableConstraintWB(IlcInt timeMin, IlcInt timeMax,
IlcInt timeStep, IlcInt energy, IlcInt db)
```

This member function creates and returns a timetable constraint taking into account both the energy and the duration of breaks per buckets for the invoking resource. This timetable constraint implies that the energy per bucket of the resource is limited to `energy`, and the duration of breaks per bucket is limited to `db`. The constraint is enforced from `timeMin` to `timeMax` and allows to change the energy and duration of breaks per buckets only at times `timeMin + i*timeStep`.

An instance of `IloSolver::SolverErrorException` is thrown if any of the following conditions occur:

- if `timeStep` is not strictly positive;
- if `timeMin` is not strictly less than `timeMax`;
- if `timeMax` minus `timeMin` is not a multiple of `timeStep`;
- if the new timetable overlaps another timetable that has already been created for the invoking resource.

```
public IlcConstraint makeTypeTimetableConstraint(IlcBool useBatch=IlcFalse)
```

This member function attaches a type timetable constraint to the resource and returns it.

The type timetable constraint uses the transition time object that was passed to the constructor of the invoking resource to propagate the transition times. This transition time object needs to have been built with an instance of `IlcTransitionTable`. An instance of `IloSolver::SolverErrorException` is thrown if no transition time object was passed to the constructor of the resource or if the transition time object was not built with an instance of `IlcTransitionTable`.

The `useBatch` parameter allows you to batch overlapping activities together that use the same resource and that are of the same transition type. Basically, if the execution time of the activities overlaps on the resource, then the activities will be constrained to start and end at the same time. If the execution time of the activities does not intersect on the resource, then the `useBatch` parameter has no effect on the activities. Although the activities must be of the same transition type, the transition time has no effect on the action of `useBatch`.

```
public void operator=(const IlcDiscreteEnergy & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument. After execution of this operator, the invoking object and the provided argument point to the same implementation object.

```
public void setBreaksDuration(IlcInt timeMin, IlcInt timeMax, IlcInt value)
```

This member function returns the duration of breaks in the buckets that intersect the interval `[timeMin, timeMax)`. If the timetable has its declaration of the duration of breaks closed, the call to the function is ignored. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public void setEnergyMax(IlcInt timeMin, IlcInt timeMax, IlcInt energyMax)
```

This member function states that at most `energyMax` can be used throughout at the interval `[timeMin, timeMax)`. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public void setEnergyMin(IlcInt timeMin, IlcInt timeMax, IlcInt energyMin)
```

This member function states that at least `energyMin` must be used throughout at the interval `[timeMin, timeMax)`. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval indicated by `[timeMin, timeMax)`.

The resource must be *closed* in order to propagate a non-zero minimal energy.

```
public void setTimetablePropagation(IlcInt level=1L)
```

When the argument `level` is equal to 1, this member function switches on extra propagation for the maximal duration and maximal capacity of resource constraints on the resource. This extra propagation globally analyzes

the timetable of the resource to identify two things:

1. Temporal intervals where the maximal duration of the activity can be supported if the resource constraint requires its minimal capacity
2. Temporal intervals where the maximal capacity of the resource constraint can be supported if the duration of the activity equals its minimal duration.

The constraint finds new upper bounds on duration and capacity that are supported by such time intervals.

For example, consider a schedule with 3 activities: *a0* (fixed start time 0, fixed end time 8), *a1* (fixed start time 15, fixed end time 20) and *a2* (earliest start time 0, latest end time 21, variable duration in [13..21]). All three activities require the same discrete energy resource *res* (time step = 7, energy = 100 on each steps [0..7),[7..14),[14..21) etc.). Suppose that *a0* and *a1* both require 10 units of *res*. The extra propagation at level 1 will deduce that, given the current timetable, the maximal capacity of *a2* is 12, and that this maximal capacity can be satisfied if *a2* starts at 5 and finishes at 18.

When the level is equal to 0, the extra propagation above is not performed. The propagation level can be changed in a reversible way during the search.

Class IlcDiscreteEnergyIterator

Definition file: ilsched/energy.h
Include file: <ilsched/ilsched.h>

An instance of this class traverses the set of discrete energy resources.

See Also: IlcDiscreteEnergy, IlcSchedule

Constructor and Destructor Summary	
public	IlcDiscreteEnergyIterator(const IlcSchedule schedule)

Method Summary	
public IlcBool	ok() const
public IlcDiscreteEnergy	operator*() const
public IlcDiscreteEnergyIterator &	operator++()

Constructors and Destructors

`public` **`IlcDiscreteEnergyIterator`**`(const IlcSchedule schedule)`

This constructor creates an iterator to traverse all the discrete energy resources of `schedule`.

Methods

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if the current position of the iterator is a valid one. It returns `IlcFalse` if all the discrete resources have been scanned by the iterator.

`public IlcDiscreteEnergy` **`operator*`**`() const`

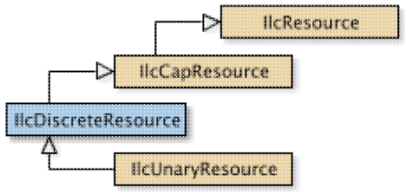
This operator accesses the instance of `IlcDiscreteEnergy` located at the current position of the iterator. If the iterator is set past the end position, this operator returns an empty handle.

`public IlcDiscreteEnergyIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator.

Class IlcDiscreteResource

Definition file: ilsched/discrete.h
Include file: <ilsched/ilsched.h>



An instance of the class `IlcDiscreteResource` represents a resource of discrete capacity. Capacity can vary over time: at any given time, the capacity represents the number of copies or instances of the resource that are available, for example, the number of milling machines available in a manufacturing shop or the number of bricklayers at work on a construction site. By *discrete*, we mean that capacity is defined to be a non-negative integer.

Each activity may require some amount of the resource capacity, for example, one milling machine or three bricklayers. This requirement is represented by *resource constraints*, and propagating these constraints entails an update of the earliest and latest start and end times of activities.

Theoretical and Maximal Capacity

The *theoreticalcapacity* of a discrete resource is a bound (that is, a limit) on the amount of capacity that can be used at any point in time. This idea can be contrasted with the *maximal capacity* that can be used "in practice" at a particular point in time or over a particular interval of time. The maximal capacity typically varies over time, while the theoretical capacity is an intrinsic property of the resource. The theoretical capacity can be infinite. Also, at any point in time, the maximal capacity cannot exceed the theoretical capacity.

Minimal Capacity

It is also possible to constrain the capacity used so that it exceeds some *minimal capacity* over some interval of time. An inconsistency will be detected if at any point in time the minimal capacity exceeds the maximal capacity.

Constraints on Discrete Resources

For discrete resources, there are several methods to take into account the constraints concerning a resource.

- The first method allows capacity to vary over time: at any given time, a maximal and a minimal capacity can be defined representing the number of instances of the resource that are available at that time. This method depends on posting a *timetable constraint*. This method is sound, which means that it ensures that any schedule on the resource satisfy the maximal and minimal capacity constraints.
- A second method deals only with requiring activities. It consists of posting a *balance constraint* to insure that the theoretical capacity is satisfied. This constraint analyzes the precedence relations between activities requiring the resource, and discovers new precedences as well as new time bounds. Provided that the maximal capacity of the resource is constant over time, this method is sound, which means that it ensures that any schedule on the resource satisfies the maximal capacity constraints. This method does not handle the minimal capacity of the resource.
- A third method consists of posting a *global disjunctive constraint*. This constraint ensures that each pair of activities that require a combined capacity that exceeds the capacity of the resource are not scheduled at the same time. In general, using this constraint alone is not enough to ensure that any schedule on the resource satisfy the maximal capacity constraints so that this method must be used in conjunction with either the timetable or the balance constraint.

Note that when you use this third method, you can increase the level of propagation even further: rather than considering only pairs of activities *(A1 A2)* to prove that *A1* must precede *A2* or vice-versa, the constraint propagation process can consider arbitrary tuples *{A1 ... An}* of activities to prove that some activity *Ai* must execute first (or must execute last) among the activities in the tuple, *{A1 ... An}*. This algorithm is known as edge-finding.

Printing or Displaying Discrete Resources

The printed representation of an instance of the class `IlcDiscreteResource` consists of its name, followed by information about its capacity enclosed in brackets. For example:

`[10]` represents a discrete resource with a capacity equal to 10.

If the Solver trace is active and the resource is not named, the string "`IlcDiscreteResource`" is followed by the address of the implementation object. The address will be enclosed in parentheses.

For more information, see Balance Constraint, Disjunctive Constraint, Edge Finder, Timetable, and Transition Time in Scheduler Engine.

See Also: IlcCapResource, IlcDiscreteResourceIterator, IlcIntTimetable, IlcResource, IlcResourceConstraint

Constructor Summary	
public	`IlcDiscreteResource()`
public	`IlcDiscreteResource(IlcDiscreteResourceI * impl)`
public	`IlcDiscreteResource(IlcSchedule schedule, IlcInt capacity, IlcBool timetable=IlcTrue)`
public	`IlcDiscreteResource(IlcSchedule schedule, IlcInt capacity, IlcTransitionTimeObject ttobj, IlcBool timetable=IlcTrue)`

Method Summary	
public IlcInt	`getCapacity() const`
public IlcInt	`getCapacityMax(IlcInt time) const`
public IlcInt	`getCapacityMaxMax(IlcInt timeMin, IlcInt timeMax) const`
public IlcInt	`getCapacityMaxMin(IlcInt timeMin, IlcInt timeMax) const`
public IlcInt	`getCapacityMin(IlcInt time) const`
public IlcInt	`getCapacityMinMax(IlcInt timeMin, IlcInt timeMax) const`
public IlcInt	`getCapacityMinMin(IlcInt timeMin, IlcInt timeMax) const`
public IlcInt	`getGlobalSlack() const`
public IlcDiscreteResourceI *	`getImpl() const`
public IlcInt	`getLocalSlack() const`
public IlcConstraint	`getTypeTimetableConstraint() const`
public IlcBool	`hasTypeTimetableConstraint() const`
public IlcConstraint	`makeDisjunctiveConstraint()`
public IlcConstraint	`makeTypeTimetableConstraint(IlcBool useBatch=IlcFalse)`
public void	`operator=(const IlcDiscreteResource & h)`
public void	`setCapacityMax(IlcInt timeMin, IlcInt timeMax, IlcInt capacityMax)`
public void	`setCapacityMin(IlcInt timeMin, IlcInt timeMax, IlcInt capacityMin)`
public void	`setEdgeFinder(IlcInt edgeFinder=1)`

public void	setPrecedencePropagation(IlcInt level=1L)
public void	setTimetablePropagation(IlcInt level=1L)
public void	storeSufficientDirectSuccessors(IloSchedulerSolution solution, IloRandom rand=0)

Inherited Methods from IlcCapResource
getImpl, getMaxTextureMeasurement, getMinTextureMeasurement, getTimetable, getTimetable, hasInitialOccupation, hasMaxTextureMeasurement, hasMinTextureMeasurement, incrDurableRequirement, incrDurableRequirement, isRedundantResource, makeBalanceConstraint, makeMaxTextureMeasurement, makeMinTextureMeasurement, makeTimetableConstraint, makeTimetableConstraint, makeTimetableConstraint, operator=, setInitialOccupation, setInitialOccupation, unsetInitialOccupation

Inherited Methods from IlcResource
close, getCalendar, getDisjunctiveConstraint, getDurableSchedule, getImpl, getLastRankedFirstRC, getLastRankedLastRC, getLastSurelyContributingRankedFirstRC, getLastSurelyContributingRankedLastRC, getName, getObject, getOldLastRankedFirstRC, getOldLastRankedLastRC, getPrecedenceGraphConstraint, getSchedule, getSolver, getSolverI, getTimetableConstraint, getTransitionTime, hasCalendar, hasDisjunctiveConstraint, hasLightPrecedenceGraphConstraint, hasPrecedenceGraphConstraint, hasPrecedenceInfo, hasRankInfo, hasTimetableConstraint, isCapacityResource, isClosed, isContinuousReservoir, isDiscreteEnergy, isDiscreteResource, isDurable, isReservoir, isStateResource, isTransitionTimeSuspended, isUnaryResource, makeFunctionalConstraint, makeIntegralConstraint, makeLightPrecedenceGraphConstraint, makePrecedenceGraphConstraint, operator!=, operator=, operator==, setCalendar, setName, setObject, setTransitionTimeObject, setTransitionTimeSuspended, whenContribution, whenDirectPredecessors, whenDirectSuccessors, whenNext, whenPossibleNext, whenPossiblePrevious, whenPredecessors, whenPrevious, whenRankedFirstRC, whenRankedLastRC, whenSuccessors

Constructors

public **IlcDiscreteResource**()

This constructor creates an instance which is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

public **IlcDiscreteResource**(IlcDiscreteResourceI * impl)

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

public **IlcDiscreteResource**(IlcSchedule schedule, IlcInt capacity, IlcBool timetable=IlcTrue)

This constructor creates a new instance of IlcDiscreteResource and adds it to the set of resources managed by schedule. The capacity of the resource is limited to capacity.

The argument timetable indicates whether the standard timetable constraint should be posted. The standard timetable constraint manages the capacity of the resource from the time origin to the time horizon of the given schedule and allows the capacity to change at any point in time; that is, it defines a time step of 1 (one).

public **IlcDiscreteResource**(IlcSchedule schedule, IlcInt capacity, IlcTransitionTimeObject ttobj, IlcBool timetable=IlcTrue)

This constructor creates a new instance of `IlcDiscreteResource` and adds it to the set of resources managed by `schedule`. The capacity of the resource is limited to `capacity`.

The argument `ttobj` indicates which transition time function will be used for the invoking resource. The argument `ttobj` must have been built with an instance of `IlcTransitionTable`. An instance of `IloSolver::SolverErrorException` is thrown if this is not the case.

Transition times are taken into account when the disjunctive constraint or the type timetable constraint is posted. Transition times are only propagated between two activities that are incompatible. As the disjunctive constraint defines incompatibility based on the resource demand of the activities and the type timetable constraint defines incompatibility based on the transition types of the activities, they do not propagate in the same manner. Please see Transition Time in Scheduler Engine and Type Timetable Constraint for more information. Note that when the precedence graph constraint is posted, transition times are also propagated between successor resource constraints.

The argument `timetable` indicates whether the standard timetable constraint should be posted. The standard timetable constraint manages the capacity of the resource from the time origin to the time horizon of the given schedule and allows the capacity to change at any point in time; that is, it defines a time step of 1 (one).

Methods

```
public IlcInt getCapacity() const
```

This member function returns the theoretical capacity of the invoking resource, that is, the capacity that was passed to the resource constructor.

```
public IlcInt getCapacityMax(IlcInt time) const
```

This member function returns the maximal capacity that can be used at the given `time`. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the given `time`.

```
public IlcInt getCapacityMaxMax(IlcInt timeMin, IlcInt timeMax) const
```

This member function returns the maximal value of the maximal resource capacity over the interval `[timeMin, timeMax)`. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public IlcInt getCapacityMaxMin(IlcInt timeMin, IlcInt timeMax) const
```

This member function returns the maximal value of the minimal resource capacity over the interval `[timeMin, timeMax)`. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public IlcInt getCapacityMin(IlcInt time) const
```

This member function returns the minimal capacity that must be used or is actually used at the given `time`. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the given `time`.

```
public IlcInt getCapacityMinMax(IlcInt timeMin, IlcInt timeMax) const
```

This member function returns the minimal value of the maximal resource capacity over the interval `[timeMin, timeMax)`. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public IlcInt getCapacityMinMin(IlcInt timeMin, IlcInt timeMax) const
```

This member function returns the minimal value of the minimal resource capacity over the interval `[timeMin, timeMax)`. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public IlcInt getGlobalSlack() const
```

This member function measures the overall capacity still available of the invoking resource. To do so, it looks at all the resource constraints that surely use the invoking resource and for which the activities have *not yet been assigned a start time* and calculates their overall minimal start time, *smin*, and their overall maximal end time, *emax*. It then computes the sum of all minimal energies of the activities that should be processed between *smin* and *emax*. It returns the maximal available energy between *smin* and *emax* minus this calculated sum of minimal energies; that is, $((emax - smin) \times getCapacity) - \sum minEnergy$

The minimal energy of an activity is defined as the minimum of the product of the duration of the activity and the required capacity of the activity on the resource.

Note that the global slack does not take into account the maximal capacity profile or the break list eventually defined on the invoking resource.

```
public IlcDiscreteResourceI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcInt getLocalSlack() const
```

This member function returns the minimal slack over all intervals *[t1, t2)* where *t1* corresponds to the earliest start time of an activity and *t2* corresponds to the latest end time of an activity and for which it holds that *[t1, t2)* is included in the interval *[smin, emax)* where *smin* is the overall minimal start time and *emax* the overall maximal end time of the set of activities that surely use the invoking resource and that have not yet been assigned a start time.

The slack in an interval *[t1, t2)* is defined as *(t2 - t1)*getCapacity - sumEnergy(t1, t2)*, where *sumEnergy(t1, t2)* is the sum of the minimal energy of activities that have to be scheduled between *t1* and *t2*. The minimal energy of an activity is defined as the minimum of the product of the duration of the activity and the required capacity of the activity on the resource.

If all activities that surely use the resource have been assigned a start time, this member function returns the product of the capacity of the resource and the difference between the scheduling horizon and the scheduling origin *((schedule.getTimeMax() - schedule.getTimeMin())*getCapacity()*, where *schedule* is the schedule of the invoking resource).

Note that the local slack does not take into account the maximal capacity profile or the break list eventually defined on the invoking resource.

Example

Let's consider an example. Let's say we're scheduling three activities, *A*, *B*, and *C*, of which we know that activity *A* has a duration of only two days and can take place anytime in the next twenty days; activity *B* lasts 5 days, cannot start before day 1, and must be finished by day 12; and finally activity *C* will last 4 days, can start on day 2, and must be finished by day 13. All 3 activities require a capacity of 1 from a resource with a capacity of 2

If we consider the problem globally, we look at all the earliest start times, and among those values, we take the minimal. That is, we take the smallest of the earliest start times (that's day 0 for activity *A*) and the last of the latest end times (that's day 20 for activity *A*), so there are 40 capacity-days (20 days * 2 capacity) available to us; and the activities take only 11 days (2+5+4) total; so we have 29 capacity days (40-11) slack globally.

However, if we refine our idea of slack by considering the earliest start time of any activity and the latest end time of any other activity, we get much tighter slack times. By considering the earliest start time of activity *B* and the latest end time of activity *C*, we'll get an overall span of 12 days. The total duration of the activities that must absolutely execute during this period is 9 days (5 days for *B* and 4 days for *C*), so we now have only 15 capacity days ((12*2)-9) of slack.

```
public IlcConstraint getTypeTimetableConstraint() const
```

This member function returns the *type timetable constraint* of the invoking resource.

```
public IlcBool hasTypeTimetableConstraint() const
```

This member function returns `IlcTrue` if the invoking resource has a *type timetable constraint*. Otherwise, it returns `IlcFalse`.

```
public IlcConstraint makeDisjunctiveConstraint()
```

This member function creates and returns the global disjunctive constraint associated with the invoking resource. That constraint has to be posted in order to be taken into account. For more information, see Disjunctive Constraint.

```
public IlcConstraint makeTypeTimetableConstraint(IlcBool useBatch=IlcFalse)
```

This member function attaches a *type timetable constraint* to the resource and returns it.

The type timetable constraint uses the transition time object that was passed to the constructor of the invoking resource to propagate the transition times. This transition time object needs to have been built with an instance of `IlcTransitionTable`. An instance of `IloSolver::SolverErrorException` is thrown if no transition time object was passed to the constructor of the resource or if the transition time object was not built with an instance of `IlcTransitionTable`.

The `useBatch` parameter allows you to batch overlapping activities together that use the same resource and that are of the same transition type. Basically, if the execution time of the activities overlaps on the resource, then the activities will be constrained to start and end at the same time. If the execution time of the activities does not intersect on the resource, then the `useBatch` parameter has no effect on the activities. Although the activities must be of the same transition type, the transition time has no effect on the action of `useBatch`.

```
public void operator=(const IlcDiscreteResource & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument. After execution of this operator, the invoking object and the provided argument point to the same implementation object.

```
public void setCapacityMax(IlcInt timeMin, IlcInt timeMax, IlcInt capacityMax)
```

This member function states that at most `capacityMax` can be used throughout the interval `[timeMin, timeMax)`. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public void setCapacityMin(IlcInt timeMin, IlcInt timeMax, IlcInt capacityMin)
```

This member function states that at least `capacityMin` must be used throughout the interval `[timeMin, timeMax)`. Only if the resource is closed will this result in propagation. If the resource is not closed, any number of resource constraints can still be added, and thus no deductions can be made. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public void setEdgeFinder(IlcInt edgeFinder=1)
```

If the parameter `edgeFinder` is 1 (one), then this member function switches on the extra propagation for Disjunctive Constraint. If `edgeFinder` is strictly greater than 1 and the resource is an instance of the class `IlcUnaryResource`, an extra level of propagation is switched on also. If `edgeFinder` is 0 (zero), all extra propagation is switched off. Note that if the invoking resource is not an instance of `IlcUnaryResource`, then the disjunctive constraints must have been created on the resource.

Extra propagation should *not* be used when a very large capacity is defined for the resource. That extra propagation requires the calculation of the available “energy” of the resource; that energy is defined as the theoretical capacity multiplied by the available time. Having such a large capacity might lead to overflow problems.

```
public void setPrecedencePropagation(IlcInt level=1L)
```

This member function should be used only if a precedence graph constraint has been created on the resource. It switches on an extra propagation for the precedence graph constraint.

If `level` is 1, the extra propagation computes new bounds for the start time and completion time of the resource constraints in the graph by analyzing their *direct* predecessors and successors.

If `level` is 2, this extra propagation computes new bounds for the start time and completion time of the resource constraints in the graph by analyzing all their predecessors and successors.

For example, consider a schedule with 4 activities: *a1* (earliest start time = 0, duration = 20), *a2* (earliest start time = 0, duration = 30), *a3* (earliest start time = 0, duration = 40) and *b* that require the same discrete resource *res (capacity = 2)*. Suppose that *a2* requires 2 units of *res* whereas all other activities require 1 unit of *res*. If, on the precedence graph of resource *res*, activity *a2* is constrained to be successor of *a1* and activity *b* is constrained to be successor of both activities *a2* and *a3*, the extra-propagation at level 1 will deduce that *b* cannot start before 50 while the extra-propagation at level 2 will deduce that *b* cannot start before 60.

```
public void setTimetablePropagation(IlcInt level=1L)
```

When the argument `level` is equal to 1, this member function switches on extra propagation for the maximal duration and maximal capacity of resource constraints on the resource. This extra propagation globally analyzes the timetable of the resource to identify two things:

1. Temporal intervals where the maximal duration of the activity can be supported should the resource constraint require its minimal capacity

2. Temporal intervals where the maximal capacity of the resource constraint can be supported should the duration of the activity equal its minimal duration.

The constraint finds new upper bounds on duration and capacity that are supported by such time intervals.

For example, consider a schedule with two activities: *a0* (fixed start time 20, fixed end time 70) and *a1* (earliest start time = 0, latest end time 100, variable duration in [10..100]). These two activities require the same discrete resource *res (capacity = 2)*. Suppose that *a0* requires 1 unit of *res* whereas *a1* requires 2 units of *res*. The extra propagation at level 1 will deduce that the maximal duration of *a1* is 30. This is because the two time intervals where *a1* could be executed, given its minimal requirement of 2 units of *res*, are [0,20] and [70,100]. The maximal duration of 30 corresponds to this last interval.

When the level is equal to 0, the extra propagation above is not performed. The propagation level can be changed in a reversible way during the search.

```
public void storeSufficientDirectSuccessors(IloSchedulerSolution solution,
IloRandom rand=0)
```

This member function stores in the solution a set of direct successors that are sufficient to ensure that the theoretical capacity of the resource is not exceeded. It can be called even if no precedence graph has been created. Only resource constraints that surely contribute and that have their start and end variables bound are taken into account. When the optional parameter `rand` is used, a non-deterministic heuristic is used to select the set of direct successors.

Class IlcDiscreteResourceIterator

Definition file: ilsched/discrete.h
Include file: <ilsched/ilsched.h>

An instance of this class traverses the set of discrete resources.

See Also: IlcDiscreteResource, IlcSchedule

Constructor and Destructor Summary	
public	IlcDiscreteResourceIterator(const IlcSchedule schedule)

Method Summary	
public IlcBool	ok() const
public IlcDiscreteResource	operator*() const
public IlcDiscreteResourceIterator &	operator++()

Constructors and Destructors

`public` **`IlcDiscreteResourceIterator`**`(const IlcSchedule schedule)`

This constructor creates an iterator to traverse all the discrete resources of `schedule`.

Methods

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if the current position of the iterator is a valid one. It returns `IlcFalse` if all the discrete resources have been scanned by the iterator.

`public IlcDiscreteResource` **`operator*`**`() const`

This operator accesses the instance of `IlcDiscreteResource` located at the current position of the iterator. If the iterator is set past the end position, this operator returns an empty handle.

`public IlcDiscreteResourceIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator.

Class IlcFollowingActivityIterator

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>

During the search, an instance of this class traverses the activities that are such that a precedence constraint holds between this activity and the activity given in the constructor of the iterator. If used before entering the search, this iterator will traverse an empty list of activities.

This class of iterators does not distinguish among the precedence constraints created by means of the member functions `IlcActivity::startsAfterStart`, `IlcActivity::startsAfterEnd`, `IlcActivity::endsAfterStart`, `IlcActivity::endsAfterEnd`, `IlcActivity::startsAtStart`, `IlcActivity::startsAtEnd`, `IlcActivity::endsAtStart`, `IlcActivity::endsAtEnd`. Whichever of those member functions is used to create the precedence constraint, the invoking activity is identified as the one that is constrained to occur after the activity passed as an argument to the member function. Likewise, the activity passed as an argument to the member function is identified as the one that is constrained to occur before the invoking activity. See the Example which follows for a program that illustrates that idea.

The order in which activities are seen by the iterator is platform-dependent and thus is not predictable.

Example

```
 Must be during search (e.g., inside a goal)

  IloSolver solver = getSolver();
  IlcScheduler schedule(solver, 0, 20);

  IlcActivity A(schedule, 5); A.setName("A");
  IlcActivity B(schedule, 5); B.setName("B");
  IlcActivity C(schedule, 5); C.setName("C");

  solver.add(A.startsAtEnd(C));
  solver.add(B.endsAtStart(C));
  solver.add(A.startsAfterEnd(B, 2));

  solver.out() << "Preceding A:" << endl;
  for(IlcPrecedingActivityIterator precedingActivityIterator(A);
      precedingActivityIterator.ok();
      ++precedingActivityIterator)
    solver.out() << " " << *precedingActivityIterator << endl;

  solver.out() << "Following C:" << endl;
  for (IlcFollowingActivityIterator followingActivityIterator(C);
       followingActivityIterator.ok();
       ++followingActivityIterator)
    solver.out() << " " << *followingActivityIterator << endl;
```

See Also: IlcActivity, IlcPrecedenceConstraint, IlcPrecedingActivityIterator

Constructor and Destructor Summary	
public	`IlcFollowingActivityIterator(IlcActivity activity, IlcResourceConstraintIteratorFilter filter=IlcAllConstraints)`

Method Summary	
`public IlcPrecedenceConstraint`	`getPrecedenceConstraint() const`
`public IlcBool`	`ok() const`

public IlcActivity	operator*() const
public IlcFollowingActivityIterator &	operator++()

Constructors and Destructors

`public` **`IlcFollowingActivityIterator`**`(IlcActivity activity, IlcResourceConstraintIteratorFilter filter=IlcAllConstraints)`

This constructor creates an iterator to traverse all the activities constrained to follow `activity`.

Methods

`public IlcPrecedenceConstraint` **`getPrecedenceConstraint`**`() const`

This member function returns the precedence constraint between the activity used by the constructor of the invoking iterator and the activity at its current position.

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if the current position of the iterator is a valid one. It returns `IlcFalse` if all the activities have been scanned by the iterator.

`public IlcActivity` **`operator*`**`() const`

This operator accesses the instance of `IlcActivity` located at the current position of the iterator. If the iterator is set past the end position, this operator returns an empty handle.

`public IlcFollowingActivityIterator &` **`operator++`**`()`

This operator shifts the current position of the iterator.

Class IlcGranularFunction

Definition file: ilsched/gfbase.h
Include file: <ilsched/ilsched.h>

An instance of `IlcGranularFunction` holds a description of a granular step-wise function.

The granular function must respect the following properties:

- It is defined over the range `[xmin,xmax)`, and takes only integer values.
- It consists only of steps with non-negative values. These steps are closed on the left and open on the right.
- Its maximum value multiplied by the width `xmax-xmin` of its definition interval must remain in the range `[0,IlcIntMax)`, in order to not overflow the platform integer representation.

These properties are checked when starting to solve the problem, and an exception will be thrown if necessary.

The positive granularity parameter is used as a scaling factor when computing the integral of the function. This allows limited representation of non-integer function values. This is particularly the case for integral expressions or constraints built with `IlcGranularFunction` (see Functional and Integral Constraints on Resources for more information).

When computing the integral of the function over a given interval (for example, the start and end time of an activity), the result is divided by the granularity, and the result rounded:

$$\text{Integral Value} = \text{ROUND}\left(\left(\int_{start}^{end} func\right) / \text{granularity}\right)$$

Four rounding modes are available when dividing by `granularity`. See Functional and Integral Constraints on Resources for more information.

Note that the member function `IlcGranularFunction::getValue` does not use the granularity, but returns the actual value stored in the function, without any scaling.

For more information, see Functional and Integral Constraints on Resources.

See Also: IlcResource, IlcFunctionalExp, IlcActivityIntegralExp, IlcGranularFunctionRoundingMode

Constructor Summary	
public	`IlcGranularFunction()`
public	`IlcGranularFunction(IlcGranularFunctionI * impl)`
public	`IlcGranularFunction(IlcManager m, IlcInt xmin, IlcInt xmax, IlcInt granularity=1, IlcGranularFunctionRoundingMode rounding=IlcGranularFunctionRoundUpward)`

Method Summary	
public void	`close() const`
public IlcInt	`getDefinitionIntervalMax() const`
public IlcInt	`getDefinitionIntervalMin() const`
public IlcInt	`getGranularity() const`
public IlcGranularFunctionI *	`getImpl() const`

public const char *	getName() const
public IlcAny	getObject() const
public IlcGranularFunctionRoundingMode	getRoundingMode() const
public IloSolver	getSolver() const
public IloSolverI *	getSolverI() const
public IlcInt	getValue(IlcInt x) const
public IlcBool	isClosed() const
public IlcIntExp	operator()(const IlcIntVar x) const
public void	operator=(const IlcGranularFunction & h)
public void	setName(const char * name) const
public void	setObject(IlcAny object) const
public void	setRoundingMode(IlcGranularFunctionRoundingMode rounding) const
public void	setValue(IlcInt x1, IlcInt x2, IlcInt value) const

Constructors

public **IlcGranularFunction**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcGranularFunction**(IlcGranularFunctionI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IlcGranularFunction**(IlcManager m, IlcInt xmin, IlcInt xmax, IlcInt granularity=1, IlcGranularFunctionRoundingMode rounding=IlcGranularFunctionRoundUpward)

This constructor creates a new instance of `IlcGranularFunction`, with granularity equal to `granularity`. The initial function is on the interval `[xmin, xmax)`, and will be set to a constant initial value of `granularity` over this interval. The parameter `rounding` selects the default rounding mode applicable for an integral constraint built with this instance.

Methods

public void **close**() const

This member function closes the invoking function. That is, it states that the function is known so constraint propagation can proceed. This is a reversible operation. Any modification of a closed function will raise an error.

public IlcInt **getDefinitionIntervalMax**() const

This member function returns the right-most point of the interval of definition of the invoking granular function.

public IlcInt **getDefinitionIntervalMin**() const

This member function returns the left-most point of the interval of definition of the invoking granular function.

```
public IlcInt getGranularity() const
```

This member function returns the value of the granularity.

```
public IlcGranularFunctionI * getImpl() const
```

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

```
public const char * getName() const
```

This member function returns the name of the invoking object.

```
public IlcAny getObject() const
```

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

```
public IlcGranularFunctionRoundingMode getRoundingMode() const
```

This member function returns the current rounding mode of the invoking granular function.

```
public IloSolver getSolver() const
```

This member function returns an instance of `IloSolver` associated with the invoking object.

```
public IloSolverI * getSolverI() const
```

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

```
public IlcInt getValue(IlcInt x) const
```

This member function returns the current value of the granular function at point `x`. This point must be inside the range `[xmin, xmax)`. Otherwise, an exception is thrown.

```
public IlcBool isClosed() const
```

This member function returns `IlcTrue` if the invoking object is closed. Otherwise, it returns `IlcFalse`.

```
public IlcIntExp operator()(const IlcIntVar x) const
```

This function creates and returns an integer expression constrained to be the value of the function at the value of variable `x`. The granularity of the function must be equal to 1, otherwise, an error will be raised.

```
public void operator=(const IlcGranularFunction & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public void setName(const char * name) const
```

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

```
public void setObject(IlcAny object) const
```

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

```
public void setRoundingMode(IlcGranularFunctionRoundingMode rounding) const
```

This member function selects the rounding mode that will be used when creating an integral constraint with the invoking granular function.

```
public void setValue(IlcInt x1, IlcInt x2, IlcInt value) const
```

This member function sets the value of the granular function to `value` over the interval `[x1, x2)`. The `IlcGranularFunction` must not be closed. The arguments `x1` and `x2` must respect `xmin <= x1 < x2 <= xmax`, and `value` must be non-negative. Otherwise, an exception will be thrown.

Class IlcGranularFunctionCursor

Definition file: ilsched/gfbase.h
Include file: <ilsched/ilsched.h>

An instance of `IlcGranularFunctionCursor` traverses the segments of a granular function.

See Also: IlcGranularFunction

Constructor Summary	
public	IlcGranularFunctionCursor(const IlcGranularFunction func, IlcInt x)

Method Summary	
public IlcInt	getSegmentMax() const
public IlcInt	getSegmentMin() const
public IlcInt	getValue() const
public IlcBool	ok() const
public void	operator++()
public void	operator--()

Constructors

`public` **`IlcGranularFunctionCursor`**`(const IlcGranularFunction func, IlcInt x)`

This constructor creates a cursor to traverse the segments of the granular function `func`. It is initialized at the segment containing the position `x`. If this position is invalid, an error will be raised.

Methods

`public IlcInt` **`getSegmentMax`**`() const`

This member function returns the right-most valid position pertaining to the current segment.

`public IlcInt` **`getSegmentMin`**`() const`

This member function returns the left-most valid position pertaining to the current segment.

`public IlcInt` **`getValue`**`() const`

This member function returns the value taken by the function on the current segment.

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if the current position of the cursor is a valid one. Otherwise, it returns `IlcFalse`.

```
public void operator++()
```

This left-increment operator shifts the current position of the cursor to the next segment of the function.

```
public void operator--()
```

This left-decrement operator shifts the current position of the cursor to the previous segment of the function.

Class IlcIntervalList

Definition file: ilsched/breaks.h
Include file: <ilsched/ilsched.h>

An instance of the class `IlcIntervalList` represents a list of non-overlapping intervals. Each interval *[timeMin, timeMax)* from the list is associated with a numerical type.

Such time intervals can be used, for instance, to represent breaks or shifts. As each interval can be associated with an integer type, some specific behaviors can be defined on activities (ignore shift or break interval, break interval possibly overlapped, and so forth).

Break and shift intervals can be attached to a calendar and then associated with resources and resource constraints (see `IlcCalendar`).

Printing or Displaying Lists of Intervals

The printed representation of an instance of the class `IlcIntervalList` consists of its name, followed by the first chronological interval and the last interval. For example:

`<[20->40)...[70->80)>` represents a list of intervals whose first interval occurs in the time interval `[20 40)` and whose last interval occurs in the time interval `[70 80)`.

If the Solver trace is active and the resource is not named, the string `"IlcIntervalList"` is followed by the address of the implementation object. The address will be enclosed in parentheses.

For more information, see Calendars.

See Also: IlcIntervalListCursor

Constructor Summary	
public	`IlcIntervalList()`
public	`IlcIntervalList(IlcIntervalListI * impl)`
public	`IlcIntervalList(const IlcSchedule schedule)`

Method Summary	
public void	`addInterval(IlcInt start, IlcInt end, IlcInt type=OL)`
public void	`addIntervalOnDuration(IlcInt start, IlcInt duration, IlcInt type=OL)`
public void	`addPeriodicInterval(IlcInt start, IlcInt duration, IlcInt period)`
public void	`addPeriodicInterval(IlcInt start, IlcInt duration, IlcInt period, IlcInt end, IlcInt type=0L)`
public void	`close()`
public IlcIntervalListI *	`getImpl() const`
public const char *	`getName() const`
public IlcAny	`getObject() const`
public IloSolver	`getSolver() const`
public IloSolverI *	`getSolverI() const`

public IlcBool	isClosed() const
public IlcBool	isEmpty() const
public void	open()
public void	operator=(const IlcIntervalList & h)
public void	removeInterval(IlcInt start, IlcInt end)
public void	removeIntervalOnDuration(IlcInt start, IlcInt duration)
public void	removePeriodicInterval(IlcInt start, IlcInt duration, IlcInt period, IlcInt end)
public void	removePeriodicInterval(IlcInt start, IlcInt duration, IlcInt period)
public void	setName(const char * name) const
public void	setObject(IlcAny object) const

Constructors

`public` **`IlcIntervalList`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcIntervalList`**`(IlcIntervalListI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IlcIntervalList`**`(const IlcSchedule schedule)`

This constructor creates a new instance of `IlcIntervalList` and adds it to the set of interval lists managed by the given `schedule`. That newly constructed list of intervals is initially empty.

Methods

`public void` **`addInterval`**`(IlcInt start, IlcInt end, IlcInt type=0L)`

This member function adds an interval of type `type` to the invoking list of intervals. The start time and end time of that newly added interval are set to `start` and `end`. By default, the type of the interval is 0.

An interval can be added to the interval list as long as it is not closed. The effect of this function is reversible.

Any attempt to add a new interval that overlaps with an already existing interval of a different type raises an error.

`public void` **`addIntervalOnDuration`**`(IlcInt start, IlcInt duration, IlcInt type=0L)`

This member function adds an interval of type `type` to the invoking list of intervals. The start time of the newly added interval is `start`; its end time is `start+duration`. By default, the type of the interval is 0.

An interval can be added to the interval list as long as it is not closed. The effect of this function is reversible.

Any attempt to add a new interval that overlaps with an already existing interval of a different type raises an error.

`public void` **`addPeriodicInterval`**`(IlcInt start, IlcInt duration, IlcInt period)`

This member function adds a set of intervals of type 0 to the invoking list of intervals. For every `i >= 0` such that `start + i * period < horizon`, an interval `[start + i * period, start + duration + i * period)` is added. `horizon` is the time horizon of the schedule that was given as an argument to the constructor of the invoking list of intervals. A periodic interval can be added to the interval list as long as it is not closed. The effect of this function is reversible.

Any attempt to add a new periodic interval that overlaps with an already existing interval of a different type raises an error.

```
public void addPeriodicInterval(IlcInt start, IlcInt duration, IlcInt period,
IlcInt end, IlcInt type=0L)
```

This member function adds a set of intervals to the invoking list of intervals. For every `i >= 0` such that `start + i * period < end`, an interval `[start + i * period, start + duration + i * period)` is added. By default, the type of the interval is 0.

A periodic interval can be added to the interval list as long as it is not closed. The effect of these functions is reversible.

Any attempt to add a new interval that overlaps with an already existing interval of a different type raises an error.

```
public void close()
```

This member function closes the interval list, which means that intervals can no longer be added or removed. The Scheduler Engine uses this knowledge to perform extra propagation. Closing an already closed interval list has no effect. This function is reversible.

Example

The following piece of code illustrates the extra propagation when a break list is closed.

```
//Must be during search (e.g., inside a goal)

IloSolver solver = getSolver();
IlcScheduler schedule(solver,0,100);
IlcUnaryResource res(schedule);
IlcCalendar cal(schedule);
IlcIntervalList bl(schedule);
cal.setBreakList(bl);
res.setCalendar(cal);
IlcActivity act(schedule, 20, IlcTrue);
solver.add(act.requires(res));
solver.add(act.startsAt(10));
solver.out() << act << endl;
// [10 -- (20) 20..90 --> 30..100]
bl.addInterval(20,50);
solver.out() << act << endl;
// [10 -- (20) 50..90 --> 60..100]
bl.close();
solver.out() << act << endl;
// [10 -- (20) 50 --> 60]
```

```
public IlcIntervalListI * getImpl() const
```

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

```
public const char * getName() const
```

This member function returns the name of the invoking object.

```
public IlcAny getObject() const
```

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

```
public IloSolver getSolver() const
```

This member function returns an instance of `IloSolver` associated with the invoking object.

```
public IloSolverI * getSolverI() const
```

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

```
public IlcBool isClosed() const
```

This member function returns `IlcTrue` if the invoking interval list is closed. Otherwise, it returns `IlcFalse`.

```
public IlcBool isEmpty() const
```

This member function returns `IlcTrue` if the invoking interval list is empty. Otherwise, it returns `IlcFalse`.

```
public void open()
```

This member function opens an interval list so that it is possible to add or remove intervals on it. Opening an already opened interval list has no effect.

This member function is available only outside the search. Trying to open an interval list in search raises an error.

```
public void operator=(const IlcIntervalList & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public void removeInterval(IlcInt start, IlcInt end)
```

This member function removes all intervals between `start` and `end`. If `start` is placed inside an interval `[start1, end1)`, that is, `start1 < start < end1`, this results in an interval `[start1, start)`. If `end` is placed inside an interval `[start2, end2)` this results in an interval `[end, end2)`. The interval list must be open. Trying to remove an interval during the search or from a closed interval list raises an error.

This member function is available only outside the search.

```
public void removeIntervalOnDuration(IlcInt start, IlcInt duration)
```

This member function removes all intervals between `start` and `start+duration`. The interval list must be open. Trying to remove an interval during the search or from a closed interval list raises an error.

This member function is available only outside the search.

```
public void removePeriodicInterval(IlcInt start, IlcInt duration, IlcInt period,
IlcInt end)
```

These member functions remove intervals from the invoking interval list. More precisely, for every `i >= 0` such that `start + i * period < end`, these functions remove all intervals between `start + i * period` and `start + duration + i * period`. The invoking interval list must be open. Trying to remove a periodic interval during the search or from a closed interval list raises an error.

This member function is available only outside the search.

```
public void removePeriodicInterval(IlcInt start, IlcInt duration, IlcInt period)
```

These member functions remove intervals from the invoking interval list. More precisely, for every `i >= 0` such that `start + i * period < horizon`, these functions remove all intervals between `start + i * period` and `start + duration + i * period`. `horizon` is the time horizon of the schedule that was given as an argument to the constructor of the invoking list. The invoking interval list must be open. Trying to remove a periodic interval during the search or from a closed interval list raises an error.

This member function is available only outside the search.

```
public void setName(const char * name) const
```

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

```
public void setObject(IlcAny object) const
```

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

Class IlcIntervalListCursor

Definition file: ilsched/brkcsor.h
Include file: <ilsched/ilsched.h>

An instance of the class `IlcIntervalList` represents a list of non-overlapping intervals. Each interval *[timeMin, timeMax)* from the list is associated with a numerical type.

Note that when two consecutive intervals of the list have the same types, these intervals are merged so that the list is always represented with the minimal number of intervals.

The class `IlcIntervalListCursor` provides a way to traverse those lists of intervals.

See Also: IlcIntervalList

Constructor and Destructor Summary	
public	`IlcIntervalListCursor(const IlcIntervalListCursor & csor)`
public	`IlcIntervalListCursor(const IlcIntervalList list, const IlcBool forward=IlcTrue)`

Method Summary	
public IlcInt	`getDuration() const`
public IlcInt	`getEnd() const`
public IlcInt	`getStart() const`
public IlcInt	`getType() const`
public IlcBool	`ok() const`
public IlcIntervalListCursor &	`operator++()`
public IlcIntervalListCursor &	`operator--()`

Constructors and Destructors

`public` **`IlcIntervalListCursor`**`(const IlcIntervalListCursor & csor)`

This copy constructor creates a cursor by copying another one. C++ relies on this constructor when you pass an interval cursor as an argument to a function.

`public` **`IlcIntervalListCursor`**`(const IlcIntervalList list, const IlcBool forward=IlcTrue)`

This constructor creates a new instance of `IlcIntervalListCursor` that traverses the intervals in `list`. If `forward` is `IlcTrue` (its default value), then the cursor starts at the first interval, that is, the interval with the earliest start time. Otherwise, the cursor starts with the last interval, that is, the interval with the latest start time.

Methods

`public IlcInt` **`getDuration`**`() const`

This member function returns the duration of the current interval, the one to which the invoking cursor points.

`public IlcInt` **`getEnd`**`() const`

This member function returns the end time of the current interval, the one to which the invoking cursor points.

`public IlcInt` **`getStart`**`() const`

This member function returns the start time of the current interval, the one to which the invoking cursor points.

`public IlcInt` **`getType`**`() const`

This member function returns the type of the current interval.

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if there is a current interval and the invoking cursor points to it. Otherwise, it returns `IlcFalse`.

`public IlcIntervalListCursor &` **`operator++`**`()`

This operator shifts the cursor to the next interval.

`public IlcIntervalListCursor &` **`operator--`**`()`

This operator shifts the cursor back to the previous interval.

Class IlcIntTimetable

Definition file: ilsched/timetabh.h
Include file: <ilsched/ilsched.h>

An instance of the handle class `IlcIntTimetable` represents a *capacity timetable*. In Scheduler Engine, capacity timetables are used to manage the capacity of resources, both the capacity already used and the capacity remaining.

A timetable is defined over an *interval*, `[timeMin, timeMax)`, where `timeMin` is the origin of the timetable and `timeMax` is its horizon. In addition to the origin and horizon, you may optionally indicate the *period* of the timetable. The period must be a positive integer, and furthermore, the size of the interval (that is, `timeMax - timeMin`) must be an integer multiple of the period. If a period is specified, then the values managed by the timetable can change only at times indicated by `timeMin + i * period`.

For each point in time in its interval, a capacity timetable retains a minimal and maximal integer value. The minimal value retained for a point in time indicates the *capacity* that has already been used; the difference between the minimal and maximal value indicates the remaining capacity at that time.

Member functions of this class let you consult or modify these minimal and maximal values. These values may change only monotonically; that is, minimal values cannot decrease, and maximal values cannot increase.

Two types of propagation events can be triggered when a capacity timetable is modified. An event of type `rangeInterval` indicates that there are some times at which some modification of minimal or maximal values occurred. An event of type `valueInterval` indicates that there are some times at which the minimal value became equal to the maximal value. In order to perform propagation, member functions allow you to associate demons with each type of event.

The information stored into a timetable is *reversible*. In particular, when modifiers are called, the state before their call will be saved by Solver.

For more information, see Timetable.

See Also: IlcAnyTimetable, IlcCapResource, IlcIntTimetableCursor, IlcIntTimetableIterator

Constructor Summary	
public	`IlcIntTimetable()`
public	`IlcIntTimetable(IlcCapTimetableI * impl)`
public	`IlcIntTimetable(IlcSchedule, IlcInt timeMin, IlcInt timeMax, IlcInt period=1, IlcInt min=IlcIntMin, IlcInt max=IlcIntMax)`

Method Summary	
public IlcCapTimetableI *	`getImpl() const`
public IlcInt	`getMax(IlcInt time) const`
public IlcInt	`getMaxMax(IlcInt timeMin, IlcInt timeMax) const`
public IlcInt	`getMaxMin(IlcInt timeMin, IlcInt timeMax) const`
public IlcInt	`getMin(IlcInt time) const`
public IlcInt	`getMinMax(IlcInt timeMin, IlcInt timeMax) const`
public IlcInt	`getMinMin(IlcInt timeMin, IlcInt timeMax) const`
public const char *	`getName() const`

public IlcAny	getObject() const
public IlcInt	getPeriod() const
public IlcInt	getRangeTimeMax() const
public IlcInt	getRangeTimeMin() const
public IloSolver	getSolver() const
public IloSolverI *	getSolverI() const
public IlcInt	getTimeMax() const
public IlcInt	getTimeMin() const
public IlcInt	getValue(IlcInt time) const
public IlcInt	getValueTimeMax() const
public IlcInt	getValueTimeMin() const
public IlcBool	isBound(IlcInt time) const
public void	operator=(const IlcIntTimetable & h)
public void	setMax(IlcInt timeMin, IlcInt timeMax, IlcInt max)
public void	setMin(IlcInt timeMin, IlcInt timeMax, IlcInt min)
public void	setName(const char * name) const
public void	setObject(IlcAny object) const
public void	setValue(IlcInt timeMin, IlcInt timeMax, IlcInt value)
public void	whenRangeInterval(const IlcDemon c) const
public void	whenValueInterval(const IlcDemon c) const

Constructors

`public` **`IlcIntTimetable`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcIntTimetable`**`(IlcCapTimetableI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IlcIntTimetable`**`(IlcSchedule, IlcInt timeMin, IlcInt timeMax, IlcInt period=1, IlcInt min=IlcIntMin, IlcInt max=IlcIntMax)`

This constructor creates a timetable to manage an integer quantity (the capacity) whose value at all times is bounded between `min` and `max`. The constructor adds that timetable to those managed by the schedule. The timetable starts at `timeMin` and extends to `timeMax`, divided into equal periods of size `period`.

An instance of `IloSolver::SolverErrorException` is thrown if any of the following conditions occur:

- `timeMax - timeMin` is not strictly positive;
- `period` is not strictly positive;
- `timeMax - timeMin` is not an integer multiple of `period`.

Methods

`public IlcCapTimetableI * ` **`getImpl`**`() const`

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcInt getMax(IlcInt time) const
```

This member function returns the maximum value at `time` of the invoking timetable. The difference between this maximum and the minimum value at `time` (returned by the member function `IlcIntTimetable::getMin`) indicates the remaining capacity at `time`.

```
public IlcInt getMaxMax(IlcInt timeMin, IlcInt timeMax) const
```

This member function returns the largest of the maximal values of the integer quantity managed by the invoking timetable. Only those maximal values that correspond to times in the interval `[timeMin timeMax)` are considered. An error ("`bad index interval`") is raised if the invoking timetable does not cover the interval `[timeMin, timeMax)` or if `timeMin` is strictly greater than `timeMax`.

```
public IlcInt getMaxMin(IlcInt timeMin, IlcInt timeMax) const
```

This member function returns the largest of the minimal values of the integer quantity managed by the invoking timetable. Only those minimal values that correspond to times in the interval `[timeMin timeMax)` are considered. An error ("`bad index interval`") is raised if the invoking timetable does not cover the interval `[timeMin, timeMax)` or if `timeMin` is strictly greater than `timeMax`.

```
public IlcInt getMin(IlcInt time) const
```

This member function returns the minimum value at `time` of the invoking timetable. This minimum value at `time` indicates the capacity that has already been used at `time`.

```
public IlcInt getMinMax(IlcInt timeMin, IlcInt timeMax) const
```

This member function returns the least of the maximal values of the integer quantity managed by the invoking timetable. Only those maximal values that correspond to times in the interval `[timeMin timeMax)` are considered. An error ("`bad index interval`") is raised if the invoking timetable does not cover the interval `[timeMin, timeMax)` or if `timeMin` is strictly greater than `timeMax`.

```
public IlcInt getMinMin(IlcInt timeMin, IlcInt timeMax) const
```

This member function returns the least of the minimal values of the integer quantity managed by the invoking timetable. Only those minimal values that correspond to times in the interval `[timeMin timeMax)` are considered. An error ("`bad index interval`") is raised if the invoking timetable does not cover the interval `[timeMin, timeMax)` or if `timeMin` is strictly greater than `timeMax`.

```
public const char * getName() const
```

This member function returns the name of the invoking object.

```
public IlcAny getObject() const
```

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

```
public IlcInt getPeriod() const
```

This member function returns the size of the periods of the invoking timetable. The meaning of this size is that the timetable may change only at times representing the beginning of periods, that is, times of the form `(getTimeMin() + i * getPeriod())`.

```
public IlcInt getRangeTimeMax() const
```

When it is called during the execution of a demon associated with a timetable by the member function `IlcIntTimetable::whenRangeInterval`, this member function returns the time `rangeTimeMax`, that is, the maximum of the interval `[rangeTimeMin, rangeTimeMax)` containing all the times at which a modification of the values occurred. The return value of this member function is not meaningful outside the execution of a demon associated with the timetable by the member function `IlcIntTimetable::whenRangeInterval`.

```
public IlcInt getRangeTimeMin() const
```

When it is called during the execution of a demon associated with a timetable by the member function `IlcIntTimetable::whenRangeInterval`, this member function returns the time `rangeTimeMin`, that is, the minimum of the interval `[rangeTimeMin, rangeTimeMax)` containing all the times at which a modification of the values occurred. The return value of this member function is not meaningful outside the execution of a demon associated with the timetable by the member function `IlcIntTimetable::whenRangeInterval`.

```
public IloSolver getSolver() const
```

This member function returns an instance of `IloSolver` associated with the invoking object.

```
public IloSolverI * getSolverI() const
```

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

```
public IlcInt getTimeMax() const
```

This member function returns the time horizon of the invoking timetable.

```
public IlcInt getTimeMin() const
```

This member function returns the time origin of the invoking timetable.

```
public IlcInt getValue(IlcInt time) const
```

This member function returns the value of the invoking timetable at `time`. An instance of `IloSolver::SolverErrorException` is thrown if the timetable is not bound at `time`.

```
public IlcInt getValueTimeMax() const
```

When it is called during the execution of a demon associated with a timetable by the member function `IlcIntTimetable::whenValueInterval`, this member function returns the time `valueTimeMax`, that is, the maximum of the interval `[valueTimeMin, valueTimeMax)` containing all the times at which the minimal value has become equal to the maximal value. The return value of this member function is not meaningful outside the execution of a demon associated with the timetable by the member function `IlcIntTimetable::whenValueInterval`.

```
public IlcInt getValueTimeMin() const
```

When it is called during the execution of a demon associated with a timetable by the member function `IlcIntTimetable::whenValueInterval`, this member function returns the time `valueTimeMin`, that is, the minimum of the interval `[valueTimeMin, valueTimeMax)` containing all the times at which the minimal value has become equal to the maximal value. The return value of this member function is not meaningful outside the execution of a demon associated with the timetable by the member function `IlcIntTimetable::whenValueInterval`.

```
public IlcBool isBound(IlcInt time) const
```

This member function returns `IlcTrue` if the invoking timetable is bound to a value at `time`. Otherwise, it returns `IlcFalse`.

```
public void operator=(const IlcIntTimetable & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public void setMax(IlcInt timeMin, IlcInt timeMax, IlcInt max)
```

This member function allows you to modify the maximal values of the integer quantity managed by the invoking timetable. At every time `t`, the new maximal value becomes `max`, provided that `max` is less than the current maximal value of `t`. The maximal value of `t` remains unchanged if `max` is greater than the current maximal value of `t`. A failure is generated if there is some time `t` for which `max` is strictly less than the minimal value of `t`. The maximal values are modified only for the times belonging to the interval `[timeMin, timeMax)`.

An instance of `IloSolver::SolverErrorException` is thrown (`bad index interval`) if the invoking timetable does not cover the interval `[timeMin, timeMax)` or if `timeMin` is strictly greater than `timeMax`.

```
public void setMin(IlcInt timeMin, IlcInt timeMax, IlcInt min)
```

This member function allows you to modify the minimal values of the integer quantity managed by the invoking timetable. At every time `t`, the new minimal value becomes `min`, provided that `min` is greater than the current minimal value in `t`. The minimal value in `t` remains unchanged if `min` is less than the current minimal value of `t`. A failure is generated if there is some time `t` for which `min` is strictly greater than the maximal value of `t`. The minimal values are modified only for the times belonging to the interval `[timeMin, timeMax)`.

An instance of `IloSolver::SolverErrorException` is thrown (`bad index interval`) if the invoking timetable does not cover the interval `[timeMin, timeMax)` or if `timeMin` is strictly greater than `timeMax`.

```
public void setName(const char * name) const
```

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

```
public void setObject(IlcAny object) const
```

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

```
public void setValue(IlcInt timeMin, IlcInt timeMax, IlcInt value)
```

This member function allows you to set the value of the integer quantity managed by the invoking timetable. Then for every value `t` on the interval `[timeMin, timeMax)`, both the minimal and the maximal values in `t` become equal to `value`. A failure is generated if there is some time `t` for which `value` is strictly less than the current minimal value of `t` or strictly greater than the current maximal value of `t`.

An instance of `IloSolver::SolverErrorException` is thrown (`bad index interval`) if the given times `timeMin` and `timeMax` do not belong to the interval of the invoking timetable or if `timeMin` is strictly greater than `timeMax`.

```
public void whenRangeInterval(const IlcDemon c) const
```

This member function associates the demon `d` with the `rangeInterval` propagation event of the invoking timetable. Whenever a `rangeInterval` propagation event occurs, demon `d` is executed.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever a `rangeInterval` propagation event occurs, the constraint will be posted and propagated.

A call to the demon `d` signifies that there are *some* times at which a modification of the values occurred. The interval `[rangeTimeMin, rangeTimeMax)` is the least interval containing all these times.

```
public void whenValueInterval(const IlcDemon c) const
```

This member function associates the demon `d` with the `valueInterval` propagation event of the invoking timetable. Whenever a `valueInterval` propagation event occurs, the demon `d` is executed.

Since a constraint is also a demon, a constraint can also be passed as an argument to this member function. Whenever a `valueInterval` propagation event occurs, the constraint will be posted and propagated.

A call to the demon `d` signifies that there are *some* times at which the minimal value became equal to the maximal value. The interval `[valueTimeMin, valueTimeMax)` is the least interval containing all these times.

Class IlcIntTimetableCursor

Definition file: ilsched/timetabh.h
Include file: <ilsched/ilsched.h>

Objects of the class `IlcIntTimetableCursor` allow you to inspect the contents of integer timetables. A *region* of an integer timetable is a subinterval `[timeMin, timeMax)` of the interval where the timetable is defined such that all the times in the region share the same information and any two adjacent regions store different information. *Cursors* are intended to iterate forward or backward over the regions of an integer timetable.

Note

The structure of a timetable cannot be changed while a cursor is being used to inspect the timetable. Therefore functions that change the structure of the timetable should not be called while the cursor is being used; for example, `IlcIntTimetable::setMax`.

See Also: IlcIntTimetable

Constructor and Destructor Summary	
public	`IlcIntTimetableCursor(const IlcIntTimetable table, IlcInt time)`

Method Summary	
public IlcInt	`getMax() const`
public IlcInt	`getMin() const`
public IlcInt	`getTimeMax() const`
public IlcInt	`getTimeMin() const`
public IlcInt	`getValue() const`
public IlcBool	`isBound() const`
public IlcBool	`ok() const`
public void	`operator++()`
public void	`operator--()`

Constructors and Destructors

`public` **`IlcIntTimetableCursor`**`(const IlcIntTimetable table, IlcInt time)`

This constructor creates a cursor to inspect the information stored in the integer timetable `table`. This cursor lets you to iterate forward or backward over the regions composing the timetable. The cursor initially indicates the region containing `time`.

Methods

`public IlcInt` **`getMax`**`() const`

This member function returns the maximal possible value of the current region. All the times in the current region share the same maximal value.

```
public IlcInt getMin() const
```

This member function returns the minimal possible value of the current region. All the times in the current region share the same minimal value.

```
public IlcInt getTimeMax() const
```

This member function returns the time ending the region currently indicated by the cursor.

```
public IlcInt getTimeMin() const
```

This member function returns the time beginning the region currently indicated by the cursor.

```
public IlcInt getValue() const
```

This member function returns the value of the region that the invoking timetable cursor indicates. An instance of `IloSolver::SolverErrorException` is thrown if the timetable is not bound at the cursor position.

```
public IlcBool isBound() const
```

This member function returns `IlcTrue` if the region indicated by the invoking timetable cursor has been bound; that is, its maximum value is equal to its minimum value. Otherwise, it returns `IlcFalse`.

```
public IlcBool ok() const
```

This member function returns `IlcFalse` if the cursor does not currently indicate a region included in the timetable. Otherwise, it returns `IlcTrue`. Any attempt to use the cursor after `ok()` returns `IlcFalse` could lead to undefined behavior.

```
public void operator++()
```

This operator moves the cursor to the region adjacent "on the right" to the current region (forward iteration).

```
public void operator--()
```

This operator moves the cursor to the region adjacent "on the left" to the current region (backward iteration).

Class IlcIntTimetableIterator

Definition file: ilsched/capacity.h
Include file: <ilsched/ilsched.h>

An instance of this class traverses the set of timetables associated with an integer capacity resource (`IlcDiscreteResource`, `IlcUnaryResourceIlcReservoir`, or `IlcDiscreteEnergy`).

See Also: IlcIntTimetable, IlcSchedule

Constructor Summary	
`public`	`IlcIntTimetableIterator(IlcCapResource res)`

Method Summary	
`public IlcIntTimetable`	`operator*()`
`public IlcIntTimetableIterator &`	`operator++()`

Constructors

`public` **`IlcIntTimetableIterator`**`(IlcCapResource res)`

This constructor creates an iterator to traverse all the timetables of a capacity resource.

Methods

`public IlcIntTimetable` **`operator*`**`()`

This operator accesses the instance of `IlcIntTimetable` located at the current position of the iterator. If the iterator is set past the end position, this operator returns an empty handle.

`public IlcIntTimetableIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator.

Class IlcIntToFloatSegmentFunction

Definition file: ilsched/segfunc.h
Include file: <ilsched/ilsched.h>

An instance of `IlcIntToFloatSegmentFunction` represents a continuous or discontinuous piecewise linear function over integers on an interval [`xmin`, `xmax`). The member functions of the class `IlcIntToFloatSegmentFunction` are not reversible.

See Also: IlcIntToFloatSegmentFunctionCursor

Constructor Summary	
public	`IlcIntToFloatSegmentFunction()`
public	`IlcIntToFloatSegmentFunction(IlcSegmentedFunctionI * impl)`
public	`IlcIntToFloatSegmentFunction(IlcManager m, IlcInt xmin=IloIntMin, IlcInt xmax=IloIntMax, IlcFloat dval=0.)`
public	`IlcIntToFloatSegmentFunction(IlcManager m, IlcIntArray x, IlcFloatArray v, IlcInt xmin=IloIntMin, IlcInt xmax=IloIntMax)`
public	`IlcIntToFloatSegmentFunction(const IlcIntToIntStepFunction & f)`

Method Summary	
public void	`addValue(IlcInt x1, IlcInt x2, IlcFloat v)`
public void	`dilate(IlcInt k)`
public IlcFloat	`getArea(IlcInt x1, IlcInt x2) const`
public IlcInt	`getDefinitionIntervalMax() const`
public IlcInt	`getDefinitionIntervalMin() const`
public IlcSegmentedFunctionI *	`getImpl() const`
public IlcFloat	`getMax(IlcInt x1, IlcInt x2) const`
public IlcFloat	`getMin(IlcInt x1, IlcInt x2) const`
public const char *	`getName() const`
public IlcAny	`getObject() const`
public IloSolver	`getSolver() const`
public IloSolverI *	`getSolverI() const`
public IlcFloat	`getValue(IlcInt x) const`
public void	`operator*=(IlcFloat k)`
public void	`operator+=(const IlcIntToFloatSegmentFunction & fct)`
public void	`operator-=(const IlcIntToFloatSegmentFunction & fct)`
public void	`operator=(const IlcIntToFloatSegmentFunction & h)`
public void	`setName(const char * name) const`
public void	`setObject(IlcAny object) const`
public void	`setPeriodic(const IlcIntToFloatSegmentFunction & f, IlcInt x0, IlcInt n=IlcIntMax, IlcFloat dval=0)`

public void	setPoints(IlcIntArray x, IlcFloatArray v)
public void	setValue(IlcInt x1, IlcFloat v1, IlcInt x2, IlcFloat v2)
public void	setValue(IlcInt x1, IlcInt x2, IlcFloat v)
public void	shift(IlcInt dx, IlcFloat dval=0)

Constructors

`public` **`IlcIntToFloatSegmentFunction`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcIntToFloatSegmentFunction`**`(IlcSegmentedFunctionI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IlcIntToFloatSegmentFunction`**`(IlcManager m, IlcInt xmin=IloIntMin, IlcInt xmax=IloIntMax, IlcFloat dval=0.)`

This constructor creates an integer piecewise linear function defined everywhere on the interval `[xmin,xmax)` with the same value `dval`.

`public` **`IlcIntToFloatSegmentFunction`**`(IlcManager m, IlcIntArray x, IlcFloatArray v, IlcInt xmin=IloIntMin, IlcInt xmax=IloIntMax)`

This constructor creates an integer piecewise linear function defined everywhere on the interval `[xmin,xmax)`; its steps are defined by the two arrays of parameters, `x` and `v`. More precisely, the size `n` of array `x` must be equal to the size of array `v` and, if the created function is defined on the interval `[xmin,xmax)`, its values will be:

- `v[0]` on interval `[xmin,x[0])`,
- `v[i]+ (t-x[i])(v[i+1]-v[i])/(x[i+1]-x[i])` for `t` in `[x[i],x[i+1])` for all `i` in `[0,n-2]` such that `x[i-1] <> x[i]`, and
- `v[n-1]` on interval `[x[n-1],xmax)`.

`public` **`IlcIntToFloatSegmentFunction`**`(const IlcIntToIntStepFunction & f)`

This copy constructor creates a new piecewise linear function. This new piecewise linear function is a copy of the integer step function `f`. They point to different implementation objects.

Methods

`public void` **`addValue`**`(IlcInt x1, IlcInt x2, IlcFloat v)`

This member function increases the value of the invoking piecewise linear function by `v` everywhere on the interval `[x1,x2)`.

`public void` **`dilate`**`(IlcInt k)`

This member function multiplies the scale of `x` by `k` for the invoking piecewise linear function. The parameter `k` must be a positive integer.

More precisely, if the invoking function was defined over an interval `[xmin,xmax)`, it will be redefined over the interval `[k*xmin,k*xmax)` and the value at `x` will be the former value at `x/k`.

```
public IlcFloat getArea(IlcInt x1, IlcInt x2) const
```

This member function returns the area of the invoking piecewise linear function over the interval `[x1,x2)`. If the interval `[x1,x2)` is not included in the interval of definition of the invoking function, an error will be raised.

```
public IlcInt getDefinitionIntervalMax() const
```

This member function returns the right-most point of the interval of definition of the invoking piecewise linear function.

```
public IlcInt getDefinitionIntervalMin() const
```

This member function returns the left-most point of the interval of definition of the invoking piecewise linear function.

```
public IlcSegmentedFunctionI * getImpl() const
```

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcFloat getMax(IlcInt x1, IlcInt x2) const
```

This member function returns the maximal value of the invoking piecewise linear function on the interval `[x1,x2)`. If the interval `[x1,x2)` is not included in the definition interval of the invoking function, an error will be raised.

```
public IlcFloat getMin(IlcInt x1, IlcInt x2) const
```

This member function returns the minimal value of the invoking piecewise linear function on the interval `[x1,x2)`. If the interval `[x1,x2)` is not included in the definition interval of the invoking function, an error will be raised.

```
public const char * getName() const
```

This member function returns the name of the invoking object.

```
public IlcAny getObject() const
```

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

```
public IloSolver getSolver() const
```

This member function returns an instance of `IloSolver` associated with the invoking object.

```
public IloSolverI * getSolverI() const
```

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

```
public IlcFloat getValue(IlcInt x) const
```

This member function returns the value of the invoking piecewise linear function at `x`. If `x` does not belong to the definition interval of the invoking function, an error will be raised.

```
public void operator*=(IlcFloat k)
```

This operator multiplies the value of the invoking integer step function by a factor `k` everywhere on the interval of definition.

```
public void operator+=(const IlcIntToFloatSegmentFunction & fct)
```

This operator adds the parameter function `fct` to the invoking piecewise linear function.

```
public void operator-=(const IlcIntToFloatSegmentFunction & fct)
```

This operator subtracts the parameter function `fct` from the invoking piecewise linear function.

```
public void operator=(const IlcIntToFloatSegmentFunction & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public void setName(const char * name) const
```

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

```
public void setObject(IlcAny object) const
```

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

```
public void setPeriodic(const IlcIntToFloatSegmentFunction & f, IlcInt x0, IlcInt
n=IlcIntMax, IlcFloat dval=0)
```

This member function initializes the invoking function as a piecewise linear function that repeats the piecewise linear function `f`, `n` times after `x0`.

More precisely, if `f` is defined on `[xfmin,xfmax)` and if the invoking function is defined on `[xmin,xmax)`, the value of the invoking function will be:

- `dval` on `[xmin, x0)`,
- `f((x-x0) % (xfmax-xfmin))` for `x` in `[x0, Min(x0+n*(xfmax-xfmin), xmax))`, and
- `dval` on `[Min(x0+n*(xfmax-xfmin), xmax), xmax)`

```
public void setPoints(IlcIntArray x, IlcFloatArray v)
```

This member function initializes the invoking function as a piecewise linear function whose segments are defined by the two parameter arrays `x` and `v`.

More precisely, the size `n` of array `x` must be equal to the size of array `v` and, if the created function is defined on the interval `[xmin,xmax)`, its values will be:

- `v[0]` on interval `[xmin,x[0])`,
- `v[i]+ (t-x[i])(v[i+1]-v[i])/(x[i+1]-x[i])` for `t` in `[x[i],x[i+1])` for all `i` in `[0,n-2]` such that `x[i-1] <> x[i]`, and
- `v[n-1]` on interval `[x[n-1],xmax)`.

```
public void setValue(IlcInt x1, IlcInt x2, IlcFloat v)
public void setValue(IlcInt x1, IlcFloat v1, IlcInt x2, IlcFloat v2)
```

This member function sets the value of the invoking piecewise linear function to be `v` on the interval `[x1,x2)`.

```
public void shift(IlcInt dx, IlcFloat dval=0)
```

This member function shifts the invoking function from `dx` to the right if `dx` > 0 or from `-dx` to the left if `dx` < 0. It has no effect if `dx` = 0.

More precisely, if the invoking function is defined on `[xmin,xmax)` and `dx` > 0, the new value of the invoking function is:

- `dval` on the interval `[xmin,xmin+dx)`,
- for all `x` in `[xmin+dx,xmax)`, the former value at `x-dx`.

If `dx` < 0, the new value of the invoking function is:

- for all `x` in `[xmin,xmax+dx)`, the former value at `x-dx`,
- `dval` on the interval `[xmax+dx,xmax)`.

Class IlcIntToFloatSegmentFunctionCursor

Definition file: ilsched/segfunc.h
Include file: <ilsched/ilsched.h>

An instance of the class `IlcIntToFloatSegmentFunctionCursor` allows you to inspect the contents of a piecewise linear function. A segment of an instance of `IlcIntToFloatSegmentFunction` is defined as an interval `[x1, x2)` over which the function is linear. Cursors iterate forward or backward over the segments of a piecewise linear function.

Note

The structure of the piecewise linear function must not be changed while a cursor is inspecting it. Therefore functions that change the structure of the piecewise linear function, such as `IlcIntToFloatSegmentFunction::setValue` , should not be called while a cursor is in use.

See Also: IlcIntToFloatSegmentFunction

Constructor and Destructor Summary	
public	`IlcIntToFloatSegmentFunctionCursor(const IlcIntToFloatSegmentFunction &, IlcInt x)`

Method Summary	
public IlcInt	`getSegmentMax() const`
public IlcInt	`getSegmentMin() const`
public IlcFloat	`getValue(IlcInt t) const`
public IlcFloat	`getValueLeft() const`
public IlcFloat	`getValueRight() const`
public IlcBool	`ok() const`
public void	`operator++()`
public void	`operator--()`

Constructors and Destructors

`public` **`IlcIntToFloatSegmentFunctionCursor`**`(const IlcIntToFloatSegmentFunction &, IlcInt x)`

This constructor creates a cursor to inspect the piecewise linear function argument. This cursor lets you iterate forward or backward over the segments of the function. The cursor initially indicates the segment of the function that contains `x`.

Methods

`public IlcInt` **`getSegmentMax`**`() const`

This member function returns the right-most point of the segment currently indicated by the cursor.

```
public IlcInt getSegmentMin() const
```

This member function returns the left-most point of the segment currently indicated by the cursor.

```
public IlcFloat getValue(IlcInt t) const
```

This member function returns the value of the function at time `t`. `t` must be inside the segment currently indicated by the cursor, that is in time interval `[getSegmentMin(), getSegmentMax())`. An instance of `IloSolver::SolverErrorException` is thrown otherwise.

```
public IlcFloat getValueLeft() const
```

This member function returns the value of the function at the left-most point of the segment currently indicated by the cursor.

```
public IlcFloat getValueRight() const
```

This member function returns the value of the function at the right-most point of the segment currently indicated by the cursor.

```
public IlcBool ok() const
```

This member function returns `IlcFalse` if the cursor does not currently indicate a segment included in the interval of definition of the piecewise linear function. Otherwise, it returns `IlcTrue`. An attempt to use the cursor after `ok` returns `IlcFalse` leads to undefined behavior.

```
public void operator++()
```

This operator moves the cursor to the segment adjacent to the current segment (forward move).

```
public void operator--()
```

This operator moves the cursor to the segment adjacent to the current segment (backward move).

Class IlcPossibleAltResIterator

Definition file: ilsched/altresh.h
Include file: <ilsched/ilsched.h>

An instance of the class `IlcPossibleAltResIterator` is an iterator that traverses the possible resources (those which could be selected) for an instance of `IlcAltResConstraint`.

If a resource has been *selected*, then there is one and only one *possible* resource among those managed by the constraint.

To make the search for a scheduling solution efficient, it is often a good idea to select one resource for the activity for which an instance of `IlcAltResSet` is defined before you schedule it.

See Also: IlcAltResConstraint, IlcResource

Constructor and Destructor Summary	
public	IlcPossibleAltResIterator(IlcAltResConstraint ct)

Method Summary	
public IlcBool	ok() const
public IlcResource	operator*() const
public IlcPossibleAltResIterator &	operator++()

Constructors and Destructors

`public` **`IlcPossibleAltResIterator`**`(IlcAltResConstraint ct)`

This constructor creates a new instance of `IlcPossibleAltResIterator` that traverses the set of possible resources which belong to `ct`.

Methods

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if there is a current possible resource and the invoking iterator points to it. Otherwise, it returns `IlcFalse`.

`public IlcResource` **`operator*`**`() const`

This operator returns the current possible resource to which the invoking iterator points. If the iterator is set past the end position, then this operator returns an empty handle.

`public IlcPossibleAltResIterator &` **`operator++`**`()`

This operator shifts the iterator to the next possible resource.

Class IlcPrecedenceConstraint

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>

Instances of the class `IlcPrecedenceConstraint` are *temporalconstraints*. These temporal constraints express precedence between activities in a schedule. (Other temporal constraints, such as instances of `IlcTimeBoundConstraint`, express constraints on the time interval in which an activity is to be scheduled.)

This class inherits from the Solver class `IlcConstraint`. That class is documented in the *IBM ILOG Solver Reference Manual*.

Instances of this class are created by these member functions:

- `IlcActivity::startsAfterStart`
- `IlcActivity::startsAfterEnd`
- `IlcActivity::endsAfterStart`
- `IlcActivity::endsAfterEnd`
- `IlcActivity::startsAtStart`
- `IlcActivity::startsAtEnd`
- `IlcActivity::endsAtStart`
- `IlcActivity::endsAtEnd`

For more information, see Metaconstraints, and `IlcConstraint` in the *IBM ILOG Solver Reference Manual*.

See Also: IlcActivity, IlcPrecedenceConstraintType, IlcTimeBoundConstraint

Constructor Summary	
`public`	`IlcPrecedenceConstraint()`
`public`	`IlcPrecedenceConstraint(IlcPrecedenceConstraintI * impl)`

Method Summary	
`public IlcInt`	`getDelay() const`
`public IlcIntVar`	`getDelayVariable() const`
`public IlcActivity`	`getFollowingActivity() const`
`public IlcPrecedenceConstraintI *`	`getImpl() const`
`public IlcActivity`	`getPrecedingActivity() const`
`public IlcPrecedenceConstraintType`	`getType() const`
`public IlcBool`	`hasDelayVariable() const`
`public void`	`operator=(const IlcPrecedenceConstraint & h)`

Constructors

`public` **`IlcPrecedenceConstraint`**`()`

This constructor creates an instance which is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

`public` **`IlcPrecedenceConstraint`**`(IlcPrecedenceConstraintI * impl)`

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

Methods

```
public IlcInt getDelay() const
```

This member function returns the delay of the invoking precedence constraint.

Example

The statement

```
m.add(act2.startsAfterEnd(act1, delay));
```

posts the constraint that at least the given `delay` must elapse between the end of the preceding activity `act1` and the start of the following activity `act2`.

```
public IlcIntVar getDelayVariable() const
```

This member function returns the delay variable of the invoking precedence constraint.

```
public IlcActivity getFollowingActivity() const
```

This member function returns the following activity of the invoking precedence constraint.

```
public IlcPrecedenceConstraintI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcActivity getPrecedingActivity() const
```

This member function returns the preceding activity of the invoking precedence constraint.

```
public IlcPrecedenceConstraintType getType() const
```

This member function returns the type of the invoking precedence constraint.

```
public IlcBool hasDelayVariable() const
```

This member function returns `IlcTrue` if the invoking precedence constraint has a delay variable. Otherwise, it returns `IlcFalse`.

```
public void operator=(const IlcPrecedenceConstraint & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument. After execution of this operator, the invoking object and the provided argument point to the same implementation object.

Class IlcPrecedingActivityIterator

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>

When used during the search, an instance of this class traverses the activities that are such that a precedence constraint holds between this activity and the activity given in the constructor of the iterator. If used before entering the search, this iterator will traverse an empty list of activities.

This class of iterators does not distinguish among the precedence constraints created by means of the member functions `IlcActivity::startsAfterStart`, `IlcActivity::startsAfterEnd`, `IlcActivity::endsAfterStart`, `IlcActivity::endsAfterEnd`, `IlcActivity::startsAtStart`, `IlcActivity::startsAtEnd`, `IlcActivity::endsAtStart`, and `IlcActivity::endsAtEnd`. Whichever of those member functions is used to create the precedence constraint, the invoking activity is identified as the one that is constrained to occur after the activity passed as an argument to the member function. Likewise, the activity passed as an argument to the member function is identified as the one that is constrained to occur before the invoking activity. See the example in the class `IlcFollowingActivityIterator` for a program that illustrates that idea.

See Also: IlcActivity, IlcFollowingActivityIterator, IlcPrecedenceConstraint

Constructor and Destructor Summary	
public	`IlcPrecedingActivityIterator(IlcActivity activity, IlcResourceConstraintIteratorFilter filter=IlcAllConstraints)`

Method Summary	
`public IlcPrecedenceConstraint`	`getPrecedenceConstraint() const`
`public IlcBool`	`ok() const`
`public IlcActivity`	`operator*() const`
`public IlcPrecedingActivityIterator &`	`operator++()`

Constructors and Destructors

`public` **`IlcPrecedingActivityIterator`**`(IlcActivity activity, IlcResourceConstraintIteratorFilter filter=IlcAllConstraints)`

This constructor creates an iterator to traverse all the activities constrained to precede `activity`.

Methods

`public IlcPrecedenceConstraint` **`getPrecedenceConstraint`**`() const`

This member function returns the precedence constraint between the activity used by the constructor of the invoking iterator and the activity at its current position.

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if the current position of the iterator is a valid one. It returns `IlcFalse` if all the activities have been scanned by the iterator.

```
public IlcActivity operator*() const
```

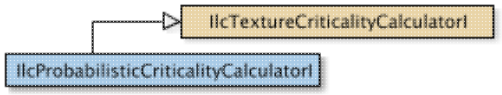
This dereference operator accesses the instance of `IlcActivity` located at the current position of the iterator. If the iterator is set past the end position, then this operator returns an empty handle.

```
public IlcPrecedingActivityIterator & operator++()
```

This left-increment operator shifts the current position of the iterator.

Class IlcProbabilisticCriticalityCalculatorI

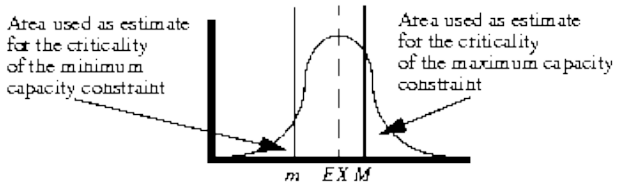
Definition file: ilsched/texturei.h
Include file: <ilsched/ilsched.h>



`IlcProbabilisticCriticalityCalculatorI` is the implementation class that implements the probabilistic texture criticality calculator.

Example

The probabilistic criticality calculation assumes that the aggregate demand and variance can be represented by a normal distribution parameterized by the expected demand (*EX*, in the following figure) and variance. The criticality for a minimum constraint, *m*, is calculated as the fraction of the area under the curve to the left of the *m*-value. Similarly, the criticality for a maximum constraint, *M*, is the fraction of the area under the curve to the right of *M*.



For more information, see Texture Measurements.

See Also: IlcResourceTexture, IlcTextureCriticalityCalculator, IlcTextureCriticalityCalculatorI, IlcRelativeDemandCriticalityCalculatorI

Constructor and Destructor Summary	
`public`	`IlcProbabilisticCriticalityCalculatorI()`

Method Summary	
`public virtual IlcFloat`	`calculateCriticalityGreaterThan(IlcFloat expectedDemand, IlcFloat constraintVal, IlcFloat expectedVariance) const`
`public virtual IlcFloat`	`calculateCriticalityLessThan(IlcFloat expectedDemand, IlcFloat constraintVal, IlcFloat expectedVariance) const`

Inherited Methods from `IlcTextureCriticalityCalculatorI`
`calculateCriticalityGreaterThan, calculateCriticalityLessThan`

Constructors and Destructors

`public` **`IlcProbabilisticCriticalityCalculatorI`**`()`

This constructor creates an instance of `IlcProbabilisticCriticalityCalculatorI`.

Methods

```
public virtual IlcFloat calculateCriticalityGreaterThan(IlcFloat expectedDemand,
IlcFloat constraintVal, IlcFloat expectedVariance) const
```

This method calculates the criticality of a maximum constraint at one time point based on a probabilistic estimation. This estimation assumes that the aggregate demand at a time point can be represented by a normal distribution with expected demand, demand and with variance, variance. The fraction of this distribution that lies to the right of `constraintVal`, is used as the criticality. This calculation is illustrated in the previous figure.

```
public virtual IlcFloat calculateCriticalityLessThan(IlcFloat expectedDemand,
IlcFloat constraintVal, IlcFloat expectedVariance) const
```

This method calculates the criticality of a minimum constraint at one time point based on a probabilistic estimation. This estimation assumes that the aggregate demand at a time point can be represented by a normal distribution with expected demand, demand and with variance, variance. The fraction of this distribution that lies to the left of constraintVal, is used as the criticality. This calculation is illustrated in the previous figure.

Class IlcRCTexture

Definition file: ilsched/texture.h
Include file: <ilsched/ilsched.h>

An instance of `IlcRCTexture` represents an individual texture curve for one resource constraint instance.

Individual curves for each resource constraint that can possibly be on a resource are aggregated to form aggregate curves represented by the `IlcResourceTexture` class.

For more information, see Texture Measurements.

See Also: IlcRCTextureIterator, IlcResourceTexture, IlcRCTextureI, IlcRCTextureFactoryI

Method Summary	
public IlcFloat	getCriticalContribution() const
public IlcResourceConstraint	getResourceConstraint() const
public IlcFloat	getTargetStart() const
public IlcBool	hasAlternatives() const
public void	setNeedToRecomputeCurve()
public void	setTargetStart(IlcFloat t) const

Methods

`public IlcFloat` **`getCriticalContribution`**`() const`

This member function returns the amount that this individual curve contributes to the aggregate curve at the critical time point.

`public IlcResourceConstraint` **`getResourceConstraint`**`() const`

This member function returns the resource constraint associated with the individual curve.

`public IlcFloat` **`getTargetStart`**`() const`

This member function returns the target start time as set by the IlcRCTexture::setTargetStart member function. If no target start time has been set, `IlcFloatMin` is returned.

`public IlcBool` **`hasAlternatives`**`() const`

This member function returns `IlcTrue` if the associated resource constraint *does not necessarily* have to be true. That is, it returns `IlcTrue` if the resource constraint is either part of an alternative resource constraint or the resource constraint is part of a meta-constraint. Otherwise, the function returns `IlcFalse`.

`public void` **`setNeedToRecomputeCurve`**`()`

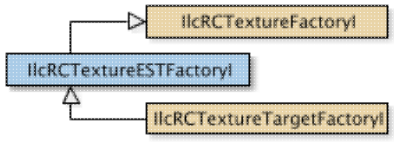
This member function forces the individual curve to be recomputed the next time the aggregate curve is updated. This function does not have to be explicitly called if the need to recompute the curve arises from changes to variables associated with the resource constraint or activity or to the target start time. Rather, this function is provided for the case where the individual texture curve is based on external user-defined data. When such data is "manually" changed, the `IlcRCTexture` object must be informed of the need to recompute the curve. Note that using this function generally implies that you have redefined the `IlcRCTextureI::updateDataPoints` method in your user-defined subclass of `IlcRCTextureI`.

```
public void setTargetStart(IlcFloat t) const
```

This member function allows the setting of the target start time for an instance of `IlcRCTexture`. The target start time of an `IlcRCTexture` is the preferred start time of the corresponding activity derived by some external source (such as user preferences or a cooperating solver). The target start time is relevant only if the implementation class (a subclass of `IlcRCTextureI`) of the invoking object calculates the individual curve using the target. Of the predefined subclasses, only `IlcRCTextureTargetI` uses the target start time. User-defined subclasses of `IlcRCTexture` may form their individual curve using the target start time

Class IlcRCTextureESTFactoryI

Definition file: ilsched/texturei.h
Include file: <ilsched/ilsched.h>



`IlcRCTextureESTFactoryI` is an implementation object that allocates instances of `IlcRCTextureESTI`.

For more information, see Texture Measurements.

See Also: IlcResourceTexture, IlcRCTextureI, IlcRCTextureESTI

Constructor and Destructor Summary	
`public`	`IlcRCTextureESTFactoryI(IloSolver solver)`

Method Summary	
`public virtual IlcRCTexture`	`createRCTexture(IlcResourceConstraint, IlcResourceTexture) const`

Inherited Methods from `IlcRCTextureFactoryI`
`createRCTexture, getSolver, hasRealZeroCriticality`

Constructors and Destructors

`public` **`IlcRCTextureESTFactoryI`**`(IloSolver solver)`

This constructor creates an instance of `IlcRCTextureESTFactoryI`.

Methods

`public virtual IlcRCTexture` **`createRCTexture`**`(IlcResourceConstraint, IlcResourceTexture) const`

This method returns a pointer to a newly allocated instance of `IlcRCTexture` representing the individual curve of the resource constraint for the resource associated with the resource texture.

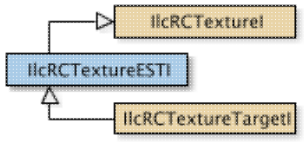
Example

The `createRCTexture()` method could be written as follows:

```
IlcRCTexture
IlcRCTextureESTFactoryI::createRCTexture(IlcResourceConstraint rct,
                  IlcResourceTexture texture) const {
  IloSolver solver = getSolver();
  return new (solver.getHeap())
    IlcRCTextureESTI(solver, rct, texture);
}
```

Class IlcRCTextureESTI

Definition file: ilsched/texturei.h
Include file: <ilsched/ilsched.h>



`IlcRCTextureESTI` is the implementation class for the earliest start time individual texture curve. This individual curve represents the individual demand of the associated resource constraint for its resource based on the assumption that it will start at its earliest start time. The points used in the texture curve can be seen in the example in the documentation for IlcRCTextureESTI::calculateIndividualCurve.

For more information, see Texture Measurements.

See Also: IlcRCTextureIterator, IlcResourceTexture, IlcRCTextureESTFactoryI, IlcRCTextureProbabilisticI, IlcRCTextureI, IlcRCTextureTargetI

Constructor and Destructor Summary	
protected	`IlcRCTextureESTI(IlcManager m, IlcResourceConstraint rc, IlcResourceTexture resTexture)`

Method Summary	
protected virtual void	`calculateIndividualCurve()`

Inherited Methods from `IlcRCTextureI`
`calcAltWeight, calculateIndividualCurve, getAltWeight, getCapacity, getDuration, getEnd, getResourceConstraint, getStart, getTargetStart, insertEvent, setAltWeight, setCapacity, setDuration, setEnd, setStart, setTargetStart, updateDataPoints`

Constructors and Destructors

`protected` **`IlcRCTextureESTI`**`(IlcManager m, IlcResourceConstraint rc, IlcResourceTexture resTexture)`

This protected constructor creates an instance of `IlcRCTextureESTI` for resource constraint `rc` and resource texture `resTexture`. The created class represents the individual contribution of `rc` to the aggregate curve of `resTexture`. As this constructor is protected, it should only be called from subclasses of `IlcRCTextureESTI` or from the `IlcRCTextureESTFactoryI` friend class.

Methods

`protected virtual void` **`calculateIndividualCurve`**`()`

This virtual, protected function creates the actual individual curve for the invoking object.

Example

Here is an example of how the `updateDataPoints()` and `calculateIndividualCurve()` methods might be written.

```
IlcBool IlcRCTextureESTI::updateDataPoints() {
  IlcResourceConstraint rc = getResourceConstraint();
  IlcFloat newAltWeight = calcAltWeight();
  IlcInt dur, cap, startMin;
  IlcAltResConstraint altRct = rc.getAlternative();
  if (0 != altRct.getImpl()) {
    IlcResource res = rc.getResource();
    startMin = altRct.getStartMin(res)
    dur = altRct.getDurationMax(res);
    cap = altRct.getCapacityMax(res);
  }
  else {
    IlcActivity act = rc.getActivity();
    startMin = act.getStartMin();
    dur = act.getDurationMax();
    cap = (rc.isVariableResourceConstraint() ?
          rc.getCapacityVariable().getMax() :
          rc.getCapacity());
  }
  IlcBool changed = IlcFalse;
  if ((getStart() != startMin) ||
      (getDuration() != dur) ||
      (getCapacity() != cap) ||
      (getAltWeight() != newAltWeight)) {
    changed = IlcTrue;
    setStart(startMin);
    setDuration(dur);
    setEnd(startMin + dur);
    setCapacity(cap);
    setAltWeight(newAltWeight);
  }
  return changed;
}
void IlcRCTextureESTI::calculateIndividualCurve() {
  IlcFloat demand = getCapacity() * getAltWeight();
```

```
    insertEvent(getStart(), demand, demand);
    insertEvent(getEnd(), 0, -demand);
  }
```

Class IlcRCTextureFactory

Definition file: ilsched/texture.h
Include file: <ilsched/ilsched.h>

`IlcRCTextureFactory` is the handle class for `IlcRCTextureFactoryI` and for its subclasses.

For more information, see Texture Measurements.

Predefined RCTexture Factories

These functions return instances of `IlcRCTextureFactory`.

- `IlcRCTextureFactory IlcRCTextureESTFactory (IloSolver);`
- `IlcRCTextureFactory IlcRCTextureProbabilisticFactory (IloSolver);`
- `IlcRCTextureFactory IlcRCTextureTargetFactory (IloSolver);`

See Also: IlcResourceTexture, IlcRCTextureFactoryI, IlcRCTexture, IlcRCTextureI, IlcRCTextureESTI, IlcRCTextureTargetI, IlcRCTextureProbabilisticI

Constructor Summary	
public	`IlcRCTextureFactory()`
public	`IlcRCTextureFactory(IlcRCTextureFactoryI * impl)`

Method Summary	
public IlcRCTexture	`createRCTexture(IlcResourceConstraint, IlcResourceTexture) const`
public IlcRCTextureFactoryI *	`getImpl() const`
public IlcBool	`hasRealZeroCriticality() const`
public void	`operator=(const IlcRCTextureFactory & h)`

Constructors

`public` **`IlcRCTextureFactory`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcRCTextureFactory`**`(IlcRCTextureFactoryI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public IlcRCTexture` **`createRCTexture`**`(IlcResourceConstraint, IlcResourceTexture) const`

This method returns a newly allocated instance of `IlcRCTexture`. This is the function used internally by the `IlcResourceTexture` object to generate an individual curve for each resource constraint on a resource.

```
public IlcRCTextureFactoryI * getImpl() const
```

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcBool hasRealZeroCriticality() const
```

This method defines a characteristic of the `IlcRCTexture` subclasses created by the invoking factory. If this function returns `IlcTrue`, it means that when the individual demand at some time point, *t*, for `IlcRCTextureI` instances allocated by the invoking factory is 0, it will remain at 0 until there is a backtrack in the search. Otherwise, if `IlcFalse` is returned, a time point may become zero and then some other value without backtracking.

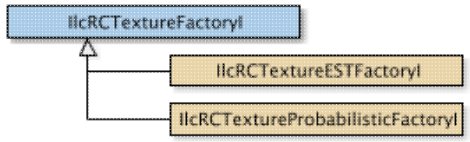
See `IlcRCTextureFactoryI` for more details about this method.

```
public void operator=(const IlcRCTextureFactory & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcRCTextureFactoryI

Definition file: ilsched/texturei.h
Include file: <ilsched/ilsched.h>



`IlcRCTextureFactoryI` is the abstract implementation base class for a class that creates an object that is a subclass `IlcRCTextureI`. Instances of subclasses of this class are passed (via a handle class, `IlcRCTextureFactory`) to an `IlcResourceTexture` and used internally whenever a new individual curve object is required. Whenever a user-defined subclass of `IlcRCTextureI` is defined, a corresponding `IlcRCTextureFactoryI` subclass must also be created. Its sole purpose is to allocate the `IlcRCTextureI` subclass.

For more information, see Texture Measurements.

See Also: IlcResourceTexture, IlcRCTextureI

Constructor and Destructor Summary	
public	`IlcRCTextureFactoryI(IloSolver solver)`

Method Summary	
public virtual IlcRCTexture	`createRCTexture(IlcResourceConstraint, IlcResourceTexture) const`
public IloSolver	`getSolver() const`
public virtual IlcBool	`hasRealZeroCriticality() const`

Constructors and Destructors

`public` **`IlcRCTextureFactoryI`**`(IloSolver solver)`

This constructor creates an instance of `IlcRCTextureFactoryI`. As the class is abstract, this constructor should only be called by the constructor of subclasses of `IlcRCTextureFactoryI`.

Methods

`public virtual IlcRCTexture` **`createRCTexture`**`(IlcResourceConstraint, IlcResourceTexture) const`

This pure virtual method returns a pointer to a newly allocated subclass of `IlcRCTextureI` which represents the individual curve of resource constraint `rct` for the resource associated with the resource texture `texture`. This is the function used internally by the `IlcResourceTexture` object to generate an individual curve for each resource constraint on a resource. The individual curve should be allocated on the solver heap. See the example code in the documentation of `IlcRCTextureESTFactoryI`.

`public IloSolver` **`getSolver`**`() const`

This member function returns the `IloSolver` object passed in the constructor of the invoking object.

```
public virtual IlcBool hasRealZeroCriticality() const
```

This virtual method defines a characteristic of the `IlcRCTextureI` subclasses created by the invoking factory. If this function returns `IlcTrue`, it means that when the individual demand at some time point, *t*, for `IlcRCTextureI` instances allocated by the invoking factory is 0, it will remain at 0 until there is a backtrack in the search. Otherwise, if `IlcFalse` is returned, a time point may become zero and then some other value without backtracking.

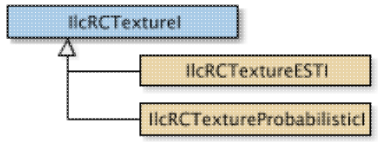
For example, the `IlcRCTextureProbabilisticI` object has this characteristic because all possible time points at which the associated activity can execute have a non-zero demand. When a time point is given a zero demand it is because the possible time window of the resource constraint has been pruned in a monotonic fashion. In contrast, the `IlcRCTextureESTI` object does not have this characteristic. For example, the time point corresponding to `getStartMin() + getDuration() + 1` has a zero individual demand. In a future search state however, the minimum start time may be increased, resulting in a non-zero demand for that time point.

This function is used to optimize the calculation of texture measurements in the cases where it returns `IlcTrue`.

The default return value for the method is `IlcFalse`.

Class IlcRCTexturel

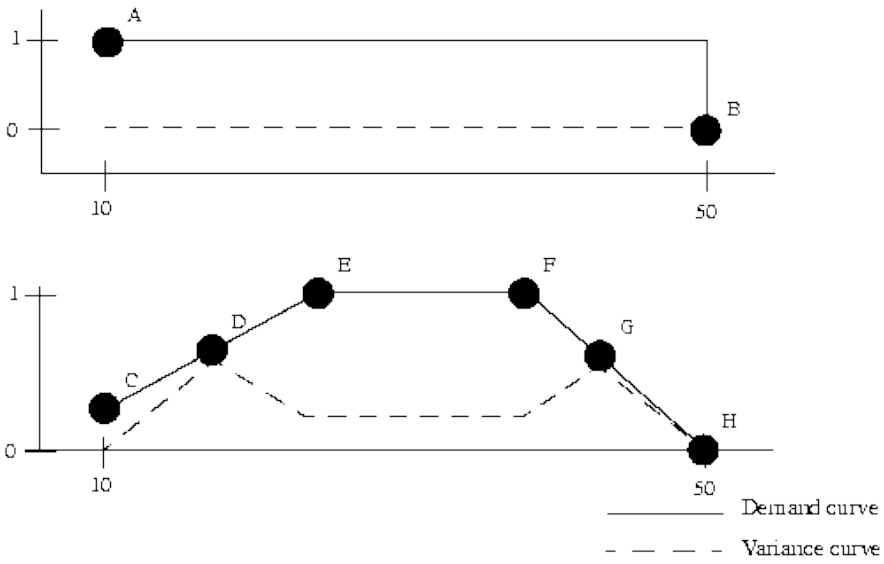
Definition file: ilsched/texturei.h
Include file: <ilsched/ilsched.h>



`IlcRCTextureI` is the implementation base class for the representation of the individual texture curve of a resource constraint on a resource. `IlcRCTextureI` is documented so that the user can create subclasses specifying the different types of individual curves. For example, by default the `IlcRCTextureESTI` subclass uses the maximum duration for the individual curve. If the user would rather use the minimum duration, a subclass of `IlcRCTextureESTI` could be written. See `IlcRCTextureESTI` for an example of subclassing `IlcRCTextureI`.

The individual curve is a piecewise linear curve represented by a set of points each of which have a time point and a demand. Optionally a point may represent a variance curve as well as the demand curve. The curves may be discontinuous, in which case the point representing the discontinuity must explicitly contain its magnitude and direction (that is, positive or negative). For example, the following figure represents two individual curves. The values of each of the data points (A through H) are listed in the table that follows.

Example of Demand Curve and Variance Curve



Data Points for the Demand Curve

Note that the data points are floating point values and can be negative.

Point	Time	Demand	Demand Discontinuity	Variance	Variance Discontinuity
A	10	1	1	0	0
B	50	0	-1	0	0
C	10	0.25	0.25	0	0
D	16.66	0.5	0	0.4	0

E	23.33	1	0	0.125	0
F	36.66	1	0	0.125	0
G	43.33	0.5	0	0.4	0
H	50	0	0	0	0

Users have complete control over the definition of the individual curve. A number of data methods (such as `IlcRCTextureI::getCapacity`, `IlcRCTextureI::setCapacity`, `IlcRCTextureI::getStart`, `IlcRCTextureI::setStart`, etc.) are provided for the maintenance of data upon which the individual curves may be built.

The points of the individual curve must be inserted in ascending temporal order by the `IlcRCTextureI::insertEvent` method call from the `IlcRCTextureI::calculateIndividualCurve` method. See the example code for `IlcRCTextureESTI::calculateIndividualCurve`

.

For more information, see Texture Measurements.

See Also: IlcRCTextureIterator, IlcResourceTexture, IlcRCTextureFactoryI, IlcRCTextureESTI, IlcRCTextureProbabilisticI, IlcRCTextureTargetI

Constructor and Destructor Summary	
protected	`IlcRCTextureI(IlcManager m, IlcResourceConstraint, IlcResourceTexture)`

Method Summary	
protected IlcFloat	`calcAltWeight() const`
protected virtual void	`calculateIndividualCurve()`
public IlcFloat	`getAltWeight() const`
public IlcFloat	`getCapacity() const`
public IlcFloat	`getDuration() const`
public IlcFloat	`getEnd() const`
public IlcResourceConstraint	`getResourceConstraint() const`
public IlcFloat	`getStart() const`
public IlcFloat	`getTargetStart() const`
protected void	`insertEvent(IlcFloat t, IlcFloat demand, IlcFloat demandDisc=0, IlcFloat var=0, IlcFloat varDisc=0)`
protected void	`setAltWeight(IlcFloat altW)`
protected void	`setCapacity(IlcFloat cap)`
protected void	`setDuration(IlcFloat dur)`
protected void	`setEnd(IlcFloat lft)`
protected void	`setStart(IlcFloat est)`
public void	`setTargetStart(IlcFloat)`
protected virtual IlcBool	`updateDataPoints()`

Constructors and Destructors

`protected` **`IlcRCTextureI`**`(IlcManager m, IlcResourceConstraint, IlcResourceTexture)`

This protected constructor creates an instance of `IlcRCTextureI` for resource constraint `rct` and the resource texture, `texture`. The created class represents the individual contribution of `rct` to the aggregate curve of `texture`. As this constructor is protected, it should only be called from subclasses of `IlcRCTextureI`.

Methods

`protected IlcFloat` **`calcAltWeight`**`() const`

This protected function calculates the alternative weight of the invoking `IlcRCTextureI` object. The alternative weight is an estimate of the probability that the resource constraint associated with the invoking object will be selected. The alternative weight is defined to be `1/nbAlt`, where `nbAlt` is the number of possible alternatives for the alternative resource constraint associated with the invoking object. If the resource constraint associated with the invoking object is not an alternative resource constraint and does not participate in any meta-constraints, the alternative weight is 1. If the resource constraint associated with the invoking object is involved in meta-constraints, there is no general method for estimating the alternative weight. Therefore, in such a case, the alternative weight is defined to be 0.5.

Note that this function calculates and returns a value. There are no side-effects. In particular, if you want to store the alternative weight in the invoking object, you must call `IlcRCTextureI::setAltWeight`.

`protected virtual void` **`calculateIndividualCurve`**`()`

This pure virtual, protected function creates the actual individual curve for the invoking object. Typically, the curve will be formulated based on the data points updated in the `IlcRCTextureI::updateDataPoints` method. However, no restriction is placed on the basis on which the individual curve is formed.

Note

WARNING! When this member function is called, it is necessary to completely recreate the curve by inserting all the curve elements in ascending temporal order using `IlcRCTextureI::insertEvent`.

`public IlcFloat` **`getAltWeight`**`() const`

This function returns the alternative weight of the invoking `IlcRCTextureI` object. The alternative weight is the value assigned by the `IlcRCTextureI::setAltWeight` function. See the member function `IlcRCTextureI::calcAltWeight` for a detailed description of alternative weight. If the alternative weight has never been set, 1 is returned.

`public IlcFloat` **`getCapacity`**`() const`

This function returns the capacity value used to calculate the individual curve of the invoking object. This value is assigned using `IlcRCTextureI::setCapacity`. Note that even though the associated resource constraint may have a variable capacity, a single value, which is the maximum capacity of the resource constraint on the resource associated with the `IlcResourceTexture` instance, is used by default to calculate the individual curve. You can, of course, create your own subclass of `IlcRCTextureI` where a different value or even multiple values of capacity are used to form the individual curve.

```
public IlcFloat getDuration() const
```

This method returns the duration value used to calculate the individual curve of the invoking object. This value is assigned using `setDuration`. Note that even though the associated resource constraint may have a variable duration, a single value, which is the maximum duration of the resource constraint on the resource associated with the `IlcResourceTexture` instance, is used by default to calculate the individual curve. You can, of course, create your own subclass of `IlcRCTextureI` where a different value or even multiple values of capacity are used to form the individual curve.

```
public IlcFloat getEnd() const
```

This method returns the time value of the final point in the individual curve. This value is set using `IlcRCTextureI::setEnd`. Note that the value used here may not have any resemblance to the maximum end time of the activity associated with the resource constraint of the invoking object: it is simply the maximum time value of the end point of the individual curve. For example, in `IlcRCTextureESTI`, it can be seen that `IlcRCTextureI::setEnd` is used to assign the maximum end time to be `IlcRCTextureI::getStart` plus `IlcRCTextureI::getDuration`.

```
public IlcResourceConstraint getResourceConstraint() const
```

This method returns a pointer to the implementation class of the resource constraint associated with the invoking individual texture curve.

```
public IlcFloat getStart() const
```

This method returns the time value of the initial point in the individual curve. This value is set using `IlcRCTextureI::setStart`. Note that the value used here may not have any resemblance to the minimum start time of the activity associated with the resource constraint of the invoking object: it is simply the minimum time value of the start point of the individual curve.

```
public IlcFloat getTargetStart() const
```

This public method returns the time value that has been set from an external source using `IlcRCTextureI::setTargetStart`. The start time differs from the minimum start time in an important way: the minimum start time is the minimum possible time value for any elements of the individual curve. The start time is an expression of preference. Based on some external source, this time is the preferred start time. The calculation of the individual curve may take this information into account. For example, the `IlcRCTextureTargetI` class creates the individual curve assuming that the start time is the only possible start time. In contrast, `IlcRCTextureProbabilisticI` ignores the start time.

```
protected void insertEvent(IlcFloat t, IlcFloat demand, IlcFloat demandDisc=0,
IlcFloat var=0, IlcFloat varDisc=0)
```

This member function inserts an event into the internal representation of the individual curve. There are three major restrictions with the use of this function:

1. This method may only be called from the `IlcRCTextureI::calculateIndividualCurve` function.
2. All the points of the curve must be reinserted in each call to `IlcRCTextureI::calculateIndividualCurve`.

3. A series of consecutive calls to this function (within the same call to `IlcRCTextureI::calculateIndividualCurve`) must insert elements in ascending temporal order to ensure the correct connectivity.

The arguments to this method are as follows:

1. `t` - This is the time point of the event.
2. `demand` - This is the value of the demand for the associated resource constraint for the resource at time point `t`.
3. `demandDisc` - This is the discontinuity in the demand curve at time `t`. Discontinuity means that the demand curve jumps non-continuously up or down by amount `|demandDisc|` at time `t`. A negative value indicates a negative jump. See points B and C in the figure at the start of this class.
4. `var` - This is the variance of the demand that the associated resource constraint has for the resource at time point `t`. This value is only non-zero when the demand is the expected value of some probabilistic estimation of resource demand.
5. `varDisc` - This is the discontinuity in the variance curve.

```
protected void setAltWeight(IlcFloat altW)
```

This protected method allows the alternative weight data point of the invoking object to be set.

```
protected void setCapacity(IlcFloat cap)
```

This protected method allows the capacity data point of the invoking object to be set.

```
protected void setDuration(IlcFloat dur)
```

This protected method allows the duration data point of the invoking object to be set.

```
protected void setEnd(IlcFloat lft)
```

This protected method allows the maximum end data point of the invoking object to be set.

```
protected void setStart(IlcFloat est)
```

This protected method allows the minimum start time data point of the invoking object to be set.

```
public void setTargetStart(IlcFloat)
```

This protected method allows the preferred start time data point of the invoking object to be set.

```
protected virtual IlcBool updateDataPoints()
```

This virtual method is called automatically during the process of updating the invoking individual texture curve. It must update all the data points upon which the individual curve is calculated. In particular, the version of this function for `IlcRCTextureI` updates the following items.

- the minimum start time to the minimum start time of the associated resource constraint,

- the maximum end time to the maximum end time of the associated resource constraint,
- the duration to the maximum duration of the associated resource constraint,
- the capacity to be the maximum capacity of the associated resource constraint, and
- the alternative weight to be the value returned by the member function `IlcRCTextureI::setAltWeight`.

If one or more of the data points has changed, `IlcTrue` must be returned. Otherwise, `IlcFalse` is returned. The correct return value (depending on whether the data points have been modified) is critical for the correct updating of the individual texture curve. The user can override this function to update new data points that have been introduced in subclasses of `IlcRCTextureI`.

Class IlcRCTextureIterator

Definition file: ilsched/texture.h
Include file: <ilsched/ilsched.h>

An instance of `IlcRCTextureIterator` can be used to iterate through the list of `IlcRCTexture` objects associated with an instance of `IlcResourceTexture`.

See Also: IlcRCTexture

Constructor Summary	
`public`	`IlcRCTextureIterator(IlcResourceTexture texture)`

Method Summary	
`public IlcBool`	`ok() const`
`public IlcRCTexture`	`operator*()`
`public void`	`operator++()`

Constructors

`public` **`IlcRCTextureIterator`**`(IlcResourceTexture texture)`

This constructor creates an instance of `IlcRCTextureIterator` over the `IlcRCTexture` elements associated with `texture`.

Methods

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if the current position of the iterator is valid. It returns `IlcFalse` if all of the `IlcRCTexture` elements on the corresponding `IlcResourceTexture` have been scanned.

`public IlcRCTexture` **`operator*`**`()`

This operator accesses the instance of `IlcRCTexture` located at the current position of the iterator. If the iterator is set past the end position, this operator returns an empty handle.

`public void` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator.

Class IlcRCTextureProbabilisticFactoryI

Definition file: ilsched/texturei.h
Include file: <ilsched/ilsched.h>

`IlcRCTextureProbabilisticFactoryI` is an implementation object that allocates instances of `IlcRCTextureProbabilisticI`.

For more information, see Texture Measurements.

See Also: IlcResourceTexture, IlcRCTextureI, IlcRCTextureProbabilisticI

Constructor and Destructor Summary	
`public`	`IlcRCTextureProbabilisticFactoryI(IloSolver solver)`

Method Summary	
`public virtual IlcRCTexture`	`createRCTexture(IlcResourceConstraint, IlcResourceTexture) const`

Inherited Methods from `IlcRCTextureFactoryI`
`createRCTexture, getSolver, hasRealZeroCriticality`

Constructors and Destructors

`public` **`IlcRCTextureProbabilisticFactoryI`**`(IloSolver solver)`

This constructor creates an instance of `IlcRCTextureProbabilisticFactoryI`.

Methods

`public virtual IlcRCTexture` **`createRCTexture`**`(IlcResourceConstraint, IlcResourceTexture) const`

This method returns a pointer to a newly allocated instance of `IlcRCTexture` representing the individual curve of the resource constraint for the resource associated with the resource texture.

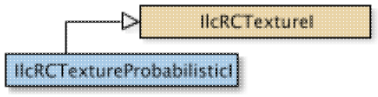
Example

The `createRCTexture()` method could be written as follows:

```
IlcRCTexture IlcRCTextureProbabilisticFactoryI::createRCTexture(
                 IlcResourceConstraint rct,
                 IlcResourceTexture texture) const {
  IloSolver solver = getSolver();
  return new (solver.getHeap())
    IlcRCTextureProbabilisticI(solver, rct, texture);
}
```

Class IlcRCTextureProbabilisticI

Definition file: ilsched/texturei.h
Include file: <ilsched/ilsched.h>



`IlcRCTextureProbabilisticI` is the implementation class for the probabilistic individual texture curve. This individual curve represents the individual demand (and variance of that demand) of the associated resource constraint for its resource. The curve is based on the assumption that each time in the domain of the start time variable of the activity associated with the resource constraint is uniformly likely to be assigned. Based on that assumption, the curve consists of four (time, demand) data points (here expressed in terms of the `IlcRCTextureI` API):

- (`getStartMin(), h/STD`)
- (`getStartMin()+getDuration(), h*min(STD,getDuration())/STD`)
- (` getEndMax()-getDuration(), h*min(STD,getDuration())/STD`)
- (`getEndMax(), 0`)

Where:

- `h = getAltWeight() * getCapacity()`
- `STD = getEndMax() - getStartMin() - getDuration() + 1`

Note that all these calculations are based on the data points of the invoking `IlcRCTextureProbabilisticI` object. These points are updated in the `IlcRCTextureProbabilisticI::updateDataPoints` method using the methods of `IlcRCTextureI` (for example, `IlcRCTextureI::setStart`, `IlcRCTextureI::setCapacity`). These values are assigned as discussed for `IlcRCTextureI::updateDataPoints`.

By subclassing this class and overriding the `IlcRCTextureProbabilisticI::updateDataPoints` method, users can redefine these data points and so change the individual curve.

For more information, see Texture Measurements.

See Also: IlcRCTextureIterator, IlcResourceTexture, IlcRCTextureProbabilisticFactoryI, IlcRCTextureESTI, IlcRCTextureI, IlcRCTextureTargetI

Constructor and Destructor Summary	
protected	`IlcRCTextureProbabilisticI(IlcManager m, IlcResourceConstraint rc, IlcResourceTexture resTexture)`

Method Summary	
protected virtual void	`calculateIndividualCurve()`
protected virtual IlcBool	`updateDataPoints()`

Inherited Methods from `IlcRCTextureI`
`calcAltWeight, calculateIndividualCurve, getAltWeight, getCapacity, getDuration, getEnd, getResourceConstraint, getStart, getTargetStart, insertEvent, setAltWeight, setCapacity, setDuration, setEnd, setStart, setTargetStart, updateDataPoints`

Constructors and Destructors

`protected` **`IlcRCTextureProbabilisticI`**`(IlcManager m, IlcResourceConstraint rc, IlcResourceTexture resTexture)`

This protected constructor creates an instance of `IlcRCTextureProbabilisticI` for resource constraint `rc` and resource texture, `texture`. The created class represents the individual contribution of `rc` to the aggregate curve of `texture`. As this constructor is protected, it should only be called from subclasses of `IlcRCTextureProbabilisticI` or from the `IlcRCTextureProbabilisticFactoryI` friend class.

Methods

`protected virtual void` **`calculateIndividualCurve`**`()`

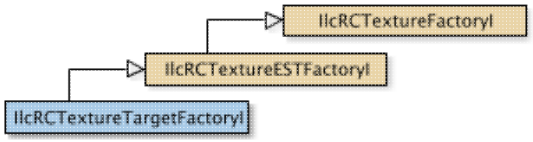
This virtual, protected function creates the actual individual curve for the invoking object.

`protected virtual IlcBool` **`updateDataPoints`**`()`

This virtual method is called automatically during the process of updating the invoking individual texture curve. It updates as discussed for the member function `IlcRCTextureI::updateDataPoints`.

Class IlcRCTextureTargetFactoryI

Definition file: ilsched/texturei.h
Include file: <ilsched/ilsched.h>



`IlcRCTextureTargetFactoryI` is an implementation object that allocates instances of `IlcRCTextureTargetI`.

For more information, see Texture Measurements.

See Also: IlcResourceTexture, IlcRCTextureI, IlcRCTextureTargetI

Constructor and Destructor Summary	
`public`	`IlcRCTextureTargetFactoryI(IloSolver solver)`

Method Summary	
`public virtual IlcRCTexture`	`createRCTexture(IlcResourceConstraint, IlcResourceTexture) const`

Inherited Methods from `IlcRCTextureESTFactoryI`
`createRCTexture`

Inherited Methods from `IlcRCTextureFactoryI`
`createRCTexture, getSolver, hasRealZeroCriticality`

Constructors and Destructors

`public` **`IlcRCTextureTargetFactoryI`**`(IloSolver solver)`

This constructor creates an instance of `IlcRCTextureTargetFactoryI`.

Methods

`public virtual IlcRCTexture` **`createRCTexture`**`(IlcResourceConstraint, IlcResourceTexture) const`

This method returns a pointer to a newly allocated instance of `IlcRCTexture` representing the individual curve of the resource constraint for the resource associated with the resource texture.

Example

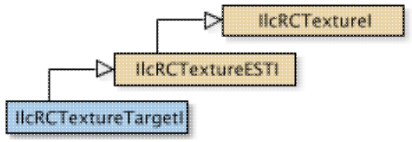
The `createRCTexture()` method could be written as follows:

```
IlcRCTexture
IlcRCTextureTargetFactoryI::createRCTexture(IlcResourceConstraint rct,
                  IlcResourceTexture texture) const {
```

```
  IloSolver solver = getSolver();
  return new (solver.getHeap())
    IlcRCTextureTargetI(solver, rct, texture);
}
```

Class IlcRCTextureTargetI

Definition file: ilsched/texturei.h
Include file: <ilsched/ilsched.h>



`IlcRCTextureTargetI` is the implementation class for the target start time individual texture curve. This individual curve represents the individual demand of the associated resource constraint for its resource based on the assumption that it will start at an externally defined target start time. The points used in the texture curve can be seen in the example code below.

For more information, see Texture Measurements.

See Also: IlcRCTextureIterator, IlcResourceTexture, IlcRCTextureTargetFactoryI, IlcRCTextureProbabilisticI, IlcRCTextureI, IlcRCTexture

Constructor and Destructor Summary	
public	`IlcRCTextureTargetI(IlcManager m, IlcResourceConstraint rc, IlcResourceTexture resTexture)`

Method Summary	
protected virtual IlcBool	`updateDataPoints()`

Inherited Methods from `IlcRCTextureESTI`
`calculateIndividualCurve`

Inherited Methods from `IlcRCTextureI`
`calcAltWeight, calculateIndividualCurve, getAltWeight, getCapacity, getDuration, getEnd, getResourceConstraint, getStart, getTargetStart, insertEvent, setAltWeight, setCapacity, setDuration, setEnd, setStart, setTargetStart, updateDataPoints`

Constructors and Destructors

`public` **`IlcRCTextureTargetI`**`(IlcManager m, IlcResourceConstraint rc, IlcResourceTexture resTexture)`

This protected constructor creates an instance of `IlcRCTextureTargetI` for resource constraint `rc` and the resource texture `resTexture`. The created class represents the individual contribution of `rc` to the aggregate curve of `resTexture`. As this constructor is protected, it should only be called from subclasses of `IlcRCTextureTargetI` or from the `IlcRCTextureTargetFactoryI` friend class.

Methods

`protected virtual IlcBool` **`updateDataPoints`**`()`

This virtual, protected function updates all the data points on which the curve is calculated.

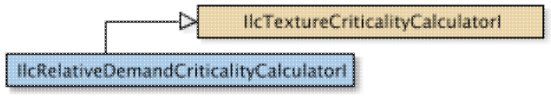
Example

Here is an example of how the `updateDataPoints()` method might be written.

```
IlcBool IlcRCTextureTargetI::updateDataPoints() {
  IlcResourceConstraint rc = getResourceConstraint();
  IlcFloat newAltWeight = calcAltWeight();
  IlcInt dur, cap;
  IlcFloat startMin = getStartTime();

  IlcAltResConstraint altRct = rc.getAlternative();
  if (0 != altRct.getImpl()) {
    IlcResource res = rc.getResource();
    dur = altRct.getDurationMax(res);
    cap = altRct.getCapacityMax(res);
  }

  else {
    IlcActivity act = rc.getActivity();
    dur = act.getDurationMax();
    cap = (rc.isVariableResourceConstraint() ?
          rc.getCapacityVariable().getMax() :
          rc.getCapacity());
  }

  IlcBool changed = IlcFalse;
  if ((getStartMin() != startMin) ||
      (getDuration() != dur) ||
      (getCapacity() != cap) ||
      (getAltWeight() != newAltWeight)) {
    changed = IlcTrue;

    setStart(startMin);
    setDuration(dur);
    setEnd(startMin + dur);
    setCapacity(cap);
    setAltWeight(newAltWeight);
  }

  return changed;
}
```

Class IlcRelativeDemandCriticalityCalculatorI

Definition file: ilsched/texturei.h
Include file: <ilsched/ilsched.h>



`IlcRelativeDemandCriticalityCalculatorI` is the implementation class that implements the relative demand texture criticality calculator.

For more information, see Texture Measurements.

See Also: IlcResourceTexture, IlcTextureCriticalityCalculatorI, IlcTextureCriticalityCalculator, IlcProbabilisticCriticalityCalculatorI

Constructor and Destructor Summary	
`public`	`IlcRelativeDemandCriticalityCalculatorI()`

Method Summary	
`public virtual IlcFloat`	`calculateCriticalityGreaterThan(IlcFloat expectedDemand, IlcFloat constraintVal, IlcFloat expectedVariance) const`
`public virtual IlcFloat`	`calculateCriticalityLessThan(IlcFloat expectedDemand, IlcFloat constraintVal, IlcFloat expectedVariance) const`

Inherited Methods from `IlcTextureCriticalityCalculatorI`
`calculateCriticalityGreaterThan, calculateCriticalityLessThan`

Constructors and Destructors

`public` **`IlcRelativeDemandCriticalityCalculatorI`**`()`

This constructor creates an instance of `IlcRelativeDemandCriticalityCalculatorI`.

Methods

`public virtual IlcFloat` **`calculateCriticalityGreaterThan`**`(IlcFloat expectedDemand, IlcFloat constraintVal, IlcFloat expectedVariance) const`

This method calculates the criticality of a maximum constraint at one time point based on the ratio of `demand` to `constraintVal`. For sample code, see `IlcRelativeDemandCriticalityCalculatorI::calculateCriticalityLessThan`.

`public virtual IlcFloat` **`calculateCriticalityLessThan`**`(IlcFloat expectedDemand, IlcFloat constraintVal, IlcFloat expectedVariance) const`

This method calculates the criticality of a minimum constraint at one time point based on the ratio of `constraintVal` to `demand`.

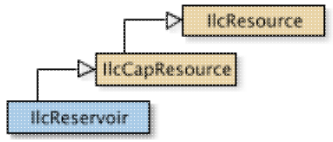
Example

The two criticality calculation methods may be written as follows:

```
IlcFloat IlcRelativeDemandCriticalityCalculatorI::
  calculateCriticalityGreaterThan(IlcFloat demand,
                                  IlcFloat constraintVal,
                                  IlcFloat) const {
  if (demand <= 0)
    return 0;
  else if (constraintVal <= 0)
    return IlcFloatMax;
  return demand / constraintVal;
}

IlcFloat IlcRelativeDemandCriticalityCalculatorI::
  calculateCriticalityLessThan(IlcFloat demand,
                               IlcFloat constraintVal,
                               IlcFloat = 0.) const {
  if (constraintVal <= 0)
    return 0;
  else if (demand <= 0)
    return IlcFloatMax;
  return constraintVal / demand;
}
```

Class IlcReservoir

Definition file: ilsched/reservoi.h
Include file: <ilsched/ilsched.h>



An instance of the class `IlcReservoir` represents a resource for which activities can both *provide* capacity and also *require* capacity. You can define required and provided capacity so that the Scheduler Engine will insure that no more capacity is ever used than provided.

Furthermore, if you define a maximal level of the reservoir, then this maximal level will never be exceeded.

When the model of your problem represents an ongoing process, you may be faced with the fact that the reservoir already has some non-zero level. You can simply pass an initial level like that to the constructor of `IlcReservoir`.

The capacity of a reservoir can vary over time. You can define temporary maximal and minimal levels by using member functions of `IlcReservoir`.

Closing a Reservoir

The member function `IlcResource::close` is crucial for propagation affecting the class `IlcReservoir`. If `IlcResource::close` is not called, new activities providing or requiring capacity can still be added. This means that no propagation can take place before the reservoir is closed.

Printing or Displaying Reservoirs

The printed representation of an instance of the class `IlcReservoir` consists of its name, if it exists, and its theoretical capacity followed by its initial level. The two values are enclosed in brackets and separated by a dash (-). For example:

`r1[100 - 10]` represents the reservoir named `r1` which has a capacity equal to 100 and an initial level equal to 10.

If the Solver trace is active and the resource is not named, the string "`IlcReservoir`" is followed by the address of the implementation object. The address will be enclosed in parentheses.

If the theoretical capacity of the reservoir is equal to its maximal value (that is `IlcIntMax/2)`, the string `Maximum Capacity` is displayed instead of its numerical value.

For more information, see Timetable, and Balance Constraint.

See Also: IlcCapResource, IlcReservoirIterator, IlcResource, IlcResourceConstraint, IlcSchedule

Constructor Summary	
public	`IlcReservoir()`
public	`IlcReservoir(IlcReservoirI * impl)`
public	`IlcReservoir(IlcSchedule schedule, IlcInt capacity=IlcMaxCapacityReservoir, IlcInt initialLevel=0L, IlcBool timetable=IlcTrue)`

Method Summary	
public IlcInt	`getCapacity() const`

public IlcReservoirI *	getImpl() const
public IlcInt	getInitialLevel() const
public IlcInt	getLevelMax(IlcInt time) const
public IlcInt	getLevelMaxMax(IlcInt timeMin, IlcInt timeMax) const
public IlcInt	getLevelMaxMin(IlcInt timeMin, IlcInt timeMax) const
public IlcInt	getLevelMin(IlcInt time) const
public IlcInt	getLevelMinMax(IlcInt timeMin, IlcInt timeMax) const
public IlcInt	getLevelMinMin(IlcInt timeMin, IlcInt timeMax) const
public void	operator=(const IlcReservoir & h)
public void	setLevelMax(IlcInt timeMin, IlcInt timeMax, IlcInt levelMax)
public void	setLevelMin(IlcInt timeMin, IlcInt timeMax, IlcInt levelMin)

Inherited Methods from `IlcCapResource`
getImpl, getMaxTextureMeasurement, getMinTextureMeasurement, getTimetable, getTimetable, hasInitialOccupation, hasMaxTextureMeasurement, hasMinTextureMeasurement, incrDurableRequirement, incrDurableRequirement, isRedundantResource, makeBalanceConstraint, makeMaxTextureMeasurement, makeMinTextureMeasurement, makeTimetableConstraint, makeTimetableConstraint, makeTimetableConstraint, operator=, setInitialOccupation, setInitialOccupation, unsetInitialOccupation

Inherited Methods from `IlcResource`
close, getCalendar, getDisjunctiveConstraint, getDurableSchedule, getImpl, getLastRankedFirstRC, getLastRankedLastRC, getLastSurelyContributingRankedFirstRC, getLastSurelyContributingRankedLastRC, getName, getObject, getOldLastRankedFirstRC, getOldLastRankedLastRC, getPrecedenceGraphConstraint, getSchedule, getSolver, getSolverI, getTimetableConstraint, getTransitionTime, hasCalendar, hasDisjunctiveConstraint, hasLightPrecedenceGraphConstraint, hasPrecedenceGraphConstraint, hasPrecedenceInfo, hasRankInfo, hasTimetableConstraint, isCapacityResource, isClosed, isContinuousReservoir, isDiscreteEnergy, isDiscreteResource, isDurable, isReservoir, isStateResource, isTransitionTimeSuspended, isUnaryResource, makeFunctionalConstraint, makeIntegralConstraint, makeLightPrecedenceGraphConstraint, makePrecedenceGraphConstraint, operator!=, operator=, operator==, setCalendar, setName, setObject, setTransitionTimeObject, setTransitionTimeSuspended, whenContribution, whenDirectPredecessors, whenDirectSuccessors, whenNext, whenPossibleNext, whenPossiblePrevious, whenPredecessors, whenPrevious, whenRankedFirstRC, whenRankedLastRC, whenSuccessors

Constructors

public **IlcReservoir**()

This constructor creates an instance which is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

public **IlcReservoir**(IlcReservoirI * impl)

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

```
public IlcReservoir(IlcSchedule schedule, IlcInt capacity=IlcMaxCapacityReservoir,
IlcInt initialLevel=OL, IlcBool timetable=IlcTrue)
```

This constructor creates a new instance of `IlcReservoir` and adds it to the set of resources managed in the given `schedule`. The `capacity` argument expresses the capacity of the new reservoir. The capacity may be consumed by certain activities and produced by others. The argument `initialLevel` defines an initial amount in the reservoir at the time origin of the schedule. By default, the reservoir is assumed to be empty at the time origin; that is, the initial level is 0 (zero). The default value of the theoretical capacity is `IlcIntMax/2`. That is the maximal theoretical capacity that is allowed. Any capacity greater than `IlcIntMax/2` will be treated as if it were equal to `IlcIntMax/2`.

If `timetable` is `IlcTrue`, then a timetable constraint is posted, defining the level of the reservoir to be between 0 (zero) and `capacity` over the interval `[timeMin timeMax)`, where `timeMin` is the origin and `timeMax` is the horizon of the schedule. An instance of `IloSolver::SolverErrorException` is thrown if `capacity` is strictly negative.

Methods

```
public IlcInt getCapacity() const
```

This member function returns the theoretical capacity of the invoking reservoir, that is, the capacity that was passed to the resource constructor. If the theoretical capacity is unlimited, then this member function returns `IlcIntMax/2` (a platform-dependent Solver constant indicating the greatest possible integer).

```
public IlcReservoirI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcInt getInitialLevel() const
```

This member function returns the initial level of the reservoir; that is, the initial level that was passed to the reservoir constructor.

```
public IlcInt getLevelMax(IlcInt time) const
```

This member function returns the maximal level that is present at the given `time`. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking reservoir do not cover the given `time`.

```
public IlcInt getLevelMaxMax(IlcInt timeMin, IlcInt timeMax) const
```

This member function returns the maximal level throughout the interval `[timeMin, timeMax)` (that is, the maximal value over the interval `[timeMin, timeMax)` of the maximal reservoir level). An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking reservoir do not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public IlcInt getLevelMaxMin(IlcInt timeMin, IlcInt timeMax) const
```

This member function returns the maximal valiue, over the interval `[timeMin, timeMax)`, of the minimal reservoir level. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking reservoir do not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public IlcInt getLevelMin(IlcInt time) const
```

This member function returns the minimal level that is present at the given `time`. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking reservoir do not cover the given `time`.

```
public IlcInt getLevelMinMax(IlcInt timeMin, IlcInt timeMax) const
```

This member function returns the minimal value, over the interval `[timeMin, timeMax)`, of the maximal reservoir level. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking reservoir do not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public IlcInt getLevelMinMin(IlcInt timeMin, IlcInt timeMax) const
```

This member function returns the minimal level throughout the interval `[timeMin, timeMax)` (that is, the minimal value over the interval `[timeMin, timeMax)` of the minimal reservoir level). An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking reservoir do not cover the complete interval indicated by `[timeMin, timeMax)`.

```
public void operator=(const IlcReservoir & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument. After execution of this operator, the invoking object and the provided argument point to the same implementation object.

```
public void setLevelMax(IlcInt timeMin, IlcInt timeMax, IlcInt levelMax)
```

This member function states that the level of the reservoir can be at most `levelMax` throughout the interval `[timeMin, timeMax)`. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking reservoir do not cover the complete interval indicated by `[timeMin, timeMax)`. The reservoir must be *closed* in order to propagate constraints.

```
public void setLevelMin(IlcInt timeMin, IlcInt timeMax, IlcInt levelMin)
```

This member function states that the level of the reservoir must be at least `levelMin` throughout the interval `[timeMin timeMax)`. An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking reservoir do not cover the complete interval indicated by `[timeMin, timeMax)`. The reservoir must be *closed* in order to propagate constraints.

Class IlcReservoirIterator

Definition file: ilsched/reservoi.h
Include file: <ilsched/ilsched.h>

An instance of this class traverses the set of reservoirs.

See Also: IlcReservoir, IlcSchedule

Constructor and Destructor Summary	
public	IlcReservoirIterator(const IlcSchedule schedule)

Method Summary	
public IlcBool	ok() const
public IlcReservoir	operator*() const
public IlcReservoirIterator &	operator++()

Constructors and Destructors

`public` **`IlcReservoirIterator`**`(const IlcSchedule schedule)`

This constructor creates an iterator to traverse all the reservoirs of `schedule`.

Methods

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if the current position of the iterator is a valid one. It returns `IlcFalse` if all the reservoirs have been scanned by the iterator.

`public IlcReservoir` **`operator*`**`() const`

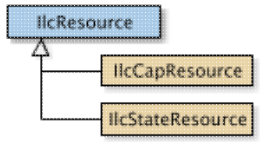
This operator accesses the instance of `IlcReservoir` located at the current position of the iterator. If the iterator is set past the end position, this operator returns an empty handle.

`public IlcReservoirIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator.

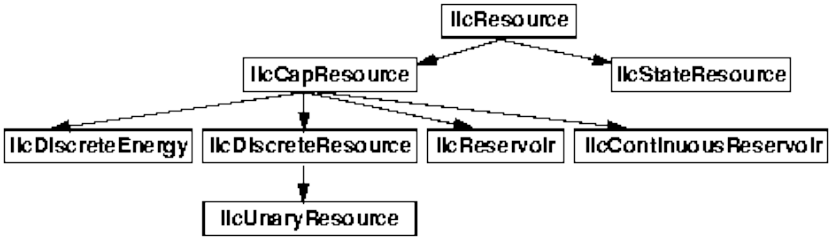
Class IlcResource

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>



In the Scheduler Engine, a resource is represented by an instance of the abstract class `IlcResource`. Activities in a schedule may require or provide resources, so there is a data member in the class `IlcResource` to distinguish between resources "to be required" and those "to be provided" by activities. Each resource belongs to a unique schedule, an instance of `IlcSchedule`. The member function `IlcResource::getSchedule` returns that unique schedule whenever it is invoked on a resource.

There are several predefined subclasses of `IlcResource`:



Closing a Resource

The member function `close` specifies that all the activities requiring the invoking resource are known; that is, they have been linked to the resource. This information allows additional constraint propagation to take place, for example, the propagation of minimal capacity constraints. Propagating minimal capacity constraints is particularly useful in the case of resource allocation problems for which some minimal amount of provided capacity must be reached. With such information, indeed, the system can eliminate situations in which minimal capacity amounts cannot be reached if no other (not deduced) activity can execute to provide the resource.

An instance of `IloSolver::SolverErrorException` is thrown when a new resource constraint (see the `IlcResourceConstraint` class) is posted on a closed resource.

The member function `IlcResource::close` is also crucial for propagation affecting the `IlcReservoir` and `IlcContinuousReservoir` classes. If `IlcResource::close` is not called, new activities providing or requiring capacity can still be added. This means that no propagation can take place before the reservoir is closed.

For more information, see Calendars, Disjunctive Constraint, Durability, Timetable, Transition Times, Precedence Graph Constraints, and Functional and Integral Constraints on Resources.

See Also: IlcAltResSet, IlcIntervalList, IlcCapResource, IlcResourceConstraint, IlcResourceIterator, IlcSchedule, IlcSchedVariable, IlcGranularFunction

Constructor Summary	
`public`	`IlcResource()`
`public`	`IlcResource(IlcResourceI * impl)`

Method Summary

public void	close()
public IlcCalendar	getCalendar() const
public IlcConstraint	getDisjunctiveConstraint() const
public IlcSchedule	getDurableSchedule() const
public IlcResourceI *	getImpl() const
public IlcResourceConstraint	getLastRankedFirstRC() const
public IlcResourceConstraint	getLastRankedLastRC() const
public IlcResourceConstraint	getLastSurelyContributingRankedFirstRC() const
public IlcResourceConstraint	getLastSurelyContributingRankedLastRC() const
public const char *	getName() const
public IlcAny	getObject() const
public IlcResourceConstraint	getOldLastRankedFirstRC() const
public IlcResourceConstraint	getOldLastRankedLastRC() const
public IlcConstraint	getPrecedenceGraphConstraint() const
public IlcSchedule	getSchedule() const
public IloSolver	getSolver() const
public IloSolverI *	getSolverI() const
public IlcConstraint	getTimetableConstraint() const
public IlcInt	getTransitionTime(const IlcResourceConstraint ct1, const IlcResourceConstraint ct2) const
public IlcBool	hasCalendar() const
public IlcBool	hasDisjunctiveConstraint() const
public IlcBool	hasLightPrecedenceGraphConstraint() const
public IlcBool	hasPrecedenceGraphConstraint() const
public IlcBool	hasPrecedenceInfo() const
public IlcBool	hasRankInfo() const
public IlcBool	hasTimetableConstraint() const
public IlcBool	isCapacityResource() const
public IlcBool	isClosed() const
public IlcBool	isContinuousReservoir() const
public IlcBool	isDiscreteEnergy() const
public IlcBool	isDiscreteResource() const
public IlcBool	isDurable() const
public IlcBool	isReservoir() const
public IlcBool	isStateResource() const
public IlcBool	isTransitionTimeSuspended() const
public IlcBool	isUnaryResource() const
public IlcConstraint	makeFunctionalConstraint(IlcSchedVariable leftVariable, const IlcGranularFunction func, IlcSchedVariable rightVariable=IlcDurationVariable, IlcBool fste=IlcFalse) const
public IlcConstraint	makeIntegralConstraint(IlcSchedVariable leftVariable, const IlcGranularFunction func,

	IlcBool ignoreSuspensionAtStartEnd=IlcTrue, IlcBool fste=IlcFalse) const
public IlcConstraint	makeLightPrecedenceGraphConstraint()
public IlcConstraint	makePrecedenceGraphConstraint()
public IlcBool	operator!=(const IlcResource & resource) const
public void	operator=(const IlcResource & h)
public IlcBool	operator==(const IlcResource & resource) const
public void	setCalendar(IlcCalendar cal)
public void	setName(const char * name) const
public void	setObject(IlcAny object) const
public void	setTransitionTimeObject(IlcTransitionTimeObject ttobj)
public void	setTransitionTimeSuspended(IlcBool suspended=IlcTrue)
public void	whenContribution(const IlcResourceDemon d)
public void	whenDirectPredecessors(const IlcResourceDemon d)
public void	whenDirectSuccessors(const IlcResourceDemon d)
public void	whenNext(const IlcResourceDemon d)
public void	whenPossibleNext(const IlcResourceDemon d)
public void	whenPossiblePrevious(const IlcResourceDemon d)
public void	whenPredecessors(const IlcResourceDemon d)
public void	whenPrevious(const IlcResourceDemon d)
public void	whenRankedFirstRC(const IlcDemon demon) const
public void	whenRankedLastRC(const IlcDemon demon) const
public void	whenSuccessors(const IlcResourceDemon d)

Inner Enumeration
IlcResource::RankFilter

Inner Class
IlcResource::ResourceConstraintDeltaIterator
IlcResource::ResourceConstraintIterator

Constructors

`public` **`IlcResource`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcResource`**`(IlcResourceI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public void` **`close`**`()`

This member function closes the invoking resource. That is, it states that all the activities requiring or providing the invoking resource are known so constraint propagation can proceed.

`public IlcCalendar` **`getCalendar`**`() const`

This member function returns the calendar attached to the invoking resource, if such an object exists.

`public IlcConstraint` **`getDisjunctiveConstraint`**`() const`

This member function returns the disjunctive constraint of the invoking resource.

`public IlcSchedule` **`getDurableSchedule`**`() const`

This member function returns the durable schedule on which the invoking durable resource was constructed. It returns an empty handle if the invoking resource is not durable.

`public IlcResourceI * ` **`getImpl`**`() const`

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

`public IlcResourceConstraint` **`getLastRankedFirstRC`**`() const`

This member function returns the last resource constraint that was ranked first on the resource. If no resource constraint has been ranked first, it returns an empty handle.

This function should be used only if the ranking information is available on the resource (see `IlcResource::hasRankInfo`).

`public IlcResourceConstraint` **`getLastRankedLastRC`**`() const`

This member function returns the last resource constraint that was ranked last on the resource. If no resource constraint has been ranked last, it returns an empty handle.

This function should be used only if the ranking information is available on the resource (see `IlcResource::hasRankInfo`).

`public IlcResourceConstraint` **`getLastSurelyContributingRankedFirstRC`**`() const`

This member function returns the last resource constraint that was ranked first on the resource and surely affects the availability of the resource (strictly positive duration and capacity requirement). If no resource constraint meets these conditions, it returns an empty handle.

This function should be used only if the ranking information is available on the resource (see `IlcResource::hasRankInfo`).

```
public IlcResourceConstraint getLastSurelyContributingRankedLastRC() const
```

This member function returns the last resource constraint that was ranked last on the resource and surely affects the availability of the resource (strictly positive duration and capacity requirement). If no resource constraint meets these conditions, it returns an empty handle.

This function should be used only if the ranking information is available on the resource (see `IlcResource::hasRankInfo`).

```
public const char * getName() const
```

This member function returns the name of the invoking object.

```
public IlcAny getObject() const
```

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

```
public IlcResourceConstraint getOldLastRankedFirstRC() const
```

When called by the execution of a demon `d` attached with the event `IlcResource::whenRankedFirstRC` this member function returns the last resource constraint that has been ranked first during the last triggering of the event. If the event is triggered for the first time, this function returns an empty handle. The delta iterator `IlcResource::ResourceConstraintDeltaIterator` with argument `RankedFirst` allows iteration in chronological order (with respect to the start/end time of activities) over all the resource constraints between the one returned by `getOldLastRankedFirstRC` (excluded) and the one returned by `IlcResource::getLastRankedFirstRC` (included).

When called outside the execution of a demon `d` attached with the event `IlcResource::whenRankedFirstRC` this member function returns the last resource constraint that was ranked first on the resource. That is, it returns exactly the same value as the member function `IlcResource::getLastRankedFirstRC`.

This function should be used only if the ranking information is available on the resource (see `IlcResource::hasRankInfo`).

```
public IlcResourceConstraint getOldLastRankedLastRC() const
```

When called by the execution of a demon `d` attached with the event `IlcResource::whenRankedLastRC` this member function returns the last resource constraint that has been ranked last during the last triggering of the event. If the event is triggered for the first time, this function returns an empty handle. The delta iterator `IlcResource::ResourceConstraintDeltaIterator` with argument `RankedLast` allows iteration in anti-chronological order (with respect to the start/end time of activities) over all the resource constraints between the one returned by `getOldLastRankedLastRC` (excluded) and the one returned by `IlcResource::getLastRankedLastRC` (included).

When called outside the execution of a demon `d` attached with the event `IlcResource::whenRankedLastRC` this member function returns the last resource constraint that was ranked last on the resource. That is, it returns exactly the same value as the member function `IlcResource::getLastRankedLastRC`.

This function should be used only if the ranking information is available on the resource (see `IlcResource::hasRankInfo`).

```
public IlcConstraint getPrecedenceGraphConstraint() const
```

This member function returns the precedence graph constraint associated with the invoking resource.

```
public IlcSchedule getSchedule() const
```

This member function returns the schedule to which the invoking resource belongs. Each resource belongs to a unique schedule, an instance of `IlcSchedule`.

```
public IloSolver getSolver() const
```

This member function returns an instance of `IloSolver` associated with the invoking object.

```
public IloSolverI * getSolverI() const
```

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

```
public IlcConstraint getTimetableConstraint() const
```

This member function returns the timetable constraint of the invoking resource.

```
public IlcInt getTransitionTime(const IlcResourceConstraint ct1, const
IlcResourceConstraint ct2) const
```

This member function returns the transition time between the two activities corresponding to `ct1` and `ct2`. By default, that is when no transition time object has been defined for the resource, this function returns 0.

Transition times can be taken into account by the disjunctive constraint or by the type timetable constraint. See Transition Time in Scheduler Engine for more details.

```
public IlcBool hasCalendar() const
```

This member function returns `IlcTrue` if a calendar has been attached to the invoking resource. Otherwise, it returns `IlcFalse`.

```
public IlcBool hasDisjunctiveConstraint() const
```

This member function returns `IlcTrue` if the invoking resource has a disjunctive constraint. Otherwise, it returns `IlcFalse`.

```
public IlcBool hasLightPrecedenceGraphConstraint() const
```

This member function returns `IlcTrue` if and only if a light precedence graph constraint has been created on the resource.

```
public IlcBool hasPrecedenceGraphConstraint() const
```

This member function returns `IlcTrue` if the invoking resource is associated with a precedence graph constraint. Otherwise, it returns `IlcFalse`.

```
public IlcBool hasPrecedenceInfo() const
```

This member function returns `IlcTrue` if and only if some precedence information is available on the resource. This is the case when the resource has been associated with a precedence graph constraint or when the resource is a unary resource with a sequence constraint.

When this member function returns `IlcTrue`, all the member functions allowed when `IlcResource::hasRankInfo` returns `IlcTrue` are allowed, along with the following additional functions:

- `IlcResource::whenDirectSuccessors`, `IlcResource::whenDirectPredecessors`, `IlcResource::whenSuccessors`, `IlcResource::whenPredecessors`, `IlcResource::whenPossiblePrevious`, `IlcResource::whenPossibleNext`.
- `IlcResourceConstraint::setNotNext`, `IlcResourceConstraint::setNotSetup`, `IlcResourceConstraint::setNotTeardown`, `IlcResourceConstraint::isDirectlySucceededBy`, `IlcResourceConstraint::isSucceededBy`, `IlcResourceConstraint::hasAsPossibleNext`, `IlcResourceConstraint::isPossibleSetup`, `IlcResourceConstraint::isPossibleTeardown`.

```
public IlcBool hasRankInfo() const
```

This member function returns `IlcTrue` if and only if some rank information is available on the resource. This is the case when the resource is a unary resource and has been associated with a light precedence graph constraint, a precedence graph constraint, a disjunctive constraint or a sequence constraint, or when the resource is a state resource with a precedence graph constraint or a disjunctive constraint.

When this member function returns `IlcTrue`, the following functions can be used during the search:

- `IlcResource::getLastRankedFirstRC`, `IlcResource::getLastRankedLastRC`, `IlcResource::getOldLastRankedFirstRC`, `IlcResource::getOldLastRankedLastRC`, `IlcResource::getLastSurelyContributingRankedFirstRC`, `IlcResource::getLastSurelyContributingRankedLastRC`, `IlcResource::whenRankedFirstRC`, `IlcResource::whenRankedLastRC`, `IlcResource::whenPrevious`, `IlcResource::whenNext`, `IlcResource::whenContribution`.
- `IlcUnaryResource::isRanked`, `IlcUnaryResource::getSetupRC`, `IlcUnaryResource::getTeardownRC`, `IlcUnaryResource::hasSetupRC`, `IlcUnaryResource::hasTeardownRC`.
- `IlcResourceConstraint::rankFirst`, `IlcResourceConstraint::rankLast`, `IlcResourceConstraint::rankNotFirst`, `IlcResourceConstraint::rankNotLast`, `IlcResourceConstraint::setNext`, `IlcResourceConstraint::setSetup`, `IlcResourceConstraint::setTeardown`, `IlcResourceConstraint::setSuccessor`, `IlcResourceConstraint::isPossibleFirst`, `IlcResourceConstraint::isPossibleLast`, `IlcResourceConstraint::isRankedFirst`, `IlcResourceConstraint::isRankedLast`, `IlcResourceConstraint::isRanked`, `IlcResourceConstraint::getNextRC`, `IlcResourceConstraint::getPrevRC`, `IlcResourceConstraint::hasAsNext`, `IlcResourceConstraint::hasNextRC`, `IlcResourceConstraint::hasPrevRC`, `IlcResourceConstraint::isSetup`, `IlcResourceConstraint::isTeardown`, `IlcResourceConstraint::surelyContributes`, `IlcResourceConstraint::possiblyContributes`.
- `IlcTryRankFirst`, `IlcTryRankLast`, `IlcRank`, `IlcRankBackward`.

```
public IlcBool hasTimetableConstraint() const
```

This member function returns `IlcTrue` if the invoking resource has a timetable constraint. Otherwise, it returns `IlcFalse`.

```
public IlcBool isCapacityResource() const
```

This member function distinguishes between the classes of resources available in the Scheduler Engine. It returns `IlcTrue` if the invoking resource is an instance of the class `IlcCapResource`. Otherwise, it returns `IlcFalse`.

```
public IlcBool isClosed() const
```

This member function returns `IlcTrue` if the invoking resource is closed; that is, the resource no longer accepts new resource constraints being declared for it. The member function returns `IlcFalse` if the invoking resource is still open.

```
public IlcBool isContinuousReservoir() const
```

This member function distinguishes among the classes of resources available in the Scheduler Engine. It returns `IlcTrue` if the invoking resource is an instance of the class `IlcContinuousReservoir`. Otherwise, it returns `IlcFalse`.

```
public IlcBool isDiscreteEnergy() const
```

This member function distinguishes among the classes of resources available in the Scheduler Engine. It returns `IlcTrue` if the invoking resource is an instance of the class `IlcDiscreteEnergy`. Otherwise, it returns `IlcFalse`.

```
public IlcBool isDiscreteResource() const
```

This member function distinguishes among the classes of resources available in the Scheduler Engine. It returns `IlcTrue` if the invoking resource is an instance of the class `IlcDiscreteResource`. Otherwise, it returns `IlcFalse`.

```
public IlcBool isDurable() const
```

This member function returns `IlcTrue` if the invoking resource was constructed on a durable schedule. Otherwise, it returns `IlcFalse`.

```
public IlcBool isReservoir() const
```

This member function distinguishes among the classes of resources available in the Scheduler Engine. It returns `IlcTrue` if the invoking resource is an instance of the class `IlcReservoir`. Otherwise, it returns `IlcFalse`.

```
public IlcBool isStateResource() const
```

This member function distinguishes among the classes of resources available in the Scheduler Engine. It returns `IlcTrue` if the invoking resource is an instance of the class `IlcStateResource`. Otherwise, it returns `IlcFalse`.

```
public IlcBool isTransitionTimeSuspended() const
```

This member function returns `IlcTrue` if the transition time on the invoking resource has been declared to be suspended by breaks.

```
public IlcBool isUnaryResource() const
```

This member function distinguishes among the classes of resources available in the Scheduler Engine. It returns `IlcTrue` if the invoking resource is an instance of the class `IlcUnaryResource`. Otherwise, it returns `IlcFalse`.

```
public IlcConstraint makeFunctionalConstraint(IlcSchedVariable leftVariable, const
IlcGranularFunction func, IlcSchedVariable rightVariable=IlcDurationVariable,
IlcBool fste=IlcFalse) const
```

This member function creates a functional constraint from the function `func` on all the activities requiring the invoking resource. If the time extent is `IlcNever` or `IlcAlways`, the resource constraint will be ignored.

For each such activity, the integral of the function `func` is evaluated from the value of the variable designated by `rightVariable`, and set to be equal to the variable designated by `leftVariable`:

```
leftVariable = f(rightVariable)
```

The `func` object must be closed, otherwise an error will be raised. Whenever the processing time is used (`IloProcessingTimeVariable`), every activity executing on the resource must be breakable, and the granular function `func` must take a value less than or equal to its granularity. Otherwise an error will be raised when starting to solve the problem.

```
public IlcConstraint makeIntegralConstraint(IlcSchedVariable leftVariable, const
IlcGranularFunction func, IlcBool ignoreSuspensionAtStartEnd=IlcTrue, IlcBool
fste=IlcFalse) const
```

This member function creates an integral constraint from the function `func` on all the activities requiring the invoking resource. If the time extent is `IlcNever` or `IlcAlways`, the resource constraint will be ignored.

For each such activity, the integral of the function `func` is computed over the duration, divided by the granularity, and properly rounded (see `IlcGranularFunction`), It is then set to be equal to the variable designated by `leftVar`:

$$\text{leftVar} = (\int_{\text{start}}^{\text{end}} \text{func})/\text{granularity}$$

The leftVar argument should only be one of the following variable types: `IlcExternalVariable`, `IlcProcessingTimeVariable`, `IlcCapacityVariable`, `IlcEnergyVariable`. The `func` object must be closed, otherwise an error will be raised. Whenever the processing time is used (`IlcProcessingTimeVariable`), every activity executing on the resource must be breakable, and the

granular function `func` must take a value less than or equal to its granularity. Otherwise an error will be raised when starting to solve the problem.

The suspension of activities at the start or end (see `IlcActivity::canBeSuspendedAtStart` and `IlcActivity::canBeSuspendedAtEnd`) is by default not taken into account. To take forbidden suspensions into account, the argument `ignoreSuspensionAtStartEnd` may be set to `IlcFalse`. Then, the resulting integral constraint will accordingly prevent activities to start/end in intervals where the granular function func has zero values.

```
public IlcConstraint makeLightPrecedenceGraphConstraint()
```

This member function allows the creation and return of a light precedence graph constraint on the invoking unary resource. That constraint has to be posted in order to be taken into account.

```
public IlcConstraint makePrecedenceGraphConstraint()
```

This member function creates and returns the precedence graph constraint associated with the invoking resource. That constraint has to be posted in order to be taken into account.

```
public IlcBool operator!=(const IlcResource & resource) const
```

This operator returns `IlcTrue` if and only if `resource` does not refer to the same implementation object as the invoking resource.

```
public void operator=(const IlcResource & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public IlcBool operator==(const IlcResource & resource) const
```

This operator returns `IlcTrue` if and only if `resource` refers to the same implementation object as the invoking resource.

```
public void setCalendar(IlcCalendar cal)
```

This member function attaches the calendar `cal` to the invoking resource.

Outside the search, it is possible to attach a new calendar to a resource that already has a calendar attached. In that case the new attachment replaces the previous one.

During search, it is possible to attach a calendar only to a resource that does not have a previously attached calendar. In that case, the attachment is reversible. During search, any attempt to attach a new calendar to a resource that already has an attached calendar raises an error.

When two different calendars are attached respectively to a resource constraint and to its corresponding resource, only the one on the resource constraint is taken into account. That is, if some breaks are attached to the resource *Res* using the calendar *Cal1* and some shifts are attached to a resource constraint *Rct* of *Res* using the calendar *Cal2*, then only shifts are taken into account on *Rct* (*Cal2*). In other words, breaks of *cal1* are ignored (*Cal1*).

Notice that calendars can be shared between resources and resource constraints, and that `setCalendar` does not imply that a local copy of the calendar is made; one should be aware of the fact that any change to a shared calendar holds for all resources and resource constraints sharing the calendar.

```
public void setName(const char * name) const
```

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

```
public void setObject(IlcAny object) const
```

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

```
public void setTransitionTimeObject(IlcTransitionTimeObject ttobj)
```

This member function allows setting `ttobj` as the new transition time function of the invoking resource. During the search, any attempt to change the transition function of a resource will raise an error.

```
public void setTransitionTimeSuspended(IlcBool suspended=IlcTrue)
```

This member function allows specifying whether or not the transition time on the invoking resource should be suspended by breaks. By default, the transition time of the resource is not suspended by breaks.

```
public void whenContribution(const IlcResourceDemon d)
```

This member function associates the resource demon `d` with contribution events of resource constraints of the invoking resource. When the contribution status of a resource constraint changes (from possibly contributing to surely contributing or to not possibly contributing), the demon `d` is executed on that resource constraint.

This member function should be used only in search and only if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`).

```
public void whenDirectPredecessors(const IlcResourceDemon d)
```

This member function associates the resource demon `d` with changes to the set of resource constraints that are direct predecessors of a resource constraint of the invoking resource. When the set of resource constraints that are direct predecessors of a resource constraint changes because some new direct predecessors have appeared, the demon `d` is executed on the resource constraint whose set of direct predecessors has changed.

This member function should be used only in search and only if some precedence information is available on the resource (see member function `IlcResource::hasPrecedenceInfo`).

```
public void whenDirectSuccessors(const IlcResourceDemon d)
```

This member function associates the resource demon `d` with changes to the set of resource constraints that are direct successors of a resource constraint of the invoking resource. When the set of resource constraints that are direct successors of a resource constraint changes because some new direct successors have appeared, the demon `d` is executed on the resource constraint whose set of direct successors has changed.

This member function should be used only in search and only if some precedence information is available on the resource (see member function `IlcResource::hasPrecedenceInfo`).

```
public void whenNext(const IlcResourceDemon d)
```

This member function associates the resource demon `d` with the next resource constraint of resource constraints of the invoking resource. When the next resource constraint of a resource constraint is known, the demon `d` is executed on the resource constraint whose next resource constraints have become known. This next resource constraint can then be accessed with the member function `IlcResourceConstraint::getNextRC`. When the setup resource constraints is known, the resource demon `d` is executed on an empty handle resource constraint.

This member function should be used only in search and only if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`).

```
public void whenPossibleNext(const IlcResourceDemon d)
```

This member function associates the resource demon `d` with changes to the set of resource constraints that are possibly next to a resource constraint of the invoking resource. When the set of resource constraints that are possibly next to a resource constraint changes because some resource constraint that was possibly next is no longer possibly next, the demon `d` is executed on the resource constraint whose set of possibly next resource constraints has changed. In case the possible setup resource constraints change, the resource demon `d` is executed on an empty handle resource constraint.

This member function should be used only in search and only if some precedence information is available on the resource (see member function `IlcResource::hasPrecedenceInfo`).

```
public void whenPossiblePrevious(const IlcResourceDemon d)
```

This member function associates the resource demon `d` with changes to the set of resource constraints that are possibly previous to a resource constraint of the invoking resource. When the set of resource constraints that are possibly previous to a resource constraint changes because some resource constraint that was possibly previous is no longer possibly previous, the demon `d` is executed on the resource constraint whose set of possibly previous resource constraints has changed. In case the possible teardown resource constraints change, the resource demon `d` is executed on an empty handle resource constraint.

This member function should be used only in search and only if some precedence information is available on the resource (see member function `IlcResource::hasPrecedenceInfo`).

```
public void whenPredecessors(const IlcResourceDemon d)
```

This member function associates the resource demon `d` with changes to the set of resource constraints that are predecessors of a resource constraint of the invoking resource. When the set of resource constraints that are predecessors of a resource constraint changes because some new predecessors have appeared, the demon `d` is executed on the resource constraint whose set of predecessors has changed.

This member function should be used only in search and only if some precedence information is available on the resource (see member function `IlcResource::hasPrecedenceInfo`).

```
public void whenPrevious(const IlcResourceDemon d)
```

This member function associates the resource demon `d` with the previous resource constraint of resource constraints of the invoking resource. When the previous resource constraint of a resource constraint is known, the demon `d` is executed on the resource constraint whose previous resource constraints have become known. This previous resource constraint can then be accessed with the member function `IlcResourceConstraint::getPrevRC`. When the teardown resource constraints is known, the resource

demon `d` is executed on an empty handle resource constraint.

This member function should be used only in search and only if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`).

```
public void whenRankedFirstRC(const IlcDemon demon) const
```

This member function associates a demon with a change of the set of resource constraints currently ranked first on the resource. When this set changes because some resource constraint has just been ranked first, the demon is executed. In the execution code of the demon, the delta iterator `IlcResource::ResourceConstraintDeltaIterator` is available to iterate over the freshly ranked first resource constraints.

Since a constraint is also a demon, a constraint can be passed as an argument to this member function. When the change event is triggered, the constraint is posted and propagated.

This function should be used only if the ranking information is available on the resource (see `IlcResource::hasRankInfo`).

```
public void whenRankedLastRC(const IlcDemon demon) const
```

This member function associates a demon with a change of the set of resource constraints currently ranked last on the resource. When this set changes because some resource constraint has just been ranked last, the demon is executed. In the execution code of the demon, the delta iterator `IlcResource::ResourceConstraintDeltaIterator` is available to iterate over the freshly ranked last resource constraints.

Since a constraint is also a demon, a constraint can be passed as an argument to this member function. When the change event is triggered, the constraint is posted and propagated.

This function should be used only if the ranking information is available on the resource (see `IlcResource::hasRankInfo`).

```
public void whenSuccessors(const IlcResourceDemon d)
```

This member function associates the resource demon `d` with changes to the set of resource constraints that are successors of a resource constraint of the invoking resource. When the set of resource constraints that are successors of a resource constraint changes because some new successors have appeared, the demon `d` is executed on the resource constraint whose set of successors has changed.

This member function should be used only in search and only if some precedence information is available on the resource (see member function `IlcResource::hasPrecedenceInfo`).

Inner Enumerations

Enumeration RankFilter

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>

This enumeration allows specifying a subset of resource constraints to traverse with the iterators `IlcResource::ResourceConstraintIterator`, and `IlcResource::ResourceConstraintDeltaIterator`.

`RankedFirst` indicates the subset of resource constraints that have been ranked first on the resource.

`RankedLast` indicates the subset of resource constraints that have been ranked last on the resource.

`NotRanked` indicates the subset of resource constraints that have not yet been ranked first or last on the resource.

`PossibleFirst` indicates the subset of resource constraints that can possibly be ranked first on the resource. This is the set of resource constraints that has not yet been ranked first nor ranked last, nor otherwise determined to be unavailable to be ranked first.

`PossibleLast` indicates the subset of resource constraints that can possibly be ranked last on the resource. This is the set of resource constraints that has not yet been ranked first nor ranked last, nor otherwise determined to be unavailable to be ranked last.

See Also: IlcResource::ResourceConstraintIterator, IlcResource::ResourceConstraintDeltaIterator

Fields:

```
RankedFirst = 1

RankedLast = 2

Ranked = 3

NotRanked = 4

PossibleFirst = 8

PossibleLast = 16
```

Class IlcResourceConstraint

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>

Instances of the class `IlcResourceConstraint` are *resource constraints*. This class inherits from the Solver class `IlcConstraint`, which is documented in the *Solver Reference Manual*.

Instances of this class are created by these member functions:

- `IlcActivity::requires`
- `IlcActivity::provides`
- `IlcActivity::consumes`
- `IlcActivity::produces`
- `IlcActivity::requiresNot`

For more information, see Precedence Graph Constraints, Sequence Constraint, Metaconstraints, and `IlcConstraint` in the IBM ILOG Solver Reference Manual.

See Also: IlcActivity, IlcResource, IlcResourceConstraintIterator, IlcTimeExtent, IlcTryRankFirst, IlcTryRankLast, IlcUnaryResource

Constructor Summary	
`public`	`IlcResourceConstraint()`
`public`	`IlcResourceConstraint(IlcResourceConstraintI * impl)`

Method Summary	
`public IlcActivity`	`getActivity() const`
`public IlcCalendar`	`getCalendar() const`
`public IlcInt`	`getCapacity() const`
`public IlcIntVar`	`getCapacityVariable() const`
`public IlcResourceConstraintI *`	`getImpl() const`
`public IlcResourceConstraint`	`getNextRC() const`
`public IlcIntVar`	`getNextVar() const`
`public IlcResourceConstraint`	`getPrevRC() const`
`public IlcIntVar`	`getPrevVar() const`
`public IlcResource`	`getResource() const`
`public IlcInt`	`getSequenceIndex() const`
`public IlcShape`	`getShape() const`
`public IlcFloat`	`getSlope() const`
`public IlcSlopeConstraintMode`	`getSlopeConstraintMode() const`
`public IlcAny`	`getState() const`
`public IlcAnySet`	`getStateSet() const`
`public IlcAnySetVar`	`getStateSetVariable() const`
`public IlcAnyVar`	`getStateVariable() const`
`public IlcTimeExtent`	`getTimeExtent() const`

public IlcBool	hasAsNext(IlcResourceConstraint) const
public IlcBool	hasAsPossibleNext(IlcResourceConstraint) const
public IlcBool	hasCalendar() const
public IlcBool	hasNextRC() const
public IlcBool	hasPrevRC() const
public IlcBool	hasShape() const
public IlcBool	hasSlope() const
public IlcBool	isCapacityConstraint() const
public IlcBool	isDirectlySucceededBy(IlcResourceConstraint) const
public IlcBool	isInwardConstraint() const
public IlcBool	isNegativeConstraint() const
public IlcBool	isNotVisited() const
public IlcBool	isPossibleFirst() const
public IlcBool	isPossibleLast() const
public IlcBool	isPossibleSetup() const
public IlcBool	isPossibleTeardown() const
public IlcBool	isProvidingConstraint() const
public IlcBool	isRanked() const
public IlcBool	isRankedFirst() const
public IlcBool	isRankedLast() const
public IlcBool	isSetup() const
public IlcBool	isStateConstraint() const
public IlcBool	isStateSetConstraint() const
public IlcBool	isSucceededBy(IlcResourceConstraint) const
public IlcBool	isTeardown() const
public IlcBool	isVariableResourceConstraint() const
public IlcBool	isVirtualNode() const
public IlcBool	isVisited() const
public IlcVariableSlopeShape	makeVariableSlopeShape(IlcFloatVar slope) const
public void	operator=(const IlcResourceConstraint & h)
public IlcBool	possiblyContributes() const
public void	rankFirst()
public void	rankLast()
public void	rankNotFirst()
public void	rankNotLast()
public void	removeShape() const
public void	setCalendar(IlcCalendar cal)
public void	setNext(IlcResourceConstraint ct)
public void	setNotNext(IlcResourceConstraint ct)
public void	setNotSetup()
public void	setNotTeardown()

public void	setNotVisited()
public void	setSetup()
public void	setSlope(IlcFloat slope, IlcSlopeConstraintMode mode=IlcRoundedCapacity)
public void	setSuccessor(IlcResourceConstraint ct)
public void	setTeardown()
public void	setVisited()
public IlcBool	surelyContributes() const
public void	unsetNext()
public void	unsetSetup()
public void	unsetSuccessor(IlcResourceConstraint ct)
public void	unsetTeardown()

Constructors

public **IlcResourceConstraint**()

This constructor creates an instance which is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

public **IlcResourceConstraint**(IlcResourceConstraintI * impl)

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

Methods

public IlcActivity **getActivity**() const

This member function returns the activity of the invoking resource constraint.

public IlcCalendar **getCalendar**() const

This member function returns the calendar attached to the invoking resource constraint, if such an object exists.

public IlcInt **getCapacity**() const

This member function returns the required or provided quantity of the invoking resource constraint.

public IlcIntVar **getCapacityVariable**() const

This member function returns the variable representing the required or provided quantity of the invoking resource constraint.

public IlcResourceConstraintI * **getImpl**() const

This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcResourceConstraint getNextRC() const
```

This member function returns the resource constraint of the next activity of the invoking resource constraint. The resource constraint must be of time extent `IlcFromStartToEnd`. This member function should be called only if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`).

```
public IlcIntVar getNextVar() const
```

This member function returns the next variable of the invoking resource constraint. The resource constraint must be of time extent `IlcFromStartToEnd`. The resource must be an instance of `IlcUnaryResource`, closed, and with its sequence constraint created. Let `Nb` be the number of resource constraints of time extent `IlcFromStartToEnd`. When the sequence constraint is posted, the next variable is initialized to the following set.

$$\{-1\} \cup \langle 1, Nb + 1 \rangle$$

If the value of the next variable is `Nb + 1`, the resource constraint is the last on the resource. If its value is -1, the resource constraint is not visited, that is, either the processing time or the required capacity is zero.

```
public IlcResourceConstraint getPrevRC() const
```

This member function returns the resource constraint of the previous activity of the invoking resource constraint. The resource constraint must be of time extent `IlcFromStartToEnd`. This member function should be called only if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`).

```
public IlcIntVar getPrevVar() const
```

This member function returns the previous variable of the invoking resource constraint. The resource constraint must be of time extent `IlcFromStartToEnd`. The resource must be an instance of `IlcUnaryResource`, closed, and with its sequence constraint created. Let `Nb` be the number of resource constraints of time extent `IlcFromStartToEnd`. When the sequence constraint is posted, the previous variable is initialized to the following set.

$$\{-1\} \cup \langle 0, Nb \rangle$$

If the value of the previous variable is 0 (zero), the resource constraint is the first on the resource. If its value is -1, the resource constraint is not visited, that is, either the processing time or the required capacity is zero.

```
public IlcResource getResource() const
```

This member function returns the resource of the invoking resource constraint.

```
public IlcInt getSequenceIndex() const
```

This member function returns the unique index of the invoking resource constraint used by the sequence constraint attached to its resource.

The resource constraint must be of time extent `IlcFromStartToEnd`. The resource must be an instance of `IlcUnaryResource`, closed, and with its sequence constraint created.

To obtain the resource constraint from an index, use the accessor `IlcUnaryResource::getSequenceRC`.

```
public IlcShape getShape() const
```

This function returns the instance of `IlcShape` associated with the resource constraint. An error will be raised if no such shape has been created.

```
public IlcFloat getSlope() const
```

This member function returns the slope value of the slope constraint created by the member function `setSlope`. An instance of `IloSolver::SolverErrorException` is thrown if the invoking resource constraint has no slope constraint or if its resource is not a continuous reservoir.

```
public IlcSlopeConstraintMode getSlopeConstraintMode() const
```

This member function returns the rounding mode of the slope constraint created by the member function `IlcResourceConstraint::setSlope`. An instance of `IloSolver::SolverErrorException` is thrown if the invoking resource constraint has no slope constraint or if its resource is not a continuous reservoir.

```
public IlcAny getState() const
```

This member function returns the required state of the invoking resource constraint.

```
public IlcAnySet getStateSet() const
```

This member function returns the required set of states of the invoking resource constraint.

```
public IlcAnySetVar getStateSetVariable() const
```

This member function returns the variable representing the required set of states of the invoking resource constraint.

```
public IlcAnyVar getStateVariable() const
```

This member function returns the variable representing the required state of the invoking resource constraint.

```
public IlcTimeExtent getTimeExtent() const
```

This member function returns the time extent of the invoking resource constraint.

```
public IlcBool hasAsNext(IlcResourceConstraint) const
```

Before search, this function returns `IlcTrue` if and only if a next relation has been added with the member function `IlcResourceConstraint::setNext(rct)`. In search, this member function returns `IlcTrue` if `rct`

is next to the invoking resource constraint. Otherwise, it returns `IlcFalse`. This member function should be called only if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`).

```
public IlcBool hasAsPossibleNext(IlcResourceConstraint) const
```

This member function returns `IlcTrue` if it is possible that `rct` is next to the invoking resource constraint. Otherwise, it returns `IlcFalse`.

This member function should be used only in search and only if some precedence information is available on the resource (see member function `IlcResource::hasPrecedenceInfo`).

```
public IlcBool hasCalendar() const
```

This member function returns `IloTrue` if a calendar has been attached to the invoking resource constraint. Otherwise, it returns `IloFalse`.

```
public IlcBool hasNextRC() const
```

This member function returns `IlcTrue` if the immediately following activity of the activity of the invoking resource constraint is known. Otherwise it returns `IlcFalse`.

The resource constraint must be of time extent `IlcFromStartToEnd`. This member function should be called only if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`)

There is no possible next resource constraint if the invoking resource constraint does not contribute or is the teardown resource constraint.

```
public IlcBool hasPrevRC() const
```

This member function returns `IlcTrue` if the immediately preceding activity of the activity of the invoking resource constraint is known. Otherwise it returns `IlcFalse`.

The resource constraint must be of time extent `IlcFromStartToEnd`. This member function should be called only if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`)

There is no possible previous resource constraint if the invoking resource constraint does not contribute or is the setup resource constraint.

```
public IlcBool hasShape() const
```

This function returns `IlcTrue` if a shape has been associated with the resource constraint

```
public IlcBool hasSlope() const
```

This member function returns `IlcTrue` if the resource of the invoking resource constraint is a continuous reservoir and if a slope constraint has been defined by the member function `IlcResourceConstraint::setSlope`. Otherwise, it returns `IlcFalse`.

```
public IlcBool isCapacityConstraint() const
```

This member function returns `IlcTrue` if and only if the invoking resource constraint indicates that a quantity (and thus not a state) is required or provided.

```
public IlcBool isDirectlySucceededBy(IlcResourceConstraint) const
```

This member function returns `IlcTrue` if the invoking resource constraint is directly succeeded by the resource constraint `rct`. Otherwise, it returns `IlcFalse`.

This member function should be called only if some precedence information is available on the resource (see member function `IlcResource::hasPrecedenceInfo`).

```
public IlcBool isInwardConstraint() const
```

This member function returns `IlcTrue` if and only if the occupancy of the resource by the invoking constraint is to be rounded inward towards the nearest valid time that corresponds to a time step. This rounding is important only when one of the timetables of the resource has a time step greater than 1 (one).

```
public IlcBool isNegativeConstraint() const
```

This member function returns `IlcTrue` if and only if the invoking constraint was constructed by the member function `IlcActivity::requiresNot`, or was extracted from an `IloResourceConstraint` that was constructed by the member function `IloActivity::requiresNot`

```
public IlcBool isNotVisited() const
```

This member function returns `IlcTrue` if the activity of the invoking resource constraint is not visited by the path defined by the sequence constraint attached to its resource. Otherwise it returns `IlcFalse`. Not visited means that the processing time or the required capacity is zero.

The resource constraint must be of time extent `IlcFromStartToEnd`. The resource must be an instance of `IlcUnaryResource`, closed, and with its sequence constraint created.

```
public IlcBool isPossibleFirst() const
```

This member function returns `IlcTrue` if the activity that corresponds to the invoking resource constraint can be ranked first among those that have not yet been ranked. Otherwise, it returns `IlcFalse`. In particular, it returns `IlcFalse` if the activity has already been ranked.

This member function should be used *only* if the resource that corresponds to the invoking resource constraint is either a unary resource (an instance of `IlcUnaryResource`) or a state resource (an instance of `IlcStateResource`).

This function should be used only if the ranking information is available on the resource (see `IlcResource::hasRankInfo`).

```
public IlcBool isPossibleLast() const
```

This member function returns `IlcTrue` if the activity that corresponds to the invoking resource constraint can be ranked last among those that have not yet been ranked. Otherwise, it returns `IlcFalse`. In particular, it returns `IlcFalse` if the activity has already been ranked.

This member function should be used *only* if the resource that corresponds to the invoking resource constraint is either a unary resource (an instance of `IlcUnaryResource`) or a state resource (an instance of `IlcStateResource`).

This function should be used only if the ranking information is available on the resource (see `IlcResource::hasRankInfo`).

```
public IlcBool isPossibleSetup() const
```

This member function returns `IlcTrue` if it is possible that the invoking resource constraint is a setup resource constraint. Otherwise, it returns `IlcFalse`.

This member function should be used only in search and only if some precedence information is available on the resource (see member function `IlcResource::hasPrecedenceInfo`).

```
public IlcBool isPossibleTeardown() const
```

This member function returns `IlcTrue` if it is possible that the invoking resource constraint is a teardown resource constraint. Otherwise, it returns `IlcFalse`.

This member function should be used only in search and only if some precedence information is available on the resource (see member function `IlcResource::hasPrecedenceInfo`).

```
public IlcBool isProvidingConstraint() const
```

This member function returns `IlcTrue` if and only if the invoking constraint was constructed by the member functions `IlcActivity::provides`, or `IlcActivity::produces`, or was extracted from an `IloResourceConstraint` that was constructed by the member function `IloActivity::requiresNot`.

```
public IlcBool isRanked() const
```

This member function returns `IlcTrue` if and only if the invoking resource constraint has been ranked first or last on the resource; otherwise it returns `IlcFalse`.

This member function is available in search only. It should be used *only* if the resource that corresponds to the invoking resource constraint is either a unary resource (an instance of `IlcUnaryResource`) or a state resource (an instance of `IlcStateResource`). This member function is available in search only.

This function should be used only if the ranking information is available on the resource (see `IlcResource::hasRankInfo`).

```
public IlcBool isRankedFirst() const
```

This member function returns `IlcTrue` if and only if the invoking resource constraint has been ranked first on the resource; otherwise it returns `IlcFalse`.

This member function is available in search only. It should be used *only* if the resource that corresponds to the invoking resource constraint is either a unary resource (an instance of `IlcUnaryResource`) or a state resource

(an instance of IlcStateResource). This member function is available in search only.

This function should be used only if the ranking information is available on the resource (see IlcResource::hasRankInfo).

```
public IlcBool isRankedLast() const
```

This member function returns IlcTrue if and only if the invoking resource constraint has been ranked last on the resource; otherwise it returns IlcFalse.

This member function is available in search only. It should be used *only* if the resource that corresponds to the invoking resource constraint is either a unary resource (an instance of IlcUnaryResource) or a state resource (an instance of IlcStateResource). This member function is available in search only.

This function should be used only if the ranking information is available on the resource (see IlcResource::hasRankInfo).

```
public IlcBool isSetup() const
```

Before search, this function returns IlcTrue if and only if the invoking resource constraint has been constrained to be a setup resource constraint with the member function IlcResourceConstraint::setSetup.

In search, this member function returns IlcTrue if the invoking resource constraint is a setup resource constraint. Otherwise, it returns IlcFalse.

This member function should be called only if some rank information is available on the resource (see member function IlcResource::hasRankInfo).

```
public IlcBool isStateConstraint() const
```

This member function returns IlcTrue if and only if the invoking resource constraint indicates that a single state (and thus not a quantity or one of a set of states) is required.

```
public IlcBool isStateSetConstraint() const
```

This member function returns IlcTrue if and only if the invoking resource constraint indicates that one of a set of states (and thus not a quantity or a single state) is required.

```
public IlcBool isSucceededBy(IlcResourceConstraint) const
```

Before search, this function returns IlcTrue if and only if a successor relation has been added with the member function IlcResourceConstraint::setSuccessor(rct).

In search, this member function returns IlcTrue if the invoking resource constraint is succeeded by the resource constraint rct. Otherwise, it returns IlcFalse.

This member function should be called only if some precedence information is available on the resource (see member function IlcResource::hasPrecedenceInfo).

```
public IlcBool isTeardown() const
```

Before search, this function returns `IlcTrue` if and only if the invoking resource constraint has been constrained to be a teardown resource constraint with the member function `IlcResourceConstraint::setTeardown`.

In search, this member function returns `IlcTrue` if the invoking resource constraint is a teardown resource constraint. Otherwise, it returns `IlcFalse`.

This member function should be called only if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`).

```
public IlcBool isVariableResourceConstraint() const
```

This member function returns `IlcTrue` if and only if the invoking resource constraint has a variable representing the required quantity or state or provided quantity.

```
public IlcBool isVirtualNode() const
```

This member function returns true if and only if the resource constraint requires a unary resource and represents the sequence virtual node of this unary resource. The sequence virtual node of a unary resource is an automatically created resource constraint that do not affect the availability of the resource and that is used in the sequence goals and selectors to represent the virtual initial (in case of a chronological sequence goal like `IlcSequence`) or final (in case of an anti-chronological sequence goal like `IlcSequenceBackward`) resource constraint in the sequence of resource constraints of the unary resource .

```
public IlcBool isVisited() const
```

This member function returns `IlcTrue` if the activity of the invoking resource constraint is visited by the path defined by the sequence constraint attached to its resource. Otherwise it returns `IlcFalse`. Visited means that the processing time is strictly positive and the required capacity is one.

The resource constraint must be of time extent `IlcFromStartToEnd`. The resource must be an instance of `IlcUnaryResource`, closed, and with its sequence constraint created.

```
public IlcVariableSlopeShape makeVariableSlopeShape(IlcFloatVar slope) const
```

This function associates an instance of `IlcVariableSlopeShape` with the resource constraint. Shapes are only available on continuous reservoirs. An exception will be thrown if the minimal value of the slope variable is strictly negative.

See Also: IlcShape, IlcVariableSlopeShape

```
public void operator=(const IlcResourceConstraint & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument. After execution of this operator, the invoking object and the provided argument point to the same implementation object.

```
public IlcBool possiblyContributes() const
```

This member function returns `IlcTrue` if the invoking resource constraint possibly affects the availability of the resource. Otherwise, it returns `IlcFalse`.

`public void` **`rankFirst`**`()`

This member function states that the activity that corresponds to the invoking resource constraint is the *first* activity to execute among those that have not yet been ranked. This member function should be used *only* if the resource that corresponds to the invoking resource constraint is either a unary resource (an instance of `IlcUnaryResource`) or a state resource (an instance of `IlcStateResource`).

The ranking has no effect on the start and end times of activities that do not truly require the resource, that is, activities for which the duration or the amount of required capacity can be 0 (zero). In other words, if the activity that corresponds to the invoking resource constraint truly requires its resource, then it is constrained to execute *before* any other unranked activity that truly requires the resource.

Any of the following cases will raise an error:

- if no rank information (see `IlcResource::hasRankInfo`) is associated with the resource of the invoking resource constraint;
- if the time extent of the invoking resource constraint is not `IlcFromStartToEnd`; (Indeed, it does not make sense to rank the activity if the time extent is not `IlcFromStartToEnd`;
- if the invoking resource constraint has already been ranked (using either `rankFirst` or `rankLast`).

`public void` **`rankLast`**`()`

This member function states that the activity that corresponds to the invoking resource constraint is the *last* activity to execute among those that have not yet been ranked. This member function should be used *only* if the resource that corresponds to the invoking resource constraint is either a unary resource (an instance of `IlcUnaryResource`) or a state resource (an instance of `IlcStateResource`).

The ranking has no effect on the start and end times of activities that do not truly require the resource, that is, activities for which the duration or the amount of required capacity can be 0 (zero). In other words, if the activity that corresponds to the invoking resource constraint truly requires its resource, then it is constrained to execute *after* any other unranked activity that truly requires the resource.

Any of the following cases will raise an error:

- if no rank information (see `IlcResource::hasRankInfo`) is associated with the resource of the invoking resource constraint;
- if the time extent of the invoking resource constraint is not `IlcFromStartToEnd`; (Indeed, it does not make sense to rank the activity if the time extent is not `IlcFromStartToEnd`, a value of the enumeration `IlcTimeExtent`.)
- if the invoking resource constraint has already been ranked (using either `rankFirst` or `rankLast`).

`public void` **`rankNotFirst`**`()`

This member function states that the activity that corresponds to the invoking resource constraint is not the first activity to execute among those that have not yet been ranked. This member function should be used *only* if the resource that corresponds to the invoking resource constraint is either a unary resource (an instance of `IlcUnaryResource`) or a state resource (an instance of `IlcStateResource`).

Any of the following cases will raise an error:

- if no rank information (see `IlcResource::hasRankInfo`) is associated with the resource of the invoking resource constraint;
- if the time extent of the invoking resource constraint is not `IlcFromStartToEnd`. (Indeed, it does not make sense to rank the activity if the time extent is not `IlcFromStartToEnd`, a value of the enumeration `IlcTimeExtent`.)

```
public void rankNotLast()
```

This member function states that the activity that corresponds to the invoking resource constraint is not the last activity to execute among those that have not yet been ranked. This member function should be used *only* if the resource that corresponds to the invoking resource constraint is either a unary resource (an instance of `IlcUnaryResource`) or a state resource (an instance of `IlcStateResource`).

Any of the following cases will raise an error:

- if no rank information (see `IlcResource::hasRankInfo`) is associated with the resource of the invoking resource constraint;
- if the time extent of the invoking resource constraint is not `IlcFromStartToEnd`. (Indeed, it does not make sense to rank the activity if the time extent is not `IlcFromStartToEnd`, a value of the enumeration `IlcTimeExtent`.)

```
public void removeShape() const
```

This function remove the instance of `IlcShape` associated with the resource constraint. This member function must be used outside the search. An error will be raised if no such shape has been created.

```
public void setCalendar(IlcCalendar cal)
```

This member function attaches the calendar `cal` to the invoking resource constraint.

Outside the search, it is possible to attach a new calendar to a resource constraint that already has a calendar attached. In that case the new attachment replaces the previous one.

During search, it is possible to attach a calendar only to a resource constraint that does not have a previously attached calendar. In that case, the attachment is reversible. During search, any attempt to attach a new calendar to a resource constraint that already has an attached calendar raises an error.

When two different calendars are attached respectively to a resource constraint and to its corresponding resource, only the one on the resource constraint is taken into account. That is, if some breaks are attached to the resource *Res* using the calendar *Cal1* and some shifts are attached to a resource constraint *Rct* of *Res* using the calendar *Cal2*, then only shifts are taken into account on *Rct* (*Cal2*). In other words, breaks of *cal1* are ignored (*Cal1*).

Notice that calendars can be shared between resources and resource constraints, and that `setCalendar` does not imply that a local copy of the calendar is made; one should be aware of the fact that any change to a shared calendar holds for all resources and resource constraints sharing the calendar.

```
public void setNext(IlcResourceConstraint ct)
```

This member function states that `ct` is next to the invoking resource constraint. This means that there cannot exist any resource constraint *ct0* such that *ct0* definitely affects the availability of the resource and *ct0* starts or finishes between the end of the invoking resource constraint and the start of `ct`.

This member function should be called only on unary resources and if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`).

```
public void setNotNext(IlcResourceConstraint ct)
```

This member function states that `ct` cannot be next to the invoking resource constraint. This means that there must exist a resource constraint *ct0* such that *ct0* definitely affects the availability of the resource and *ct0* starts or finishes between the end of the invoking resource constraint and the start of `ct`.

This member function should be called only on unary resources and only if some precedence information is available on the resource (see member function `IlcResource::hasPrecedenceInfo`).

```
public void setNotSetup()
```

This member function states that the invoking resource constraint cannot be a setup resource constraint which means that some resource constraint must necessarily be previous to it.

This member function should be used only if a precedence graph constraint or a sequence constraint has been created on the resource. It can be used before or during search.

This member function should be called only on unary resources and if some precedence information is available on the resource (see member function `IlcResource::hasPrecedenceInfo`).

```
public void setNotTeardown()
```

This member function states that the invoking resource constraint cannot be a teardown resource constraint which means that some other resource constraint *rct* must exist such that the invoking resource constraint is previous to *rct*.

This member function should be called only on unary resources and if some precedence information is available on the resource (see member function `IlcResource::hasPrecedenceInfo`).

```
public void setNotVisited()
```

This member function sets the activity of the invoking resource constraint to "not visited" by the path defined by the sequence constraint attached to its resource. Not visited means that the processing time or the required capacity is zero.

The resource constraint must be of time extent `IlcFromStartToEnd`. The resource must be an instance of `IlcUnaryResource`, closed, and with its sequence constraint created.

```
public void setSetup()
```

This member function states that the invoking resource constraint is a setup resource constraint which means that no resource constraint can be previous to it.

This member function should be called only on unary resources and only if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`).

```
public void setSlope(IlcFloat slope, IlcSlopeConstraintMode
mode=IlcRoundedCapacity)
```

This member function creates a slope constraint for the invoking resource constraint, that is the ratio between the capacity of the invoking resource constraint and the duration of its activity is constrained to be `slope`, with a rounding mode defined by `mode`. This constraint is internally added to the solver.

The resource of the invoking resource constraint must be a continuous reservoir. The slope of a resource constraint cannot be modified in search but a slope can be set to a resource constraint with no slope defined yet.

```
public void setSuccessor(IlcResourceConstraint ct)
```

This member function states that the invoking resource constraint has the resource constraint `ct` as successor on the precedence graph of the resource. That is, this member function adds an edge on the precedence graph.

This member function should be called only if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`).

```
public void setTeardown()
```

This member function states that the invoking resource constraint is a teardown resource constraint which means that no resource constraint *rct* can exist such that the invoking resource constraint is previous to *rct*.

This member function should be called only on unary resources and if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`).

```
public void setVisited()
```

This member function sets the activity of the invoking resource constraint to "visited" by the path defined by the sequence constraint attached to its resource. Visited means that the processing time is strictly positive and the required capacity is one.

The resource constraint must be of time extent `IlcFromStartToEnd`. The resource must be an instance of `IlcUnaryResource`, closed, and with its sequence constraint created.

```
public IlcBool surelyContributes() const
```

This member function returns `IlcTrue` if the invoking resource constraint definitely affects the availability of the resource. Otherwise, it returns `IlcFalse`.

```
public void unsetNext()
```

This member function removes a next relation that was previously added on the graph with `IlcResourceConstraint::setNext`.

This member function should be used only before search and only if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`).

```
public void unsetSetup()
```

This member function removes the information that the invoking resource constraint must be a setup resource constraint as stated by `IlcResourceConstraint::setSetup`.

This member function should be used only before search and only if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`).

```
public void unsetSuccessor(IlcResourceConstraint ct)
```

This member function removes a successor relation that was previously added on the graph with `IlcResourceConstraint::setSuccessor`.

This member function should be called only if some precedence information is available on the resource (see member function `IlcResource::hasPrecedenceInfo`).

```
public void unsetTeardown()
```

This member function removes the information that the invoking resource constraint must be a teardown resource constraint as stated by `IlcResourceConstraint::setTeardown`.

This member function should be used only before search and only if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`).

Class IlcResourceConstraintDeltaIterator

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>

An instance of this class traverses a delta set of resource constraints (for example, the new direct predecessors of a given resource constraint).

See Also: IlcActivity, IlcResource, IlcResourceConstraint, IlcResourceConstraintIteratorFilter

Constructor and Destructor Summary	
public	`IlcResourceConstraintDeltaIterator(IlcResourceConstraint constraint, IlcResourceConstraintIteratorFilter filter)`

Method Summary	
`public IlcActivity`	`getActivity() const`
`public IlcResource`	`getResource() const`
`public IlcBool`	`ok() const`
`public IlcResourceConstraint`	`operator*() const`
`public IlcResourceConstraintDeltaIterator &`	`operator++()`

Constructors and Destructors

`public` **`IlcResourceConstraintDeltaIterator`**`(IlcResourceConstraint constraint, IlcResourceConstraintIteratorFilter filter)`

This constructor creates a delta iterator to traverse the elements of the subset of resource constraints specified by the filter whose status has changed with respect to `constraint`. The possible filters are `IlcDirectPredecessors`, `IlcDirectSuccessors`, `IlcPredecessors`, `IlcSuccessors`. The possible statuses of a resource constraint with respect to `constraint` are: unranked, direct predecessor, direct successor, indirect predecessor, indirect successor.

Thus, with the filter `IlcDirectSuccessors` or `IlcDirectPredecessors`, the delta iterator traverses the set of new direct successors or direct predecessors of `constraint`.

With the filter `IlcSuccessors` or `IlcPredecessors`, the delta iterator traverses the union of the set of new direct successors and the set of new indirect successors or the union of the set of new direct predecessors and the set of new indirect predecessors of `constraint`. This delta set is a superset of the set of new successors or new predecessors of `constraint` because any resource constraint whose status changes from direct successor to indirect successor or from direct predecessor to indirect predecessor will be traversed by this delta iterator even though it was already a successor or predecessor.

Note

The delta sets of resource constraints are emptied when all the demons attached to the graph events of a resource constraint have been executed. Thus, any attempt to traverse a delta set of resource constraints outside the execution of such a demon may lead to unexpected behavior.

This constructor can be used only when a resource precedence graph is associated with the resource required by `constraint`. If the resource is not associated with a precedence graph, an instance of `IloSolver::SolverErrorException` is thrown.

Methods

```
public IlcActivity getActivity() const
```

This member function returns the activity involved in the resource constraint located at the current position of the invoking iterator.

```
public IlcResource getResource() const
```

This member function returns the resource involved in the resource constraint located at the current position of the invoking iterator.

```
public IlcBool ok() const
```

This member function returns `IlcTrue` if the current position of the iterator is a valid one. It returns `IlcFalse` if all the constraints have been scanned by the iterator.

```
public IlcResourceConstraint operator*() const
```

This operator accesses the instance of `IlcResourceConstraint` located at the current position of the iterator. If the iterator is set past the end position, this operator returns an empty handle.

```
public IlcResourceConstraintDeltaIterator & operator++()
```

This left-increment operator shifts the current position of the iterator.

Class IlcResourceConstraintIterator

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>

An instance of this class traverses a set of resource constraints (for example, the resource constraints on a given activity or the resource constraints on a given resource, etc.).

Iterating over all the resource constraints on the resources required or provided by an activity is actually an iteration over all the resources that *might be* required or provided by the activity. If you want to iterate *only* the resource constraints that are surely required or provided by the activity, then you should use the `isTrue` member function of the class of constraints involved. The following example shows a program that correctly makes this distinction.

Example

The following program correctly makes the distinction between iterating over all the resources that might be used by an activity versus iterating only those resources that are surely used by the activity.

```
Must be during search (e.g., inside a goal)

 IloSolver solver = getSolver();
 IlcScheduler schedule(solver, 0, 50);

 IlcUnaryResource resource1(schedule);
 resource1.setName("resource 1");

 IlcUnaryResource resource2(schedule);
 resource2.setName("resource 2");

 IlcUnaryResource resource3(schedule);
 resource3.setName("resource 3");

 IlcActivity activity(schedule, 5);
 activity.setName("activity");

 solver.add(activity.requires(resource1));
 solver.add(activity.requires(resource3) || activity.requires(resource2));

 for (IlcResourceConstraintIterator ite(activity); ite.ok();
     ++ite)
   solver.out() << activity << " might require "
     << ite.getResource() << endl;
 solver.out() << endl;

 for (IlcResourceConstraintIterator ite2(activity); ite2.ok();
     ++ite2)
   if ((*ite2).isTrue())
     solver.out() << activity << " requires "
        << ite2.getResource() << endl;
 solver.out() << endl;
```

The output of that program looks like this:

```
 activity[0..45 -- 5 --> 5..50] might require resource 2[1]
 activity[0..45 -- 5 --> 5..50] might require resource 3[1]
 activity[0..45 -- 5 --> 5..50] might require resource 1[1]

 activity[0..45 -- 5 --> 5..50] requires resource 1[1]
```

For more information, see Precedence Graph Constraints.

See Also: IlcActivity, IlcResource, IlcResourceConstraint, IlcResourceConstraintIteratorFilter

Constructor and Destructor Summary	
public	IlcResourceConstraintIterator(IlcActivity activity, IlcResourceConstraintIteratorFilter filter=IlcAllConstraints)
public	IlcResourceConstraintIterator(IlcActivity activity, IlcTimeExtent extent, IlcResourceConstraintIteratorFilter filter=IlcAllConstraints)
public	IlcResourceConstraintIterator(IlcResource resource, IlcResourceConstraintIteratorFilter filter=IlcAllConstraints)
public	IlcResourceConstraintIterator(IlcResource resource, IlcTimeExtent extent, IlcResourceConstraintIteratorFilter filter=IlcAllConstraints)
public	IlcResourceConstraintIterator(IlcResourceConstraint constraint, IlcResourceConstraintIteratorFilter filter)

Method Summary	
public IlcActivity	getActivity() const
public IlcResource	getResource() const
public IlcBool	ok() const
public IlcResourceConstraint	operator*() const
public IlcResourceConstraintIterator &	operator++()

Constructors and Destructors

public **IlcResourceConstraintIterator**(IlcActivity activity, IlcResourceConstraintIteratorFilter filter=IlcAllConstraints)

This constructor creates an iterator to traverse all the resources required or provided by `activity`.

public **IlcResourceConstraintIterator**(IlcActivity activity, IlcTimeExtent extent, IlcResourceConstraintIteratorFilter filter=IlcAllConstraints)

This constructor creates an iterator to traverse all the resources required or provided by `activity` throughout the time extent indicated by `extent`.

public **IlcResourceConstraintIterator**(IlcResource resource, IlcResourceConstraintIteratorFilter filter=IlcAllConstraints)

This constructor creates an iterator to traverse all the activities that require or provide `resource`.

public **IlcResourceConstraintIterator**(IlcResource resource, IlcTimeExtent extent, IlcResourceConstraintIteratorFilter filter=IlcAllConstraints)

This constructor creates an iterator to traverse all the activities that require or provide `resource` throughout the time extent indicated by `extent`.

public **IlcResourceConstraintIterator**(IlcResourceConstraint constraint, IlcResourceConstraintIteratorFilter filter)

When a resource precedence graph is associated with the resource required by `constraint`, this constructor creates an iterator to traverse the subset of resource constraints specified by the filter given as the second argument. If the resource is not associated with a precedence graph, this constructor raises an error.

Before entering the search, only the filter `IlcSuccessors` is permitted. It allows the definition of an iterator that traverses the subset of resource constraints `rct0` for which the successor relation `(rct,rct0)` has been added via the member function `rct.IlcResourceConstraint::setSuccessor(rct0)`. Any attempt to use another filter before entering the search will raise an error. In search, all the filters are allowed.

For example, the following loop, during the search, displays the set of resource constraints that are direct successors of the resource constraint `rct` in the precedence graph of the resource:

```
for (IlcResourceConstraintIterator ite(rct, IlcDirectSuccessors);
       ite.ok();
       ++ite) {
    solver.out() << *ite << endl;
}
```

Methods

`public IlcActivity` **`getActivity`**`() const`

This member function returns the activity involved in the resource constraint located at the current position of the invoking iterator.

`public IlcResource` **`getResource`**`() const`

This member function returns the resource involved in the resource constraint located at the current position of the invoking iterator.

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if the current position of the iterator is a valid one. It returns `IlcFalse` if all the constraints have been scanned by the iterator.

`public IlcResourceConstraint` **`operator*`**`() const`

This operator accesses the instance of `IlcResourceConstraint` located at the current position of the iterator. If the iterator is set past the end position, then this operator returns an empty handle.

`public IlcResourceConstraintIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator.

Class IlcResourceDemon

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>

An instance of `IlcResourceDemon` represents a demon that is associated with all the resource constraints on a resource. An example is a demon that is triggered every time the set of successors of a resource constraint on the resource changes. An instance of this class can be created by the macro `ILCRESOURCEDEMON`.

See Also: ILCRESOURCEDEMON, IlcResource

Constructor Summary	
public	IlcResourceDemon()
public	IlcResourceDemon(IlcResourceDemonI * impl)

Method Summary	
public IlcResourceDemonI *	getImpl() const
public void	operator=(const IlcResourceDemon & h)

Constructors

`public` **`IlcResourceDemon`**`()`

This constructor creates an instance which is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

`public` **`IlcResourceDemon`**`(IlcResourceDemonI * impl)`

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

Methods

`public IlcResourceDemonI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public void` **`operator=`**`(const IlcResourceDemon & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument. After execution of this operator, the invoking object and the provided argument point to the same implementation object.

Class IlcResourceIterator

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>

An instance of this class traverses a set of resources (for example, all the resources managed by a given schedule).

See Also: IlcResource, IlcSchedule

Constructor and Destructor Summary	
public	IlcResourceIterator(const IlcSchedule schedule)

Method Summary	
public IlcBool	ok() const
public IlcResource	operator*() const
public IlcResourceIterator &	operator++()

Constructors and Destructors

`public` **`IlcResourceIterator`**`(const IlcSchedule schedule)`

This constructor creates an iterator to traverse all the resources of `schedule`.

Methods

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if the current position of the iterator is a valid one. It returns `IlcFalse` if all the resources have been scanned by the iterator.

`public IlcResource` **`operator*`**`() const`

This operator accesses the instance of `IlcResource` located at the current position of the iterator. If the iterator is set past the end position, then this operator returns an empty handle.

`public IlcResourceIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator.

Class IlcResourceTexture

Definition file: ilsched/texture.h
Include file: <ilsched/ilsched.h>

An instance of `IlcResourceTexture` represents a measure of the criticality of the resource over time. Criticality is a non-negative floating point value with 0 indicating that the minimum or maximum resource capacity constraint will be satisfied at all time points given the current time windows for all resource constraints on the resource. The higher the criticality value the more likely the resource capacity constraint will be broken at a time point.

Such a measure of criticality can be used as a basis for heuristic decision making. For example, it might be useful to identify the resource and time point with the highest overall criticality and then make a heuristic decision involving the resource constraints that possibly demand that resource at that time point.

The criticality is calculated by the aggregation of demand from each of the resource constraints that can possibly execute on the resource. This individual demand is represented, for each resource constraint, by an instance of the class `IlcRCTexture`. A simple aggregation is first performed by summing the demand (and variance of the demand) at each time point from each `IlcRCTexture` instance. These aggregate demand and variance curves can be examined using an instance of `IlcResourceTextureIterator`. The criticality curve is formed by a transformation of the aggregate demand and variance curves. This transformation takes into account the value of the resource capacity constraint (minimum or maximum) at each time point. The class `IlcTextureCriticalityCalculatorI` allows you to define the transformation from aggregate demand and aggregate variance to criticality. The common transformations are predefined in the classes `IlcProbabilisticCriticalityCalculatorI` and `IlcRelativeDemandCriticalityCalculatorI`.

You can also define the individual curves which represent the demand (and variance) of a single resource constraint for a single resource. This can be done by defining a subclass of `IlcRCTextureI` and a subclass of `IlcRCTextureFactoryI`. Again, a number of commonly used classes are predefined: `IlcRCTextureESTI`, `IlcRCTextureProbabilisticI`, and `IlcRCTextureTargetI` together with their corresponding factories: `IlcRCTextureESTFactoryI`, `IlcRCTextureProbabilisticFactoryI`, and `IlcRCTextureTargetFactoryI`.

The individual curves represented by the instances of `IlcRCTexture` and the aggregate curves represented by the instances of `IlcResourceTexture` are automatically updated when the possible time window of an activity changes.

An `IlcResourceTexture` instance must be created by using either the method `IlcCapResource::makeMaxTextureMeasurement` or `IlcCapResource::makeMinTextureMeasurement`.

The following concepts are used in the discussion of the `IlcResourceTexture` class.

- *Commitment*: A commitment is a heuristic decision. Typically, it is one branch of a choice point. For example, a commitment might be the assignment of a start time to an activity, the addition of a precedence constraint between a pair of activities, or the assignment of an alternative resource constraint to a resource. Commitments do not have to be “positive” decisions. For example, we also consider the following as commitments: specifying that the start time of an activity must *not* be a particular time point, or specifying that an alternative resource constraint must *not* execute on a particular resource.
- *Critical Time Point*: The critical time point on a resource is the time point with the highest criticality in the current search state.

For more information and examples of the use of `IlcResourceTexture` curves, see Texture Measurements, and the texture curve example in the *IBM ILOG Scheduler User's Manual*, "Using The Trace Facilities to Handle An Overconstrained Problem."

For more information, see Texture Measurements.

See Also: IlcRCTexture, IlcResourceTextureIterator, IlcDiscreteResource, IlcRCTextureFactoryI, IlcTextureCriticalityCalculatorI, IlcTextureSuccessorGoal, IlcTextureAltSuccessorGoal

Constructor Summary	
public	IlcResourceTexture()
public	IlcResourceTexture(IlcResourceTextureI * impl)

Method Summary	
public IlcFloat	getCriticalContribution(const IlcResourceConstraint ct) const
public IlcRCTextureArray	getCriticalityOrderedRCTextures() const
public IlcResourceTextureI *	getImpl() const
public IlcFloat	getMaxCriticality() const
public const char *	getName() const
public IlcAny	getObject() const
public IloSolver	getSolver() const
public IloSolverI *	getSolverI() const
public IlcFloat	getTimeOfMaxCriticality() const
public IlcBool	hasPossibleCommitments() const
public void	operator=(const IlcResourceTexture & h)
public void	resetNoCommitments() const
public void	setName(const char * name) const
public void	setNoCommitmentsAtCriticalPoint() const
public void	setObject(IlcAny object) const
public void	setRandomGenerator(IloRandom rg, IlcFloat beta=1.) const

Constructors

public **IlcResourceTexture**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcResourceTexture**(IlcResourceTextureI * impl)

This constructor creates a handle object from a pointer to an implementation object.

Methods

public IlcFloat **getCriticalContribution**(const IlcResourceConstraint ct) const

This member function returns the amount that the individual curve associated with `ct` contributes to the overall curve at the critical time point.

public IlcRCTextureArray **getCriticalityOrderedRCTextures**() const

This member function returns an array of `IlcRCTexture` instances associated with this `IlcResourceTexture` instance, ordered in descending order of the magnitude of their contribution to the aggregate curve at the critical time point.

Note

Instances of `IlcRCTextureArray` are defined using the Solver macro `ILCARRAY(IlcRCTexture)`. Refer to the *IBM ILOG Solver Reference Manual* for information on using the macro to create array classes.

```
public IlcResourceTextureI * getImpl() const
```

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcFloat getMaxCriticality() const
```

This member function returns the highest level of criticality of the texture curve.

```
public const char * getName() const
```

This member function returns the name of the invoking object.

```
public IlcAny getObject() const
```

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

```
public IloSolver getSolver() const
```

This member function returns an instance of `IloSolver` associated with the invoking object.

```
public IloSolverI * getSolverI() const
```

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

```
public IlcFloat getTimeOfMaxCriticality() const
```

This member function returns the time point of the highest criticality of the texture curve.

```
public IlcBool hasPossibleCommitments() const
```

Returns IlcTrue if there are any time points where a commitment is possible.

This member function tests if there are any time points at which there can possibly be commitments to be performed among the contending resource constraints. This function returns `IlcTrue` if there are any time points at which there is an aggregate criticality of greater than 0. Otherwise, it returns `IlcFalse`.

```
public void operator=(const IlcResourceTexture & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public void resetNoCommitments() const
```

This function removes all intervals that were previously declared to contain no possible commitments by the `IlcResourceTexture::setNoCommitmentsAtCriticalPoint` function. This function can be useful when your search consists of multiple separate goals. For example, imagine you are solving a problem involving both resource allocation and activity sequencing. You may want to assign all alternative resources using one goal and then use a different goal to add precedence constraints between resource constraints assigned to the same resource. Depending on the goal, the use of `setNoCommitmentsAtCriticalPoint()` has different semantics. In the alternative resource assignment goal, the `IlcResourceTexture::setNoCommitmentsAtCriticalPoint` is used to inform the texture measurement that there are no alternative resource assignments to be made over some time interval. When switching to the resource constraint sequencing goal, the no commitment intervals are no longer relevant: the goal is trying to add precedence constraints and so the fact that there are no alternative resource assignments to be made is irrelevant. In such a case, you can use `resetNoCommitment()` to remove all the intervals and recalculate the criticality.

```
public void setName(const char * name) const
```

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

```
public void setNoCommitmentsAtCriticalPoint() const
```

This member function informs the `IlcResourceTexture` instance that there are no commitments at its current critical point, and therefore any time points that involve only those activities that have a positive contribution for the current critical point should actually have a criticality of 0.

An example of such a situation is the following:

Activity *A*: [0 .. 10 — 10 —> 10 .. 20] and requires *R0*

Activity *B*: [10 .. 20 — 10 —> 20 .. 30] and requires *R0*

A ends before the start of *B*.

If *A* and *B* are the only activities on unary resource *R0*, and *R0* is available throughout the time interval [0, 30], then it is clear that the criticality is 0 at all time points on the interval [0, 30]. There is no possibility that the capacity constraint on *R0* will be exceeded. However, for reasons of computational complexity, the `IlcResourceTexture` may consider each activity individually in aggregating individual demand into overall criticality. This estimation leads to a non-zero criticality value on the interval [10, 20). That is, based on their time windows, both *A* and *B* can be executing at, for example, time 15. Because the texture aggregation procedure does not take into account the existence of a precedence constraint between *A* and *B*, it estimates that there is a non-zero likelihood that *bothA* and *B* will execute at time 15. Therefore, the estimated criticality is greater than 0. If 15 were estimated to be the critical time point on *R0*, some heuristics may attempt to reduce that criticality by posting a precedence constraint between the two activities only to find that such a constraint already exists and, in fact, that the true criticality is 0. At that point, this member function could be used to inform the `IlcResourceTexture` that there are no possible commitments at its critical point and to cause it to recalculate its criticality. In fact, this function analyzes all the activities that can possibly execute at the critical point and expands the interval with no possible commitments as far as possible. The reasoning is that if there are no possible commitments among the activities competing for the critical point, then there are no possible commitments at any time point where only those activities (or a subset) can possibly execute. In our example, the new texture measurement would assign a criticality of 0 to all points on the interval [0 30).

```
public void setObject(IlcAny object) const
```

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

```
public void setRandomGenerator(IloRandom rg, IlcFloat beta=1.) const
```

This member function sets the random number generator and the beta value to be used with the random number generator. The random number generator is used to choose the critical time point when a number of time points have similar criticalities. The beta value controls the interval of values which will be considered similar. If a random generator has been set, the critical time point on a texture will be found by selecting with uniform probability one time point from all of those whose criticality value lies in the interval `[`beta `* maxCrit, maxCrit]`, where `maxCrit` is the maximum criticality value of all time points on the resource. In other words, all the time points whose criticality values lie within the interval are considered to have the same criticality and so one of them is randomly selected.

Class IlcResourceTextureIterator

Definition file: ilsched/texture.h
Include file: <ilsched/ilsched.h>

An instance of `IlcResourceTextureIterator` can be used to examine the criticality and aggregate demand curves represented by an instance of `IlcResourceTexture`.

See Also: IlcResourceTexture

Constructor Summary	
public	IlcResourceTextureIterator(const IlcResourceTexture texture)
public	IlcResourceTextureIterator(const IlcResourceTexture texture, IlcFloat time)

Method Summary	
public IlcFloat	getCriticality() const
public IlcFloat	getDemand() const
public IlcFloat	getDemandDiscontinuity() const
public IlcFloat	getDemandSlope() const
public IlcFloat	getTime() const
public IlcBool	ok() const
public void	operator++()
public void	operator--()

Constructors

`public` **`IlcResourceTextureIterator`**`(const IlcResourceTexture texture)`

This constructor creates an iterator to traverse `texture`. The iterator initially points to the first segment of the texture curve.

`public` **`IlcResourceTextureIterator`**`(const IlcResourceTexture texture, IlcFloat time)`

This constructor creates an iterator to traverse `texture`. The iterator initially points to the segment of the texture curve containing `time`.

Methods

`public IlcFloat` **`getCriticality`**`() const`

This member function returns the value of the criticality at the start time of the current segment of the texture curve pointed to by the invoking iterator.

`public IlcFloat` **`getDemand`**`() const`

This member function returns the value of the aggregate demand at the start time of the current segment of the texture curve pointed to by the invoking iterator.

```
public IlcFloat getDemandDiscontinuity() const
```

This member function returns the discontinuity of the aggregate demand at the start time of the current segment of the texture curve pointed to by the invoking iterator.

```
public IlcFloat getDemandSlope() const
```

This member function returns the slope of the aggregate demand at the start time of the current segment of the texture curve pointed by the invoking iterator.

```
public IlcFloat getTime() const
```

This member function returns the start time of the current segment of the texture curve pointed to by the invoking iterator.

```
public IlcBool ok() const
```

This member function returns `IlcFalse` if the iterator does not currently indicate a segment included in the texture curve. Otherwise, it returns `IlcTrue`.

```
public void operator++()
```

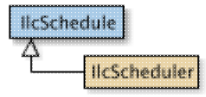
This operator moves the iterator to the segment adjacent to the current segment of the texture curve (forward move).

```
public void operator--()
```

This operator moves the iterator to the segment adjacent to the current segment of the texture curve (backward move).

Class IlcSchedule

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>



An instance of the class `IlcSchedule` is an object which represents a schedule. Although most applications will use only one schedule, it is possible to use multiple schedules, for example, to simulate distributed scheduling. A schedule manages *resources* that are required or provided by *activities*.

A schedule is associated with a time interval defined by `timeMin`, the *origin* of the schedule, and `timeMax`, the *horizon* of the schedule. By convention, the time interval is considered closed on the left and open on the right, denoted like this: `[timeMin, timeMax)`. This convention makes it possible to define time-varying parameters (such as the number of resources available) at any given time. In the Scheduler Engine library, integers are used to represent time so the value assumed by a parameter at a given `time` is the value assumed over the entire *interval* `[time, (time + 1))`. The time origin and the time horizon are used by default to initialize timetables of resources as well as earliest start times and latest end times of activities.

Printing or Displaying a Schedule

The printed representation of an instance of the class `IlcSchedule` consists of two parts: its name followed by the values of its two constructor arguments, `timeMin` and `timeMax`. These two values are enclosed in brackets and separated by two dots, like this: `[0..10]`.

If the Solver trace is active and the schedule has not been named, the string `"IlcSchedule"` followed by the address of the implementation object precedes the values of the two constructor arguments, `timeMin` and `timeMax`, enclosed in brackets.

Inverse Links

An instance of the class `IlcSchedule` may be a data member of another "external" object. In such a case, it may be useful to find the external object from the instance of `IlcSchedule`. The member functions `getObject` and `setObject` are provided to manage such an inverse link.

Handles and Implementation Classes

Like Solver, Scheduler Engine implements most of its entities by means of handle classes and implementation classes, where an object of the handle class contains a pointer (the handle pointer) to an instance of the corresponding implementation class (the implementation object). These two levels allow Scheduler to do most of the memory management for you. For more details about handle and implementation classes, see that topic in the *Solver Reference Manual*. Normally, as a Scheduler Engine user, you will exploit handles.

There are two cases in which you should use the implementation class:

1. When you define a transition time object and choose not to use the macro `IlcTransitionTime`. See `IlcTransitionTimeObjectI` for an example of deriving a new transition time object.
2. When you define a transition cost object and choose not to use the macro `IlcTransitionCost`. See `IlcTransitionCostObjectI` for an example of deriving a new transition cost object.

For more information, see Durability, and the Solver Reference Manual.

See Also: IlcActivity, IlcResource

Constructor Summary	
`public`	`IlcSchedule()`

public	IlcSchedule(IlcScheduleI * impl)

Method Summary	
public void	close()
public IlcScheduleI *	getImpl() const
public const char *	getName() const
public IlcInt	getNumberOfActivities() const
public IlcInt	getNumberOfResources() const
public IlcAny	getObject() const
public IlcConstraint	getPrecedenceGraphConstraint() const
public IloSolver	getSolver() const
public IloSolverI *	getSolverI() const
public IlcInt	getTimeMax() const
public IlcInt	getTimeMin() const
public IlcBool	hasPrecedenceGraphConstraint() const
public IlcBool	isClosed() const
public IlcBool	isDurable() const
public void	lock(IlcInt sizeRes, IlcResource * arrayRes, IlcInt sizeAlt, IlcAltResSet * arrayAlt)
public void	lock(IlcResourceArray array)
public void	lock(IlcInt size, IlcResource * array)
public IlcConstraint	makePrecedenceGraphConstraint()
public IlcBool	operator!=(const IlcSchedule & schedule) const
public void	operator=(const IlcSchedule & h)
public IlcBool	operator==(const IlcSchedule & schedule) const
public void	setDurable() const
public void	setName(const char * name) const
public void	setObject(IlcAny object) const
public void	unlock(IlcInt sizeRes, IlcResource * arrayRes, IlcInt sizeAlt, IlcAltResSet * arrayAlt)
public void	unlock(IlcResourceArray array)
public void	unlock(IlcInt size, IlcResource * array)
public void	whenDirectPredecessors(const IlcScheduleDemon)
public void	whenDirectSuccessors(const IlcScheduleDemon)
public void	whenPredecessors(const IlcScheduleDemon)
public void	whenSuccessors(const IlcScheduleDemon)

Constructors

public **IlcSchedule**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcSchedule**(IlcScheduleI * impl)

This constructor creates a handle object from a pointer to an implementation object.

Methods

```
public void close()
```

After creation of all its durable resources, a durable schedule may be closed by calling this member function. After closing, no further durable resources can be created on the invoking schedule. Closing a durable schedule is not reversible.

Calling this member function is mandatory only in multi-threaded applications and must occur before using any of the durable resources constructed on the invoking durable schedule.

This member function has relevance for durable schedules only.

```
public IlcScheduleI * getImpl() const
```

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

```
public const char * getName() const
```

This member function returns the name of the invoking object.

```
public IlcInt getNumberOfActivities() const
```

This member function returns the number of activities managed by the invoking schedule.

```
public IlcInt getNumberOfResources() const
```

This member function returns the number of resources managed by the invoking schedule.

```
public IlcAny getObject() const
```

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

```
public IlcConstraint getPrecedenceGraphConstraint() const
```

This member function returns the precedence graph constraint associated with the invoking schedule.

```
public IloSolver getSolver() const
```

This member function returns an instance of `IloSolver` associated with the invoking object.

```
public IloSolverI * getSolverI() const
```

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

```
public IlcInt getTimeMax() const
```

This member function returns the time *horizon* of the invoking schedule. This value is defined in the `IloSchedulerEnv` associated with the model extracted by the `IlcScheduler` object.

```
public IlcInt getTimeMin() const
```

This member function returns the time *origin* of the invoking schedule. This value is defined in the `IloSchedulerEnv` associated with the model extracted by the `IlcScheduler` object.

```
public IlcBool hasPrecedenceGraphConstraint() const
```

This member function returns `IlcTrue` if the invoking schedule is associated with a precedence graph constraint.

```
public IlcBool isClosed() const
```

This member function returns `IlcTrue` if the invoking schedule has been closed. Otherwise it returns `IlcFalse`.

```
public IlcBool isDurable() const
```

This member function returns `IlcTrue` if the invoking schedule has been declared as durable. Otherwise it returns `IlcFalse`.

```
public void lock(IlcInt size, IlcResource * array)
public void lock(IlcInt sizeRes, IlcResource * arrayRes, IlcInt sizeAlt,
IlcAltResSet * arrayAlt)
public void lock(IlcResourceArray array)
```

These member functions allow the invoking schedule object to lock one or several durable resources. The resources passed as arguments must be durable resources constructed on the same durable schedule.

Note

The class `IlcResourceArray` is created using the Solver macro `ILCARRAY`. Refer to the *IBM ILOG Solver Reference Manual* for information on using the macro to create array classes.

Before using one or several durable resources to define a scheduling problem, the schedule associated with the problem must lock those durable resources.

After calling `lock`, the resources are considered to be locked by the invoking schedule. The invoking schedule is called the computation schedule and its solver the computation solver.

Trying to lock resources already locked by another schedule in the same thread will raise an error. Trying to lock resources already locked by other schedules in different threads may cause the current thread to be blocked while waiting for all arguments to be unlocked by other threads.

Note that in multi-threaded applications, the use of `lock` in loops like:

```
    for(IlcInt i = 0; i < SIZE; i++)
       compSchedule.lock(1, resourceArray[i]);
```

may result in deadlocks. The correct use should be:

```
compSchedule.lock(SIZE, resourceArray);
```

Trying to lock a resource already locked by the same schedule has no effect.

The method `lock` is reversible under backtracking and the associated reversible action is `IlcSchedule::unlock` with the same arguments.

After locking by `compSchedule`, the following equalities are true for a locked resource:

```
 resource.getSchedule() == compSchedule;
 resource.getSolver() == compSchedule.getSolver();
```

A resource is locked if and only if the following test succeeds:

```
resource.getDurableSchedule() != resource.getSchedule()
```

A resource that has just been locked does not have any resource constraints or global constraints, except global constraints created on the allocation solver. If global constraints were already added on the durable schedule, they will be added automatically to the computation schedule.

Locked resources can be used for creating and adding resource constraints on the solver of the computation schedule.

```
public IlcConstraint makePrecedenceGraphConstraint()
```

This member function creates and returns the precedence graph constraint associated with the invoking schedule. The constraint must be posted in order to be taken into account.

```
public IlcBool operator!=(const IlcSchedule & schedule) const
```

This operator returns `IlcTrue` if and only if schedule does not refer to the same implementation object as the invoking schedule.

```
public void operator=(const IlcSchedule & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public IlcBool operator==(const IlcSchedule & schedule) const
```

This operator returns `IlcTrue` if and only if schedule refers to the same implementation object as the invoking schedule.

```
public void setDurable() const
```

This member function declares the invoking schedule as *durable*. This member function must be used outside the search. This solver is called the *allocation solver*.

Durable schedules should be used only to create resources and global constraints (timetable, break, and disjunctive constraints) on these resources. Such resources are called *durable resources*. Durable resources can

be grouped into alternatives (`IlcAltResSet`) on the durable schedule.

Activities cannot be created on a durable schedule and resource constraints cannot be added to the allocation solver. A schedule already containing some resources and/or activities cannot be set as durable.

Setting a schedule as durable is not reversible.

```
public void setName(const char * name) const
```

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

```
public void setObject(IlcAny object) const
```

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

```
public void unlock(IlcInt size, IlcResource * array)
public void unlock(IlcInt sizeRes, IlcResource * arrayRes, IlcInt sizeAlt,
IlcAltResSet * arrayAlt)
public void unlock(IlcResourceArray array)
```

These member functions allow the invoking schedule object to unlock one or several durable resources. The resources passed as arguments must be durable resources constructed on the same durable schedule. Unlocking may be performed during search.

Note

The class `IlcResourceArray` is created using the Solver macro `ILCARRAY`. Refer to the *IBM ILOG Solver Reference Manual* for information on using the macro to create array classes.

All constraints (resource and global) that are using the unlocked resources become "inhibited," meaning that no further propagation will occur through them. Unlocked resources keep the timetable information that was computed before they were unlocked.

For any unlocked resource the following equalities are true:

```
resource.getSchedule() == resource.getDurableSchedule();
resource.getSolver() == resource.getDurableSchedule().getSolver();
```

Unlocked resources are in a state allowing them to be locked. Further computation performed on the schedule does not affect the unlocked resource, except backtracking. Backtracking a decision that modified the timetable of a resource will undo the modification, even if the resource has been unlocked.

Do not call `unlock` during propagation. Trying to unlock a resource which is not locked by the invoking schedule has no effect.

Note that `unlock` is not reversible, that is, an unlocked resource will not be locked again under backtracking and inhibited constraints remain inhibited.

```
public void whenDirectPredecessors(const IlcScheduleDemon)
```

This member function associates the schedule demon `d` with changes to the set of activities that are direct predecessors of an activity of the invoking schedule. When the set of activities that are direct predecessors of an activity changes because some new direct predecessors have appeared, the demon `d` is executed on the activity whose set of direct predecessors has changed.

This member function has no effect until a precedence graph constraint has been created on the schedule.

```
public void whenDirectSuccessors(const IlcScheduleDemon)
```

This member function associates the schedule demon `d` with changes to the set of activities that are direct successors of an activity of the invoking schedule. When the set of activities that are direct successors of an activity changes because some new direct successors have appeared, the demon `d` is executed on the activity whose set of direct successors has changed.

This member function has no effect until a precedence graph constraint has been created on the schedule.

```
public void whenPredecessors(const IlcScheduleDemon)
```

This member function associates the schedule demon `d` with changes to the set of activities that are predecessors of an activity of the invoking schedule. When the set of activities that are predecessors of an activity changes because some new predecessors have appeared, the demon `d` is executed on the activity whose set of predecessors has changed.

This member function has no effect until a precedence graph constraint has been created on the schedule.

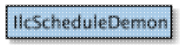
```
public void whenSuccessors(const IlcScheduleDemon)
```

This member function associates the schedule demon `d` with changes to the set of activities that are successors of an activity of the invoking schedule. When the set of activities that are successors of an activity changes because some new successors have appeared, the demon `d` is executed on the activity whose set of successors has changed.

This member function has no effect until a precedence graph constraint has been created on the schedule.

Class IlcScheduleDemon

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>



An instance of `IlcScheduleDemon` represents a demon that is associated with all the activities on a schedule. An example could be a demon that is triggered every time the set of successors of an activity in the schedule changes. An instance of this class can be created by the macro `ILCSCHEDULEDEMON`.

See Also: ILCSCHEDULEDEMON, IlcSchedule

Constructor Summary	
public	IlcScheduleDemon()
public	IlcScheduleDemon(IlcScheduleDemonI * impl)

Method Summary	
public IlcScheduleDemonI *	getImpl() const
public void	operator=(const IlcScheduleDemon & h)

Constructors

`public` **`IlcScheduleDemon`**`()`

This constructor creates an instance which is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

`public` **`IlcScheduleDemon`**`(IlcScheduleDemonI * impl)`

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

Methods

`public IlcScheduleDemonI *` **`getImpl`**`() const`

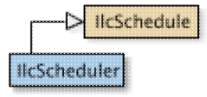
This member function returns a pointer to the implementation object of the invoking handle.

`public void` **`operator=`**`(const IlcScheduleDemon & h)`

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument. After execution of this operator, the invoking object and the provided argument point to the same implementation object.

Class IlcScheduler

Definition file: ilsched/ilocpschedextr.h
Include file: <ilsched/iloscheduler.h>



The class `IlcScheduler` is the repository of all the information used during the solve process of a scheduling problem. It allows retrieval of the search object (`Ilc`) that has been extracted from a scheduler extractable (`Ilo`).

For more information, see the concept Assert and NDEBUG in the IBM ILOG Solver Reference Manual.

Constructor Summary	
public	IlcScheduler(IloSolver solver)

Method Summary	
public IlcActivity	getActivity(const IloActivity ext) const
public IlcAltResConstraint	getAltResConstraint(const IloResourceConstraint ext) const
public IlcAltResSet	getAltResSet(const IloAltResSet ext) const
public IlcCalendar	getCalendar(const IloCalendar ext) const
public IlcCapResource	getCapResource(const IloCapResource ext) const
public IlcContinuousReservoir	getContinuousReservoir(const IloContinuousReservoir ext) const
public IlcDiscreteEnergy	getDiscreteEnergy(const IloDiscreteEnergy ext) const
public IlcDiscreteResource	getDiscreteResource(const IloDiscreteResource ext) const
public IloGranularFunction	getExtractable(IlcGranularFunction sobj) const
public IloTransitionCost	getExtractable(IlcTransitionCostObject sobj) const
public IloTransitionTime	getExtractable(IlcTransitionTimeObject sobj) const
public IloResource	getExtractable(IlcResource sobj) const
public IloActivity	getExtractable(IlcActivity sobj) const
public IloAltResSet	getExtractable(IlcAltResSet sobj) const
public IloResourceConstraint	getExtractable(IlcAltResConstraint sobj) const
public IloResourceConstraint	getExtractable(IlcResourceConstraint sobj) const
public IloTimeBoundConstraint	getExtractable(IlcTimeBoundConstraint sobj) const
public IloPrecedenceConstraint	getExtractable(IlcPrecedenceConstraint sobj) const
public IlcGranularFunction	getGranularFunction(const IloGranularFunction ext) const
public IlcPrecedenceConstraint	getPrecedenceConstraint(const IloPrecedenceConstraint ext) const
public IlcReservoir	getReservoir(const IloReservoir ext) const
public IlcResource	getResource(const IloResource ext) const
public IlcResourceConstraint	

	getResourceConstraint(const IloResourceConstraint ext) const
public IlcShiftObject	getShiftObject(const IloShiftObject ext) const
public IloSolver	getSolver() const
public IlcStateResource	getStateResource(const IloStateResource ext) const
public IlcTimeBoundConstraint	getTimeBoundConstraint(const IloTimeBoundConstraint ext) const
public IlcTransitionCostObject	getTransitionCostObject(const IloTransitionParam ext) const
public IlcTransitionCostObject	getTransitionCostObject(const IloTransitionCostObject ext) const
public IlcTransitionCostObject	getTransitionCostObject(const IloTransitionCost ext) const
public IlcTransitionTimeObject	getTransitionTimeObject(const IloTransitionTimeObject ext) const
public IlcTransitionTimeObject	getTransitionTimeObject(const IloTransitionTime ext) const
public IlcUnaryResource	getUnaryResource(const IloUnaryResource ext) const
public IloBool	hasAlternative(const IloResourceConstraint ext) const

Inherited Methods from `IlcSchedule`
close, getImpl, getName, getNumberOfActivities, getNumberOfResources, getObject, getPrecedenceGraphConstraint, getSolver, getSolverI, getTimeMax, getTimeMin, hasPrecedenceGraphConstraint, isClosed, isDurable, lock, lock, lock, makePrecedenceGraphConstraint, operator!=, operator=, operator==, setDurable, setName, setObject, unlock, unlock, unlock, whenDirectPredecessors, whenDirectSuccessors, whenPredecessors, whenSuccessors

Constructors

public **IlcScheduler**(IloSolver solver)

This constructor creates a new instance of `IlcScheduler` if none currently exists on the given instance of `IloSolver`. If a schedule has already been created on the solver, then the new handle uses it, and points to the same implementation. If no instance of `IloSchedulerEnv` had been created on the `IloEnv` that the solver is built on, then one is created.

Methods

public IlcActivity **getActivity**(const IloActivity ext) const

This member function returns the instance of `IlcActivity` that has been extracted from `ext`, the given `IloActivity`. If `ext` has not been extracted, an empty handle is returned.

public IlcAltResConstraint **getAltResConstraint**(const IloResourceConstraint ext) const

This member function returns the instance of `IlcAltResConstraint` that has been extracted from `ext`, the

given IloResourceConstraint. If ext has not been extracted, an empty handle is returned. In debug mode, an assertion will be violated if ext is not an alternative resource constraint.

```
public IlcAltResSet getAltResSet(const IloAltResSet ext) const
```

This member function returns the instance of IlcAltResSet that has been extracted from ext, the given IloAltResSet. If ext has not been extracted, an empty handle is returned.

```
public IlcCalendar getCalendar(const IloCalendar ext) const
```

This member function returns the instance of IlcCalendar that has been extracted from ext, the given IloCalendar. If extHas not been extracted, an empty handle is returned.

```
public IlcCapResource getCapResource(const IloCapResource ext) const
```

This member function returns the instance of IlcCapResource that has been extracted from ext, the given IloCapResource. If ext has not been extracted, an empty handle is returned.

```
public IlcContinuousReservoir getContinuousReservoir(const IloContinuousReservoir
ext) const
```

This member function returns the instance of IlcContinuousReservoir that has been extracted from ext, the given IloContinuousReservoir. If ext has not been extracted, an empty handle is returned.

```
public IlcDiscreteEnergy getDiscreteEnergy(const IloDiscreteEnergy ext) const
```

This member function returns the instance of IlcDiscreteEnergy that has been extracted from ext, the given IloDiscreteEnergy. If ext has not been extracted, an empty handle is returned.

```
public IlcDiscreteResource getDiscreteResource(const IloDiscreteResource ext) const
```

This member function returns the instance of IlcDiscreteResource that has been extracted from ext, the given IloDiscreteResource. If ext has not been extracted, an empty handle is returned.

```
public IloGranularFunction getExtractable(IlcGranularFunction sobj) const
```

This member function returns the instance of IloGranularFunction from which sobj has been extracted. If sobj has not been generated via an extraction, then it returns an empty handle.

```
public IloTransitionCost getExtractable(IlcTransitionCostObject sobj) const
```

This member function returns the instance of IloTransitionCost from which sobj has been extracted. If sobj has not been generated via an extraction, then it returns an empty handle.

```
public IloTransitionTime getExtractable(IlcTransitionTimeObject sobj) const
```

This member function returns the instance of `IloTransitionTime` from which `sobj` has been extracted. If `sobj` has not been generated via an extraction, then it returns an empty handle.

```
public IloResource getExtractable(IlcResource sobj) const
```

This member function returns the instance of `IloResource` from which `sobj` has been extracted. If `sobj` has not been generated via an extraction, then it returns an empty handle.

```
public IloActivity getExtractable(IlcActivity sobj) const
```

This member function returns the instance of `IloActivity` from which `sobj` has been extracted. If `sobj` has not been generated via an extraction, then it returns an empty handle.

```
public IloAltResSet getExtractable(IlcAltResSet sobj) const
```

This member function returns the instance of `IloAltResSet` from which `sobj` has been extracted. If `sobj` has not been generated via an extraction, then it returns an empty handle.

```
public IloResourceConstraint getExtractable(IlcAltResConstraint sobj) const
```

This member function returns the instance of `IloResourceConstraint` from which `sobj` has been extracted. If `sobj` has not been generated via an extraction, then it returns an empty handle.

```
public IloResourceConstraint getExtractable(IlcResourceConstraint sobj) const
```

This member function returns the instance of `IloResourceConstraint` from which `sobj` has been extracted. If `sobj` has not been generated via an extraction, then it returns an empty handle.

```
public IloTimeBoundConstraint getExtractable(IlcTimeBoundConstraint sobj) const
```

This member function returns the instance of `IloTimeBoundConstraint` from which `sobj` has been extracted. If `sobj` has not been generated via an extraction, then it returns an empty handle.

```
public IloPrecedenceConstraint getExtractable(IlcPrecedenceConstraint sobj) const
```

This member function returns the instance of `IloPrecedenceConstraint` from which `sobj` has been extracted. If `sobj` has not been generated via an extraction, then it returns an empty handle.

```
public IlcGranularFunction getGranularFunction(const IloGranularFunction ext) const
```

This member function returns the instance of `IlcGranularFunction` that has been extracted from `ext`, the given `IloGranularFunction`. If `ext` has not been extracted, an empty handle is returned.

```
public IlcPrecedenceConstraint getPrecedenceConstraint(const
IloPrecedenceConstraint ext) const
```

This member function returns the instance of `IlcPrecedenceConstraint` that has been extracted from `ext`, the given `IloPrecedenceConstraint`. If `ext` has not been extracted, an empty handle is returned.

```
public IlcReservoir getReservoir(const IloReservoir ext) const
```

This member function returns the instance of `IlcReservoir` that has been extracted from `ext`, the given `IloReservoir`. If `ext` has not been extracted, an empty handle is returned.

```
public IlcResource getResource(const IloResource ext) const
```

This member function returns the instance of `IlcResource` that has been extracted from `ext`, the given `IloResource`. If `ext` has not been extracted, an empty handle is returned.

```
public IlcResourceConstraint getResourceConstraint(const IloResourceConstraint ext)
const
```

This member function returns the instance of `IlcResourceConstraint` that has been extracted from `ext`, the given `IloResourceConstraint`. If `ext` has not been extracted, an empty handle is returned. In debug mode, an assertion will be violated if `ext` is an alternative resource constraint.

```
public IlcShiftObject getShiftObject(const IloShiftObject ext) const
```

This member function returns the instance of `IlcShiftObject` that has been extracted from `ext`, the given `IloShiftObject`. If `ext`has not been extracted, an empty handle is returned.

```
public IloSolver getSolver() const
```

This member function returns the instance of `IloSolver` on which was built the called object.

```
public IlcStateResource getStateResource(const IloStateResource ext) const
```

This member function returns the instance of `IlcStateResource` that has been extracted from `ext`, the given `IloStateResource`. If `ext` has not been extracted, an empty handle is returned.

```
public IlcTimeBoundConstraint getTimeBoundConstraint(const IloTimeBoundConstraint
ext) const
```

This member function returns the instance of `IlcTimeBoundConstraint` that has been extracted from `ext`, the given `IloTimeBoundConstraint`. If `ext` has not been extracted, an empty handle is returned.

```
public IlcTransitionCostObject getTransitionCostObject(const IloTransitionParam
ext) const
```

This member function returns the instance of `IlcTransitionCostObject` corresponding to the extraction of `ext`, the given `IloTransitionParam`. If `ext` has not been extracted, an empty handle is returned. This member function is useful to get a transition cost object that was not associated with any unary resource in the model; for example, to use it in a selector for the goals `IlcSequence` or `IlcSequenceBackward`.

```
public IlcTransitionCostObject getTransitionCostObject(const
IloTransitionCostObject ext) const
```

This member function returns the instance of `IlcTransitionCostObject` that has been extracted from `ext`, the given `IloTransitionCostObject`. If `ext` has not been extracted, an empty handle is returned.

```
public IlcTransitionCostObject getTransitionCostObject(const IloTransitionCost ext)
const
```

This member function returns the instance of `IlcTransitionCostObject` that has been extracted from `ext`, the given `IloTransitionCost`. If `ext` has not been extracted, an empty handle is returned.

```
public IlcTransitionTimeObject getTransitionTimeObject(const
IloTransitionTimeObject ext) const
```

This member function returns the instance of `IlcTransitionTimeObject` that has been extracted from `ext`, the given `IlcTransitionTimeObject`. If `ext` has not been extracted, an empty handle is returned.

```
public IlcTransitionTimeObject getTransitionTimeObject(const IloTransitionTime ext)
const
```

This member function returns the instance of `IlcTransitionTimeObject` that has been extracted from `ext`, the given `IloTransitionTime`. If `ext` has not been extracted, an empty handle is returned.

```
public IlcUnaryResource getUnaryResource(const IloUnaryResource ext) const
```

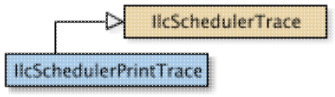
This member function returns the instance of `IlcUnaryResource` that has been extracted from `ext`, the given `IloUnaryResource`. If `ext` has not been extracted, an empty handle is returned.

```
public IloBool hasAlternative(const IloResourceConstraint ext) const
```

This member function returns `IloTrue` if the extracted counterpart of the argument `ext` is an instance of the class `IlcAltResConstraint`. It returns `IlcFalse` if the extracted counterpart of the argument `ext` is an instance of the class `IlcResourceConstraint`.

Class IlcSchedulerPrintTrace

Definition file: ilsched/schedtrace.h
Include file: <ilsched/ilsched.h>



The `IlcSchedulerPrintTrace` class is used to get printed messages of every trace event. To get those events, the manager must be in trace mode. See `IloSolver::setTraceMode` in the *IBM ILOG Solver Reference Manual*.

Example

```
solver.setTraceMode(IlcTrue);
IlcSchedulerPrintTrace trace(schedule);
trace.traceAllActivities();
trace.traceAllResources();
```

See Also: IlcSchedulerTrace, IlcSchedulerTraceFilter

Constructor Summary	
public	`IlcSchedulerPrintTrace()`
public	`IlcSchedulerPrintTrace(IlcSchedulerPrintTraceI * impl)`
public	`IlcSchedulerPrintTrace(IlcSchedule s, IlcBool traceFails=IlcTrue, const char * name=0)`

Method Summary	
public IlcSchedulerPrintTraceI *	`getImpl() const`
public void	`operator=(const IlcSchedulerPrintTrace & h)`
public void	`resetFilter() const`
public void	`setFilter(IlcSchedulerTraceFilter filter) const`

Inherited Methods from `IlcSchedulerTrace`
`trace, trace, trace, trace, trace, trace, trace, traceAllActivities, traceAllFailures, traceAllResources`

Constructors

`public` **`IlcSchedulerPrintTrace`**`()`

This constructor creates an instance which is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

`public` **`IlcSchedulerPrintTrace`**`(IlcSchedulerPrintTraceI * impl)`

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

`public` **`IlcSchedulerPrintTrace`**`(IlcSchedule s, IlcBool traceFails=IlcTrue, const char * name=0)`

This constructor creates an instance of `IlcSchedulerPrintTrace` for the `IlcSchedule` given as argument. Member functions of the base class `IlcSchedulerTrace` like `IlcSchedulerTrace::traceAllActivities` will act on the objects of this instance of `IlcSchedule`.

If the value of `traceFails` is `IlcTrue` (the default), then all the failures will be traced. Otherwise, only failures triggered by Solver variables of traced Scheduler objects will be displayed. A name can be given to the trace.

Methods

```
public IlcSchedulerPrintTraceI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public void operator=(const IlcSchedulerPrintTrace & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument. After execution of this operator, the invoking object and the provided argument point to the same implementation object.

```
public void resetFilter() const
```

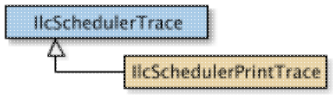
This member function resets the filter associated with the calling object. After the filter is reset, all events will be displayed.

```
public void setFilter(IlcSchedulerTraceFilter filter) const
```

This member function associates the specified filter to the calling object. Only events for which the function given as argument returns `IlcTrue` will be displayed.

Class IlcSchedulerTrace

Definition file: ilsched/schedtrace.h
Include file: <ilsched/ilsched.h>



The `IlcSchedulerTrace` class is used to determine which Scheduler objects (activities, resources, constraints) will be traced.

The implementation class, `IlcSchedulerTraceI`, has several virtual methods that can be overloaded to get different behaviors.

Example

```
MySchedTraceI* traceI = new (solver.getHeap())
       MySchedTraceI(schedule);
IlcSchedulerTrace trace(traceI);
trace.traceAllActivities();
trace.traceAllResources();
```

See Also: IlcSchedulerPrintTrace, IlcSchedulerTraceI

Method Summary	
public void	trace(IlcTimeBoundConstraint tbCt, IlcBool withActivity=IlcTrue) const
public void	trace(IlcPrecedenceConstraint precCt, IlcBool withActivities=IlcTrue) const
public void	trace(IlcAltResSet resSet) const
public void	trace(IlcResource res, IlcBool withResourceConstraints=IlcTrue) const
public void	trace(IlcAltResConstraint altRct, IlcBool withActivity=IlcTrue) const
public void	trace(IlcResourceConstraint rct, IlcBool withActivity=IlcTrue) const
public void	trace(IlcActivity act, IlcBool withResourceConstraints=IlcTrue) const
public void	traceAllActivities() const
public void	traceAllFailures() const
public void	traceAllResources() const

Methods

`public void` **`trace`**`(IlcTimeBoundConstraint tbCt, IlcBool withActivity=IlcTrue) const`

This member function states that the time bound constraint must be traced: if this constraint has a variable date, modifications of this variable will trigger the call of specific member functions of the implementation class of the invoking object.

If the argument `withActivity` equals `IlcTrue`, then the activity involved in the time bound constraint will also be traced.

`public void` **`trace`**`(IlcPrecedenceConstraint precCt, IlcBool withActivities=IlcTrue) const`

This member function states that the precedence constraint must be traced: if this precedence constraint has a variable delay, modifications of this variable will trigger the call of specific member functions of the implementation class of the invoking object.

If the argument `withActivities` equals `IlcTrue`, then the two activities involved in the precedence constraint will also be traced.

```
public void trace(IlcAltResSet resSet) const
```

This member function states that the given alternative resource set must be traced; all the alternative resource constraints that refer to this set will be traced.

```
public void trace(IlcResource res, IlcBool withResourceConstraints=IlcTrue) const
```

This member function states that the resource must be traced: this means that the structures used in the global constraints posted on this resource (timetables, precedence graphs) will be traced. Modifications of those data structures will trigger the call of specific member functions of the implementation class of the invoking object.

If the argument `withResourceConstraints` equals `IlcTrue`, then all the resource constraints in which this resource is involved, and their corresponding activities will also be traced.

```
public void trace(IlcAltResConstraint altRct, IlcBool withActivity=IlcTrue) const
```

This member function states that the given alternative resource constraint must be traced: any modification of any variable that this resource constraint owns (index of chosen resource, capacity, or required set) will call a specific member function of the implementation class of the invoking object.

If the argument `withActivity` equals `IlcTrue`, then the activity that corresponds to this alternative resource constraint will also be traced.

```
public void trace(IlcResourceConstraint rct, IlcBool withActivity=IlcTrue) const
```

This member function states that the given resource constraint must be traced: any modification of any variable that this resource constraint owns (capacity or required set) will call a specific member function of the implementation class of the invoking object.

If the argument `withActivity` equals `IlcTrue`, then the activity that corresponds to this resource constraint will also be traced.

```
public void trace(IlcActivity act, IlcBool withResourceConstraints=IlcTrue) const
```

This member function states that the given activity must be traced: any modification of any variable that this activity owns will call a specific member function of the implementation class of the invoking object.

If the argument `withResourceConstraints` equals `IlcTrue`, then all the resource constraints, alternative resource constraints, precedence constraints and time bound constraints in which this activity is involved will also be traced.

```
public void traceAllActivities() const
```

This member function states that all the activities must be traced. The constraints in which they are involved (time bound, precedence, resource, and alternative resource constraints) will also be traced.

```
public void traceAllFailures() const
```

This member function states that all the failures must be traced. The corresponding member functions of the implementation class of the invoking object will be called. The default behavior of a Scheduler trace object (that is, if this method is not called), is that only failures triggered by Solver variables of traced Scheduler objects will be traced.

```
public void traceAllResources() const
```

This member function states that all the resources must be traced. The constraints in which they are involved (resource constraints, alternative resource constraints), and their corresponding activities, will also be traced.

Note that temporal constraints will not be traced, and activities that do not require a resource will not be traced either. Use the member function `IlcSchedulerTrace::traceAllActivities` for such a case.

Class IlcSchedulerTraceI

Definition file: ilsched/schedtracei.h
Include file: <ilsched/ilsched.h>

The class `IlcSchedulerTraceI` is the base class used to build a trace for a scheduling problem. It provides virtual methods that are called when a change occurs on a traced scheduling object, and is intended to be sub-classed by the user so that its behavior fits the user's needs.

The default behavior of all the virtual methods is to do nothing, so that one can overload only the methods one is interested in.

Tracing modifications of objects

When a modification on a traced scheduling object (see `IlcSchedulerTrace`) occurs, the virtual method that corresponds to this change (named `xxxChange`, where `xxx` is the type of object that changes) will be called twice: first just before the modification, then just after. The timing of a particular call (whether just before or just after the modification) is indicated by the first argument, a boolean value that equals `IlcTrue` if that call occurred just before the modification.

The other arguments give the context of the modification: the instance of object that is being modified, the type of change, and the value that is used to change the object (for example, the new minimum value or the value removed from the possible set).

Tracing failures

When a failure occurs, it will be traced if either of the two following conditions hold:

- it is triggered by a Solver variable of a traced Scheduler object.
- the member function `IlcSchedulerTrace::traceAllFailures` was called.

If a failure is traced, then one or more virtual methods of the class `IlcSchedulerTraceI` will be called.

- If the failure occurs because a variable gets an empty domain, then either `failIntVar` or `failIntSetVar` will be called, depending on the type of variable.
- If the failure occurs because a goal, a demon or a constraint explicitly calls `fail()`, then `failDemon` will be called.
- In all fail cases, `failManager` will be called.

Example

```
class MyTraceI : public IlcSchedulerTraceI {
public:
  MyTraceI(IlcScheduleI* schedule, const char* name=0);
  virtual ~MyTraceI() {}

  ...

};

int main () {

  ...

  MyTraceI* myTraceI = new (solver.getGlobalHeap())
       MyTraceI(scheduler, "MyTrace"));
  IlcSchedulerTrace myTrace(myTraceI);

  ...

}
```

See Also: IlcSchedulerPrintTrace, IlcSchedulerTrace

Constructor and Destructor Summary	
public	IlcSchedulerTraceI(IlcScheduleI * schedule, const char * name=0)

Method Summary	
public virtual void	activityChange(IlcBool isBeginEvent, const IlcActivity act, IlcSchedulerChange change, IlcSolverChange solverChange, const IlcActivity act2, IlcInt val)
public virtual void	altResConstraintChange(IlcBool isBeginEvent, const IlcAltResConstraint altResCt, IlcSchedulerChange change, IlcSolverChange solverChange, IlcInt intVal)
public virtual void	anyTimetableChange(IlcBool isBeginEvent, const IlcAnyTimetable tt, IlcSchedulerChange change, IlcInt t1, IlcInt t2, IlcAnySet anySet, IlcAny anyVal)
public virtual void	failDemon(const IlcDemon demon)
public virtual void	failIntSetVar(const IlcIntSetVar setVar)
public virtual void	failIntVar(const IlcIntExp exp)
public virtual void	failManager(IlcInt nbFails)
public IlcActivity	getCurrentActivity1() const
public IlcActivity	getCurrentActivity2() const
public IlcAltResConstraint	getCurrentAltResConstraint() const
public IlcAltResSet	getCurrentAltResSet() const
public IlcPrecedenceConstraint	getCurrentPrecedenceConstraint() const
public IlcResource	getCurrentResource() const
public IlcResourceConstraint	getCurrentResourceConstraint1() const
public IlcResourceConstraint	getCurrentResourceConstraint2() const
public IlcTimeBoundConstraint	getCurrentTimeBoundConstraint() const
public IlcInt	getCurrentTimeMax() const
public IlcInt	getCurrentTimeMin() const
public IlcFailReason	getFailReason() const
public const char *	getMessage(IlcFailReason reason) const
public const char *	getMessage(IlcSolverChange chg) const
public const char *	getMessage(IlcSchedulerChange chg) const
public IlcScheduleI *	getScheduleI() const
public IloSolver	getSolver() const
public	ILCSTD(ostream) const
public virtual void	intTimetableChange(IlcBool isBeginEvent, const IlcIntTimetable tt, IlcSchedulerChange change, IlcInt t1, IlcInt t2, IlcInt intVal)
public virtual void	precedenceConstraintChange(IlcBool isBeginEvent, const IlcPrecedenceConstraint precCt, IlcSchedulerChange change, IlcSolverChange

	solverChange, IlcInt intVal)
public virtual void	resourceConstraintChange(IlcBool isBeginEvent, const IlcResourceConstraint resCt, IlcSchedulerChange change, IlcSolverChange solverChange, const IlcResourceConstraint resCt2, IlcInt intVal, IlcAny anyVal)
public virtual void	timeBoundConstraintChange(IlcBool isBeginEvent, const IlcTimeBoundConstraint tbCt, IlcSchedulerChange change, IlcSolverChange solverChange, IlcInt intVal)

Constructors and Destructors

public **IlcSchedulerTraceI**(IlcScheduleI * schedule, const char * name=0)

This constructor creates a new instance of IlcSchedulerTraceI and adds it to the given schedule. It can be used to trace all objects in schedule. Its name is set to name.

Methods

public virtual void **activityChange**(IlcBool isBeginEvent, const IlcActivity act, IlcSchedulerChange change, IlcSolverChange solverChange, const IlcActivity act2, IlcInt val)

This member function is called when a change occurs on the traced activity act. If activityChange is called just before the modification, then isBeginEvent equals IlcTrue; it equals IlcFalse if the modification has just occurred. The argument change indicates what is changed in the activity (for example, the start variable or end variable). If the change is related to an underlying Solver object (such as the start variable), then solverChange indicates how this object is changed (for example, setMin, setMax, setValue). Otherwise, this parameter equals IlcUndefinedSolverChange.

Some events (such as IlcActivitySetSuccessor) are related to two activities. In this case, act2 is a handle on this other activity. Otherwise, it is an empty handle.

When an integer Solver variable is changed, the value used (such as the new minimum) is given in val. When an activity is postponed (either forward or backward), the date from which it is postponed is given in val. Otherwise, the value of this argument is IlcIntMin.

public virtual void **altResConstraintChange**(IlcBool isBeginEvent, const IlcAltResConstraint altResCt, IlcSchedulerChange change, IlcSolverChange solverChange, IlcInt intVal)

This member function is called when a change occurs on the traced alternative resource constraint altResCt. If altResConstraintChange is called just before the modification, then isBeginEvent equals IlcTrue; it equals IlcFalse if the modification has just occurred. The argument change indicates what is changed in the alternative resource constraint (for example, the index variable). If the change is related to an underlying Solver object (such as the index variable), then solverChange indicates how this object is changed (for example, setMin, setMax, setValue). Otherwise, this parameter equals IlcUndefinedSolverChange.

Some events (such as IlcResourceConstraintSetSuccessor) are related to two alternative resource constraints. In this case, resCt2 is a handle on this other alternative resource constraint. Otherwise, it is an empty handle.

When an integer Solver variable is changed, the value used (such as the new minimum) is given in intVal. Otherwise, the value of this argument is IlcIntMin.

```
public virtual void anyTimetableChange(IlcBool isBeginEvent, const IlcAnyTimetable
tt, IlcSchedulerChange change, IlcInt t1, IlcInt t2, IlcAnySet anySet, IlcAny
anyVal)
```

This member function is called when a change occurs on the traced timetable `tt`. If `resourceConstraintChange` is called just before the modification, then `isBeginEvent` equals `IlcTrue`; it equals `IlcFalse` if the modification has just occurred. The argument `change` indicates what is changed in the timetable.

The parameters `t1` and `t2` define the interval on which the timetable is being modified.

The value used to change the timetable (for example, the set of states removed from the possible states set) is given in `anySet`. If the value consists of only one `IlcAny`, then `anySet` is an empty handle, and this value is given in `anyVal`. The value used to change the timetable (such as the value removed from the possible states set) is given in `anyVal`, if it is only a single state. Otherwise, this parameter equals 0 and the values are given in `anySet`.

```
public virtual void failDemon(const IlcDemon demon)
```

This method is called when `demon` triggers a failure.

```
public virtual void failIntSetVar(const IlcIntSetVar setVar)
```

This method is called when a modification of `setVar` triggers a failure.

```
public virtual void failIntVar(const IlcIntExp exp)
```

This method is called when a modification of `exp` triggers a failure.

```
public virtual void failManager(IlcInt nbFails)
```

This method is called when a failure occurs. The new number of fails is `nbFails`.

```
public IlcActivity getCurrentActivity1() const
```

When a fail occurs and the activity related to the failure is known to Scheduler Engine, then this method returns this activity. Otherwise it returns an empty handle. This method must only be called when inside a call to `failDemon`, `failIntVar`, `failIntSetVar` or `failManager`.

```
public IlcActivity getCurrentActivity2() const
```

When a fail occurs and a second activity related to the failure is known to the Scheduler Engine (e.g. one activity is said to be next to another, and this leads to a fail), then this method returns this second activity. Otherwise it returns an empty handle. This method must only be called when inside a call to `failDemon`, `failIntVar`, `failIntSetVar` or `failManager`.

```
public IlcAltResConstraint getCurrentAltResConstraint() const
```

When a fail occurs and an instance of `IlcAltResConstraint` is known to the Scheduler Engine, then this method returns this alternative resource constraint. Otherwise it returns an empty handle. This method must only be called when inside a call to `failDemon`, `failIntVar`, `failIntSetVar` or `failManager`.

```
public IlcAltResSet getCurrentAltResSet() const
```

When a fail occurs and an instance of `IlcAltResSet` related to the failure is known to the Scheduler Engine, then this method returns this alternative resource set. Otherwise it returns an empty handle. This method must only be called when inside a call to `failDemon`, `failIntVar`, `failIntSetVar` or `failManager`.

```
public IlcPrecedenceConstraint getCurrentPrecedenceConstraint() const
```

When a fail occurs and a precedence constraint related to the failure is known to the Scheduler Engine, then this method returns this precedence constraint. Otherwise it returns an empty handle. This method must only be called when inside a call to `failDemon`, `failIntVar`, `failIntSetVar` or `failManager`.

```
public IlcResource getCurrentResource() const
```

When a fail occurs and a resource related to the failure is known to the Scheduler Engine, then this method returns this resource. Otherwise it returns an empty handle. This method must only be called when inside a call to `failDemon`, `failIntVar`, `failIntSetVar` or `failManager`.

```
public IlcResourceConstraint getCurrentResourceConstraint1() const
```

When a fail occurs and a resource constraint related to the failure is known to the Scheduler Engine, then this method returns this resource constraint. Otherwise it returns an empty handle. This method must only be called when inside a call to `failDemon`, `failIntVar`, `failIntSetVar` or `failManager`.

```
public IlcResourceConstraint getCurrentResourceConstraint2() const
```

When a fail occurs and a second resource constraint related to the failure is known to the Scheduler Engine, then this method returns this second resource constraint. Otherwise it returns an empty handle. This method must only be called when inside a call to `failDemon`, `failIntVar`, `failIntSetVar` or `failManager`.

```
public IlcTimeBoundConstraint getCurrentTimeBoundConstraint() const
```

When a fail occurs and a time bound constraint related to the failure is known to the Scheduler Engine, then this method returns this time bound constraint. Otherwise it returns an empty handle. This method must only be called when inside a call to `failDemon`, `failIntVar`, `failIntSetVar` or `failManager`.

```
public IlcInt getCurrentTimeMax() const
```

When a fail occurs and the time interval related to the failure is known to the Scheduler Engine, then this method returns the upper bound of this interval. Otherwise it returns `IlcIntMax`. This method must only be called when inside a call to `failDemon`, `failIntVar`, `failIntSetVar` or `failManager`.

```
public IlcInt getCurrentTimeMin() const
```

When a fail occurs and the time interval related to the failure is known to the Scheduler Engine, then this method returns the lower bound of this interval. Otherwise it returns `IlcIntMin`. This method must only be called when inside a call to `failDemon`, `failIntVar`, `failIntSetVar` or `failManager`.

```
public IlcFailReason getFailReason() const
```

When a fail occurs and the reason for the fail is known to the Scheduler Engine, then this method returns the reason. If the reason of the fail is unknown to the Scheduler Engine, this method will return `IlcFailReasonUnknown`. This method must only be called when inside a call to `failDemon`, `failIntVar`, `failIntSetVar` or `failManager`.

```
public const char * getMessage(IlcSchedulerChange chg) const
public const char * getMessage(IlcFailReason reason) const
public const char * getMessage(IlcSolverChange chg) const
```

This member function returns a string containing the name of the change given as argument.

```
public IlcScheduleI * getScheduleI() const
```

This member function returns the instance of `IlcScheduleI` on which the trace was built.

```
public IloSolver getSolver() const
```

This member function returns the instance of `IloSolver` that is associated with the instance of `IlcSchedule` on which the trace was built.

```
public ILCSTD(ostream) const
```

This member function returns the stream that can be used to output trace information. This is the stream that is attached to the solver.

```
public virtual void intTimetableChange(IlcBool isBeginEvent, const IlcIntTimetable
tt, IlcSchedulerChange change, IlcInt t1, IlcInt t2, IlcInt intVal)
```

This member function is called when a change occurs on the traced integer timetable `tt`. If `timeBoundConstraintChange` is called just before the modification, then `isBeginEvent` equals `IlcTrue`; it equals `IlcFalse` if the modification has just occurred. The argument `change` indicates what is changed in the timetable.

The value used to change the timetable (such as the new minimum) is given by `intVal`. The parameters `t1` and `t2` define the interval on which the timetable is being modified.

```
public virtual void precedenceConstraintChange(IlcBool isBeginEvent, const
IlcPrecedenceConstraint precCt, IlcSchedulerChange change, IlcSolverChange
solverChange, IlcInt intVal)
```

This member function is called when a change occurs on the traced precedence constraint `precCt`. If `precedenceConstraintChange` is called just before the modification, then `isBeginEvent` equals `IlcTrue`; it equals `IlcFalse` if the modification has just occurred. The argument `change` indicates what is changed in the precedence constraint (for example, the `delay` variable). If the change is related to an underlying Solver object (such as the `delay` variable), then `solverChange` indicates how this object is changed (for example, `setMin`, `setMax`, `setValue`). Otherwise, this parameter equals `IlcUndefinedSolverChange`.

When an integer Solver variable is changed, the value used (such as the new minimum) is given in `intVal`. Otherwise, the value of this argument is `IlcIntMin`.

```
public virtual void resourceConstraintChange(IlcBool isBeginEvent, const
IlcResourceConstraint resCt, IlcSchedulerChange change, IlcSolverChange
solverChange, const IlcResourceConstraint resCt2, IlcInt intVal, IlcAny anyVal)
```

This member function is called when a change occurs on the traced resource constraint `resCt`. If `resourceConstraintChange` is called just before the modification, then `isBeginEvent` equals `IlcTrue`; it equals `IlcFalse` if the modification has just occurred. The argument `change` indicates what is changed in the resource constraint (for example, the `capacity` variable). If the change is related to an underlying Solver object (such as the `capacity` variable), then `solverChange` indicates how this object is changed (for example, `setMin`, `setMax`, `setValue`). Otherwise, this parameter equals `IlcUndefinedSolverChange`.

Some events (such as `IlcResourceConstraintSetSuccessor`) are related to two resource constraints. In this case, `resCt2` is a handle on this other resource constraint. Otherwise, it is an empty handle.

When an integer Solver variable is changed, the value used (for example, the new minimum) is given in `intVal`. Otherwise, this parameter equals `IlcIntMin`. When an `IlcAnyVar` is changed (such as the `state` required), the value used (such as the `state` removed) is given in `anyVal`. Otherwise this parameter equals 0.

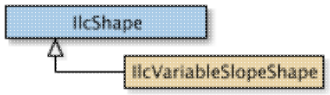
```
public virtual void timeBoundConstraintChange(IlcBool isBeginEvent, const
IlcTimeBoundConstraint tbCt, IlcSchedulerChange change, IlcSolverChange
solverChange, IlcInt intVal)
```

This member function is called when a change occurs on the traced time bound constraint `tbCt`. If `timeBoundConstraintChange` is called just before the modification, then `isBeginEvent` equals `IlcTrue`; it equals `IlcFalse` if the modification has just occurred. The argument `change` indicates what is changed in the time bound constraint (for example, the `date` variable). If the change is related to an underlying Solver object (such as the `date` variable), then `solverChange` indicates how this object is changed (for example, `setMin`, `setMax`, `setValue`). Otherwise, this parameter equals `IlcUndefinedSolverChange`.

When an integer Solver variable is changed, the value used (for example, the new minimum) is given in `intVal`. Otherwise, this parameter equals `IlcIntMin`.

Class IlcShape

Definition file: ilsched/shaperct.h
Include file: <ilsched/ilsched.h>



Instances of `IlcShape` are generic objects describing shapes associated with resource constraints on continuous reservoirs.

See Also: IlcResourceConstraint, IlcVariableSlopeShape, IloShape, IloVariableSlopeShape

Constructor Summary	
public	IlcShape()
public	IlcShape(IlcShapeI * impl)

Method Summary	
public IlcShapeI *	getImpl() const
public const char *	getName() const
public IlcAny	getObject() const
public IlcResourceConstraint	getResourceConstraint() const
public IloSolver	getSolver() const
public IloSolverI *	getSolverI() const
public IlcBool	isVariableSlopeShape() const
public void	operator=(const IlcShape & h)
public void	setName(const char * name) const
public void	setObject(IlcAny object) const

Constructors

public **IlcShape**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcShape**(IlcShapeI * impl)

This constructor creates a handle object from a pointer to an implementation object.

Methods

public IlcShapeI * **getImpl**() const

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

public const char * **getName**() const

This member function returns the name of the invoking object.

```
public IlcAny getObject() const
```

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

```
public IlcResourceConstraint getResourceConstraint() const
```

This member function returns the instance of `IlcResourceConstraint` with which the shape is associated.

```
public IloSolver getSolver() const
```

This member function returns an instance of `IloSolver` associated with the invoking object.

```
public IloSolverI * getSolverI() const
```

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

```
public IlcBool isVariableSlopeShape() const
```

This member function returns `IlcTrue` if the invoking handle relates to an instance of `IlcVariableSlopeShape`. In this case, the handle can be safely down-cast into a `IlcVariableSlopeShape` handle, using the corresponding copy-constructor.

See Also: IlcVariableSlopeShape

```
public void operator=(const IlcShape & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public void setName(const char * name) const
```

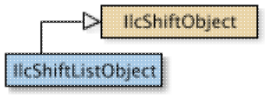
This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

```
public void setObject(IlcAny object) const
```

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

Class IlcShiftListObject

Definition file: ilsched/shifts.h



The class `IlcShiftListObject` inherits from the class `IlcShiftObject`). It allows expression of shifts as a list of forbidden time intervals. Depending on the type of the shift list object, the time restriction can concern the whole activity execution, or only its start or its end. Indeed three different types are defined:

- `OnStart`: Shifts only concern the start of the activity. For instance, if the shift is the interval *[a,b)*, then the start of the activity must be strictly smaller than *a* or greater than *b*.
- `OnEnd`: Shifts only concern the end of the activity. For instance, if the shift is the interval *[a,b)*, then the end of the activity must be strictly smaller than *a* or greater than *b*.
- `OnOverlap`: Shifts concern the whole activity. That is, the activity cannot overlap shifts. For example, if the shift is the interval *[a,b)*, then the end of the activity must be strictly smaller than *a* or the start must be greater than *b*.

In addition, each time interval (see `IlcIntervalList`) can be associated with an integer type. In such cases, activities can specify which types have to be ignored.

For more information, see Calendars and Shift Object Semantic.

Constructor Summary	
public	`IlcShiftListObject()`
public	`IlcShiftListObject(IlcShiftListObjectI * impl)`
public	`IlcShiftListObject(IlcSchedule sched)`
public	`IlcShiftListObject(IlcSchedule sched, IlcIntervalList shiftList, IlcShiftListObject::Type type)`

Method Summary	
public IlcShiftListObjectI *	`getImpl() const`
public IlcIntervalList	`getShiftList() const`
public IlcShiftListObject::Type	`getType() const`
public void	`operator=(const IlcShiftListObject & h)`
public void	`setShiftList(IlcIntervalList shiftList, IlcShiftListObject::Type type)`

Inherited Methods from `IlcShiftObject`
`getImpl, operator=`

Inner Enumeration
IlcShiftListObject::Type

Constructors

`public` **`IlcShiftListObject`**`()`

This constructor creates an instance which is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

```
public IlcShiftListObject(IlcShiftListObjectI * impl)
```

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

```
public IlcShiftListObject(IlcSchedule sched)
```

This constructor creates a new instance of `IlcShiftListObject`. By default, the shift list is empty and the type is `OnStart`.

```
public IlcShiftListObject(IlcSchedule sched, IlcIntervalList shiftList,
IlcShiftListObject::Type type)
```

This constructor creates a new instance of `IlcShiftListObject`. The shift list is set to `shiftList` and the type is set to `type`.

Methods

```
public IlcShiftListObjectI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcIntervalList getShiftList() const
```

This member function returns the time interval list that represents the set of forbidden dates of the invoking shift object.

```
public IlcShiftListObject::Type getType() const
```

This member function returns the type that defines the behavior of the shifts during the search (see `IlcShiftListObject::Type`).

```
public void operator=(const IlcShiftListObject & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument. After execution of this operator, the invoking object and the provided argument point to the same implementation object.

```
public void setShiftList(IlcIntervalList shiftList, IlcShiftListObject::Type type)
```

This member function sets `shiftList` as the new shift list of the invoking shift object with the type `type`.

Inner Enumerations

Enumeration Type

Definition file: ilsched/shifts.h

The `Type` of `IlcShiftListObject` allows definition of the behavior during search regarding the variables of concerned activities. The possible types are:

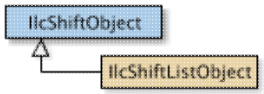
- `OnStart`: Shifts only concern the start of the activity. For instance, if the shift is the interval *[a,b)*, then the start of the activity must be strictly smaller than *a* or greater than *b*.
- `OnEnd`: Shifts only concern the end of the activity. For instance, if the shift is the interval *[a,b)*, then the end of the activity must be strictly smaller than *a* or greater than *b*.
- `OnOverlap`: Shifts concern the whole activity. That is, the activity cannot overlap shifts. For instance, if the shift is the interval *[a,b)*, then the end of the activity must be strictly smaller than *a* or the start must be greater than *b*.

Fields:

```
OnStart = 0

OnEnd = 1

OnOverlap = 2
```

Class IlcShiftObject

Definition file: ilsched/shifts.h



The class `IlcShiftObject` allows definition of shifts which constrain concerned activities. Shifts are forbidden time intervals, not necessarily defined in extension `ILCUSERSHIFTOBJECT`), which restrict possible starts, ends or whole executions of activities.

To express shifts it is possible to enumerate all intervals with `IlcShiftListObject`, or to write an intention definition using `ILCUSERSHIFTOBJECT`).

For more information, see Calendars and Shift Object Semantic.

Constructor Summary	
public	IlcShiftObject()
public	IlcShiftObject(IlcShiftObjectI * impl)
public	IlcShiftObject(void * impl)

Method Summary	
public IlcShiftObjectI *	getImpl() const
public void	operator=(const IlcShiftObject & h)

Constructors

`public` **`IlcShiftObject`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcShiftObject`**`(IlcShiftObjectI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IlcShiftObject`**`(void * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public IlcShiftObjectI *` **`getImpl`**`() const`

This constructor creates an object by copying another one.

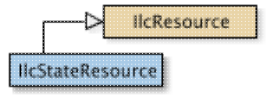
This member function returns a pointer to the implementation object of the invoking handle.

```
public void operator=(const IlcShiftObject & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcStateResource

Definition file: ilsched/state.h
Include file: <ilsched/ilsched.h>



An instance of the class `IlcStateResource` represents a resource of infinite capacity, the *state* of which can vary over time. Each activity may, throughout its execution, require a state resource to be in a given state (or in any of a given set of states). Consequently, two activities may not overlap if they require incompatible states during their execution. Two methods—similar to those used for discrete and unary resources—are available to take into account the constraints concerning the requirement or the provision of a state resource.

- The first method allows the set of possible states to vary over time: at any given time, the resource may or may not be allowed to be in a given state.
- The second method deals only with requiring activities: it consists of updating the earliest and latest start and end times of activities to insure that the time intervals over which two activities that require incompatible states cannot overlap.

For state resources, you can define *transition times* with the second method.

State

In the context of an instance of `IlcStateResource`, a *state* is defined as a pointer to any type of object (that is, a pointer of type `IlcAny`). The class `IlcAnyTimetable` is used to represent the *evolution* of the state of the resource over time.

Printing or Displaying State Resources

The printed representation of an instance of the class `IlcStateResource` consists of its name, followed by a list of states. If there are more than 10 states, only the number of states is displayed.

For example:

`[0x1, 0x2, 0x3]` represents a state resource whose possible states are the ones represented by the pointers `0x1`, `0x2`, and `0x3`.

`[12 possible states]` represents a state resource whose number of possible states is equal to 12.

If the Solver trace is active and the resource is not named, the string `"IlcStateResource"` is followed by the address of the implementation object. The address will be enclosed in parentheses.

Special Considerations for State Resources: Limitations of Disjunctive Constraints

Throughout this explanation, we have assumed that each activity which requires a state resource requires it in the same state (either a constant or a Solver variable) from the beginning to the end of its execution.

In addition, an activity may require a state resource in any of a given *set* of states, and yet allow the state to *change* during the execution of the activity, provided that it remains in the given set. Similarly, an activity may simply require the state resource *not* to be in a given state (or in any of a given set of states) throughout its execution.

A disjunctive constraint cannot manage these cases as well as a timetable.

For example, assume three activities A, B, and C intersect in time. Assume the resource constraints are as follows:

- A requires the resource in any of the states s1 and s2 (the state is allowed to change during the

execution of the activity but must always be either s1 or s2);
- B requires the resource in any of the states s1 and s3 (the state is allowed to change during the execution of the activity but must always be either s1 or s3);
- C requires the resource in any of the states s2 and s3 (the state is allowed to change during the execution of the activity but must always be either s2 or s3).

In such a case, the disjunctive constraint fails to detect the incompatibility of the three constraints because any two of these constraints are fully compatible.

In contrast, the timetable mechanism perfectly detects the incompatibility because the variable representing the state of the resource at the time at which the three activities intersect can assume no value. A disjunctive constraint may still be useful in such a case, as a *redundant constraint* for activities requiring only a state, or to take *transition times* into account.

More precisely, there are two *types* of resource constraints that can be posted to insist that an activity requires a state resource:

- `IlcActivity::requires` means that the activity requires the resource in a state or given set of states that cannot change during the execution of the activity (the state or states may be a Solver variable, but this variable has to be instantiated in a solution to the problem under consideration).
- `IlcActivity::requiresNot` means that the activity requires the resource not to be in a given state or given set of states (which can be a Solver variable); in other words, the state of the resource can change during the execution of the activity, but must never be the given state.

As explained before, a disjunctive constraint deals perfectly with constraints of the type `requires`. A disjunctive constraint deals *imperfectly* with constraints of the type `requiresNot`. Instead, a timetable is needed to deal with such constraints.

For more information, see Ranking , Disjunctive Constraint, Timetable, and Transition Time in Scheduler Engine.

See Also: IlcAnyTimetable, IlcResource, IlcResourceConstraint, IlcStateResourceIterator

Constructor Summary	
public	`IlcStateResource()`
public	`IlcStateResource(IlcStateResourceI * impl)`
public	`IlcStateResource(IlcSchedule schedule, IlcAnySet states, IlcBool timeTable=IlcTrue)`
public	`IlcStateResource(IlcSchedule schedule, IlcAnySet states, IlcTransitionTimeObject ttobj, IlcBool disjAndTimetable=IlcTrue)`

Method Summary	
public IlcStateResourceI *	`getImpl() const`
public IlcAny	`getState(IlcInt time) const`
public IlcAnyTimetable	`getTimetable() const`
public IlcAnyTimetable	`getTimetable(IlcInt time) const`
public IlcConstraint	`getTypeTimetableConstraint() const`
public IlcBool	`hasTypeTimetableConstraint() const`
public IlcBool	`isAlwaysInUse(IlcInt timeMin, IlcInt timeMax) const`
public IlcBool	`isAlwaysPossible(IlcAny state, IlcInt timeMin, IlcInt timeMax) const`
public IlcBool	`isAlwaysRequired(IlcAny state, IlcInt timeMin, IlcInt timeMax) const`
public IlcBool	`isBound(IlcInt time) const`

public IlcBool	isEverInUse(IlcInt timeMin, IlcInt timeMax) const
public IlcBool	isEverPossible(IlcAny state, IlcInt timeMin, IlcInt timeMax) const
public IlcBool	isEverRequired(IlcAny state, IlcInt timeMin, IlcInt timeMax) const
public IlcBool	isInUse(IlcInt time) const
public IlcBool	isPossible(IlcAny state, IlcInt time) const
public IlcBool	isRequired(IlcAny state, IlcInt time) const
public IlcConstraint	makeDisjunctiveConstraint()
public IlcConstraint	makeTimetableConstraint(IlcInt timeMin, IlcInt timeMax, IlcInt timeStep, IlcAnySet states)
public IlcConstraint	makeTimetableConstraint(IlcInt timeStep=1)
public IlcConstraint	makeTypeTimetableConstraint(IlcBool useBatch=IlcFalse)
public void	operator=(const IlcStateResource & h)
public void	removePossibleStates(IlcInt timeMin, IlcInt timeMax, IlcAnySet states)
public void	removeState(IlcInt timeMin, IlcInt timeMax, IlcAny state)
public void	setMustBeInUse(IlcInt timeMin, IlcInt timeMax)
public void	setPossibleStates(IlcInt timeMin, IlcInt timeMax, IlcAnySet states)
public void	setState(IlcInt timeMin, IlcInt timeMax, IlcAny state)

Inherited Methods from `IlcResource`
close, getCalendar, getDisjunctiveConstraint, getDurableSchedule, getImpl, getLastRankedFirstRC, getLastRankedLastRC, getLastSurelyContributingRankedFirstRC, getLastSurelyContributingRankedLastRC, getName, getObject, getOldLastRankedFirstRC, getOldLastRankedLastRC, getPrecedenceGraphConstraint, getSchedule, getSolver, getSolverI, getTimetableConstraint, getTransitionTime, hasCalendar, hasDisjunctiveConstraint, hasLightPrecedenceGraphConstraint, hasPrecedenceGraphConstraint, hasPrecedenceInfo, hasRankInfo, hasTimetableConstraint, isCapacityResource, isClosed, isContinuousReservoir, isDiscreteEnergy, isDiscreteResource, isDurable, isReservoir, isStateResource, isTransitionTimeSuspended, isUnaryResource, makeFunctionalConstraint, makeIntegralConstraint, makeLightPrecedenceGraphConstraint, makePrecedenceGraphConstraint, operator!=, operator=, operator==, setCalendar, setName, setObject, setTransitionTimeObject, setTransitionTimeSuspended, whenContribution, whenDirectPredecessors, whenDirectSuccessors, whenNext, whenPossibleNext, whenPossiblePrevious, whenPredecessors, whenPrevious, whenRankedFirstRC, whenRankedLastRC, whenSuccessors

Constructors

public **IlcStateResource**()

This constructor creates an instance which is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

public **IlcStateResource**(IlcStateResourceI * impl)

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

```
public IlcStateResource(IlcSchedule schedule, IlcAnySet states, IlcBool
timeTable=IlcTrue)
```

This constructor creates a new instance of `IlcStateResource` and adds it to the set of resources managed in the given `schedule`. The argument `states` is the set of pointers that can be accepted as possible states for the resource. The argument `timeTable` indicates whether the standard timetable constraint manages the profile of possible states of the resource from the time origin to the time horizon of the given `schedule`; if so, it allows this set of possible states to change at any point in time; that is, it defines a time step of 1 (one).

```
public IlcStateResource(IlcSchedule schedule, IlcAnySet states,
IlcTransitionTimeObject ttobj, IlcBool disjAndTimetable=IlcTrue)
```

This constructor creates a new instance of `IlcStateResource` and adds it to the set of resources managed in the given `schedule`. The argument `states` is the set of pointers that can be accepted as possible states for the resource. The argument `ttobj` indicates which transition time function will be used for the invoking resource. The argument `disjAndTimetable` indicates whether both the standard timetable constraint *and* the disjunctive constraint will be added to the solver.

Transition times are taken into account when the timetable or the disjunctive constraint are posted. Transition times are only propagated between two activities that are incompatible. As the disjunctive constraint defines incompatibility based on the resource demand of the activities and the type timetable constraint defines incompatibility based on the transition types of the activities, they do not propagate in the same manner. Please see Transition Time in Scheduler Engine and Type Timetable Constraint for more information.

If the argument `ttobj` has not been built with an instance of `IlcTransitionTable`, the type timetable constraint will be unable to take the transition times into account. Please see Type Timetable Constraint for more information.

The standard timetable constraint manages the profile of possible states of the resource from the time origin to the time horizon of the given schedule; it allows this set of possible states to change at any point in time; that is, it defines a time step of 1 (one).

Methods

```
public IlcStateResourceI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcAny getState(IlcInt time) const
```

This member function returns the state of the invoking resource at `time`.

```
public IlcAnyTimetable getTimetable() const
```

This member function returns the first (in chronological order) timetable of the invoking resource. An instance of `IloSolver::SolverErrorException` is thrown if no timetable exists for the invoking resource.

```
public IlcAnyTimetable getTimetable(IlcInt time) const
```

This member function returns the timetable which includes `time`. An instance of `IloSolver::SolverErrorException` is thrown if no timetable is defined at `time`.

```
public IlcConstraint getTypeTimetableConstraint() const
```

This member function returns the type timetable constraint of the invoking resource.

```
public IlcBool hasTypeTimetableConstraint() const
```

This member function returns `IlcTrue` if the invoking resource has a type timetable constraint. Otherwise, it returns `IlcFalse`.

```
public IlcBool isAlwaysInUse(IlcInt timeMin, IlcInt timeMax) const
```

This member function returns `IlcTrue` if the resource is constantly in use over the interval `[timeMin, timeMax)`; that is, the resource is never idle in that interval. Otherwise, it returns `IlcFalse`.

An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval `[timeMin, timeMax)`.

```
public IlcBool isAlwaysPossible(IlcAny state, IlcInt timeMin, IlcInt timeMax) const
```

This member function returns `IlcTrue` if and only if it is possible that the invoking resource is in the given `state`*over the entire* interval `[timeMin, timeMax)`. Otherwise, it returns `IlcFalse`.

An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval `[timeMin, timeMax)`.

```
public IlcBool isAlwaysRequired(IlcAny state, IlcInt timeMin, IlcInt timeMax) const
```

This member function returns `IlcTrue` if and only if it is certain that the invoking resource is in the given `state`*over the entire* interval `[timeMin, timeMax)`. Otherwise, it returns `IlcFalse`.

An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval `[timeMin, timeMax)`.

```
public IlcBool isBound(IlcInt time) const
```

This member function returns `IlcTrue` if and only if the state of the invoking resource at the given `time` is known. Otherwise, it returns `IlcFalse`.

An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover `time`.

```
public IlcBool isEverInUse(IlcInt timeMin, IlcInt timeMax) const
```

This member function returns `IlcTrue` if the invoking resource is ever in use over the interval `[timeMin, timeMax)`; that is, there is ever a time in the interval when the invoking resource is not idle. Otherwise, the member function returns `IlcFalse`.

An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval `[timeMin, timeMax)`.

```
public IlcBool isEverPossible(IlcAny state, IlcInt timeMin, IlcInt timeMax) const
```

This member function returns `IlcTrue` if and only if it is possible that the invoking resource is in the given `state`*at some point* in the interval `[timeMin, timeMax)`. Otherwise, it returns `IlcFalse`.

An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval `[timeMin, timeMax)`.

```
public IlcBool isEverRequired(IlcAny state, IlcInt timeMin, IlcInt timeMax) const
```

This member function returns `IlcTrue` if and only if it is certain that the invoking resource is in the given `state`*at some point* in the interval `[timeMin, timeMax)`. Otherwise, it returns `IlcFalse`.

An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval `[timeMin, timeMax)`.

```
public IlcBool isInUse(IlcInt time) const
```

This member function returns `IlcTrue` if the invoking resource is in use at the time indicated by `time`; that is, the resource is not idle at `time`. Otherwise, it returns `IlcFalse`.

An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover `time`.

```
public IlcBool isPossible(IlcAny state, IlcInt time) const
```

This member function returns `IlcTrue` if and only if it is possible that the invoking resource is in the given `state` at the given `time`. Otherwise, it returns `IlcFalse`.

An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover `time`.

```
public IlcBool isRequired(IlcAny state, IlcInt time) const
```

This member function returns `IlcTrue` if and only if it is certain that the invoking resource is in the given `state` at the given `time`. Otherwise, it returns `IlcFalse`.

An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover `time`.

```
public IlcConstraint makeDisjunctiveConstraint()
```

This member function creates and returns the global disjunctive constraint associated with the invoking resource. For more information see Disjunctive Constraint.

```
public IlcConstraint makeTimetableConstraint(IlcInt timeMin, IlcInt timeMax, IlcInt
```

```
timeStep, IlcAnySet states)
```

This member function creates and returns a new timetable constraint for the invoking resource. The argument `states` indicates the set of possible states. From `timeMin` to `timeMax`, the state of the resource is allowed to change *only* at times `timeMin + i * timeStep`. The state must remain constant over each interval `[t (t + timeStep))` with `t = timeMin + i * timeStep` and `t` at least as great as `timeMin` and strictly less than `timeMax`.

An instance of `IloSolver::SolverErrorException` is thrown if any of the following conditions occur:

- if `timeMin` is not strictly less than `timeMax`;
- if `timeStep` is not strictly positive;
- if `timeMax - timeMin` is not a multiple of `timeStep`;
- if `states` contains an element that was not in the set of possible states that was passed to the resource constructor;
- if the new timetable overlaps a timetable that has already been created for the invoking resource.

```
public IlcConstraint makeTimetableConstraint(IlcInt timeStep=1)
```

This member function creates and returns a new timetable constraint for the invoking resource. The set of possible states is given by the set passed to the `IlcStateResource` constructor. From the time origin `timeMin` to the time horizon, the state of the resource is allowed to change only at times `timeMin + i * timeStep`. The state must remain constant over each interval `[t (t + timeStep))` with `t = timeMin + i * timeStep` and `t` at least as great as the time origin and less than the time horizon.

An instance of `IloSolver::SolverErrorException` is thrown if any of the following conditions occur:

- if `timeStep` is not strictly positive;
- if the time horizon minus the time origin is not a multiple of `timeStep`;
- if the new timetable overlaps a timetable that has already been created for the invoking resource.

```
public IlcConstraint makeTypeTimetableConstraint(IlcBool useBatch=IlcFalse)
```

This member function attaches a type timetable constraint to the resource and returns it.

The type timetable constraint uses the transition time object that was passed to the constructor of the invoking resource to propagate the transition times. This transition time object needs to have been built with an instance of `IlcTransitionTable`. An instance of `IloSolver::SolverErrorException` is thrown if no transition time object was passed to the constructor of the resource or if the transition time object was not built with an instance of `IlcTransitionTable`.

```
public void operator=(const IlcStateResource & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument. After execution of this operator, the invoking object and the provided argument point to the same implementation object.

```
public void removePossibleStates(IlcInt timeMin, IlcInt timeMax, IlcAnySet states)
```

This member function states that the invoking resource must not be in any of the given `states` at any time in the interval `[timeMin, timeMax)`. The set of “impossible” states can be provided as an instance of `IlcAnySet`. That class is documented in the *IBM ILOG Solver Reference Manual*.

An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval `[timeMin, timeMax)`.

```
public void removeState(IlcInt timeMin, IlcInt timeMax, IlcAny state)
```

This member function states that the invoking resource must not be in the given `state` at any time in the interval `[timeMin, timeMax)`.

An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval `[timeMin, timeMax)`.

```
public void setMustBeInUse(IlcInt timeMin, IlcInt timeMax)
```

This member function states that the invoking resource must be in use and that it cannot be idle over the interval `[timeMin, timeMax)`.

The resource must be *closed* to propagate.

An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval `[timeMin, timeMax)`.

```
public void setPossibleStates(IlcInt timeMin, IlcInt timeMax, IlcAnySet states)
```

This member function states that the invoking resource must be in some of the given `states` at all times in the interval `[timeMin, timeMax)`. The set of possible states can be provided as an instance of `IlcAnySet`. That class is documented in the *Solver Reference Manual*.

An instance of `IloSolver::SolverErrorException` is thrown if the timetables of the invoking resource do not cover the complete interval `[timeMin, timeMax)`.

```
public void setState(IlcInt timeMin, IlcInt timeMax, IlcAny state)
```

This member function states that the invoking resource must be in the given `state` at all times in the interval `[timeMin, timeMax)`.

An instance of `IloSolver::SolverErrorException` is thrown if the timetable of the invoking resource does not cover the complete interval `[timeMin, timeMax)`.

Class IlcStateResourceIterator

Definition file: ilsched/schedulerdoc.h
Include files: <ilsched/ilsched.h> and <ilsched/ilsched.h>

An instance of this class traverses the set of state resources.

An instance of this class traverses the set of timetables associated with a state resource.

See Also: IlcStateResource, IlcSchedule, IlcAnyTimetable, IlcSchedule

Constructor and Destructor Summary	
public	IlcStateResourceIterator(const IlcSchedule schedule)

Method Summary	
public IlcBool	ok() const
public IlcStateResource	operator*() const
public IlcStateResourceIterator &	operator++()

Constructors and Destructors

public **IlcStateResourceIterator**(const IlcSchedule schedule)

This constructor creates an iterator to traverse all the state resources of `schedule`.

Methods

public IlcBool **ok**() const

This member function returns `IlcTrue` if the current position of the iterator is a valid one. It returns `IlcFalse` if all the state resources have been scanned by the iterator.

public IlcStateResource **operator***() const

This operator accesses the instance of `IlcStateResource` located at the current position of the iterator. If the iterator is set past the end position, this operator returns an empty handle.

public IlcStateResourceIterator & **operator++**()

This left-increment operator shifts the current position of the iterator.

Class IlcTextureCriticalityCalculator

Definition file: ilsched/texture.h
Include file: <ilsched/ilsched.h>

`IlcTextureCriticalityCalculator` is the handle class for `IlcTextureCriticalityCalculatorI` and for its subclasses.

For more information, see Texture Measurements.

Predefined Texture Criticality Calculators

These functions return instances of `IlcTextureCriticalityCalculator`.

- `IlcTextureCriticalityCalculator IlcProbabilisticCriticalityCalculator (IloSolver solver);`
- `IlcTextureCriticalityCalculator IlcRelativeDemandCriticalityCalculator (IloSolver solver);`

See Also: IlcResourceTexture, IlcTextureCriticalityCalculatorI, IlcProbabilisticCriticalityCalculatorI, IlcRelativeDemandCriticalityCalculatorI

Constructor Summary	
`public`	`IlcTextureCriticalityCalculator()`
`public`	`IlcTextureCriticalityCalculator(IlcTextureCriticalityCalculatorI * impl)`

Method Summary	
`public IlcFloat`	`calculateCriticalityGreaterThan(IlcFloat expectedDemand, IlcFloat constraintVal, IlcFloat expectedVariance=0.) const`
`public IlcFloat`	`calculateCriticalityLessThan(IlcFloat expectedDemand, IlcFloat constraintVal, IlcFloat expectedVariance=0.) const`
`public IlcTextureCriticalityCalculatorI *`	`getImpl() const`
`public void`	`operator=(const IlcTextureCriticalityCalculator & h)`

Constructors

`public` **`IlcTextureCriticalityCalculator`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcTextureCriticalityCalculator`**`(IlcTextureCriticalityCalculatorI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

```
public IlcFloat calculateCriticalityGreaterThan(IlcFloat expectedDemand, IlcFloat
constraintVal, IlcFloat expectedVariance=0.) const
```

This method calculates the criticality of a maximum capacity constraint of the resource at one time point. Given the expected demand (`expectedDemand`), the variance (`expectedVariance`), and the value of the constraint (`constraintVal`), for a time point, this function calculates a measure of the likelihood that the underlying set of resource constraints represented by `expectedDemand` and `expectedVariance` will demand more than `constraintVal` units of resource. The returned value must be a non-negative value.

The actual calculation depends on the semantics of the criticality calculation. See `IlcProbabilisticCriticalityCalculatorI` and `IlcRelativeDemandCriticalityCalculatorI` for more details.

```
public IlcFloat calculateCriticalityLessThan(IlcFloat expectedDemand, IlcFloat
constraintVal, IlcFloat expectedVariance=0.) const
```

This method calculates the criticality of a minimum capacity constraint of the resource at one time point. Given the expected demand `expectedDemand`, the variance `expectedVariance`, and the value of the constraint `constraintVal`, for a time point, this function calculates a measure of the likelihood that the underlying set of resource constraints represented by `expectedDemand` and `expectedVariance` will demand less than `constraintVal` units of resource. The returned value must be a non-negative value.

The actual calculation depends on the semantics of the criticality calculation. See `IlcProbabilisticCriticalityCalculatorI` and `IlcRelativeDemandCriticalityCalculatorI` for more details.

```
public IlcTextureCriticalityCalculatorI * getImpl() const
```

This constructor creates an object by copying another one.

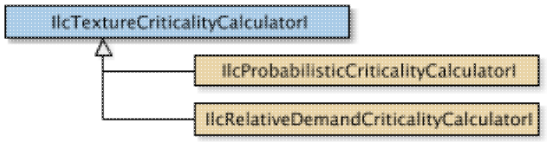
This member function returns a pointer to the implementation object of the invoking handle.

```
public void operator=(const IlcTextureCriticalityCalculator & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcTextureCriticalityCalculatorI

Definition file: ilsched/texturei.h
Include file: <ilsched/ilsched.h>



`IlcTextureCriticalityCalculatorI` is the abstract base implementation class for objects which calculate the criticality of a texture measurement at a single time point. An instance of the subclass `IlcTextureCriticalityCalculatorI` is used internally by `IlcResourceTexture` to transform the aggregate demand (and, optionally, the aggregate variance) curves into a criticality curve.

For more information, see Texture Measurements.

See Also: IlcResourceTexture, IlcTextureCriticalityCalculator, IlcProbabilisticCriticalityCalculatorI, IlcRelativeDemandCriticalityCalculatorI

Constructor and Destructor Summary	
`public`	`IlcTextureCriticalityCalculatorI()`

Method Summary	
`public virtual IlcFloat`	`calculateCriticalityGreaterThan(IlcFloat expectedDemand, IlcFloat constraintVal, IlcFloat expectedVariance) const`
`public virtual IlcFloat`	`calculateCriticalityLessThan(IlcFloat expectedDemand, IlcFloat constraintVal, IlcFloat expectedVariance) const`

Constructors and Destructors

`public` **`IlcTextureCriticalityCalculatorI()`**

This constructor creates an instance of `IlcTextureCriticalityCalculatorI`. As the class is pure abstract, this constructor should only be used automatically in the constructor of subclasses.

Methods

`public virtual IlcFloat` **`calculateCriticalityGreaterThan`**`(IlcFloat expectedDemand, IlcFloat constraintVal, IlcFloat expectedVariance) const`

This pure virtual method calculates the criticality of a maximum capacity constraint of the resource at one time point. For a time point, given `expectedDemand` and `expectedVariance`, and the value of the constraint, `constraintVal`, this function calculates a measure of the likelihood that the underlying set of resource constraints represented by demand and variance will demand more than `constraintVal` units of resource. The returned value must be non-negative.

See `IlcProbabilisticCriticalityCalculatorI` and `IlcRelativeDemandCriticalityCalculatorI` for more details.

`public virtual IlcFloat` **`calculateCriticalityLessThan`**`(IlcFloat expectedDemand,`

```
IlcFloat constraintVal, IlcFloat expectedVariance) const
```

This method calculates the criticality of a minimum capacity constraint of the resource at one time point. For a time point, given `expectedDemand` and `expectedVariance`, and the value of the constraint `constraintVal`, this function calculates a measure of the likelihood that the underlying set of resource constraints represented by demand and variance will demand less than `constraintVal` units of resource. The returned value must be a non-negative value.

See `IlcProbabilisticCriticalityCalculatorI` and `IlcRelativeDemandCriticalityCalculatorI` for more details.

Class IlcTimeBoundConstraint

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>

Instances of the class `IlcTimeBoundConstraint` are *temporalconstraints*. These temporal constraints express constraints on the time interval in which an activity is to be scheduled. (Other temporal constraints—instances of `IlcPrecedenceConstraint`—express precedence between activities in a schedule.)

This class inherits from the Solver class `IlcConstraint`, which is documented in the *Solver Reference Manual*.

Instances of this class are created by these member functions:

- `IlcActivity::startsBefore`
- `IlcActivity::endsBefore`
- `IlcActivity::startsAt`
- `IlcActivity::endsAt`
- `IlcActivity::startsAfter`
- `IlcActivity::endsAfter`

For more information, see Metaconstraints, and `IlcConstraint` in the IBM ILOG Solver Reference Manual.

See Also: IlcActivity, IlcPrecedenceConstraint, IlcTimeBoundConstraintType

Constructor Summary	
`public`	`IlcTimeBoundConstraint()`
`public`	`IlcTimeBoundConstraint(IlcTimeBoundConstraintI * impl)`

Method Summary	
`public IlcActivity`	`getActivity() const`
`public IlcTimeBoundConstraintI *`	`getImpl() const`
`public IlcInt`	`getTimeBound() const`
`public IlcIntVar`	`getTimeBoundVariable() const`
`public IlcTimeBoundConstraintType`	`getType() const`
`public IlcBool`	`hasTimeBoundVariable() const`
`public void`	`operator=(const IlcTimeBoundConstraint & h)`

Constructors

`public` **`IlcTimeBoundConstraint`**`()`

This constructor creates an instance which is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

`public` **`IlcTimeBoundConstraint`**`(IlcTimeBoundConstraintI * impl)`

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

Methods

```
public IlcActivity getActivity() const
```

This member function returns the activity of the invoking time-bound constraint.

```
public IlcTimeBoundConstraintI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcInt getTimeBound() const
```

This member function returns the time bound of the invoking time-bound constraint.

```
public IlcIntVar getTimeBoundVariable() const
```

This member function returns the time-bound variable of the invoking time-bound constraint.

```
public IlcTimeBoundConstraintType getType() const
```

This member function returns the type of the invoking time-bound constraint.

```
public IlcBool hasTimeBoundVariable() const
```

This member function returns `IlcTrue` if the invoking time-bound constraint has a time-bound variable. Otherwise, it returns `IlcFalse`.

```
public void operator=(const IlcTimeBoundConstraint & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument. After execution of this operator, the invoking object and the provided argument point to the same implementation object.

Class IlcTransitionCostObject

Definition file: ilsched/trancost.h
Include file: <ilsched/ilsched.h>

An instance of this class can be passed as an argument to the function `IlcUnaryResource::addNextTransitionCost` or `IlcUnaryResource::addPrevTransitionCost` to declare a cost to factor into a sequence constraint added to the resource. It defines a set of functions that calculate the transition, setup, and teardown costs between resource constraints.

The transition cost is the cost to have the activity of one resource constraint as the successor to the activity of another. The setup cost is the cost for the activity of a resource constraint to be sequenced first. The teardown cost is the cost for the activity of a resource constraint to be sequenced last.

The class `IlcTransitionCostObject` is the handle class of the class `IlcTransitionCostObjectI`. To define an `IlcTransitionCostObject` class adapted to your needs, you must encode the virtual functions declared in the `IlcTransitionCostObjectI` class. For an example of this, refer to `IlcTransitionCostObjectI`.

You can also use an instance of `IlcTransitionTable` as the argument of the function `IlcMakeTransitionCost`.

For more information, see Sequence Constraint.

See Also: IlcMakeTransitionCost, IlcResourceConstraint, IlcTransitionCost, IlcTransitionCostObjectI

Constructor Summary	
public	`IlcTransitionCostObject()`
public	`IlcTransitionCostObject(IlcTransitionCostObjectI * impl)`

Method Summary	
public IlcTransitionCostObjectI *	`getImpl() const`
public IlcInt	`getSetupCost(const IlcResourceConstraint srct1) const`
public IlcInt	`getSetupCostMax(const IlcResourceConstraint srct1) const`
public IlcInt	`getSetupCostMin(const IlcResourceConstraint srct1) const`
public IlcInt	`getTeardownCost(const IlcResourceConstraint srct1) const`
public IlcInt	`getTeardownCostMax(const IlcResourceConstraint srct1) const`
public IlcInt	`getTeardownCostMin(const IlcResourceConstraint srct1) const`
public IlcInt	`getTransitionCost(const IlcResourceConstraint srct1, const IlcResourceConstraint srct2) const`
public IlcInt	`getTransitionCostMax(const IlcResourceConstraint srct1, const IlcResourceConstraint srct2) const`
public IlcInt	

	getTransitionCostMin(const IlcResourceConstraint srct1, const IlcResourceConstraint srct2) const
public IlcBool	isVariable() const
public void	operator=(const IlcTransitionCostObject & h)

Constructors

public **IlcTransitionCostObject**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcTransitionCostObject**(IlcTransitionCostObjectI * impl)

This constructor creates a handle object from a pointer to an implementation object.

Methods

public IlcTransitionCostObjectI * **getImpl**() const

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

public IlcInt **getSetupCost**(const IlcResourceConstraint srct1) const

This member function returns the constant setup cost of the activity of the resource constraint `srct1`. The setup cost is the cost for this activity to be first in the sequence.

public IlcInt **getSetupCostMax**(const IlcResourceConstraint srct1) const

This member function returns the maximal variable setup cost of the activity of the resource constraint `srct1`. The setup cost is the cost for this activity to be first in the sequence.

public IlcInt **getSetupCostMin**(const IlcResourceConstraint srct1) const

This member function returns the minimal variable setup cost of the activity of the resource constraint `srct1`. The setup cost is the cost for this activity to be first in the sequence.

public IlcInt **getTeardownCost**(const IlcResourceConstraint srct1) const

This member function returns the constant teardown cost of the activity of the resource constraint `srct1`. The teardown cost is the cost for this activity to be last in the sequence.

public IlcInt **getTeardownCostMax**(const IlcResourceConstraint srct1) const

This member function returns the maximal variable teardown cost of the activity of the resource constraint `srct1`. The teardown cost is the cost for this activity to be last in the sequence.

```
public IlcInt getTeardownCostMin(const IlcResourceConstraint srct1) const
```

This member function returns the minimal variable teardown cost of the activity of the resource constraint `srct1`. The teardown cost is the cost for this activity to be last in the sequence.

```
public IlcInt getTransitionCost(const IlcResourceConstraint srct1, const
IlcResourceConstraint srct2) const
```

This member function returns the constant transition cost if the activity of the resource constraint `srct1` immediately precedes the activity of the resource constraint `srct2` in the sequence.

```
public IlcInt getTransitionCostMax(const IlcResourceConstraint srct1, const
IlcResourceConstraint srct2) const
```

This member function returns the maximal variable transition cost if the activity of the resource constraint `srct1` immediately precedes the activity of the resource constraint `srct2` in the sequence.

```
public IlcInt getTransitionCostMin(const IlcResourceConstraint srct1, const
IlcResourceConstraint srct2) const
```

This member function returns the minimal variable transition cost if the activity of the resource constraint `srct1` immediately precedes the activity of the resource constraint `srct2` in the sequence.

```
public IlcBool isVariable() const
```

This member function returns `IlcTrue` if the invoking instance of `IlcTransitionCostObject` is variable. Otherwise, it returns `IlcFalse`.

```
public void operator=(const IlcTransitionCostObject & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IlcTransitionCostObjectI

Definition file: ilsched/trancost.h
Include file: <ilsched/ilsched.h>

The class `IlcTransitionCostObjectI` defines a set of functions that calculate the transition, setup, and teardown costs between resource constraints. Its handle class can be passed as an argument to the function `IlcUnaryResource::addNextTransitionCost` or `IlcUnaryResource::addPrevTransitionCost` to declare that it is a cost to factor into a sequence constraint added to the resource.

The transition cost is the cost to have the activity of one resource constraint as the successor to the activity of another. The setup cost of a resource constraint is the cost for its activity to be sequenced first. The teardown cost of a resource constraint is the cost for its activity to be sequenced last.

The transition cost can be variable or constant. By constant we mean that it only depends upon the precedence relationship between two activities. By variable we mean that the evaluation of the transition cost depends upon current knowledge about the other variables and constraints involved. If it is variable, the transition cost object must define its minimal and maximal value given the current knowledge about the sequence.

To define an `IlcTransitionCostObjectI` subclass adapted to your needs, you have to encode the virtual functions declared in the `IlcTransitionCostObjectI` class. For a constant transition cost, you have only to redefine the virtual function `IlcTransitionCostObjectI::getTransitionCost` and, if needed, `IlcTransitionCostObjectI::getSetupCost` and `IlcTransitionCostObjectI::getTeardownCost`. For a variable transition cost, you have only to redefine the virtual functions `IlcTransitionCostObjectI::getTransitionCostMax`, `IlcTransitionCostObjectI::getTransitionCostMin` and, if needed, `IlcTransitionCostObjectI::getSetupCostMax`, `IlcTransitionCostObjectI::getSetupCostMin`, `IlcTransitionCostObjectI::getTeardownCostMax`, and `IlcTransitionCostObjectI::getTeardownCostMin`.

See the *Example* which follows.

Example

Suppose you have a unary resource that represents a machine. Between the execution of two tasks on the machine, some workers are required to perform a maintenance operation to reset the machine. You know that this maintenance operation has a given duration (the transition time between two jobs), starts after the end of the first task, and ends before the start of the next task.

To express this last constraint you need to be able to access the start variable of the next task. As you don't know in advance which task will be the next of an activity, this may pose a problem. However, the variable transition cost facility is well suited to express this constraint.

```
class StartTransitionCostI : public IlcTransitionCostObjectI {
public:
  StartTransitionCostI();
  ~StartTransitionCostI();
  virtual IlcInt getTransitionCostMax
               (const IlcResourceConstraint srct1,
                const IlcResourceConstraint srct2) const;
  virtual IlcInt getTransitionCostMin
               (const IlcResourceConstraint srct1,
                const IlcResourceConstraint srct2) const;
  virtual IlcInt getSetupCostMin
               (const IlcResourceConstraint srct1) const;
  virtual IlcInt getSetupCostMax
               (const IlcResourceConstraint srct1) const;
  virtual IlcInt getTeardownCostMin
               (const IlcResourceConstraint srct1) const;
  virtual IlcInt getTeardownCostMax
```

```
                (const IlcResourceConstraint srct1) const;
};

StartTransitionCostI::StartTransitionCostI()
  // the transition cost is variable
  :IlcTransitionCostObjectI(IlcTrue)
{}

StartTransitionCostI::~StartTransitionCostI() {}

IlcInt
StartTransitionCostI::getTransitionCostMax
                     (const IlcResourceConstraint,
                      const IlcResourceConstraint srct2) const {
  return srct2.getActivity().getStartMax();
}

IlcInt
StartTransitionCostI::getTransitionCostMin
                           (const IlcResourceConstraint,
                            const IlcResourceConstraint srct2) const {
  return srct2.getActivity().getStartMin();
}

IlcInt
StartTransitionCostI::getSetupCostMin
                           (const IlcResourceConstraint srct1) const {
  return srct1.getActivity().getStartMin();
}

IlcInt
StartTransitionCostI::getSetupCostMax
                           (const IlcResourceConstraint srct1) const {
  return srct1.getActivity().getStartMax();
}

IlcInt
StartTransitionCostI::getTeardownCostMin
                           (const IlcResourceConstraint srct1) const {
  return srct1.getActivity().getSchedule().getTimeMax();
}

IlcInt
StartTransitionCostI::getTeardownCostMax
                           (const IlcResourceConstraint srct1) const {
  return srct1.getActivity().getSchedule().getTimeMax();
}

// Must be used during search (e.g., inside a goal)
IloSolver solver = getSolver();
IlcScheduler schedule(solver, 0, 1000);

IlcUnaryResource machine(schedule);
IlcDiscreteResource workers(schedule, 10);

// Must be used during search (e.g., inside a goal)

IlcActivity task1(schedule, 100);
IlcResourceConstraint rct1 = task1.requires(machine);
solver.add(rct1);

// Must be used during search (e.g., inside a goal)
machine.close();

solver.add(machine.makeSequenceConstraint());
IlcTransitionCostObject startObj =
     new (solver.getHeap()) StartTransitionCostI();
machine.addNextTransitionCost(startObj);

// Must be used during search (e.g., inside a goal)
IlcActivity maintenance1(schedule, 20);
solver.add(maintenance1.getStartVariable() >=
           task1.getEndVariable());

solver.add(maintenance1.getEndVariable() <=
```

```
            machine.getNextTransitionCostVar(startObj, rct1));
 solver.add(maintenance1.requires(workers, 3));
```

For more information, see Sequence Constraint.

See Also: IlcResourceConstraint, IlcTransitionCostObject

Constructor and Destructor Summary	
public	IlcTransitionCostObjectI(IlcBool isVariable=IlcFalse)

Method Summary	
public virtual IlcInt	getSetupCost(const IlcResourceConstraint srct1) const
public virtual IlcInt	getSetupCostMax(const IlcResourceConstraint srct1) const
public virtual IlcInt	getSetupCostMin(const IlcResourceConstraint srct1) const
public virtual IlcInt	getTeardownCost(const IlcResourceConstraint srct1) const
public virtual IlcInt	getTeardownCostMax(const IlcResourceConstraint srct1) const
public virtual IlcInt	getTeardownCostMin(const IlcResourceConstraint srct1) const
public virtual IlcInt	getTransitionCost(const IlcResourceConstraint srct1, const IlcResourceConstraint srct2) const
public virtual IlcInt	getTransitionCostMax(const IlcResourceConstraint srct1, const IlcResourceConstraint srct2) const
public virtual IlcInt	getTransitionCostMin(const IlcResourceConstraint srct1, const IlcResourceConstraint srct2) const
public IlcBool	isVariable() const

Constructors and Destructors

public **IlcTransitionCostObjectI**(IlcBool isVariable=IlcFalse)

This constructor creates an instance of `IlcTransitionCostObjectI`. The Boolean argument `isVariable` specifies whether the transition cost is or is not variable.

Methods

public virtual IlcInt **getSetupCost**(const IlcResourceConstraint srct1) const

This virtual member function returns the constant setup cost of the activity of the resource constraint `srct1`. The setup cost is the cost for this activity to be first in the sequence. By default, it returns 0.

public virtual IlcInt **getSetupCostMax**(const IlcResourceConstraint srct1) const

This virtual member function returns the maximum variable setup cost of the activity of the resource constraint `srct1`. The setup cost is the cost for this activity to be first in the sequence. By default, it returns 0.

public virtual IlcInt **getSetupCostMin**(const IlcResourceConstraint srct1) const

This virtual member function returns the minimum variable setup cost of the activity of the resource constraint `srct1`. The setup cost is the cost for this activity to be first in the sequence. By default, it returns 0.

```
public virtual IlcInt getTeardownCost(const IlcResourceConstraint srct1) const
```

This virtual member function returns the constant teardown cost of the activity of the resource constraint `srct1`. The teardown cost is the cost for this activity to be last in the sequence. By default, it returns 0.

```
public virtual IlcInt getTeardownCostMax(const IlcResourceConstraint srct1) const
```

This virtual member function returns the maximum variable teardown cost of the activity of the resource constraint `srct1`. The teardown cost is the cost for this activity to be last in the sequence. By default, it returns 0.

```
public virtual IlcInt getTeardownCostMin(const IlcResourceConstraint srct1) const
```

This virtual member function returns the minimum variable teardown cost of the activity of the resource constraint `srct1`. The teardown cost is the cost for this activity to be last in the sequence. By default, it returns 0.

```
public virtual IlcInt getTransitionCost(const IlcResourceConstraint srct1, const
IlcResourceConstraint srct2) const
```

This virtual member function returns the constant transition cost if the activity of resource constraint `srct1` immediately precedes the activity of resource constraint `srct2` in the sequence. By default, it raises an error.

```
public virtual IlcInt getTransitionCostMax(const IlcResourceConstraint srct1, const
IlcResourceConstraint srct2) const
```

This virtual member function returns the maximum variable transition cost if the activity of the resource constraint `srct1` immediately precedes the activity of the resource constraint `srct2` in the sequence. By default, it raises an error if the invoking instance is variable and returns the constant transition cost if the invoking instance is constant.

```
public virtual IlcInt getTransitionCostMin(const IlcResourceConstraint srct1, const
IlcResourceConstraint srct2) const
```

This virtual member function returns the minimum variable transition cost if the activity of the resource constraint `srct1` immediately precedes the activity of the resource constraint `srct2` in the sequence. By default, it raises an error if the invoking instance is variable and returns the constant transition cost if the invoking instance is constant.

```
public IlcBool isVariable() const
```

This member function returns `IlcTrue` if the invoking instance of `IlcTransitionCostObjectI` is variable. Otherwise, it returns `IlcFalse`.

Class IlcTransitionTable

Definition file: ilsched/trancost.h
Include file: <ilsched/ilsched.h>

An instance of `IlcTransitionTable` is a square table of non-negative integers that defines the data for instances of `IlcTransitionCostObject` or `IlcTransitionTimeObject`.

The transition type of an instance of `IlcActivity` is used as the index of a table to calculate the transition cost or time. You can create an instance of `IlcTransitionCostObject` by calling the function `IlcMakeTransitionCost`. You can create an instance of `IlcTransitionTimeObject` by calling the function `IlcMakeTransitionTime`.

The table must be filled with non-negative integers. By default, it is initially filled with zeros. The table may or may not be symmetric, that is, the transition cost may or may not be different if an activity follows or precedes another one. If a table is declared as symmetric, the Scheduler Engine allocates only the required triangular half of the table and only that half needs to be filled. The index of the line of the table is the transition type of the preceding activity. The index of the column of the table is the transition type of the following activity.

Once the transition table has been associated with a transition time or a transition cost, the table is considered as frozen; that is, it cannot be modified any more with the member function `IlcTransitionTable::setValue`.

For more information, see Transition Time in Scheduler Engine, Disjunctive Constraint, Transition Cost (Setup and Teardown Costs) In Scheduler Engine, and Sequence Constraint.

See Also: IlcMakeTransitionCost, IlcMakeTransitionTime

Constructor Summary	
public	`IlcTransitionTable()`
public	`IlcTransitionTable(IlcTransitionTableI * impl)`
public	`IlcTransitionTable(IlcSchedule schedule, IlcInt size, IlcBool isSymmetric=IlcTrue)`
public	`IlcTransitionTable(IlcSchedule schedule, IlcInt size, IlcInt ** _table)`

Method Summary	
public IlcTransitionTableI *	`getImpl() const`
public const char *	`getName() const`
public IlcAny	`getObject() const`
public IlcSchedule	`getSchedule() const`
public IlcInt	`getSize() const`
public IloSolver	`getSolver() const`
public IloSolverI *	`getSolverI() const`
public IlcInt	`getValue(IlcInt line, IlcInt column) const`
public IlcBool	`operator!=(const IlcTransitionTable & table) const`
public void	`operator=(const IlcTransitionTable & h)`
public IlcBool	`operator==(const IlcTransitionTable & table) const`
public void	`setName(const char * name) const`
public void	`setObject(IlcAny object) const`

public void	setValue(IlcInt line, IlcInt column, IlcInt value)

Constructors

public **IlcTransitionTable**()

This constructor creates an empty handle. You must initialize it before you use it.

public **IlcTransitionTable**(IlcTransitionTableI * impl)

This constructor creates a handle object from a pointer to an implementation object.

public **IlcTransitionTable**(IlcSchedule schedule, IlcInt size, IlcBool isSymmetric=IlcTrue)

This constructor creates a new instance of IlcTransitionTable. The size argument, which must be a strictly non-negative integer, gives the number of lines and columns of the transition table. The table is initially filled with zeroes. The Boolean argument isSymmetric expresses the fact that the table is symmetric. If isSymmetric is true, only half of the table needs to be defined.

public **IlcTransitionTable**(IlcSchedule schedule, IlcInt size, IlcInt ** _table)

This constructor creates a new instance of IlcTransitionTable. The size argument, which must be a strictly non-negative integer, gives the number of lines and columns of the transition table. The table is filled with the contents of the _table argument. The _table argument must fit size and be filled with non-negative integers.

Methods

public IlcTransitionTableI * **getImpl**() const

This constructor creates an object by copying another one. This constructor creates an object by copying another one. This member function returns a pointer to the implementation object of the invoking handle.

public const char * **getName**() const

This member function returns the name of the invoking object.

public IlcAny **getObject**() const

This member function returns a pointer to the external object associated with the invoking object, if there is such an association. It returns 0 (zero) otherwise.

public IlcSchedule **getSchedule**() const

This member function returns the schedule to which the invoking transition table belongs. Each transition table belongs to a unique schedule, an instance of IlcSchedule.

public IlcInt **getSize**() const

This member function returns the size of the invoking transition table.

public IloSolver **getSolver**() const

This member function returns an instance of `IloSolver` associated with the invoking object.

```
public IloSolverI * getSolverI() const
```

This member function returns a pointer to the implementation object of the solver where the invoking object was extracted.

```
public IlcInt getValue(IlcInt line, IlcInt column) const
```

This member function returns the positive integer of the invoking transition table for the line `line` and for the column `column`. The arguments `line` and `column` must be non-negative integers strictly smaller than the size of the table. The argument `line` is the transition type of the preceding activity. The argument `column` is the transition type of the following activity.

```
public IlcBool operator!=(const IlcTransitionTable & table) const
```

This operator returns `IlcTrue` if and only if `table` does *not* refer to the same implementation object as the invoking transition table.

```
public void operator=(const IlcTransitionTable & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

```
public IlcBool operator==(const IlcTransitionTable & table) const
```

This operator returns `IlcTrue` if and only if `table` refers to the same implementation object as the invoking transition table.

```
public void setName(const char * name) const
```

This member function sets the name of the invoking object to a copy of name. This assignment is a reversible action.

```
public void setObject(IlcAny object) const
```

This member function establishes a link between the invoking object and an external object of which the invoking object might be a data member.

```
public void setValue(IlcInt line, IlcInt column, IlcInt value)
```

This member function sets the argument `value` as the value of the invoking transition table for the arguments `line` and `column`. The arguments `line` and `column` must be non-negative integers, strictly smaller than the size of the table. The argument `value` must be a non-negative integer. The argument `line` is the transition type of the preceding activity. The argument `column` is the transition type of the following activity.

Class IlcTransitionTimeObject

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>

An instance of the class `IlcTransitionTimeObject` can be passed to the constructors of the classes `IlcUnaryResource` and `IlcStateResource`. It then defines which transition time function will be used for the resource being constructed.

The simplest way to define a transition time object is to use the macro `IlcTransitionTime`.

A more general way to define a transition time object is to define a new class of transition time objects. An instance of `IlcTransitionTimeObject` uses the virtual member function `IlcTransitionTimeObjectI::getTransitionTime` to define a transition time function. For an example of this, refer to `IlcTransitionTimeObjectI`.

See Also: IlcMakeTransitionTime, IlcStateResource, IlcTransitionTime, IlcTransitionTimeObjectI, IlcUnaryResource

Constructor Summary	
public	`IlcTransitionTimeObject(const IlcTransitionTimeObject & ttobj)`
public	`IlcTransitionTimeObject(IlcTransitionTimeObjectI * impl)`

Method Summary	
public IlcTransitionTimeObjectI *	`getImpl() const`
public void	`operator=(const IlcTransitionTimeObject & ttobj)`

Constructors

`public` **`IlcTransitionTimeObject`**`(const IlcTransitionTimeObject & ttobj)`

This copy constructor creates a transition time object by copying another one. After execution of this constructor, both the newly created object and `ttobj` point to the same implementation object. C++ relies on this constructor when you pass a transition time object as an argument to a function.

`public` **`IlcTransitionTimeObject`**`(IlcTransitionTimeObjectI * impl)`

This constructor creates an instance of the handle class `IlcTransitionTimeObject` from the pointer to an instance of its implementation class `IlcTransitionTimeObjectI`.

Methods

`public IlcTransitionTimeObjectI * ` **`getImpl`**`() const`

This member function returns a pointer to the implementation object associated with the invoking transition time object.

```
public void operator=(const IlcTransitionTimeObject & ttobj)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the argument `ttobj`. After the execution of this operator, the invoking object and the `ttobj` object both point to the same implementation object. A transition time object must be assigned before it can be used; this assignment operator is useful for that purpose.

Class IlcTransitionTimeObjectI

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>

The class `IlcTransitionTimeObjectI` defines a set of functions that calculate the transition time between resource constraints.

To define an `IlcTransitionTimeObjectI` class adapted to your needs, you have to encode the virtual functions declared in the `IlcTransitionTimeObjectI` class.

See Also: IlcMakeTransitionTime, IlcTransitionTimeObject

Method Summary	
`public virtual IlcInt`	`getTransitionTime(const IlcResourceConstraint srct1, const IlcResourceConstraint srct2) const`

Methods

`public virtual IlcInt` **`getTransitionTime`**`(const IlcResourceConstraint srct1, const IlcResourceConstraint srct2) const`

This virtual member function returns the transition time if the activity of resource constraint `rct1` immediately precedes the activity of resource constraint `rct2` in the sequence. By default, it raises an error.

Example

To define an `IlcTransitionTimeObjectI` subclass adapted to your needs, you have to encode the virtual functions declared in the `IlcTransitionTimeObjectI` class. To do so, you must define a subclass of the implementation class `IlcTransitionTimeObjectI`. In the subclass, you must then redefine the virtual member function `getTransitionTime`.

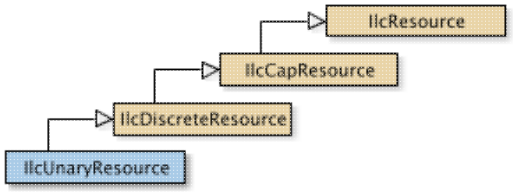
```
class TransTimeObject: public IlcTransitionTimeObjectI {
private:
  IlcInt _distance;
public:
  TransTimeObject(IlcInt distance);
  IlcInt getTransitionTime(const IlcResourceConstraint rct1,
                           const IlcResourceConstraint rct2)const;
};

TransTimeObject::TransTimeObject(IlcInt distance)
:_distance(distance)
{}

IlcInt
TransTimeObject::getTransitionTime
 (const IlcResourceConstraint,
  const IlcResourceConstraint) const
{
  return _distance;
}

IlcTransitionTimeObject myTransTimeObject(IloSolver s, *
                                          IlcInt distance) {
  return new (s.getImpl()) TransTimeObject(distance);
}
```

Now you can use this last function to define the transition time of a resource, like this:

```
IlcUnaryResource resource(schedule,
                          myTransitionTimeObject(solver, 2));
```

Class IlcUnaryResource

Definition file: ilsched/unary.h
Include file: <ilsched/ilsched.h>



An instance of the class `IlcUnaryResource` represents a discrete resource with capacity one.

As with discrete resources, there are two methods to take into account the constraints concerning the requirement of a *unary* resource. For reasons of efficiency, both methods specialize the methods of discrete resources.

- The first method allows capacity to vary over time: at any given time, the resource may or may not be in use.
- The second method deals only with requiring activities: it consists of posting a global, disjunctive constraint to insure that the time intervals over which two activities require the unary resource cannot overlap in time. No timetable needs to be created if this method is used.

In fact, that second method automatically *updates* the earliest and latest start and end times of activities by means of that posted global, disjunctive constraint. When using the second method, you can increase the level of propagation in a similar way as for discrete resources with the member function `IlcDiscreteResource::setEdgeFinder`.

The second method (of dealing only with requiring activities) allows you to define *transitiontimes* between any two activities that require the same unary resource. Given two activities A1 and A2, the transition time between A1 and A2 is an amount of time that must elapse between the end of A1 and the beginning of A2 when A1 precedes A2. The member function IlcResource::getTransitionTime returns the transition time between two activities that require the resource under consideration (that is, the invoking resource). See Transition Time in Scheduler Engine.

Sequence Constraints

Since a unary resource can only process one activity at a time, all activities requiring the same unary resource must be chronologically ordered to find a solution. As a result, in any solution to a problem that includes a unary resource, each unary resource defines a directed path through all the activities requiring it.

The nodes of such a path correspond to resource constraints of the time extent `IlcFromStartToEnd`. The links between the resource constraints can hold transition costs. See `IlcTransitionCostObject` for more details.

The path has, for its first node, a virtual node before any activities. The link between this first node and the first activity on the resource holds the setup cost. This first activity is called the setup activity.

The path also has, for its last node, a virtual node after all activities. The link between this last node and the last activity on the resource holds the teardown cost. This last activity is called the teardown activity.

Because the path has a directed cost, there are two possible orientations for calculating a transition cost: between a node and the set of its possible successors or between a node and the set of its possible predecessors.

In first case, the variable cost of a node is calculated between the virtual setup node and the set of possible setup activities, and between an activity and the set of its possible following activities. There is no teardown cost.

In the second case, the variable cost of a node is calculated between an activity and the set of its possible preceding activities, and between the virtual teardown node and the set of possible teardown activities. There is

no setup cost.

Printing or Displaying Unary Resources

The printed representation of an instance of the class `IlcUnaryResource` consists of its name followed its capacity, which is 1, enclosed in brackets.

If the Solver trace is active and the resource is not named, the string `"IlcUnaryResource"` is followed by the address of the implementation object. The address will be enclosed in parentheses.

For more information, see the concepts Disjunctive Constraint, Edge Finder, Ranking , Sequence Constraint, Timetable, Transition Cost (Setup and Teardown Costs) In Scheduler Engine, and Transition Time in Scheduler Engine.

See Also: IlcDiscreteResource, IlcIntTimetable, IlcRank, IlcResource, IlcTransitionCostObject, IlcUnaryResourceIterator

Constructor Summary	
public	`IlcUnaryResource()`
public	`IlcUnaryResource(IlcUnaryResourceI * impl)`
public	`IlcUnaryResource(IlcSchedule schedule, IlcBool disjunctive=IlcTrue)`
public	`IlcUnaryResource(IlcSchedule schedule, IlcTransitionTimeObject ttobj, IlcBool disjunctive=IlcTrue)`

Method Summary	
public IlcUnaryResourceI *	`getImpl() const`
public IlcConstraint	`getSequenceConstraint() const`
public IlcResourceConstraint	`getSetupRC() const`
public IlcResourceConstraint	`getTeardownRC() const`
public IlcResourceConstraint	`getVirtualNodeRC() const`
public IlcBool	`hasSequenceConstraint() const`
public IlcBool	`hasSetupRC() const`
public IlcBool	`hasTeardownRC() const`
public IlcBool	`isRanked() const`
public IlcBool	`isSequenced() const`
public void	`operator=(const IlcUnaryResource & h)`

Inherited Methods from `IlcDiscreteResource`
`getCapacity, getCapacityMax, getCapacityMaxMax, getCapacityMaxMin, getCapacityMin, getCapacityMinMax, getCapacityMinMin, getGlobalSlack, getImpl, getLocalSlack, getTypeTimetableConstraint, hasTypeTimetableConstraint, makeDisjunctiveConstraint, makeTypeTimetableConstraint, operator=, setCapacityMax, setCapacityMin, setEdgeFinder, setPrecedencePropagation, setTimetablePropagation, storeSufficientDirectSuccessors`

Inherited Methods from `IlcCapResource`
`getImpl, getMaxTextureMeasurement, getMinTextureMeasurement, getTimetable, getTimetable, hasInitialOccupation, hasMaxTextureMeasurement, hasMinTextureMeasurement, incrDurableRequirement, incrDurableRequirement, isRedundantResource, makeBalanceConstraint, makeMaxTextureMeasurement, makeMinTextureMeasurement, makeTimetableConstraint, makeTimetableConstraint,`

makeTimetableConstraint, operator=, setInitialOccupation, setInitialOccupation, unsetInitialOccupation

Inherited Methods from `IlcResource`
close, getCalendar, getDisjunctiveConstraint, getDurableSchedule, getImpl, getLastRankedFirstRC, getLastRankedLastRC, getLastSurelyContributingRankedFirstRC, getLastSurelyContributingRankedLastRC, getName, getObject, getOldLastRankedFirstRC, getOldLastRankedLastRC, getPrecedenceGraphConstraint, getSchedule, getSolver, getSolverI, getTimetableConstraint, getTransitionTime, hasCalendar, hasDisjunctiveConstraint, hasLightPrecedenceGraphConstraint, hasPrecedenceGraphConstraint, hasPrecedenceInfo, hasRankInfo, hasTimetableConstraint, isCapacityResource, isClosed, isContinuousReservoir, isDiscreteEnergy, isDiscreteResource, isDurable, isReservoir, isStateResource, isTransitionTimeSuspended, isUnaryResource, makeFunctionalConstraint, makeIntegralConstraint, makeLightPrecedenceGraphConstraint, makePrecedenceGraphConstraint, operator!=, operator=, operator==, setCalendar, setName, setObject, setTransitionTimeObject, setTransitionTimeSuspended, whenContribution, whenDirectPredecessors, whenDirectSuccessors, whenNext, whenPossibleNext, whenPossiblePrevious, whenPredecessors, whenPrevious, whenRankedFirstRC, whenRankedLastRC, whenSuccessors

Constructors

```
public IlcUnaryResource()
```

This constructor creates an instance which is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

```
public IlcUnaryResource(IlcUnaryResourceI * impl)
```

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

```
public IlcUnaryResource(IlcSchedule schedule, IlcBool disjunctive=IlcTrue)
```

This constructor creates a new instance of `IlcUnaryResource` and adds it to the set of resources managed in the given `schedule`. The capacity of the resource is 1 (one). The argument `disjunctive` indicates whether the disjunctive constraint should be posted.

```
public IlcUnaryResource(IlcSchedule schedule, IlcTransitionTimeObject ttobj,
IlcBool disjunctive=IlcTrue)
```

This constructor creates a new instance of `IlcUnaryResource` and adds it to the set of resources managed in the given `schedule`. The capacity of the resource is 1 (one). The argument `ttobj` indicates which transition time function will be used for the invoking resource. The argument `disjunctive` indicates whether the disjunctive constraint should be posted.

Transition times are taken into account when the disjunctive constraint or the type timetable constraint is posted. If both constraints are posted, transition times will be taken into account only by the disjunctive constraint.

If the argument `ttobj` has not been built with an instance of `IlcTransitionTable`, the type timetable constraint will be unable to take transition times into account. See Transition Time in Scheduler Engine and Type Timetable Constraint for more information.

Methods

`public IlcUnaryResourceI * getImpl() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public IlcConstraint getSequenceConstraint() const`

This member function returns the sequence constraint attached to the resource, if it exists.

`public IlcResourceConstraint getSetupRC() const`

This member function returns the resource constraint required by the setup activity of the invoking resource. This member function should be called only if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`).

`public IlcResourceConstraint getTeardownRC() const`

This member function returns the resource constraint required by the teardown activity of the invoking resource. This member function should be called only if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`).

`public IlcResourceConstraint getVirtualNodeRC() const`

This member function returns the sequence virtual node of the invoking unary resource. The sequence virtual node of a unary resource is an automatically created resource constraint that do not affect the availability of the resource and that is used in the sequence goals and selectors to represent the virtual initial (in case of a chronological sequence goal like `IlcSequence`) or final (in case of an anti-chronological sequence goal like `IlcSequenceBackward`) resource constraint in the sequence of resource constraints of the unary resource .

`public IlcBool hasSequenceConstraint() const`

This member function returns `IlcTrue` if a sequence constraint is attached on the invoking resource. Otherwise it returns `IlcFalse`.

`public IlcBool hasSetupRC() const`

This member function returns `IlcTrue` if the setup activity of invoking resource is known. Otherwise it returns `IlcFalse`. This member function should be called only if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`).

`public IlcBool hasTeardownRC() const`

This member function returns `IlcTrue` if the invoking resource has a teardown activity. Otherwise it returns `IlcFalse`. This member function should be called only if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`).

`public IlcBool isRanked() const`

This member function returns `IlcTrue` if all resource constraints defined on the invoking resource have been ranked. Otherwise, it returns `IlcFalse`. This member function should be called only if some rank information is available on the resource (see member function `IlcResource::hasRankInfo`).

```
public IlcBool isSequenced() const
```

This member function returns `IlcTrue` if the invoking resource is sequenced. That is, if each visited activity has a predecessor (or is the setup activity) and a successor (or is the teardown activity). Otherwise it returns `IlcFalse`.

```
public void operator=(const IlcUnaryResource & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument. After execution of this operator, the invoking object and the provided argument point to the same implementation object.

Class IlcUnaryResourceIterator

Definition file: ilsched/schedulerdoc.h
Include file: <ilsched/ilsched.h>

An instance of this class traverses the set of unary resources.

See Also: IlcUnaryResource, IlcSchedule

Constructor and Destructor Summary	
public	IlcUnaryResourceIterator(const IlcSchedule schedule)

Method Summary	
public IlcBool	ok() const
public IlcUnaryResource	operator*() const
public IlcUnaryResourceIterator &	operator++()

Constructors and Destructors

`public` **`IlcUnaryResourceIterator`**`(const IlcSchedule schedule)`

This constructor creates an iterator to traverse all the unary resources of `schedule`.

Methods

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if the current position of the iterator is a valid one. It returns `IlcFalse` if all the unary resources have been scanned by the iterator.

`public IlcUnaryResource` **`operator*`**`() const`

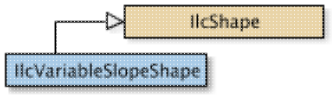
This operator accesses the instance of `IlcUnaryResource` located at the current position of the iterator. If the iterator is set past the end position, this operator returns an empty handle.

`public IlcUnaryResourceIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator.

Class IlcVariableSlopeShape

Definition file: ilsched/shaperct.h
Include file: <ilsched/ilsched.h>



An instance of `IlcVariableSlopeShape` holds a description of a shape with variable slope.

See Also: IlcResourceConstraint

Constructor Summary	
public	`IlcVariableSlopeShape()`
public	`IlcVariableSlopeShape(IlcShapeI * impl)`
public	`IlcVariableSlopeShape(const IlcShape & shape)`

Method Summary	
public IlcShapeI *	`getImpl() const`
public IlcFloatVar	`getSlopeVariable() const`
public void	`operator=(const IlcVariableSlopeShape & h)`

Inherited Methods from `IlcShape`
`getImpl, getName, getObject, getResourceConstraint, getSolver, getSolverI, isVariableSlopeShape, operator=, setName, setObject`

Constructors

`public` **`IlcVariableSlopeShape`**`()`

This constructor creates an instance which is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

`public` **`IlcVariableSlopeShape`**`(IlcShapeI * impl)`

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

`public` **`IlcVariableSlopeShape`**`(const IlcShape & shape)`

This copy-constructor provides a safe down-cast of a generic instance of `IlcShape` into an instance of `IlcVariableSlopeShape`. In debug mode, an assertion failure will be raised if the `IlcShape` is not an instance of `IlcVariableSlopeShape`.

Methods

`public IlcShapeI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object of the invoking handle.

```
public IlcFloatVar getSlopeVariable() const
```

This member function returns the variable parameterizing the slope of the shape. This parameter is supplied at construction time, when invoking `IlcResourceConstraint::makeVariableSlopeShape`.

See Also: IlcResourceConstraint

```
public void operator=(const IlcVariableSlopeShape & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument. After execution of this operator, the invoking object and the provided argument point to the same implementation object.

Class IlcWorkServer

Definition file: ilsched/workserv.h
Include file: <ilsched/workserv.h>

This class offers facilities for writing multi-threaded applications. It allows you to create a specified number of threads and then to execute different pieces of code on these threads asynchronously. Each thread is associated with a unique environment that is an instance of `IloEnv`; the code executed by each thread is a Solver goal.

Upon request, the workserver object provides you with the environment or solver of a thread waiting for work. An instance of `IloGoal` can be constructed in the environment. It can then be extracted to the solver of the corresponding thread and executed on the thread.

For more information, see Durability.

See Also: IlcGetThreadId

Constructor Summary	
public	IlcWorkServer()
public	IlcWorkServer(IlcBaseAgentServerI * impl)
public	IlcWorkServer(const IlcScheduler sched0, IlcInt nbOfThreads, const char * prefix=0)

Method Summary	
public void	end()
public IloEnv	getIdleEnv()
public IloSolver	getIdleSolver()
public IlcBaseAgentServerI *	getImpl() const
public void	launch(IloGoal goal)
public void	operator=(const IlcWorkServer & h)

Constructors

`public` **`IlcWorkServer`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IlcWorkServer`**`(IlcBaseAgentServerI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IlcWorkServer`**`(const IlcScheduler sched0, IlcInt nbOfThreads, const char * prefix=0)`

This constructor creates a work server that starts `nbOfThreads` worker threads. Each worker thread has a different environment and waits to receive work. If the number of threads is 0, then no worker thread is created

and further calls to the member function `launch` of this `IlcWorkServer` will be treated sequentially.

The created work server will be used to solve solver goals that use the durable resources of the schedule `sched0`. Note that `sched0` must be a durable and closed schedule. If `sched0` is not a durable schedule or is not closed, any attempt to launch a goal on this work server will raise an error.

The argument `prefix` serves as a base for creating names for the output streams of each thread: the first thread will output in `prefix0.out`, the second in `prefix1.out`, etc. If `prefix` is 0, then the default output stream is considered.

Note

An application containing a workserver object can be compiled as a single threaded application. In this case, the actual value of the parameter `nbOfThreads` is ignored and is considered to be zero.

Methods

```
public void end()
```

This member function waits for all worker threads to finish their work, then stops them and calls `end()` on each worker's environment.

```
public IloEnv getIdleEnv()
```

This member function returns the environment associated with a thread that is waiting for work. If there is no such thread, this call may block while waiting for a free worker.

```
public IloSolver getIdleSolver()
```

This member function returns the solver associated with a thread that is waiting for work. If there is no such thread, this call may block while waiting for a free worker.

```
public IlcBaseAgentServerI * getImpl() const
```

This constructor creates an object by copying another one.

This member function returns a pointer to the implementation object of the invoking handle.

```
public void launch(IloGoal goal)
```

The argument `goal` must be a solver goal constructed on the same environment as the solver obtained by a previous call to `getIdleSolver`. The waiting worker whose solver was returned by the previous call at `getIdleSolver` starts working by extracting and executing the `goal`. Calls to `getIdleSolver` and `launch` must be interleaved. At the end of the execution of the goal, the memory allocated on the idle solver is collected.

Example

Let us consider the very simple scheduling problem that consists of creating and scheduling an activity which requires a given durable resource. Suppose that ten instances of this problem are to be solved, each instance characterized by the particular resource it needs. The inherent parallelism of this application can by exploited by solving different instances on different threads.

The following Solver goal defines and solves an instance of the scheduling problem.

```
#include <ilsched/iloscheduler.h>
#include <ilsched/workserv.h>

 ILCGOAL1(ProblemIlc,
         IlcDiscreteResource, resource) {
   IloSolver solver = getSolver();
   IlcSchedule schedule(solver, 0, 365);
   IlcActivity activity(scheduler, 10);
   scheduler.lock(1, resource);
   solver.add(activity.requires(resource, 3));
   solver.startNewSearch(IlcSetTimes(schedule));
   solver.next();
   schedule.unlock(1, resource);
   solver.endSearch();
   return 0;
 }

 ILOCPGOALWRAPPER2(Problem, solver,
                   IlcScheduler, durableSched,
                   IloDiscreteResource, resource) {
   return ProblemIlc(solver,
                     durableSched.getDiscreteResource(resource));
 }

 int main(){

   // Creating the durable resources in a model:
   IloEnv env0;
   IloModel model0(env0);
   const IloInt nbOfResources = 2;
   IloDiscreteResource resources[nbOfResources];
   for (IloInt i=0; i < nbOfResources; i++) {
     resources[i] = IloDiscreteResource(env0, 7);
     model0.add(resources[i]);
   }

   // Creating the durable scheduler:
   IloSolver solver0(env0);
   IlcScheduler scheduler0(solver0);
   scheduler0.setDurable();
   solver0.extract(model0);
   scheduler0.close();

   // Creating the worker threads:
   IlcWorkServer server(scheduler0, 3);

   // Solving ten instances:
   for (IloInt k=0; k<10; k++){
     IloInt resourceIdx = k % nbOfResources;

     // Getting the solver of a waiting worker thread:
     IloEnv env = server.getIdleEnv();

     // Creating the problem instance:
     IloGoal instance = Problem(env,
                                scheduler0,
                                resources[resourceIdx]);

    // Solving it on the corresponding thread:
     server.launch(instance);
   }

   // Stopping all worker threads:
   server.end();
   env0.end();
   return 0;
 }
```

```
public void operator=(const IlcWorkServer & h)
```

This operator assigns an address to the handle pointer of the invoking object. That address is the location of the implementation object of the provided argument.

Class IloActivity

Definition file: ilsched/iloactivity.h
Include file: <ilsched/iloscheduler.h>

An instance of `IloActivity` allows modeling tasks in Scheduler. The properties of an instance of `IloActivity` can be divided into three categories: time interval, requirement of resources, and compliance with calendars (for example, the ability to be suspended by breaks).

This class inherits from the IBM® ILOG® Concert Technology class `IloExtractable`. That class is documented in the *IBM ILOG Concert Technology Reference Manual*.

Activity as a Time Interval

Several Concert Technology numerical variables define the underlying time interval of an activity. These variables are the start and end variables, which are the bounds of the time interval, and the processing time variable, which is the quantity of time for which the activity must be processed to achieve its execution according to calendar definition. Scheduler allows creating constraints on the time interval in a natural way.

A time-bound constraint enforces limits on the start or end variables of the time interval. The variable must be greater than, less than, or equal to a numerical constant or variable. The time-bound constraints `startsAfter` and `endsBefore` allow managing release dates and due dates on activities.

A temporal constraint synchronizes the execution of two activities. More precisely, the start or end variable of the preceding activity must be less than or equal to the start or end variable of the following activity, with a given delay. The delay is either a numerical constant or variable. For example, the temporal constraint `startsAfterEnd` is used to enforce that the invoking activity follows another activity.

For example:

```
IloEnv env;
IloModel m(env);
IloActivity act1(env, 5);
IloActivity act2(env, 8);
IloNumVar delay(env, -6, 10);
m.add(act1.startsAfterEnd(act2, delay);
m.add(act1.startsAfter(6));
IloNumVar duedate(env, 45, 60);
m.add(act2.endsBefore(duedate));
```

For more information, see Temporal Relations.

Activity as a Resource Requirement

A resource constraint enforces the fact that on a certain time interval a resource is used to process the activity. The class `IloActivity` offers a set of member functions to create a resource constraint.

The default member function to declare a resource constraint on a discrete resource or a state resource is `IloActivity::requires`. It means that the activity will reserve the resource from its start time to its end time.

The default member functions to declare a resource constraint on a reservoir are `IloActivity::produces`, which is used to fill the reservoir at the end of the activity, and `IloActivity::consumes`, used to empty the reservoir at the start of the activity.

For example:

```
IloEnv env;
IloModel m(env);
IloActivity act(env, 5);
IloDiscreteResource d(env, 12);
IloReservoir r(env, 200);
```

```
IloNumVar c(env, 3, 5);
m.add(act.requires(d, c));
m.add(act.consumes(r, 25));
```

Activity and Calendars

Resources and resource constraints can be associated with a calendar. A calendar allows expression of complex behavior such as suspension due to breaks, disjunctions due to shifts, or duration increase due to efficiency. For more information, see Calendars.

An activity that allows suspension because of breaks is a *breakable* activity. An activity that takes into account the efficiency function of calendars is a *useEfficiency* activity.

The behavior of an activity with respect to the calendars must be declared by the user. It allows the solver to partition the processing time of an activity between its start and end times.

The class `IloActivity` provides a full set of member functions associated with parameter classes that manage the behavior of an activity with respect to calendars.

Functional and Integral Constraints and External Variables

Functional and integral constraints are of the form *yrct = f(xrct)* or *yrct = sum{startrct -> endrct}f(t).dt* and hold for every resource constraint *rct* on a given resource (see Functional and Integral Constraints on Resources. External variables are useful when *xrct* or *yrct* are not already associated with the resource constraint (start, end, capacity demand, and so forth). In such cases, member functions of `IloActivity` allow designating IBM® ILOG® Concert Technology variables as external variables, and using these variables in functional and integral constraints (see the enumeration `IloSchedVariable`).

Parameter Classes

Instances of `IloActivityConstraintsParam` are used to choose what kind of constraints on activities should be taken into account.

Instances of `IloActivityBasicParam` are used to specify basic behavior of the activity regarding calendars, such as the activity is breakable, the activity use efficiency, and some general parameters on the breakable activity such as the maximum duration of breaks.

Instances of `IloActivityBreakParam` are used to specify the way breakable activities should be executed; for example, the type of breaks, the durations of disjunctive breaks, and the possibility to suspend the activity at its start or its end.

Instances of `IloActivityShiftParam` are used to specify the way activities should be behave in case of shifts; for instance, some shifts can be ignored regarding their type, or their duration.

Instances of `IloActivityOverlapParam` are used to specify in which conditions an activity can overlap breaks, for example, if the activity can start or end in a break.

Refer to Scheduler Overview for more information on how to share parameters among resources, and how the direct modification of parameters through the resource API may affect them.

For more information, see Calendars, Functional and Integral Constraints on Resources, Temporal Relations, Transition Costs, and Transition Times.

See Also: IloActivityBasicParam, IloActivityBreakParam, IloActivityShiftParam, IloActivityConstraintsParam, IloActivityOverlapParam, IloSchedulerSolution, IloResourceConstraint, IloTransitionParam, IloPrecedenceConstraint, IloTimeBoundConstraint, IloSchedVariable

Constructor Summary	
`public`	`IloActivity()`
`public`	`IloActivity(IloActivityI * impl)`

public	IloActivity(const IloEnv env, IloNum processingTime, const char * name)
public	IloActivity(const IloEnv env, IloNum processingTime, IloInt transitionType=0, const char * name=0)
public	IloActivity(const IloEnv env, const IloNumVar processingTimeVariable, IloInt transitionType=0, const char * name=0)

Method Summary	
public void	addDisjunctiveBreakType(const IloIntSet types) const
public void	addDisjunctiveBreakType(IloInt type) const
public void	addEndBreakOverlapType(const IloIntSet types) const
public void	addEndBreakOverlapType(IloInt type) const
public void	addIgnoredBreakType(const IloIntSet types) const
public void	addIgnoredBreakType(IloInt type) const
public void	addIgnoredShiftType(const IloIntSet types) const
public void	addIgnoredShiftType(IloInt type) const
public void	addStartBreakOverlapType(const IloIntSet types) const
public void	addStartBreakOverlapType(IloInt type) const
public IloBool	areCoverConstraintsIgnored() const
public IloBool	arePrecedenceConstraintsIgnored() const
public IloBool	areResourceConstraintsIgnored() const
public IloBool	areTimeBoundConstraintsIgnored() const
public IloBool	canBeSuspendedAtEnd() const
public IloBool	canBeSuspendedAtStart() const
public void	clearDisjunctiveBreakType() const
public void	clearEndBreakOverlapType() const
public void	clearIgnoredBreakType() const
public void	clearIgnoredShiftType() const
public void	clearStartBreakOverlapType() const
public IloAltResConstraint	consumes(const IloAltResSet, const IloNumVar capVar) const
public IloAltResConstraint	consumes(const IloAltResSet, IloNum cap=1) const
public IloResourceConstraint	consumes(const IloCapResource, const IloNumVar capVar) const
public IloResourceConstraint	consumes(const IloCapResource, IloNum cap=1) const
public IloCoverConstraint	covers() const
public IloTimeBoundConstraint	endsAfter(const IloNumVar time) const
public IloTimeBoundConstraint	endsAfter(IloNum time) const
public IloPrecedenceConstraint	endsAfterEnd(const IloActivity act, const IloNumVar delay) const
public IloPrecedenceConstraint	endsAfterEnd(const IloActivity act, IloNum delay=0) const
public IloPrecedenceConstraint	

	endsAfterStart(const IloActivity act, const IloNumVar delay) const
public IloPrecedenceConstraint	endsAfterStart(const IloActivity act, IloNum delay=0) const
public IloTimeBoundConstraint	endsAt(const IloNumVar time) const
public IloTimeBoundConstraint	endsAt(IloNum time) const
public IloPrecedenceConstraint	endsAtEnd(const IloActivity act, const IloNumVar delay) const
public IloPrecedenceConstraint	endsAtEnd(const IloActivity act, IloNum delay=0) const
public IloPrecedenceConstraint	endsAtStart(const IloActivity act, const IloNumVar delay) const
public IloPrecedenceConstraint	endsAtStart(const IloActivity act, IloNum delay=0) const
public IloTimeBoundConstraint	endsBefore(const IloNumVar time) const
public IloTimeBoundConstraint	endsBefore(IloNum time) const
public IloIntExprArg	getDurationExpr() const
public IloNum	getDurationMax() const
public IloNum	getDurationMaxNormalBreaks() const
public IloNum	getDurationMin() const
public IloNum	getDurationMinNormalBreaks() const
public IloNum	getEndBreakOverlapMax() const
public IloNum	getEndBreakOverlapMin() const
public IloIntExprArg	getEndExpr() const
public IloNum	getEndMax() const
public IloNum	getEndMin() const
public IloNum	getExecutionDurationMin() const
public IloNum	getExternalValue() const
public IloNumVar	getExternalVariable() const
public IloActivityI *	getImpl() const
public IloNum	getProcessingTimeMax() const
public IloNum	getProcessingTimeMin() const
public IloNumVar	getProcessingTimeVariable() const
public IloNum	getStartBreakOverlapMax() const
public IloNum	getStartBreakOverlapMin() const
public IloIntExprArg	getStartExpr() const
public IloNum	getStartMax() const
public IloNum	getStartMin() const
public IloInt	getTransitionType() const
public IloBool	hasDisjunctiveBreakType() const
public IloBool	hasEndBreakOverlapType() const
public IloBool	hasIgnoredBreakType() const
public IloBool	hasIgnoredShiftType() const

public IloBool	hasStartBreakOverlapType() const
public void	ignoreBreakDisjunctivity(IloBool ignored=IloTrue) const
public void	ignoreCoverConstraints(IloBool ignored=IloTrue) const
public void	ignorePrecedenceConstraints(IloBool ignored=IloTrue) const
public void	ignoreResourceConstraints(IloBool ignored=IloTrue) const
public void	ignoreTimeBoundConstraints(IloBool ignored=IloTrue) const
public IloBool	isBreakable() const
public IloBool	isBreakDisjunctivityIgnored() const
public IloBool	isDisjunctiveBreakType(IloInt type) const
public IloBool	isEndBreakOverlapType(IloInt type) const
public IloBool	isIgnoredBreakType(IloInt type) const
public IloBool	isIgnoredShiftType(IloInt type) const
public IloBool	isStartBreakOverlapType(IloInt type) const
public IloAltResConstraint	produces(const IloAltResSet, const IloNumVar capVar) const
public IloAltResConstraint	produces(const IloAltResSet, IloNum cap=1) const
public IloResourceConstraint	produces(const IloContinuousReservoir, const IloNumVar capVar) const
public IloResourceConstraint	produces(const IloContinuousReservoir, IloNum cap=1) const
public IloResourceConstraint	produces(const IloReservoir, const IloNumVar capVar) const
public IloResourceConstraint	produces(const IloReservoir, IloNum cap=1) const
public IloAltResConstraint	provides(const IloAltResSet, const IloNumVar capVar, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloFalse) const
public IloAltResConstraint	provides(const IloAltResSet, IloNum cap=1, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloFalse) const
public IloResourceConstraint	provides(const IloReservoir, const IloNumVar capVar, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloFalse) const
public IloResourceConstraint	provides(const IloReservoir, IloNum cap=1, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloFalse) const
public void	removeDisjunctiveBreakType(const IloIntSet types) const
public void	removeDisjunctiveBreakType(IloInt type) const
public void	removeEndBreakOverlapType(const IloIntSet types) const
public void	removeEndBreakOverlapType(IloInt type) const
public void	removeIgnoredBreakType(const IloIntSet types) const

public void	removeIgnoredBreakType(IloInt type) const
public void	removeIgnoredShiftType(const IloIntSet types) const
public void	removeIgnoredShiftType(IloInt type) const
public void	removeStartBreakOverlapType(const IloIntSet types) const
public void	removeStartBreakOverlapType(IloInt type) const
public IloResourceConstraint	requires(const IloStateResource, const IloAnySetVar states, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
public IloResourceConstraint	requires(const IloStateResource, const IloAnySet states, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
public IloResourceConstraint	requires(const IloStateResource, const IloAnyVar state, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
public IloResourceConstraint	requires(const IloStateResource, IloAny state, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
public IloAltResConstraint	requires(const IloAltResSet, const IloNumVar capVar, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
public IloAltResConstraint	requires(const IloAltResSet, IloNum cap=1, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
public IloResourceConstraint	requires(const IloCapResource, const IloNumVar capVar, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
public IloResourceConstraint	requires(const IloCapResource, IloNum cap=1, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
public IloResourceConstraint	requiresNot(const IloStateResource, const IloAnySetVar states, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
public IloResourceConstraint	requiresNot(const IloStateResource, const IloAnySet states, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
public IloResourceConstraint	requiresNot(const IloStateResource, const IloAnyVar state, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
public IloResourceConstraint	requiresNot(const IloStateResource, IloAny state, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
public void	setActivityBasicParam(const IloActivityBasicParam param) const
public void	setActivityBreakParam(const IloActivityBreakParam param) const
public void	

	setActivityConstraintsParam(const IloActivityConstraintsParam param) const
public void	setActivityOverlapParam(const IloActivityOverlapParam param) const
public void	setActivityShiftParam(const IloActivityShiftParam param) const
public void	setBreakable(IloBool breakable=IloTrue) const
public void	setCanBeSuspendedAtEnd(IloBool val=IloTrue) const
public void	setCanBeSuspendedAtStart(IloBool val=IloTrue) const
public void	setDurationMax(IloNum duration)
public void	setDurationMaxNormalBreaks(IloNum max) const
public void	setDurationMin(IloNum duration)
public void	setDurationMinNormalBreaks(IloNum min) const
public void	setEndBreakOverlapMax(IloNum max) const
public void	setEndBreakOverlapMin(IloNum min) const
public void	setEndMax(IloNum endMax) const
public void	setEndMin(IloNum endMin) const
public void	setExecutionDurationMin(IloNum min) const
public void	setExternalValue(IloNum val)
public void	setExternalVariable(IloNumVar var)
public void	setProcessingTimeMax(IloNum processingTime) const
public void	setProcessingTimeMin(IloNum processingTime) const
public void	setStartBreakOverlapMax(IloNum max) const
public void	setStartBreakOverlapMin(IloNum min) const
public void	setStartMax(IloNum startMax) const
public void	setStartMin(IloNum startMin) const
public void	setTransitionType(IloInt type) const
public void	setUseEfficiency(IloBool useEfficiency=IloTrue) const
public void	shareEndWithEnd(IloActivity activity)
public void	shareEndWithStart(IloActivity activity)
public void	shareStartWithEnd(IloActivity activity)
public void	shareStartWithStart(IloActivity activity)
public IloTimeBoundConstraint	startsAfter(const IloNumVar time) const
public IloTimeBoundConstraint	startsAfter(IloNum time) const
public IloPrecedenceConstraint	startsAfterEnd(const IloActivity act, const IloNumVar delay) const
public IloPrecedenceConstraint	startsAfterEnd(const IloActivity act, IloNum delay=0) const
public IloPrecedenceConstraint	startsAfterStart(const IloActivity act, const IloNumVar delay) const
public IloPrecedenceConstraint	startsAfterStart(const IloActivity act, IloNum delay=0) const

public IloTimeBoundConstraint	startsAt(const IloNumVar time) const
public IloTimeBoundConstraint	startsAt(IloNum time) const
public IloPrecedenceConstraint	startsAtEnd(const IloActivity act, const IloNumVar delay) const
public IloPrecedenceConstraint	startsAtEnd(const IloActivity act, IloNum delay=0) const
public IloPrecedenceConstraint	startsAtStart(const IloActivity act, const IloNumVar delay) const
public IloPrecedenceConstraint	startsAtStart(const IloActivity act, IloNum delay=0) const
public IloTimeBoundConstraint	startsBefore(const IloNumVar time) const
public IloTimeBoundConstraint	startsBefore(IloNum time) const
public void	unshare()
public IloBool	useEfficiency() const

Constructors

public **IloActivity**()

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

public **IloActivity**(IloActivityI * impl)

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

public **IloActivity**(const IloEnv env, IloNum processingTime, const char * name)

This constructor creates a new instance of `IloActivity`. The new activity is constrained to execute between the time origin and the time horizon of the schedule environment. Its processing time is set to the given `processingTime`. Its name is set to `name`.

public **IloActivity**(const IloEnv env, IloNum processingTime, IloInt transitionType=0, const char * name=0)

This constructor creates a new instance of `IloActivity`. The new activity is constrained to execute between the time origin and the time horizon of the schedule environment. Its processing time is set to the given `processingTime`. `transitionType` represents its transition type. Its name is set to `name`.

public **IloActivity**(const IloEnv env, const IloNumVar processingTimeVariable, IloInt transitionType=0, const char * name=0)

This constructor creates a new instance of `IloActivity`. The processing time variable of the new activity is set to `processingTimeVariable`. `transitionType` represents its transition type. Its name is set to `name`.

Methods

public void **addDisjunctiveBreakType**(const IloIntSet types) const

This member function adds the set of types `types` to the set of disjunctive break types of the invoking activity. If a break type belongs to the set of disjunctive break types of a breakable activity, the breakable activity must be completely processed either before or after that type of break.

Initially, an activity is created with an empty set of disjunctive break types.

This member function has no effect if the invoking activity is not breakable.

```
public void addDisjunctiveBreakType(IloInt type) const
```

This member function adds the type `type` to the set of disjunctive break types of the invoking activity. If a break type belongs to the set of disjunctive break types of an activity, the breakable activity must be completely processed either before or after that type of break.

Initially, an activity is created with an empty set of disjunctive break types.

This member function has no effect if the invoking activity is not breakable.

```
public void addEndBreakOverlapType(const IloIntSet types) const
```

This member function adds the set of types `types` to the set of overlap break types of the invoking activity. If a break type belongs to the set of overlap break types of an activity, the activity can be processed at its end time during that break. Initially, an activity is created with an empty set of overlap break types.

```
public void addEndBreakOverlapType(IloInt type) const
```

This member function adds the type `type` to the set of overlap break types of the invoking activity. If a break type belongs to the set of overlap break types of an activity, the activity can be processed at its end time during that break. Initially, an activity is created with an empty set of overlap break types.

```
public void addIgnoredBreakType(const IloIntSet types) const
```

This member function adds the set of types `types` to the set of ignored break types of the invoking activity. That is, the invoking activity behaves as if breaks of type included in `types` do not exist.

Initially, an activity is created with an empty set of ignored break types.

```
public void addIgnoredBreakType(IloInt type) const
```

This member function adds the type `type` to the set of ignored break types of the invoking activity. That is, the invoking activity behaves as if breaks of type `type` do not exist.

Initially, an activity is created with an empty set of ignored break types.

```
public void addIgnoredShiftType(const IloIntSet types) const
```

This member function adds the set of types `types` to the set of ignored shift types of the invoking activity. That is, the invoking activity behaves as if shifts of type included in `types` do not exist.

Initially, an activity is created with an empty set of ignored shift types.

```
public void addIgnoredShiftType(IloInt type) const
```

This member function adds the type `type` to the set of ignored shift types of the invoking activity. That is, the invoking activity behaves as if shifts of type `type` never exist.

Initially, an activity is created with an empty set of ignored shift types.

```
public void addStartBreakOverlapType(const IloIntSet types) const
```

This member function adds the set of types `types` to the set of overlap break types of the invoking activity. If a break type belongs to the set of overlap break types of an activity, the activity can be processed at its start time during that break. Initially, an activity is created with an empty set of overlap break types.

```
public void addStartBreakOverlapType(IloInt type) const
```

This member function adds `type` to the set of overlap break types of the invoking activity. If a break type belongs to the set of overlap break types of an activity, the activity can be processed at its start time during that break. Initially, an activity is created with an empty set of overlap break types.

```
public IloBool areCoverConstraintsIgnored() const
```

This member function returns `IloTrue` if the cover constraints defined on the invoking activity are not taken into account when searching for the solution. Otherwise, it returns `IloFalse`.

```
public IloBool arePrecedenceConstraintsIgnored() const
```

This member function returns `IloTrue` if the precedence constraints defined on the invoking activity are not taken into account when searching for the solution. Otherwise, it returns `IloFalse`.

```
public IloBool areResourceConstraintsIgnored() const
```

This member function returns `IloTrue` if the resource constraints defined on the invoking activity are not taken into account when searching for the solution. Otherwise, it returns `IloFalse`.

```
public IloBool areTimeBoundConstraintsIgnored() const
```

This member function returns `IloTrue` if the time-bound constraints defined on the invoking activity are not taken into account when searching for the solution. Otherwise, it returns `IloFalse`.

```
public IloBool canBeSuspendedAtEnd() const
```

This member function returns `IloTrue` if the invoking activity can be suspended at its end time. Otherwise, it returns `IloFalse`.

```
public IloBool canBeSuspendedAtStart() const
```

This member function returns `IloTrue` if the invoking activity can be suspended at its start time. Otherwise, it returns `IloFalse`.

```
public void clearDisjunctiveBreakType() const
```

This member function empties the set of disjunctive break types of the invoking activity.

```
public void clearEndBreakOverlapType() const
```

This member function empties the set of end overlap types of the invoking activity.

```
public void clearIgnoredBreakType() const
```

This member function empties the set of ignored break types of the invoking activity.

```
public void clearIgnoredShiftType() const
```

This member function empties the set of ignored shift types of the invoking activity.

```
public void clearStartBreakOverlapType() const
```

This member function empties the set of start overlap types of the invoking activity.

```
public IloResourceConstraint consumes(const IloCapResource, IloNum cap=1) const
public IloAltResConstraint consumes(const IloAltResSet, const IloNumVar capVar)
const
public IloAltResConstraint consumes(const IloAltResSet, IloNum cap=1) const
public IloResourceConstraint consumes(const IloCapResource, const IloNumVar capVar)
const
```

An activity *consumes* a resource if some amount of the resource capacity must be made available for the execution of the activity and the capacity is non-recoverable after the end of the activity. For example, an activity might consume a raw material in manufacturing a product.

If the resource is discrete (classes `IloDiscreteResource`, `IloReservoir`, `IloDiscreteEnergy`), the activity requires the capacity at all times after the activity's start time. The corresponding member function implies that the occupancy of the resource by the activity is rounded inward toward the nearest valid time that corresponds to a time step.

When the given resource is an instance of `IloDiscreteEnergy`, it means that the source of the energy is consumed (that is, that the given capacity can no longer be provided again after the beginning of the activity).

When the resource is discrete, the following two expressions are equivalent:

- `activity.consumes(resource, capacity);` and
- `activity.requires(resource, capacity, IlcAfterStart);`

If the resource is a continuous reservoir (class `IloContinuousReservoir`), the consumption is continuous and linear from the start time to the end time of the invoking activity. Since the time step of a timetable for a continuous reservoir is 1, the returned resource constraint has no inward/outward rounding mode. Its time extent, which does not match any case of the enumeration `IloTimeExtent`, is not defined either.

If the invoking activity consumes a resource in `set`, the consumption will be discrete if the selected resource is an instance of `IloDiscreteResource`, `IloReservoir`, or `IloDiscreteEnergy`. It will be continuous if the selected resource is an instance of `IloContinuousReservoir`.

An `IloException` is thrown when entering the search if either the capacity is negative, or if capacity is a variable with a negative minimal value.

```
public IloCoverConstraint covers() const
```

This member function builds a cover constraint for the invoking activity. The set of activities to be covered by the invoking activity is initially empty. See the member functions IloCoverConstraint::add and IloCoverConstraint::remove to see how to modify the set of covered activities. The cover constraint states that the start time of the invoking activity is equal to the earliest of the start times of the covered activities, and that the end time of the invoking activity is equal to the latest of the end times of the covered activities.

```
public IloTimeBoundConstraint endsAfter(IloNum time) const
public IloTimeBoundConstraint endsAfter(const IloNumVar time) const
```

This member function states that the invoking activity must end after or at `time`. More formally, `act.endsAfter(time)` means `end(act) >= time`.

The constraint returned by this member function must be added to the model by `IloModel::add` in order to be taken into account during the search for solutions.

```
public IloPrecedenceConstraint endsAfterEnd(const IloActivity act, IloNum delay=0)
const
public IloPrecedenceConstraint endsAfterEnd(const IloActivity act, const IloNumVar
delay) const
```

This member function states that the invoking activity ends after the end of `act`. In addition, at least the given delay must elapse between the end of `act` and the end of the invoking activity.

The member function can be invoked with a negative `delay`, which means that the invoking activity can end before the end of `act`, but the difference between the end time of `act` and the end time of the invoking activity cannot exceed `-delay`.

More formally, `act1.endsAfterEnd(act, delay)` means `end(act1) >= end(act) + delay`.

The constraint returned by this member function must be added to the model by `IloModel::add` in order to be taken into account during the search for solutions.

```
public IloPrecedenceConstraint endsAfterStart(const IloActivity act, IloNum
delay=0) const
public IloPrecedenceConstraint endsAfterStart(const IloActivity act, const
IloNumVar delay) const
```

This member function states that the invoking activity ends after the beginning of `act`. In addition, at least the given `delay` must elapse between the beginning of `act` and the end of the invoking activity.

The member function can be invoked with a negative `delay`, which means that the invoking activity can end before the beginning of `act`, but the difference between the start time of `act` and the end time of the invoking activity cannot exceed `-delay`.

More formally, `act1.endsAfterStart(act, delay)` means `end(act1) >= start(act) + delay`.

The constraint returned by this member function must be added to the model by `IloModel::add` in order to be taken into account during the search for solutions.

```
public IloTimeBoundConstraint endsAt(IloNum time) const
public IloTimeBoundConstraint endsAt(const IloNumVar time) const
```

This member function states that the invoking activity must end at `time`. More formally, `act.endsAt(time)` means `end(act) == time`.

The constraint returned by this member function must be added to the model by `IloModel::add` in order to be taken into account during the search for solutions.

```
public IloPrecedenceConstraint endsAtEnd(const IloActivity act, IloNum delay=0)
const
public IloPrecedenceConstraint endsAtEnd(const IloActivity act, const IloNumVar
delay) const
```

This member function states that exactly the given delay must elapse between the end of `act` and the end of the invoking activity. More formally, `act1.endsAtEnd(act, delay)` means `end(act1) == end(act) + delay`.

The constraint returned by this member function must be added to the model by `IloModel::add` in order to be taken into account during the search for solutions.

```
public IloPrecedenceConstraint endsAtStart(const IloActivity act, IloNum delay=0)
const
public IloPrecedenceConstraint endsAtStart(const IloActivity act, const IloNumVar
delay) const
```

This member function states that exactly the given delay must elapse between the beginning of `act` and the end of the invoking activity. More formally, `act1.endsAtStart(act, delay)` means `end(act1) == start(act) + delay`.

The constraint returned by this member function must be added to the model by `IloModel::add` in order to be taken into account during the search for solutions.

```
public IloTimeBoundConstraint endsBefore(IloNum time) const
public IloTimeBoundConstraint endsBefore(const IloNumVar time) const
```

This member function states that the invoking activity must end before or at `time`. More formally, `act.endsBefore(time)` means `end(act) <= time`.

The constraint returned by this member function must be added to the model by `IloModel::add` in order to be taken into account during the search for solutions.

```
public IloIntExprArg getDurationExpr() const
```

This member function returns an expression that represents the duration of the invoking activity.

```
public IloNum getDurationMax() const
```

This member function returns the maximum duration of the invoking activity.

```
public IloNum getDurationMaxNormalBreaks() const
```

This member function returns the maximal duration above which all breaks are considered as disjunctive. By default, the value of this maximal duration is `IloInfinity`.

```
public IloNum getDurationMin() const
```

This member function returns the minimum duration of the invoking activity.

```
public IloNum getDurationMinNormalBreaks() const
```

This member function returns the threshold duration under which all breaks are considered as disjunctive. By default, the value of this minimal duration is 1 (one) so that only the breaks with null duration are considered as disjunctive.

```
public IloNum getEndBreakOverlapMax() const
```

This member function returns the maximal value of the end break overlap variable of the invoking activity.

```
public IloNum getEndBreakOverlapMin() const
```

This member function returns the minimal value of the end break overlap variable of the invoking activity.

```
public IloIntExprArg getEndExpr() const
```

This member function returns an expression that represents the end time of the invoking activity.

```
public IloNum getEndMax() const
```

This member function returns the latest end time of the invoking activity.

```
public IloNum getEndMin() const
```

This member function returns the earliest end time of the invoking activity.

```
public IloNum getExecutionDurationMin() const
```

A breakable activity executes during a set of disjoint temporal intervals. These execution intervals are separated by intervals that correspond to the breaks that suspend the activity. This member function returns the minimal duration for any execution interval of the invoking activity.

```
public IloNum getExternalValue() const
```

This member function returns the current value of the external variable of the invoking activity. It raises an error in case the external variable is not bound.

```
public IloNumVar getExternalVariable() const
```

This member function returns the external variable of the invoking activity. By default, the external variable of an activity is a variable with a domain `[IloIntMin, IloIntMax]`.

```
public IloActivityI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloNum getProcessingTimeMax() const
```

This member function returns the maximum processing time of the invoking activity.

```
public IloNum getProcessingTimeMin() const
```

This member function returns the minimum processing time of the invoking activity.

```
public IloNumVar getProcessingTimeVariable() const
```

This member function returns the variable that represents the processing time for the invoking activity.

```
public IloNum getStartBreakOverlapMax() const
```

This member function returns the maximal value of the start break overlap variable of the invoking activity.

```
public IloNum getStartBreakOverlapMin() const
```

This member function returns the minimal value of the start break overlap variable of the invoking activity.

```
public IloIntExprArg getStartExpr() const
```

This member function returns an expression that represents the start time of the invoking activity.

```
public IloNum getStartMax() const
```

This member function returns the latest start time of the invoking activity.

```
public IloNum getStartMin() const
```

This member function returns the earliest start time of the invoking activity.

```
public IloInt getTransitionType() const
```

The transition type of an activity is an integer intended to define transition time and cost from an indexed classification of activities.

```
public IloBool hasDisjunctiveBreakType() const
```

This member function returns `IloTrue` if the set of disjunctive break types is not empty.

```
public IloBool hasEndBreakOverlapType() const
```

This member function returns `IloTrue` if the set of end break overlap types is not empty.

```
public IloBool hasIgnoredBreakType() const
```

This member function returns `IloTrue` if the set of ignored break types is not empty.

```
public IloBool hasIgnoredShiftType() const
```

This member function returns `IloTrue` if the set of ignored shift types is not empty.

```
public IloBool hasStartBreakOverlapType() const
```

This member function returns `IloTrue` if the set of start break overlap types is not empty.

```
public void ignoreBreakDisjunctivity(IloBool ignored=IloTrue) const
```

When the argument `ignored` is equal to `IloTrue`, this member function inhibits the disjunctive breaks for the invoking activity. That is, all breaks will be treated as normal, non-disjunctive breaks. Otherwise, the disjunctive breaks are taken into account for the invoking activity.

This member function has no effect if at the beginning of the search the invoking activity is not breakable.

```
public void ignoreCoverConstraints(IloBool ignored=IloTrue) const
```

When the argument `ignored` is equal to `IloTrue`, this member function inhibits the cover constraints defined on the invoking activity. Otherwise, and by default, the cover constraints defined on the invoking activity are taken into account.

```
public void ignorePrecedenceConstraints(IloBool ignored=IloTrue) const
```

When the argument `ignored` is equal to `IloTrue`, this member function inhibits the precedence constraints defined on the invoking activity. Otherwise the precedence constraints defined on the invoking activity are taken into account.

```
public void ignoreResourceConstraints(IloBool ignored=IloTrue) const
```

When the argument `ignored` is equal to `IloTrue`, this member function inhibits the resource constraints defined on the invoking activity. When the argument `ignored` is equal to `IloFalse`, this member function takes into account the resource constraints defined on the invoking activity.

```
public void ignoreTimeBoundConstraints(IloBool ignored=IloTrue) const
```

When the argument `ignored` is equal to `IloTrue`, this member function inhibits the time-bound constraints defined on the invoking activity. Otherwise the time-bound constraints defined on the invoking activity are taken into account.

```
public IloBool isBreakable() const
```

This member function returns `IloTrue` if the invoking activity is breakable. Otherwise, it returns `IloFalse`.

```
public IloBool isBreakDisjunctivityIgnored() const
```

This member function returns `IloTrue` if the break disjunctivity is ignored for the invoking activity. Otherwise, it returns `IloFalse`.

```
public IloBool isDisjunctiveBreakType(IloInt type) const
```

This member function returns `IloTrue` if the type `type` belongs to the set of disjunctive break types of the invoking activity. Otherwise, it returns `IloFalse`.

```
public IloBool isEndBreakOverlapType(IloInt type) const
```

This member function returns `IloTrue` if the type `type` belongs to the set of end overlap types of the invoking activity. Otherwise, it returns `IloFalse`.

```
public IloBool isIgnoredBreakType(IloInt type) const
```

This member function returns `IloTrue` if the type `type` belongs to the set of ignored break types of the invoking activity. Otherwise, it returns `IloFalse`.

```
public IloBool isIgnoredShiftType(IloInt type) const
```

This member function returns `IloTrue` if the type `type` belongs to the set of ignored shift types of the invoking activity. Otherwise, it returns `IloFalse`.

```
public IloBool isStartBreakOverlapType(IloInt type) const
```

This member function returns `IloTrue` if the type `type` belongs to the set of start overlap types of the invoking activity. Otherwise, it returns `IloFalse`.

```
public IloResourceConstraint produces(const IloReservoir, IloNum cap=1) const
```

An activity produces if some amount of the capacity of the (discrete or continuous) reservoir is definitely made available through the execution of the activity. This member function states that the invoking activity produces the given capacity for the given reservoir.

If the reservoir is discrete (class `IloReservoir`), this member function implies that the occupancy of the reservoir by the activity is rounded inward toward the nearest valid time that corresponds to a time step. The following two expressions are equivalent:

- `activity.produces(reservoir, capacity);`
- `activity.provides(reservoir, capacity, IloAfterEnd);`

If the reservoir is continuous (class `IloContinuousReservoir`), the production process is continuous and linear from the start time to the end time of the invoking activity. Since the time step of a timetable for a continuous reservoir is 1, the returned resource constraint has no inward/outward rounding mode. Its time extent, which does not match any case of the enumeration `IloTimeExtent`, is not defined either.

If the invoking activity produces for a reservoir in `set`, the production will be discrete if the selected reservoir is an instance of `IloReservoir`. It will be continuous if the selected reservoir is an instance of `IloContinuousReservoir`.

An `IloException` is thrown when entering the search if either the capacity is negative, or if capacity is a variable with a negative minimal value.

```
public IloResourceConstraint provides(const IloReservoir, IloNum cap=1,
IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloFalse) const
public IloAltResConstraint provides(const IloAltResSet, const IloNumVar capVar,
IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloFalse) const
public IloAltResConstraint provides(const IloAltResSet, IloNum cap=1, IloTimeExtent
extent=IloFromStartToEnd, IloBool outward=IloFalse) const
public IloResourceConstraint provides(const IloReservoir, const IloNumVar capVar,
IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloFalse) const
```

This member function states that the invoking activity provides the given capacity of the given reservoir. By default, the activity provides the reservoir from the beginning to the end of its execution. However, the optional argument `extent` is available to represent cases in which the activity provides the reservoir over a different time extent, as explained in `IloTimeExtent`.

The argument `outward` is important only when one of the resource usage enforcement intervals of the reservoir has a time step greater than 1 (one). In that case, `outward` defines whether the occupancy of the reservoir by the activity should be rounded outward or inward towards the nearest valid time that corresponds to a step. By default, `outward` is considered to be false for a providing resource constraint.

An `IloException` is thrown when entering the search if either the capacity is negative, or if capacity is a variable with a negative minimal value.

The set of alternative resources can contain a continuous reservoir if the time extent is `IloNever`, `IloAlways`, `IloAfterStart` or `IloAfterEnd`. If the time extent is `IloNever`, the activity does not provide any capacity. If the time extent is `IloAlways`, the capacity is provided at any time. If the time extent is `IloAfterStart` or `IloAfterEnd`, the capacity is not provided before the start of the activity, is totally provided after its end and linearly provided between its start and its end.

```
public void removeDisjunctiveBreakType(const IloIntSet types) const
```

This member function removes the set of types `types` from the set of disjunctive break types of the invoking activity. If a break type belongs to the set of disjunctive break types of a breakable activity, the activity must be completely processed either before or after that type of break.

```
public void removeDisjunctiveBreakType(IloInt type) const
```

This member function removes the type `type` from the set of disjunctive break types of the invoking activity. If a break type belongs to the set of disjunctive break types of a breakable activity, the activity must be completely processed either before or after that type of break.

```
public void removeEndBreakOverlapType(const IloIntSet types) const
```

For each activity, a set of break types is given that defines which break types can possibly be overlapped by the end of the activity. This member function removes all the types of `types` from this set of break types on the invoking activity.

```
public void removeEndBreakOverlapType(IloInt type) const
```

For each activity, a set of break types is given that defines which break types can possibly be overlapped by the end of the activity. This member function removes `type` from this set of break types on the invoking activity.

```
public void removeIgnoredBreakType(const IloIntSet types) const
```

This member function removes the set of types `types` from the set of ignored break types of the invoking activity.

```
public void removeIgnoredBreakType(IloInt type) const
```

This member function removes the type `type` from the set of ignored break types of the invoking activity.

```
public void removeIgnoredShiftType(const IloIntSet types) const
```

This member function removes the set of types `types` from the set of ignored shift types of the invoking activity.

```
public void removeIgnoredShiftType(IloInt type) const
```

This member function removes the type `type` from the set of ignored shift types of the invoking activity.

```
public void removeStartBreakOverlapType(const IloIntSet types) const
```

For each activity, a set of break types is given that defines which break types can possibly be overlapped by the start of the activity. This member function removes all the types of `types` from this set of break types on the invoking activity.

```
public void removeStartBreakOverlapType(IloInt type) const
```

For each activity, a set of break types is given that defines which break types can possibly be overlapped by the start of the activity. This member function removes the type `type` from this set of break types on the invoking activity.

```
public IloResourceConstraint requires(const IloStateResource, const IloAnySet
states, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
public IloResourceConstraint requires(const IloStateResource, const IloAnySetVar
states, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
```

This member function states that the invoking activity requires the given resource in one of the given set of states. The state may change during execution, but must remain in the given set of states. By default, the activity requires the resource from the beginning to the end of its execution. However, the optional argument `extent` is available to represent cases in which the activity requires the resource over a different time extent, as explained in `IloTimeExtent`.

The argument `outward` is important only when one of the resource usage enforcement intervals of the resource has a time step greater than 1 (one). In that case, `outward` defines whether the occupancy of the resource by the activity should be rounded outward or inward towards the nearest valid time that corresponds to a step. By default, `outward` is considered to be true for a requirement.

```
public IloResourceConstraint requires(const IloStateResource, IloAny state,
IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
public IloResourceConstraint requires(const IloStateResource, const IloAnyVar
state, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
```

This member function states that the invoking activity requires the given resource in the given state. By default, the activity requires the resource from the beginning to the end of its execution. However, the optional argument `extent` is available to represent cases in which the activity requires the resource over a different time extent, as explained in `IloTimeExtent`.

The argument `outward` is important only when one of the resource usage enforcement intervals of the resource has a time step greater than 1 (one). In that case, `outward` defines whether the occupancy of the resource by the activity should be rounded outward or inward towards the nearest valid time that corresponds to a step. By default, `outward` is considered to be true for a requirement.

```
public IloResourceConstraint requires(const IloCapResource, IloNum cap=1,
IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
public IloAltResConstraint requires(const IloAltResSet, const IloNumVar capVar,
IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
public IloAltResConstraint requires(const IloAltResSet, IloNum cap=1, IloTimeExtent
extent=IloFromStartToEnd, IloBool outward=IloTrue) const
public IloResourceConstraint requires(const IloCapResource, const IloNumVar capVar,
IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
```

This member function states that the invoking activity requires the given capacity of the given resource. For example, an activity might require the presence of a worker on a shift. By default, the activity requires the resource from the beginning to the end of its execution. However, the optional argument `extent` is available to represent cases in which the activity requires the resource over a different time extent, as explained in `IloTimeExtent`.

The argument `outward` is important only when one of the resource usage enforcement intervals of the resource has a time step greater than 1 (one). In that case, `outward` defines whether the occupancy of the resource by the activity should be rounded outward or inward towards the nearest valid time that corresponds to a step. When the given resource is an instance of `IloDiscreteEnergy`, it means that the given capacity is required for each unit of time in the given time extent. By default, `outward` is considered to be true for a requirement.

An `IloException` is thrown when entering the search if either the capacity is negative, or if capacity is a variable with a negative minimal value.

The member function must not be called with a capacity resource that is a continuous reservoir; however, the set of alternative resources can contain a continuous reservoir if time extent is `IloNever`, `IloAlways`, `IloAfterStart` or `IloAfterEnd`. If the time extent is `IloNever`, the activity does not require any capacity. If the time extent is `IloAlways`, the capacity is required at any time. If the time extent is `IloAfterStart` or `IloAfterEnd`, the capacity is not required before the start of the activity, is totally required after its end and linearly required between its start and its end.

```
public IloResourceConstraint requiresNot(const IloStateResource, const IloAnySet
states, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
public IloResourceConstraint requiresNot(const IloStateResource, const IloAnySetVar
states, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
```

This member function states that the invoking activity requires the given resource in any state that does not belong to the given set of states. The state may change during execution, but must never belong to the given set of states.

By default, the activity requires the resource from the beginning to the end of its execution. However, the optional argument `extent` is available to represent cases in which the activity requires the resource over a different time extent, as explained in `IloTimeExtent`.

The argument `outward` is important only when one of the resource usage enforcement intervals of the resource has a time step greater than 1 (one). In that case, `outward` defines whether the occupancy of the resource by the activity should be rounded outward or inward towards the nearest valid time that corresponds to a step. By default, `outward` is considered to be true for a requirement.

```
public IloResourceConstraint requiresNot(const IloStateResource, IloAny state,
IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
public IloResourceConstraint requiresNot(const IloStateResource, const IloAnyVar
state, IloTimeExtent extent=IloFromStartToEnd, IloBool outward=IloTrue) const
```

This member function states that the invoking activity requires the given resource in any state different from the given state. The state may change during execution, but must never be the given state.

By default, the activity requires the resource from the beginning to the end of its execution. However, the optional argument `extent` is available to represent cases in which the activity requires the resource over a different time extent, as explained in `IloTimeExtent`.

The argument `outward` is important only when one of the resource usage enforcement intervals of the resource has a time step greater than 1 (one). In that case, `outward` defines whether the occupancy of the resource by the activity should be rounded outward or inward towards the nearest valid time that corresponds to a step. By default, `outward` is considered to be true for a requirement.

```
public void setActivityBasicParam(const IloActivityBasicParam param) const
```

This member function sets `param` as the new basic activity parameter.

```
public void setActivityBreakParam(const IloActivityBreakParam param) const
```

This member function sets `param` as the new break activity parameter.

```
public void setActivityConstraintsParam(const IloActivityConstraintsParam param)
const
```

This member function sets `param` as the new constraint activity parameter.

```
public void setActivityOverlapParam(const IloActivityOverlapParam param) const
```

This member function sets `param` as the new overlap activity parameter.

```
public void setActivityShiftParam(const IloActivityShiftParam param) const
```

This member function sets `param` as the new shift activity parameter.

```
public void setBreakable(IloBool breakable=IloTrue) const
```

When the argument `breakable` is equal to `IloTrue`, this member function allows the invoking activity to be breakable. Otherwise, the invoking activity is not breakable.

```
public void setCanBeSuspendedAtEnd(IloBool val=IloTrue) const
```

If `val` is equal to `IloTrue`, this member function permits the invoking activity to be suspended at its end time.

This member function has no effect if at the beginning of the search the invoking activity is not breakable.

```
public void setCanBeSuspendedAtStart(IloBool val=IloTrue) const
```

If `val` is equal to `IloTrue`, this member function permits the invoking activity to be suspended at its start time. Otherwise, the invoking activity cannot be suspended at its start time.

This member function has no effect if at the beginning of the search the invoking activity is not breakable.

```
public void setDurationMax(IloNum duration)
```

This member function states that the duration of the invoking activity can be at most `duration`.

```
public void setDurationMaxNormalBreaks(IloNum max) const
```

This member function states that the invoking activity must be completely processed either before or after any break whose duration is strictly greater than `max`. By default, the value of this duration for a breakable activity is `IloInfinity`.

```
public void setDurationMin(IloNum duration)
```

This member function states that the duration of the invoking activity must be at least `duration`.

```
public void setDurationMinNormalBreaks(IloNum min) const
```

This member function states that the invoking activity must be completely processed either before or after any break whose duration is strictly lower than `min`.

By default, the value of this duration for a breakable activity is `1` (one) so that only the breaks with null duration are considered as disjunctive.

```
public void setEndBreakOverlapMax(IloNum max) const
```

This member function sets `max` as the new maximal value of the end break overlap duration.

```
public void setEndBreakOverlapMin(IloNum min) const
```

This member function sets `min` as the new minimal value of the end break overlap duration.

```
public void setEndMax(IloNum endMax) const
```

This member function states that the invoking activity must not end after `endMax`.

```
public void setEndMin(IloNum endMin) const
```

This member function states that the invoking activity must not end before `endMin`.

```
public void setExecutionDurationMin(IloNum min) const
```

A breakable activity executes during a set of disjoint temporal intervals. These execution intervals are separated by intervals that correspond to the breaks that suspend the activity.

This member function states that the duration of the temporal intervals during which the invoking breakable activity executes must each be greater or equal to `min`. Note that `min` must be a strictly positive integer. By default, breakable activities are created with a minimal duration for execution intervals of 1 (one).

```
public void setExternalValue(IloNum val)
```

This member function sets `val` as the value of the external variable of the invoking activity.

```
public void setExternalVariable(IloNumVar var)
```

This member function sets `var` as the external variable of the invoking activity.

```
public void setProcessingTimeMax(IloNum processingTime) const
```

This member function states that the processing time of the invoking activity can be at most `processingTime`.

```
public void setProcessingTimeMin(IloNum processingTime) const
```

This member function states that the processing time of the invoking activity must be at least `processingTime`.

```
public void setStartBreakOverlapMax(IloNum max) const
```

This member function sets `max` as the new maximal value of the start break overlap duration.

```
public void setStartBreakOverlapMin(IloNum min) const
```

This member function sets `min` as the new minimal value of the start break overlap duration.

```
public void setStartMax(IloNum startMax) const
```

This member function states that the invoking activity must not start after `startMax`.

```
public void setStartMin(IloNum startMin) const
```

This member function states that the invoking activity must not start before `startMin`.

```
public void setTransitionType(IloInt type) const
```

The transition type of an activity is an integer intended to define transition time and cost from an indexed classification of activities. It is used by transition parameters (instances of the class `IloTransitionParam`).

```
public void setUseEfficiency(IloBool useEfficiency=IloTrue) const
```

When the argument `useEfficiency` is equal to `IloTrue`, the processing time of the invoking activity is computed using the efficiency function of resource calendars.

Note that if no calendar with efficiency function is attached to resources or resource constraints required by the invoking activity, the processing time remains unbounded.

```
public void shareEndWithEnd(IloActivity activity)
```

This member function states that the invoking activity shares its end with the end of the `activity` provided as argument.

```
public void shareEndWithStart(IloActivity activity)
```

This member function states that the invoking activity shares its end with the start of the `activity` provided as argument.

```
public void shareStartWithEnd(IloActivity activity)
```

This member function states that the invoking activity shares its start with the end of the `activity` provided as argument.

```
public void shareStartWithStart(IloActivity activity)
```

This member function states that the invoking activity shares its start with the start of the `activity` provided as argument.

```
public IloTimeBoundConstraint startsAfter(IloNum time) const
public IloTimeBoundConstraint startsAfter(const IloNumVar time) const
```

This member function states that the invoking activity must start after or at `time`. More formally, `act.startsAfter(time)` means `start(act) >= time`.

The constraint returned by this member function must be added to the model by `IloModel::add` in order to be taken into account during the search for solutions.

```
public IloPrecedenceConstraint startsAfterEnd(const IloActivity act, IloNum
delay=0) const
public IloPrecedenceConstraint startsAfterEnd(const IloActivity act, const
IloNumVar delay) const
```

This member function states that the invoking activity starts after the end of `act`. (In other words, `act` precedes the invoking activity.) In addition, at least the given `delay` must elapse between the end of `act` and the beginning of the invoking activity.

The member function can be invoked with a negative `delay`, which means that the invoking activity can start before the end of `act`, but the difference between the end time of `act` and the start time of the invoking activity cannot exceed `-delay`.

More formally, `act1.startsAfterEnd(act, delay)` means `start(act1) >= end(act) + delay`.

The constraint returned by this member function must be added to the model by `IloModel::add` in order to be taken into account during the search for solutions.

```
public IloPrecedenceConstraint startsAfterStart(const IloActivity act, IloNum
delay=0) const
```

```
public IloPrecedenceConstraint startsAfterStart(const IloActivity act, const
IloNumVar delay) const
```

This member function states that the invoking activity starts after the beginning of `act`. In addition, at least the given `delay` must elapse between the beginning of `act` and the beginning of the invoking activity.

The member function can be invoked with a negative `delay`, which means that the invoking activity can start before the beginning of `act`, but the difference between the start time of `act` and the start time of the invoking activity cannot exceed `-delay`.

More formally, `act1.startsAfterStart(act, delay)` means `start(act1) >= start(act) + delay`.

The constraint returned by this member function must be added to the model by `IloModel::add` in order to be taken into account during the search for solutions.

```
public IloTimeBoundConstraint startsAt(IloNum time) const
public IloTimeBoundConstraint startsAt(const IloNumVar time) const
```

This member function states that the invoking activity must start at `time`. More formally, `act.startsAt(time)` means `start(act) == time`.

The constraint returned by this member function must be added to the model by `IloModel::add` in order to be taken into account during the search for solutions.

```
public IloPrecedenceConstraint startsAtEnd(const IloActivity act, IloNum delay=0)
const
public IloPrecedenceConstraint startsAtEnd(const IloActivity act, const IloNumVar
delay) const
```

This member function states that exactly the given delay must elapse between the end of `act` and the beginning of the invoking activity.

More formally, `act1.startsAtEnd(act, delay)` means `start(act1) == end(act) + delay`.

The constraint returned by this member function must be added to the model by `IloModel::add` in order to be taken into account during the search for solutions.

```
public IloPrecedenceConstraint startsAtStart(const IloActivity act, IloNum delay=0)
const
public IloPrecedenceConstraint startsAtStart(const IloActivity act, const IloNumVar
delay) const
```

This member function states that exactly the given `delay` must elapse between the beginning of `act` and the beginning of the invoking activity.

More formally, `act1.startsAtStart(act, delay)` means `start(act1) == start(act) + delay`.

The constraint returned by this member function must be added to the model by `IloModel::add` in order to be taken into account during the search for solutions.

```
public IloTimeBoundConstraint startsBefore(IloNum time) const
public IloTimeBoundConstraint startsBefore(const IloNumVar time) const
```

This member function states that the invoking activity must start before or at `time`. More formally, `act.startsBefore(time)` means `start(act) <= time`.

The constraint returned by this member function must be added to the model by `IloModel::add` in order to be taken into account during the search for solutions.

```
public void unshare()
```

This member function states that the invoking activity does not share its start nor its end.

```
public IloBool useEfficiency() const
```

This member function returns `IloTrue` if the processing time of the invoking activity is computed using the efficiency function of resource calendars. Otherwise, it returns `IloFalse`.

Class IloActivityBasicParam

Definition file: ilsched/iloactivityparam.h
Include file: <ilsched/iloscheduler.h>

Parameters are used to change the default behavior of activities and resources. By default, an activity is not breakable, does not consider efficiency function of potential calendars, and no interval is ignored. With `IloActivityBasicParam`, it is (for example) possible to specify if an activity is breakable, or if an activity must take into account efficiency functions. `IloActivityBasicParam` also allows ignoring sets of interval types, such as breaks and shifts.

This class inherits from the IBM® ILOG® Concert Technology class `IloExtractable`. That class is documented in the *IBM ILOG Concert Technology Reference Manual*.

For more information, see Calendars, and Parameter Classes.

See Also: IloActivity, IloSchedulerEnv

Constructor Summary	
public	`IloActivityBasicParam()`
public	`IloActivityBasicParam(IloActivityBasicParamI * impl)`
public	`IloActivityBasicParam(const IloEnv env, const char * name=0)`

Method Summary	
public void	`addIgnoredBreakType(const IloIntSet types) const`
public void	`addIgnoredBreakType(IloInt type) const`
public void	`clearIgnoredBreakType() const`
public IloActivityBasicParamI *	`getImpl() const`
public IloBool	`hasIgnoredBreakType() const`
public IloBool	`isBreakable() const`
public IloBool	`isIgnoredBreakType(IloInt type) const`
public void	`removeIgnoredBreakType(const IloIntSet types) const`
public void	`removeIgnoredBreakType(IloInt type) const`
public void	`setBreakable(IloBool breakable=IloTrue) const`
public void	`setUseEfficiency(IloBool useEfficiency=IloTrue) const`
public IloBool	`useEfficiency() const`

Constructors

`public` **`IloActivityBasicParam`**`()`

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

```
public IloActivityBasicParam(IloActivityBasicParamI * impl)
```

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

```
public IloActivityBasicParam(const IloEnv env, const char * name=0)
```

This constructor creates an instance of `IloActivityBasicParam` with the default values that an activity is not breakable, does not use efficiency, and there are no ignored breaks or shifts.

Methods

```
public void addIgnoredBreakType(const IloIntSet types) const
```

This member function adds the set of types `types` to the set of ignored break types of the invoking parameter.

```
public void addIgnoredBreakType(IloInt type) const
```

This member function adds the type `type` to the set of ignored break types of the invoking parameter.

```
public void clearIgnoredBreakType() const
```

This member function empties the set of ignored break types of the invoking parameter.

```
public IloActivityBasicParamI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloBool hasIgnoredBreakType() const
```

This member function returns `IloTrue` if the set of ignored break types is not empty.

```
public IloBool isBreakable() const
```

This member function returns `IloTrue` if the activities depending on this parameter are breakable. Otherwise, it returns `IloFalse`.

```
public IloBool isIgnoredBreakType(IloInt type) const
```

This member function returns `IloTrue` if the type `type` belongs to the set of ignored break types of the invoking parameter. Otherwise, it returns `IloFalse`.

```
public void removeIgnoredBreakType(const IloIntSet types) const
```

This member function removes the set of types `types` from the set of ignored break types of the invoking parameter.

```
public void removeIgnoredBreakType(IloInt type) const
```

This member function removes the type `type` from the set of ignored break types of the invoking parameter.

```
public void setBreakable(IloBool breakable=IloTrue) const
```

When `breakable` is equal to `IloTrue`, this member function states that activities depending on this parameter are breakable. Otherwise, they are not breakable.

```
public void setUseEfficiency(IloBool useEfficiency=IloTrue) const
```

When the argument `useEfficiency` is equal to `IloTrue`, the processing time is computed using the efficiency function of resource calendars.

Note that if no calendar with an efficiency function is attached to resources or to resource constraints required by activities used by the invoking parameter, the processing time remains unbounded.

```
public IloBool useEfficiency() const
```

This member function returns `IloTrue` if the processing time is computed using the efficiency function of resource calendars. Otherwise, it returns `IloFalse`.

Class IloActivityBreakParam

Definition file: ilsched/iloactivityparam.h
Include file: <ilsched/iloscheduler.h>

Parameters are used to change the default behavior of activities and resources. By default when an activity is breakable, the minimum duration of its execution time is equal to one, only null duration breaks are considered as disjunctive, and the breakable activity cannot be suspended at its start or end time. Instances of `IloActivityBreakParam` are used to change these characteristics in order to control the way breakable activities are executed.

It is possible to specify the minimal duration of a breakable activity executed in several parts, the ability of a breakable activity to be suspended at its start or end time, and to define the set of disjunctive breaks for activities.

This parameter has no effect on activities that are not breakable.

For more information, see Calendars and Parameter Classes.

See Also: IloActivity, IloSchedulerEnv

Constructor Summary	
public	IloActivityBreakParam()
public	IloActivityBreakParam(IloActivityBreakParamI * impl)
public	IloActivityBreakParam(const IloEnv env, const char * name=0)

Method Summary	
public void	addDisjunctiveBreakType(const IloIntSet types) const
public void	addDisjunctiveBreakType(IloInt type) const
public IloBool	canBeSuspendedAtEnd() const
public IloBool	canBeSuspendedAtStart() const
public void	clearDisjunctiveBreakType() const
public IloNum	getDurationMaxNormalBreaks() const
public IloNum	getDurationMinNormalBreaks() const
public IloNum	getExecutionDurationMin() const
public IloActivityBreakParamI *	getImpl() const
public IloBool	hasDisjunctiveBreakType() const
public IloBool	isDisjunctiveBreakType(IloInt type) const
public void	removeDisjunctiveBreakType(const IloIntSet types) const
public void	removeDisjunctiveBreakType(IloInt type) const
public void	setCanBeSuspendedAtEnd(IloBool susp=IloTrue) const
public void	setCanBeSuspendedAtStart(IloBool susp=IloTrue) const

public void	setDurationMaxNormalBreaks(IloNum maxDurNormBreaks) const
public void	setDurationMinNormalBreaks(IloNum minDurNormBreaks) const
public void	setExecutionDurationMin(IloNum minExecutionTime) const

Constructors

public **IloActivityBreakParam**()

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

public **IloActivityBreakParam**(IloActivityBreakParamI * impl)

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

public **IloActivityBreakParam**(const IloEnv env, const char * name=0)

This constructor creates an instance of `IloActivityBreakParam` with the default values that the duration of its execution time is equal to one, no breaks are considered as disjunctive, and breakable activities cannot be suspended at their start and end times.

Methods

public void **addDisjunctiveBreakType**(const IloIntSet types) const

This member function adds the set of types `types` to the set of disjunctive break types of the invoking parameter.

public void **addDisjunctiveBreakType**(IloInt type) const

This member function adds the type `type` to the set of disjunctive break types of the invoking parameter.

public IloBool **canBeSuspendedAtEnd**() const

This member function returns `IloTrue` if the activity using the invoking parameter can be suspended at its end time. Otherwise, it returns `IloFalse`.

public IloBool **canBeSuspendedAtStart**() const

This member function returns `IloTrue` if the activity using the invoking parameter can be suspended at its start time. Otherwise, it returns `IloFalse`.

public void **clearDisjunctiveBreakType**() const

This member function empties the set of disjunctive break types of activity using the invoking parameter.

```
public IloNum getDurationMaxNormalBreaks() const
```

This member function returns the threshold duration above which all breaks are considered as disjunctive.

```
public IloNum getDurationMinNormalBreaks() const
```

This member function returns the threshold duration under which all breaks are considered as disjunctive. By default, the value of this minimal duration is 1 (one) so that only the breaks with null duration are considered as disjunctive.

```
public IloNum getExecutionDurationMin() const
```

A breakable activity may execute during a set of disjoint temporal intervals. These execution intervals are separated by intervals that correspond to the breaks that suspend the activity. This member function returns the minimal duration for the execution intervals of the activities using the invoking parameter.

The default minimal duration is 1 (one). It can be redefined by calling the member function `setExecutionDurationMin`.

```
public IloActivityBreakParamI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloBool hasDisjunctiveBreakType() const
```

This member function returns `IloTrue` if the set of disjunctive break types of the invoking parameter is not empty. Otherwise, it returns `IloFalse`.

```
public IloBool isDisjunctiveBreakType(IloInt type) const
```

This member function returns `IloTrue` if the type `type` belongs to the set of disjunctive break types of the invoking activity. Otherwise, it returns `IloFalse`.

```
public void removeDisjunctiveBreakType(const IloIntSet types) const
```

This member function removes the set of types `types` from the set of disjunctive break types of the invoking parameter.

```
public void removeDisjunctiveBreakType(IloInt type) const
```

This member function removes `type` from the set of disjunctive break types of the invoking parameter.

```
public void setCanBeSuspendedAtEnd(IloBool susp=IloTrue) const
```

If `susp` is equal to `IloTrue`, this member function permits the activity using the invoking parameter to be suspended at its end time. Otherwise, these activities cannot be suspended at its end time.

```
public void setCanBeSuspendedAtStart(IloBool susp=IloTrue) const
```

If `susp` is equal to `IloTrue`, this member function permits the activity using the invoking parameter to be suspended at its start time. Otherwise, these activities cannot be suspended at its start time.

```
public void setDurationMaxNormalBreaks(IloNum maxDurNormBreaks) const
```

This member function states that the invoking activity must be completely processed either before or after any break whose duration is strictly greater than `maxDurNormBreaks`.

Increasing this maximal duration has no effect in search mode.

```
public void setDurationMinNormalBreaks(IloNum minDurNormBreaks) const
```

This member function states that the invoking activity must be completely processed either before or after any break whose duration is strictly lower than `minDurNormBreaks`.

By default, the value of this minimal duration for a breakable activity is 1 (one) so that only the breaks with null duration are considered disjunctive.

Decreasing this minimal duration has no effect in search mode.

```
public void setExecutionDurationMin(IloNum minExecutionTime) const
```

A breakable activity may execute during a set of disjoint temporal intervals. These execution intervals are separated by intervals that correspond to the breaks that suspend the activity.

This member function states that the duration of the temporal intervals during which the invoking breakable activity executes must all be greater or equal to `minExecutionTime`. Note that `minExecutionTime` must be a strictly positive integer. By default, the minimal duration for execution intervals is 1.

Decreasing the minimal duration for execution intervals has no effect in search mode.

Class IloActivityConstraintsParam

Definition file: ilsched/iloactivityparam.h
Include file: <ilsched/iloscheduler.h>

Parameters are used to change the default behavior of activities and resources. By default, all constraints on activities are taken into account. Instances of `IloActivityConstraintsParam` are used to change the default behaviour. It is possible to activate or deactivate some kinds of constraints, such as the precedence constraints, the time-bound constraints, the resource constraints, and the calendar constraints. If the activity is breakable, it is also possible to activate or deactivate disjunctive breaks.

For more information, see Calendars, Temporal Relations, and Parameter Classes.

See Also: IloActivity, IloSchedulerEnv, IloPrecedenceConstraint, IloResourceConstraint, IloTimeBoundConstraint

Constructor Summary	
public	`IloActivityConstraintsParam()`
public	`IloActivityConstraintsParam(IloActivityConstraintsParamI * impl)`
public	`IloActivityConstraintsParam(const IloEnv env, const char * name=0)`

Method Summary	
public IloBool	`areCoverConstraintsIgnored() const`
public IloBool	`arePrecedenceConstraintsIgnored() const`
public IloBool	`areResourceConstraintsIgnored() const`
public IloBool	`areShiftConstraintsIgnored() const`
public IloBool	`areTimeBoundConstraintsIgnored() const`
public IloActivityConstraintsParamI *	`getImpl() const`
public void	`ignoreBreakDisjunctivity(IloBool ignored=IloTrue) const`
public void	`ignoreCoverConstraints(IloBool ignored=IloTrue) const`
public void	`ignorePrecedenceConstraints(IloBool ignored=IloTrue) const`
public void	`ignoreResourceConstraints(IloBool ignored=IloTrue) const`
public void	`ignoreShiftConstraints(IloBool ignored=IloTrue) const`
public void	`ignoreTimeBoundConstraints(IloBool ignored=IloTrue) const`
public IloBool	`isBreakDisjunctivityIgnored() const`

Constructors

`public` **`IloActivityConstraintsParam`**`()`

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

```
public IloActivityConstraintsParam(IloActivityConstraintsParamI * impl)
```

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

```
public IloActivityConstraintsParam(const IloEnv env, const char * name=0)
```

This constructor creates a new instance of IloActivityConstraintsParam, with the default value that the following constraints are all activated: precedence constraints, time-bound constraints, resource constraints, and calendar constraints. Disjunctive breaks are also taken into account for breakable activities.

Methods

```
public IloBool areCoverConstraintsIgnored() const
```

This member function returns `IloTrue` if the cover constraints are not taken into account for the invoking parameter. Otherwise, it returns `IloFalse`.

```
public IloBool arePrecedenceConstraintsIgnored() const
```

This member function returns `IloTrue` if the precedence constraints are not taken into account for the invoking parameter. Otherwise, it returns `IloFalse`.

```
public IloBool areResourceConstraintsIgnored() const
```

This member function returns `IloTrue` if resource constraints are not taken into account for the invoking parameter. Otherwise, it returns `IloFalse`.

```
public IloBool areShiftConstraintsIgnored() const
```

This member function returns `IloTrue` if shifts of calendar constraints are not taken into account for the invoking parameter. Otherwise, it returns `IloFalse`.

```
public IloBool areTimeBoundConstraintsIgnored() const
```

This member function returns `IloTrue` if time-bound constraints are not taken into account for the invoking parameter. Otherwise, it returns `IloFalse`.

```
public IloActivityConstraintsParamI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public void ignoreBreakDisjunctivity(IloBool ignored=IloTrue) const
```

When the argument `ignored` is equal to `IloTrue`, this member function causes the invoking parameter to ignore the disjunctive breaks. Otherwise, the disjunctive constraints are taken into account.

```
public void ignoreCoverConstraints(IloBool ignored=IloTrue) const
```

When the argument `ignored` is equal to `IloTrue`, this member function causes the invoking parameter to ignore the cover constraints. Otherwise, and by default, the cover constraints are taken into account.

```
public void ignorePrecedenceConstraints(IloBool ignored=IloTrue) const
```

When the argument `ignored` is equal to `IloTrue`, this member function causes the invoking parameter to ignore the precedence constraints. Otherwise, the precedence constraints are taken into account.

```
public void ignoreResourceConstraints(IloBool ignored=IloTrue) const
```

When the argument `ignored` is equal to `IloTrue`, this member function causes the invoking parameter to ignore the resource constraints. Otherwise, the resource constraints are taken into account.

```
public void ignoreShiftConstraints(IloBool ignored=IloTrue) const
```

When the argument `ignored` is equal to `IloTrue`, this member function causes the invoking parameter to ignore the shift part of the calendar constraints. Otherwise, the shifts are taken into account.

```
public void ignoreTimeBoundConstraints(IloBool ignored=IloTrue) const
```

When the argument `ignored` is equal to `IloTrue`, this member function causes the invoking parameter to ignore the time-bound constraints. Otherwise, the constraints are taken into account.

```
public IloBool isBreakDisjunctivityIgnored() const
```

This member function returns `IloTrue` if disjunctive breaks are not taken into account for the invoking parameter. Otherwise, it returns `IloFalse`.

Class IloActivityOverlapParam

Definition file: ilsched/iloactivityparam.h
Include file: <ilsched/iloscheduler.h>

Parameters are used to change the default behavior of activities and resources. By default, an activity cannot overlap breaks. Instances of `IloActivityOverlapParam` are used to specify in which conditions an activity can overlap breaks.

A break overlap variable allows an activity to start or to finish processing inside some special breaks (called "possibly overlapped breaks"), and allows posting constraints on possible overlap duration.

This class inherits from the IBM ILOG Concert Technology class `IloExtractable`. That class is documented in the *IBM ILOG Concert Technology Reference Manual*.

For more information, see Calendars, and Parameter Classes.

See Also: IloActivity, IloSchedulerEnv

Constructor Summary	
public	`IloActivityOverlapParam()`
public	`IloActivityOverlapParam(IloActivityOverlapParamI * impl)`
public	`IloActivityOverlapParam(const IloEnv env, const char * name=0)`

Method Summary	
public void	`addEndBreakOverlapType(const IloIntSet types) const`
public void	`addEndBreakOverlapType(IloInt type) const`
public void	`addStartBreakOverlapType(const IloIntSet types) const`
public void	`addStartBreakOverlapType(IloInt type) const`
public void	`clearEndBreakOverlapType() const`
public void	`clearStartBreakOverlapType() const`
public IloNum	`getEndBreakOverlapMax() const`
public IloNum	`getEndBreakOverlapMin() const`
public IloActivityOverlapParamI *	`getImpl() const`
public IloNum	`getStartBreakOverlapMax() const`
public IloNum	`getStartBreakOverlapMin() const`
public IloBool	`hasEndBreakOverlapType() const`
public IloBool	`hasStartBreakOverlapType() const`
public IloBool	`isEndBreakOverlapType(IloInt type) const`
public IloBool	`isStartBreakOverlapType(IloInt type) const`
public void	`removeEndBreakOverlapType(const IloIntSet types) const`
public void	`removeEndBreakOverlapType(IloInt type) const`

public void	removeStartBreakOverlapType(const IloIntSet types) const
public void	removeStartBreakOverlapType(IloInt type) const
public void	setEndBreakOverlapMax(IloNum endOverlapMax) const
public void	setEndBreakOverlapMin(IloNum endOverlapMin) const
public void	setStartBreakOverlapMax(IloNum startOverlapMax) const
public void	setStartBreakOverlapMin(IloNum startOverlapMin) const

Constructors

public **IloActivityOverlapParam**()

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

public **IloActivityOverlapParam**(IloActivityOverlapParamI * impl)

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

public **IloActivityOverlapParam**(const IloEnv env, const char * name=0)

This constructor creates a new instance of `IloActivityOverlapParam` with the default values that an activity cannot overlap breaks at start and end times.

Methods

public void **addEndBreakOverlapType**(const IloIntSet types) const

This member function adds the set of types `types` to the set of end break overlap types.

public void **addEndBreakOverlapType**(IloInt type) const

This member function adds the type `type` to the set of end break overlap types.

public void **addStartBreakOverlapType**(const IloIntSet types) const

This member function adds the set of types `types` to the set of start break overlap types.

public void **addStartBreakOverlapType**(IloInt type) const

This member function adds the type `type` to the set of start break overlap types.

public void **clearEndBreakOverlapType**() const

This member function empties the set of end break overlap types.

```
public void clearStartBreakOverlapType() const
```

This member function empties the set of start break overlap types.

```
public IloNum getEndBreakOverlapMax() const
```

This member function returns the maximal value of the end break overlap variable of the activities depending on the invoking parameter.

```
public IloNum getEndBreakOverlapMin() const
```

This member function returns the minimal value of the end break overlap variable of the activities depending on the invoking parameter.

```
public IloActivityOverlapParamI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloNum getStartBreakOverlapMax() const
```

This member function returns the maximal value of the start break overlap variable of the activities depending on the invoking parameter.

```
public IloNum getStartBreakOverlapMin() const
```

This member function returns the minimal value of the start break overlap variable of the activities depending on the invoking parameter.

```
public IloBool hasEndBreakOverlapType() const
```

This member function returns `IloTrue` if the set of end break overlap types of the invoking parameter is not empty.

```
public IloBool hasStartBreakOverlapType() const
```

This member function returns `IloTrue` if the set of start break overlap types of the invoking parameter is not empty.

```
public IloBool isEndBreakOverlapType(IloInt type) const
```

This member function returns `IloTrue` if the type `type` belongs to the set of end break overlap types of the invoking parameter.

```
public IloBool isStartBreakOverlapType(IloInt type) const
```

This member function returns `IloTrue` if the type `type` belongs to the set of start break overlap types of the invoking parameter.

```
public void removeEndBreakOverlapType(const IloIntSet types) const
```

This function removes the set of types `types` from the set of end break overlap types.

```
public void removeEndBreakOverlapType(IloInt type) const
```

This member function removes the type `type` from the set of end break overlap types.

```
public void removeStartBreakOverlapType(const IloIntSet types) const
```

This function removes the set of types `types` from the set of start break overlap types.

```
public void removeStartBreakOverlapType(IloInt type) const
```

This function removes the type `type` from the set of start break overlap types.

```
public void setEndBreakOverlapMax(IloNum endOverlapMax) const
```

This member function sets `endOverlapMax` as the new maximal value of the end break overlap variable.

```
public void setEndBreakOverlapMin(IloNum endOverlapMin) const
```

This member function sets `endOverlapMin` as the new minimal value of the end break overlap variable.

```
public void setStartBreakOverlapMax(IloNum startOverlapMax) const
```

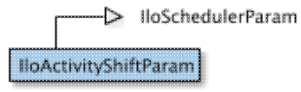
This member function sets `startOverlapMax` as the new maximal value of the start break overlap variable.

```
public void setStartBreakOverlapMin(IloNum startOverlapMin) const
```

This member function sets `startOverlapMin` as the new minimal value of the start break overlap variable.

Class IloActivityShiftParam

Definition file: ilsched/iloactivityparam.h
Include file: <ilsched/iloscheduler.h>



Parameters are used to change the default behavior of activities and resources. By default, an activity deals with all shifts. Instances of `IloActivityShiftParam` are used to define the set of ignored types of shifts for activities.

For more information, see Shift Object Semantic and Parameter Classes.

See Also: IloActivity, IloSchedulerEnv

Constructor Summary	
public	IloActivityShiftParam()
public	IloActivityShiftParam(IloActivityShiftParamI * impl)
public	IloActivityShiftParam(const IloEnv env, const char * name=0)

Method Summary	
public void	addIgnoredShiftType(const IloIntSet types) const
public void	addIgnoredShiftType(IloInt type) const
public void	clearIgnoredShiftType() const
public IloActivityShiftParamI *	getImpl() const
public IloBool	hasIgnoredShiftType() const
public IloBool	isIgnoredShiftType(IloInt type) const
public void	removeIgnoredShiftType(const IloIntSet types) const
public void	removeIgnoredShiftType(IloInt type) const

Constructors

public **IloActivityShiftParam**()

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

public **IloActivityShiftParam**(IloActivityShiftParamI * impl)

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

public **IloActivityShiftParam**(const IloEnv env, const char * name=0)

This constructor creates an instance of `IloActivityShiftParam` with the default values. No shifts are ignored.

Methods

`public void` **`addIgnoredShiftType`**`(const IloIntSet types) const`

This member function adds the set of types `types` to the set of ignored shift types of the invoking parameter.

`public void` **`addIgnoredShiftType`**`(IloInt type) const`

This member function adds the type `type` to the set of ignored shift types of the invoking parameter.

`public void` **`clearIgnoredShiftType`**`() const`

This member function empties the set of ignored shift types of the invoking parameter.

`public IloActivityShiftParamI * ` **`getImpl`**`() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public IloBool` **`hasIgnoredShiftType`**`() const`

This member function returns `IloTrue` if the set of ignored shift types is not empty.

`public IloBool` **`isIgnoredShiftType`**`(IloInt type) const`

This member function returns `IloTrue` if the type `type` belongs to the set of ignored shift types of the invoking parameter. Otherwise, it returns `IloFalse`.

`public void` **`removeIgnoredShiftType`**`(const IloIntSet types) const`

This member function removes the set of types `types` from the set of ignored shift types of the invoking parameter.

`public void` **`removeIgnoredShiftType`**`(IloInt type) const`

This member function removes the type `type` from the set of ignored shift types of the invoking parameter.

Class IloAltResConstraintIterator

Definition file: ilsched/iloresconstraint.h
Include file: <ilsched/iloscheduler.h>

An instance of this class traverses the set of alternative resource constraints defined on an environment.

Note

This class is provided for compatibility with the `IloIterator<IloAltResConstraint>` class of Scheduler 5.0. In the current version of the library the class `IloAltResConstraint` does not exist and therefore `IloIterator<IloAltResConstraint>` is not a well-formed iterator.

For more information, see `IloIterator<IloResourceConstraint>` in the Concert Reference Manual.

See Also: IloResourceConstraintIterator

Constructor Summary	
public	IloAltResConstraintIterator(const IloEnv env)

Method Summary	
public IloBool	ok()
public IloResourceConstraint	operator*()
public void	operator++()

Constructors

`public` **`IloAltResConstraintIterator`**`(const IloEnv env)`

This constructor creates an iterator to traverse all the alternative resource constraints that are defined on the environment `env`.

Methods

`public IloBool` **`ok`**`()`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all the alternative resource constraints have been scanned by the iterator.

`public IloResourceConstraint` **`operator*`**`()`

This operator returns the current instance of `IloResourceConstraint`, the one to which the invoking iterator points. This operator must not be called if the iterator does not point to a valid position, that is, one to which the member function `IloAltResConstraintIterator::ok` returns `IloFalse`.

```
public void operator++()
```

This left-increment operator shifts the current position of the iterator to the next instance of `IloResourceConstraint` with alternatives.

Class IloAltResSet

Definition file: ilsched/iloaltresset.h
Include file: <ilsched/iloscheduler.h>

Alternative Resource Set.

An instance of the class `IloAltResSet` represents a special *set* of resources to which activities can be assigned.

This class inherits from the IBM® ILOG® Concert Technology class `IloExtractable`. That class is documented in the *IBM ILOG Concert Technology Reference Manual*.

When an activity requires an instance of this class, the activity requires exactly one of the resources represented in that set. For convenience, we say that an instance of `IloAltResSet` behaves like a standard resource. To that end, the class includes member functions that reproduce the properties and behavior of a standard resource.

The set of resources (the alternatives) must consist of *capacity* resources (that is, instances of `IloDiscreteEnergy`, `IloDiscreteResource`, `IloUnaryResource`, `IloContinuousReservoir` or `IloReservoir`).

Redundant Resources

It is possible to consider the set of resources as a resource whose theoretical capacity is the sum of the capacities of the resources of the set. This resource is called the redundant resource of the set. The redundant resource of an alternative set of unary or discrete resources is a discrete resource. The redundant resource of an alternative set of discrete energy resources is a discrete energy resource. The redundant resource of an alternative set of reservoirs is a reservoir.

When created (see member functions `IloAltResSet::getRedundantResource` and `IloAltResSet::setRedundantResource`), a redundant resource can be used as a normal resource. In particular, the parameters of the redundant resource can be modified.

See Also: IloResource, IloAltResSet::Iterator

Constructor Summary	
public	`IloAltResSet()`
public	`IloAltResSet(IloAltResSetI * impl)`
public	`IloAltResSet(const IloEnv env, const char * name=0)`

Method Summary	
public void	`add(const IloResource resource) const`
public IloBool	`contains(const IloResource resource) const`
public IloAltResSetI *	`getImpl() const`
public IloCapResource	`getRedundantResource() const`
public IloBool	`hasRedundantResource() const`
public IloBool	`isKeptOpen() const`
public IloBool	`operator==(const IloAltResSet resource) const`
public void	`remove(const IloResource resource) const`
public void	`setRedundantResource(IloBool redundant=IloTrue) const`

Inner Class
IloAltResSet::Iterator

Constructors

`public` **`IloAltResSet`**`()`

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

`public` **`IloAltResSet`**`(IloAltResSetI * impl)`

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

`public` **`IloAltResSet`**`(const IloEnv env, const char * name=0)`

This constructor creates a new instance of `IloAltResSet` and adds it to those managed in the environment. This set of alternative resources is initially empty.

Methods

`public void` **`add`**`(const IloResource resource) const`

This member function adds a new `resource` to the invoking alternative resource set.

`public IloBool` **`contains`**`(const IloResource resource) const`

This member function returns `IloTrue` if `resource` currently belongs to the invoking instance of `IloAltResSet`. Otherwise, it returns `IloFalse`.

`public IloAltResSetI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public IloCapResource` **`getRedundantResource`**`() const`

This member function returns the redundant resource of the invoking alternative resource set. If this redundant resource does not currently exist - no previous call to the member function `setRedundantResource`(`IloTrue`) - this member function will create it. In that case, the current alternative resource set must not be empty and all the resources of the set must be of the same type (discrete, discrete energy, or reservoir). Otherwise, an error will be raised.

`public IloBool` **`hasRedundantResource`**`() const`

This member function returns `IloTrue` if the invoking alternative resource set is currently associated with a redundant resource. Otherwise, it will return `IloFalse`.

`public IloBool` **`isKeptOpen`**`() const`

This member function returns `IloTrue` if all resources belonging to the invoking alternative resource set must be kept open during the search. Otherwise, it returns `IloFalse`.

```
public IloBool operator==(const IloAltResSet resource) const
```

This operator returns `IloTrue` if the invoking instance and the argument `resource` are identical; that is, they are both handles with the same implementation object. Otherwise, it returns `IloFalse`.

```
public void remove(const IloResource resource) const
```

This member function removes `resource` from the invoking alternative resource set.

```
public void setRedundantResource(IloBool redundant=IloTrue) const
```

If the argument `redundant` is equal to `IloTrue`, this member function will create a redundant resource for the invoking alternative resource set. In that case, the current alternative resource set must not be empty, and all the resources of the set must be of the same type (discrete, discrete energy, or reservoir). Otherwise, an error will be raised.

If the argument `redundant` is equal to `IloFalse` and if a redundant resource already exists - from a previous call to `setRedundantResource(IloTrue)` or `getRedundantResource()` - this resource will no longer be considered as the redundant resource of the invoking alternative resource set. Note that this resource will still exist as well as its parameters and the activities that may use it.

Class IloCalendar

Definition file: ilsched/ilocalendar.h

An instance of `IloCalendar` allows modeling complex behavior for activity variables (start, end, duration and processing time) within a resource. This behaviour could represent, for example, holidays, resource performances, and so forth. For more information, see Calendars. A calendar object is defined by three components:

- A set of breaks which basically can suspend the execution of the concerned activity (see Calendars)
- A set of shifts which can, for example, forbid some start dates (see Shift Object Semantic)
- A granular step-wise function to define the efficiency of the resource along the schedule (see Functional and Integral Constraints on Resources)

Constructor Summary	
public	IloCalendar()
public	IloCalendar(IloCalendarI * impl)
public	IloCalendar(const IloEnv env, const char * name=0)

Method Summary	
public void	addShiftObject(IloShiftObject shift) const
public IloBool	areShiftObjectsIgnored() const
public IloCalendarI *	getImpl() const
public IloBool	hasBreakListParam() const
public IloBool	hasEfficiencyParam() const
public IloBool	hasShiftObject() const
public void	ignoreBreakListParam(IlcBool ignored=IloTrue) const
public void	ignoreEfficiencyParam(IlcBool ignored=IloTrue) const
public void	ignoreShiftObjects(IlcBool ignored=IloTrue) const
public IloBool	isBreakListParamIgnored() const
public IloBool	isEfficiencyParamIgnored() const
public void	removeShiftObject(IloShiftObject shift) const
public void	setBreakListParam(const IloIntervalList breakList) const
public void	setEfficiencyParam(const IloGranularFunction f) const

Inner Class
IloCalendar::ShiftObjectIterator

Constructors

`public` **`IloCalendar`**`()`

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

```
public IloCalendar(IloCalendarI * impl)
```

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

```
public IloCalendar(const IloEnv env, const char * name=0)
```

This constructor creates a new instance of `IloCalendar`. Its name is set to `name`

Methods

```
public void addShiftObject(IloShiftObject shift) const
```

This member function adds the shift object `shift` to the invoking calendar.

```
public IloBool areShiftObjectsIgnored() const
```

This member function returns `IloTrue` if shift objects of the invoking calendar are not taken into account when searching for a solution. Otherwise, it returns `IloFalse`.

```
public IloCalendarI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloBool hasBreakListParam() const
```

This member function returns `IloTrue` if a list of breaks has been attached to the invoking calendar. Otherwise, it returns `IloFalse`.

```
public IloBool hasEfficiencyParam() const
```

This member function returns `IloTrue` if an efficiency function has been attached to the invoking calendar. Otherwise, it returns `IloFalse`.

```
public IloBool hasShiftObject() const
```

This member function returns `IlcTrue` if the invoking calendar contains shift objects. If the calendar list of shift object is empty, it returns `IlcFalse`.

```
public void ignoreBreakListParam(IlcBool ignored=IloTrue) const
```

When the argument `ignored` is equal to `IloTrue`, this member function inhibits breaks for the invoking calendar. That is, all breaks will be ignored.

```
public void ignoreEfficiencyParam(IlcBool ignored=IloTrue) const
```

When the argument `ignored` is equal to `IloTrue`, this member function inhibits the efficiency behavior for the invoking calendar. That is, the efficiency function is considered as equal to the granularity on the entire horizon.

```
public void ignoreShiftObjects(IlcBool ignored=IloTrue) const
```

When the argument `ignored` is equal to `IloTrue`, this member function inhibits all shift objects added to the invoking calendar. That is, no shift is taken into account when searching for a solution.

```
public IloBool isBreakListParamIgnored() const
```

This member function returns `IloTrue` if breaks of the invoking calendar are not taken into account when searching for a solution. Otherwise, it returns `IloFalse`.

```
public IloBool isEfficiencyParamIgnored() const
```

This member function returns `IloTrue` if the efficiency of the invoking calendar is not taken into account when searching for a solution. Otherwise, it returns `IloFalse`.

```
public void removeShiftObject(IloShiftObject shift) const
```

This member function removes the shift object `shift` from the invoking calendar.

```
public void setBreakListParam(const IloIntervalList breakList) const
```

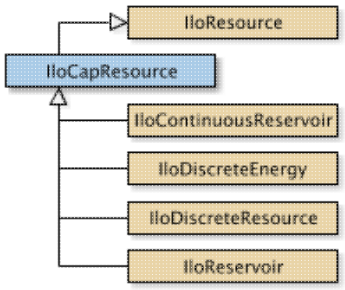
This member function sets `breakList` as the new break list of the calendar.

```
public void setEfficiencyParam(const IloGranularFunction f) const
```

This member function sets `f` as the granular step-wise function that models the efficiency of the calendar within the schedule.

Class IloCapResource

Definition file: ilsched/ilocapacity.h
Include file: <ilsched/iloscheduler.h>



`IloCapResource` is the root class for *capacity* resources, that is, resources that are defined to have a limited integer capacity over time. In Scheduler, there are five classes of capacity resources:

- `IloDiscreteEnergy`
- `IloDiscreteResource`
- `IloReservoir`
- `IloContinuousReservoir`
- `IloUnaryResource`

Theoretical and Maximal Capacity

The theoretical capacity of a capacity resource is a bound (that is, a limit) on the amount of capacity that can be available at any point in time. The maximal capacity is the capacity that can be used in practice at a particular point in time or over a particular interval of time. The maximal capacity typically varies over time (a resource *maximal capacity profile*), while the theoretical capacity is an intrinsic property of the resource. The maximal capacity can never exceed the theoretical capacity. The theoretical capacity can be infinite.

The theoretical capacity can be set by the member function `IloCapResource::setCapacity`. The maximal level is set by the member functions `setCapacityMax`, `setEnergyMax`, `setLevelMax` of the classes `IloDiscreteResource`, `IloDiscreteEnergy`, `IloReservoir` or `IloContinuousReservoir`.

Resources as Data Members

An instance of the class `IloCapResource` may be a data member of another “external” object. In such a case, it may be useful to find the external object from the instance of `IloCapResource`. The inherited member functions `IloExtractable::getObject` and `IloExtractable::setObject` are provided to manage such an inverse link.

Parameter Classes

Initial occupation: (class `IloNumToNumStepFunction` or `IloNumToNumSegmentFunction`

This parameter describes the initial occupation of the capacity resource over time (see the following section, Initial Occupation). The parameter class `IloNumToNumStepFunction` is the capacity resource of an instance of `IloDiscreteEnergy`, `IloDiscreteResource`, or `IloReservoir`. `IloNumToNumSegmentFunction` is the capacity resource of an instance of `IloContinuousReservoir`. It is directly modified by the member function `IloCapResource::setInitialOccupation`. `IloNumToNumStepFunction` and `IloNumToNumSegmentFunction` are documented in the *IBM ILOG Concert Technology Reference Manual*.

Refer to *Scheduler Overview* for more information on how to share parameters among resources, and how the direct modification of parameters through the resource API may affect them.

Initial Occupation

A resource may already be occupied by some activities before solving a scheduling problem. This initial occupation can be set up without having to declare the corresponding activities.

That facility is intended to help in solving a problem by iteratively adding a new set of activities to schedule or in improving a solution by rescheduling a subset of the activities.

The initial occupation parameter is defined with an instance of the `IloNumToNumSegmentFunction` class if the capacity resource is a continuous reservoir, and with an instance of the `IloNumToNumStepFunction` class otherwise. `IloNumToNumStepFunction` and `IloNumToNumSegmentFunction` are documented in the *IBM ILOG Concert Technology Reference Manual*.

For instances of `IloDiscreteResource` and `IloDiscreteEnergy`, the initial level is zero outside the definition domain of the function.

For instances of `IloReservoir` or `IloContinuousReservoir`, if the definition domain of the function intersects the capacity enforcement intervals defined on the invoking reservoir (see `IloResource::setCapacityEnforcementIntervalsParam`), the initial level is given by the function on the defined enforcement intervals, and by zero elsewhere. That is, the initial level of the reservoir is ignored. If the definition domain of the function does not intersect the temporal interval of the time table, the initial level of the reservoir is used as usual.

See Also: IloAltResSet, IloEnforcementLevel, IloResource, IloResourceConstraint

Constructor Summary	
public	IloCapResource()
public	IloCapResource(IloCapResourceI * impl)

Method Summary	
public void	addMaxTextureIgnoreInterval(const IloIntervalList list)
public void	addMaxTextureIgnoreInterval(IloNum start, IloNum end)
public void	addMaxTextureIgnoreIntervalOnDuration(IloNum start, IloNum duration)
public void	addMaxTexturePeriodicIgnoreInterval(IloNum start, IloNum duration, IloNum period, IloNum end)
public void	addMinTextureIgnoreInterval(const IloIntervalList list)
public void	addMinTextureIgnoreInterval(IloNum start, IloNum end)
public void	addMinTextureIgnoreIntervalOnDuration(IloNum start, IloNum duration)
public void	addMinTexturePeriodicIgnoreInterval(IloNum start, IloNum duration, IloNum period, IloNum end)
public void	emptyMaxTextureIgnoreIntervals()
public void	emptyMinTextureIgnoreIntervals()
public IloNum	getCapacity() const
public IloCapResourceI *	getImpl() const
public IloNum	getInitialOccupation(IloNum time) const
public IloNum	getInitialOccupationMax(IloNum timeMin, IloNum timeMax) const
public IloNum	getInitialOccupationMin(IloNum timeMin, IloNum timeMax) const
public IloBool	hasInitialOccupation() const
public IloBool	hasMaxTextureMeasurement() const

public IloBool	hasMinTextureMeasurement() const
public void	removeMaxTextureIgnoreInterval(const IloIntervalList list)
public void	removeMaxTextureIgnoreInterval(IloNum start, IloNum end)
public void	removeMaxTextureIgnoreIntervalOnDuration(IloNum start, IloNum duration)
public void	removeMaxTexturePeriodicIgnoreInterval(IloNum start, IloNum duration, IloNum period, IloNum end)
public void	removeMinTextureIgnoreInterval(const IloIntervalList list)
public void	removeMinTextureIgnoreInterval(IloNum start, IloNum end)
public void	removeMinTextureIgnoreIntervalOnDuration(IloNum start, IloNum duration)
public void	removeMinTexturePeriodicIgnoreInterval(IloNum start, IloNum duration, IloNum period, IloNum end)
public void	setCapacity(IloNum capacity) const
public void	setInitialOccupation(IloNum timeMin, IloNum occ1, IloNum timeMax, IloNum occ2) const
public void	setInitialOccupation(IloNum timeMin, IloNum timeMax, IloNum occ) const
public void	setInitialOccupationParam(const IloNumToNumSegmentFunction tfp) const
public void	setInitialOccupationParam(const IloNumToNumStepFunction tfp) const
public void	setMaxTextureHeuristicBeta(IloNum b)
public void	setMaxTextureParam(const IloTextureParam param) const
public void	setMaxTextureRandomGenerator(IloRandom r)
public void	setMinTextureHeuristicBeta(IloNum b)
public void	setMinTextureParam(const IloTextureParam param) const
public void	setMinTextureRandomGenerator(IloRandom r)
public void	unsetMaxTextureRandomGenerator()
public void	unsetMinTextureRandomGenerator()

Inherited Methods from `IloResource`
addCapacityEnforcementInterval, addTransitionTimeEnforcementInterval, areCalendarConstraintsIgnored, areCapacityConstraintsIgnored, arePrecedenceConstraintsIgnored, areSequenceConstraintsIgnored, areTransitionTimeConstraintsIgnored, getCalendar, getCalendarEnforcement, getCapacityEnforcement, getDurationEnforcement, getImpl, getPrecedenceEnforcement, getSequenceEnforcement, getTransitionTimeEnforcement, hasCalendar, ignoreCalendarConstraints, ignoreCapacityConstraints, ignorePrecedenceConstraints, ignoreSequenceConstraints, ignoreTransitionTimeConstraints, isCapacityResource, isContinuousReservoir, isDiscreteEnergy, isDiscreteResource, isKeptOpen, isReservoir, isStateResource, isUnaryResource, keepOpen, removeCapacityEnforcementInterval, removeTransitionTimeEnforcementInterval, setCalendar, setCalendarEnforcement, setCapacityEnforcement, setCapacityEnforcementIntervalsParam, setDurationEnforcement, setPrecedenceEnforcement, setResourceParam, setSequenceEnforcement, setTransitionTimeEnforcement, setTransitionTimeEnforcementIntervalsParam

Constructors

`public` **`IloCapResource`**`()`

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

`public` **`IloCapResource`**`(IloCapResourceI * impl)`

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

Methods

`public void` **`addMaxTextureIgnoreInterval`**`(const IloIntervalList list)`

This member function adds an interval list to the list of intervals that are ignored by the texture measurement on the maximum capacity constraint of the invoking resource. The new interval is merged with existing intervals that it overlaps, if any.

`public void` **`addMaxTextureIgnoreInterval`**`(IloNum start, IloNum end)`

This member function adds an interval to the list of intervals that are ignored by the texture measurement on the maximum capacity constraint of the invoking resource. The ignored interval is `[start, end)`. The new interval is merged with existing intervals that it overlaps, if any.

`public void` **`addMaxTextureIgnoreIntervalOnDuration`**`(IloNum start, IloNum duration)`

This member function adds an interval to the list of intervals that are ignored by the texture measurement on the maximum capacity constraint of the resource. The ignored interval is `[start,start+duration)`. The new interval is merged with existing intervals that it overlaps, if any.

`public void` **`addMaxTexturePeriodicIgnoreInterval`**`(IloNum start, IloNum duration, IloNum period, IloNum end)`

This member function adds a set of intervals to the list of intervals that are ignored by the texture measurement on the maximum capacity constraint of the invoking resource. For every `i >= 0` such that `start + i * period < end`, an interval of `[start + i * period, start + duration + i * period)` is added. Adding a new interval that overlaps with an already existing interval results in the merging of the intervals.

`public void` **`addMinTextureIgnoreInterval`**`(const IloIntervalList list)`

This member function adds an interval list to the list of intervals that are ignored by the texture measurement on the minimum capacity constraint of the invoking resource. The new interval is merged with existing intervals that it overlaps, if any.

`public void` **`addMinTextureIgnoreInterval`**`(IloNum start, IloNum end)`

This member function adds an interval to the list of intervals that are ignored by the texture measurement on the minimum capacity constraint of the invoking resource. The ignored interval is `[start, end)`. The new interval is merged with existing intervals that it overlaps, if any.

```
public void addMinTextureIgnoreIntervalOnDuration(IloNum start, IloNum duration)
```

This member function adds an interval to the list of intervals that are ignored by the texture measurement on the minimum capacity constraint of the resource. The ignored interval is `[start,start+duration)`. The new interval is merged with existing intervals that it overlaps, if any.

```
public void addMinTexturePeriodicIgnoreInterval(IloNum start, IloNum duration,
IloNum period, IloNum end)
```

This member function adds a set of intervals to the list of intervals that are ignored by the texture measurement on the minimum capacity constraint of the invoking resource. For every `i >= 0` such that `start + i * period < end`, an interval of `[start + i * period, start + duration + i * period)` is added. Adding a new interval that overlaps with an already existing interval results in the merging of the intervals.

```
public void emptyMaxTextureIgnoreIntervals()
```

This member function removes all the intervals from the ignored intervals of the texture measurement on the maximum capacity constraint of the invoking resource.

```
public void emptyMinTextureIgnoreIntervals()
```

This member function removes all the intervals from the ignored intervals of the texture measurement on the minimum capacity constraint

```
public IloNum getCapacity() const
```

This member function returns the theoretical capacity of the invoking resource.

```
public IloCapResourceI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloNum getInitialOccupation(IloNum time) const
```

Returns the initial occupation at `time`.

Initial Occupation

A resource may already be occupied by some activities before solving a scheduling problem. This initial occupation can be set up without having to declare the corresponding activities.

That facility is intended to help in solving a problem by iteratively adding a new set of activities to schedule or in improving a solution by rescheduling a subset of the activities.

The initial occupation parameter is defined with an instance of the `IloNumToNumSegmentFunction` class if the capacity resource is a continuous reservoir, and with an instance of the `IloNumToNumStepFunction` class

otherwise. IloNumToNumStepFunction and IloNumToNumSegmentFunction are documented in the *Concert Technology Reference Manual*.

For instances of IloDiscreteResource and IloDiscreteEnergy, the initial level is zero outside the definition domain of the function.

For instances of IloReservoir or IloContinuousReservoir, if the definition domain of the function intersects the capacity enforcement intervals defined on the invoking reservoir (see IloResource::setCapacityEnforcementIntervalsParam), the initial level is given by the function on the defined enforcement intervals, and by zero elsewhere. That is, the initial level of the reservoir is ignored. If the definition domain of the function does not intersect the temporal interval of the time table, the initial level of the reservoir is used as usual.

```
public IloNum getInitialOccupationMax(IloNum timeMin, IloNum timeMax) const
```

Returns the maximum level of the initial occupation function on the interval [timeMin, timeMax).

See IloCapResource::getInitialOccupation for more information about initial occupation.

```
public IloNum getInitialOccupationMin(IloNum timeMin, IloNum timeMax) const
```

Returns the minimum level of the initial occupation function on the interval [timeMin, timeMax).

See IloCapResource::getInitialOccupation for more information about initial occupation.

```
public IloBool hasInitialOccupation() const
```

This member function returns IloTrue if an initial occupation has been set up on the invoking resource. Otherwise, it returns IloFalse.

See IloCapResource::getInitialOccupation for more information about initial occupation.

```
public IloBool hasMaxTextureMeasurement() const
```

This member function returns IloTrue if a texture measurement will be created at extraction time on the maximum capacity constraint of the invoking resource. Otherwise, IloFalse is returned.

```
public IloBool hasMinTextureMeasurement() const
```

This member function returns IloTrue if a texture measurement will be created at extraction time on the minimum capacity constraint of the invoking resource. Otherwise, IloFalse is returned.

```
public void removeMaxTextureIgnoreInterval(const IloIntervalList list)
```

This member function removes all intervals ignored by the texture measurement on the invoking resource during the intervals in list.

```
public void removeMaxTextureIgnoreInterval(IloNum start, IloNum end)
```

This member function removes all intervals ignored by the texture measurement on the maximum capacity constraint of the invoking resource between `start` and `end`. If `start` is inside an existing interval `[start1, end1)`, that is, `start1 < start < end1`, this results in the ignored interval `[start1, start)`. If `end` is inside an interval `[start2, end2)` this results in an ignored interval `[end, end2)`.

```
public void removeMaxTextureIgnoreIntervalOnDuration(IloNum start, IloNum duration)
```

This member function removes all ignored intervals on the texture measurement on the maximum capacity constraint of the invoking resource between `start` and `start+duration`.

```
public void removeMaxTexturePeriodicIgnoreInterval(IloNum start, IloNum duration,
IloNum period, IloNum end)
```

This member function removes ignored intervals from the texture measurement on the maximum capacity constraint of the invoking resource. More precisely, for every `i >= 0` such that `start + i * period < end`, this function removes all intervals between`start + i * period` and `start + duration + i *period`.

```
public void removeMinTextureIgnoreInterval(const IloIntervalList list)
```

This member function removes all intervals ignored by the texture measurement on the minimum capacity constraint of the invoking resource during the intervals in list.

```
public void removeMinTextureIgnoreInterval(IloNum start, IloNum end)
```

This member function removes all intervals ignored by the texture measurement on the minimum capacity constraint of the invoking resource between `start` and `end`. If `start` is inside an existing interval `[start1, end1)`, that is, `start1 < start < end1`, this results in the ignored interval `[start1, start)`. If `end` is inside an interval `[start2, end2)` this results in an ignored interval `[end, end2)`.

```
public void removeMinTextureIgnoreIntervalOnDuration(IloNum start, IloNum duration)
```

This member function removes all ignored intervals on the texture measurement on the minimum capacity constraint of the invoking resource between `start` and `start+duration`.

```
public void removeMinTexturePeriodicIgnoreInterval(IloNum start, IloNum duration,
IloNum period, IloNum end)
```

This member function removes ignored intervals from the texture measurement on the minimum capacity constraint of the invoking resource. More precisely, for every `i >= 0` such that `start + i * period < end`, this function removes all intervals between`start + i * period` and `start + duration + i *period`.

```
public void setCapacity(IloNum capacity) const
```

This member function sets `capacity` as the new theoretical capacity of the invoking resource.

```
public void setInitialOccupation(IloNum timeMin, IloNum occ1, IloNum timeMax,
IloNum occ2) const
```

Sets the initial occupation on the interval `[timeMin, timeMax)` to be equal to `occ1` at `timeMin`, to be equal to `occ2` at `timeMax` and to be linear between these two time points. This function should be called only if the invoking capacity resource is a continuous reservoir.

See `IloCapResource::getInitialOccupation` for more information about initial occupation.

```
public void setInitialOccupation(IloNum timeMin, IloNum timeMax, IloNum occ) const
```

Sets the initial occupation on the interval [`timeMin, timeMax)` to be equal to `occ`.

See `IloCapResource::getInitialOccupation`) for more information about initial occupation.

```
public void setInitialOccupationParam(const IloNumToNumSegmentFunction tfp) const
```

This member function sets the piecewise linear function `tfp` as the initial level of the invoking resource. An exception is thrown if the invoking resource is not a continuous reservoir.

See `IloCapResource::getInitialOccupation` for more information about initial occupation.

```
public void setInitialOccupationParam(const IloNumToNumStepFunction tfp) const
```

This member function sets the argument `tfp` as the initial level of the invoking resource.

See `IloCapResource::getInitialOccupation` for more information about initial occupation.

```
public void setMaxTextureHeuristicBeta(IloNum b)
```

This member function sets the beta value to be used with the random number generator for the texture measurement on the maximum constraint of the invoking resource. If no random number generator is used, this function does nothing. For details on the use of the beta argument, see `IlcResourceTexture::setRandomGenerator`.

```
public void setMaxTextureParam(const IloTextureParam param) const
```

This member function sets the texture parameter on maximum capacity constraints to `param`. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public void setMaxTextureRandomGenerator(IloRandom r)
```

This member function sets the random number generator that will be used in choosing the critical time point for the texture measurement on the maximum constraint of the invoking resource. By default, no random number generator is used.

```
public void setMinTextureHeuristicBeta(IloNum b)
```

This member function sets the beta value to be used with the random number generator for the texture measurement on the minimum constraint of the invoking resource. If no random number generator is used, this function does nothing. For details on the use of the beta argument, see `IlcResourceTexture::setRandomGenerator`.

```
public void setMinTextureParam(const IloTextureParam param) const
```

This member function sets the texture parameter on minimum capacity constraints to `param`. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public void setMinTextureRandomGenerator(IloRandom r)
```

This member function sets the random number generator that will be used in choosing the critical time point for the texture measurement on the minimum constraint of the invoking resource. By default, no random number generator is used.

```
public void unsetMaxTextureRandomGenerator()
```

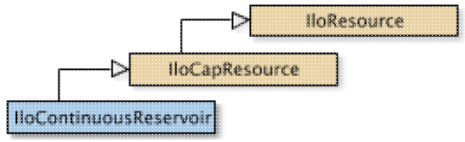
This member function removes the random number generator from the maximum capacity constraint, meaning that no random numbers will be used in choosing the critical time point in the texture measurement.

```
public void unsetMinTextureRandomGenerator()
```

This member function removes the random number generator from the minimum capacity constraint, meaning that no random numbers will be used in choosing the critical time point in the texture measurement.

Class IloContinuousReservoir

Definition file: ilsched/ilocontreservoir.h
Include file: <ilsched/iloscheduler.h>



An instance of the class `IloContinuousReservoir` represents a resource which activities can either fill or empty. The process is continuous and linear between the start and the end of the activity. If the duration of the activity is null, the filling (or emptying) process is instantaneous (so not continuous). The continuous reservoir cannot be emptied if it is already empty and, if a maximal capacity is defined, then this maximal capacity, or level, will never be exceeded.

When the model of your problem represents an ongoing process, you may be faced with the fact that a reservoir level already exists. You can simply pass an initial level like that to the constructor of `IloContinuousReservoir` or use the `IloContinuousReservoir::setInitialLevel` member function.

The maximum and minimum levels of a continuous reservoir can vary over time. You can define them by using member functions of `IloContinuousReservoir`.

Parameter classes

Minimal and maximal capacity: (class `IloNumToNumSegmentFunction`)

These parameters describe the minimal and maximal levels over time. They are directly modified by the member functions `IloContinuousReservoir::setLevelMin` and `IloContinuousReservoir::setLevelMax`. The class `IloNumToNumSegmentFunction` is documented in the *IBM ILOG Concert Technology Reference Manual*.

Refer to Scheduler Overview for more information on how to share parameters among resources, and how the direct modification of parameters through the resource API may affect them.

See Also: IloEnforcementLevel, IloCapResource, IloResourceConstraint

Constructor Summary	
`public`	`IloContinuousReservoir()`
`public`	`IloContinuousReservoir(IloContinuousReservoirI * impl)`
`public`	`IloContinuousReservoir(const IloEnv env, IloNum capacity=IloMaxCapacityReservoir, IloNum initialLevel=0, const char * name=0)`

Method Summary	
`public IloContinuousReservoirI *`	`getImpl() const`
`public IloNum`	`getInitialLevel() const`
`public IloNum`	`getLevelMax(IloNum time) const`
`public IloNum`	`getLevelMaxMax(IloNum timeMin, IloNum timeMax) const`
`public IloNum`	`getLevelMaxMin(IloNum timeMin, IloNum timeMax) const`
`public IloNum`	`getLevelMin(IloNum time) const`

public IloNum	getLevelMinMax(IloNum timeMin, IloNum timeMax) const
public IloNum	getLevelMinMin(IloNum timeMin, IloNum timeMax) const
public void	setInitialLevel(IloNum level) const
public void	setLevelMax(IloNum timeMin, IloNum timeMax, IloNum level) const
public void	setLevelMaxParam(const IloNumToNumSegmentFunction tfp) const
public void	setLevelMin(IloNum timeMin, IloNum timeMax, IloNum level) const
public void	setLevelMinParam(const IloNumToNumSegmentFunction tfp) const

Inherited Methods from `IloCapResource`
addMaxTextureIgnoreInterval, addMaxTextureIgnoreInterval, addMaxTextureIgnoreIntervalOnDuration, addMaxTexturePeriodicIgnoreInterval, addMinTextureIgnoreInterval, addMinTextureIgnoreInterval, addMinTextureIgnoreIntervalOnDuration, addMinTexturePeriodicIgnoreInterval, emptyMaxTextureIgnoreIntervals, emptyMinTextureIgnoreIntervals, getCapacity, getImpl, getInitialOccupation, getInitialOccupationMax, getInitialOccupationMin, hasInitialOccupation, hasMaxTextureMeasurement, hasMinTextureMeasurement, removeMaxTextureIgnoreInterval, removeMaxTextureIgnoreInterval, removeMaxTextureIgnoreIntervalOnDuration, removeMaxTexturePeriodicIgnoreInterval, removeMinTextureIgnoreInterval, removeMinTextureIgnoreInterval, removeMinTextureIgnoreIntervalOnDuration, removeMinTexturePeriodicIgnoreInterval, setCapacity, setInitialOccupation, setInitialOccupation, setInitialOccupationParam, setInitialOccupationParam, setMaxTextureHeuristicBeta, setMaxTextureParam, setMaxTextureRandomGenerator, setMinTextureHeuristicBeta, setMinTextureParam, setMinTextureRandomGenerator, unsetMaxTextureRandomGenerator, unsetMinTextureRandomGenerator

Inherited Methods from `IloResource`
addCapacityEnforcementInterval, addTransitionTimeEnforcementInterval, areCalendarConstraintsIgnored, areCapacityConstraintsIgnored, arePrecedenceConstraintsIgnored, areSequenceConstraintsIgnored, areTransitionTimeConstraintsIgnored, getCalendar, getCalendarEnforcement, getCapacityEnforcement, getDurationEnforcement, getImpl, getPrecedenceEnforcement, getSequenceEnforcement, getTransitionTimeEnforcement, hasCalendar, ignoreCalendarConstraints, ignoreCapacityConstraints, ignorePrecedenceConstraints, ignoreSequenceConstraints, ignoreTransitionTimeConstraints, isCapacityResource, isContinuousReservoir, isDiscreteEnergy, isDiscreteResource, isKeptOpen, isReservoir, isStateResource, isUnaryResource, keepOpen, removeCapacityEnforcementInterval, removeTransitionTimeEnforcementInterval, setCalendar, setCalendarEnforcement, setCapacityEnforcement, setCapacityEnforcementIntervalsParam, setDurationEnforcement, setPrecedenceEnforcement, setResourceParam, setSequenceEnforcement, setTransitionTimeEnforcement, setTransitionTimeEnforcementIntervalsParam

Constructors

public **IloContinuousReservoir**()

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

```
public IloContinuousReservoir(IloContinuousReservoirI * impl)
```

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

```
public IloContinuousReservoir(const IloEnv env, IloNum
capacity=IloMaxCapacityReservoir, IloNum initialLevel=0, const char * name=0)
```

This constructor creates a new instance of `IloContinuousReservoir` and adds it to the set of resources managed in the given environment. The `capacity` argument expresses the capacity of the new reservoir. The capacity may be consumed by certain activities and produced by others. The argument `initialLevel` defines an initial amount in the reservoir at the time origin of the schedule environment. By default, the continuous reservoir is assumed to be empty at the time origin; that is, the initial level is 0 (zero). If the argument `name` is defined, it is assigned as the name of the newly created continuous reservoir.

Methods

```
public IloContinuousReservoirI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloNum getInitialLevel() const
```

This member function returns the initial level of the continuous reservoir; that is, the initial level that was passed to the continuous reservoir constructor.

```
public IloNum getLevelMax(IloNum time) const
```

This member function returns the maximal level of the invoking continuous reservoir at the given `time`.

```
public IloNum getLevelMaxMax(IloNum timeMin, IloNum timeMax) const
```

This member function returns the maximal value of the level of the invoking continuous reservoir over the interval `[timeMin, timeMax)` (that is, the maximal value over the interval `[timeMin, timeMax)` of the maximal reservoir level).

```
public IloNum getLevelMaxMin(IloNum timeMin, IloNum timeMax) const
```

This member function returns the maximal value of the minimal level of the invoking continuous reservoir over the interval `[timeMin timeMax)`.

```
public IloNum getLevelMin(IloNum time) const
```

This member function returns the minimal level of the invoking continuous reservoir at the given `time`.

```
public IloNum getLevelMinMax(IloNum timeMin, IloNum timeMax) const
```

This member function returns the minimal value over the interval `[timeMin, timeMax)` of the maximal level of the invoking continuous reservoir.

```
public IloNum getLevelMinMin(IloNum timeMin, IloNum timeMax) const
```

This member function returns the minimal level of the invoking continuous reservoir throughout the interval `[timeMin, timeMax)` (that is, the minimal value over the interval `[timeMin, timeMax)` of the minimal reservoir level).

```
public void setInitialLevel(IloNum level) const
```

This member function sets the initial level of the invoking continuous reservoir.

```
public void setLevelMax(IloNum timeMin, IloNum timeMax, IloNum level) const
```

This member function states that the level of the continuous reservoir can be at most `level` throughout the time interval `[timeMin, timeMax)`. An instance of `IloException` is thrown if the timetable of the invoking continuous reservoir does not cover the complete interval indicated by `[timeMin, timeMax)`. The continuous reservoir must be *closed* in order to propagate constraints.

```
public void setLevelMaxParam(const IloNumToNumSegmentFunction tfp) const
```

This member function sets the maximal level of the continuous reservoir over time to be the function defined in `tfp`.

```
public void setLevelMin(IloNum timeMin, IloNum timeMax, IloNum level) const
```

This member function states that the level of the continuous reservoir must be at least `level` at each integer time point of the interval `[timeMin, timeMax)`. An instance of `IloException` is thrown if the timetable of the invoking continuous reservoir does not cover the complete interval indicated by `[timeMin, timeMax)`. The continuous reservoir must be *closed* in order to propagate constraints.

```
public void setLevelMinParam(const IloNumToNumSegmentFunction tfp) const
```

This member function sets the minimal level of the continuous reservoir over time to be the function defined in `tfp`.

Class IloCoverConstraint

Definition file: ilsched/iloactivity.h
Include file: <ilsched/iloscheduler.h>

Instances of the class `IloCoverConstraint` are cover constraints. A cover constraint states that an activity (the covering activity) must exactly cover a set of activities (the covered activities). More precisely, it means that the start time of the covering activity is equal to the earliest of the start times of the covered activities, and that the end time of the covering activity is equal to the latest of the end times of the covered activities.

This class inherits from the IBM® ILOG® Concert Technology class `IloConstraint`. That class is documented in the *IBM ILOG Concert Technology Reference Manual*.

Instances of this class are created by the member function `IloActivity::covers`.

See Also: IloActivity, IloActivityConstraintsParam

Constructor Summary	
public	`IloCoverConstraint()`
public	`IloCoverConstraint(IloCoverConstraintI * impl)`

Method Summary	
public void	`add(const IloActivity act) const`
public IloBool	`contains(const IloActivity act) const`
public IloActivity	`getActivity() const`
public IloCoverConstraintI *	`getImpl() const`
public void	`remove(const IloActivity act) const`

Constructors

`public` **`IloCoverConstraint`**`()`

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

`public` **`IloCoverConstraint`**`(IloCoverConstraintI * impl)`

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

Methods

`public void` **`add`**`(const IloActivity act) const`

This member function adds the activity `act` to the set of activities to be covered.

Example

The code:

```
IloCoverConstraint coverCT = act.covers();
coverCT.add(act1);
coverCT.add(act2);
model.add(coverCT);
```

adds to the model the constraint that activity `act` must cover both activities `act1` and `act2`. Another way to add the same constraint is:

```
model.add(act.covers(2, act1, act2));
```

```
public IloBool contains(const IloActivity act) const
```

This member function returns `IloTrue` if and only if the activity `act` is to be covered by the constraint.

```
public IloActivity getActivity() const
```

This member function returns the covering activity of the cover constraint. That is, it returns the activity on which a member function `IloActivity::covers` was called to build the invoking cover constraint.

```
public IloCoverConstraintI * getImpl() const
```

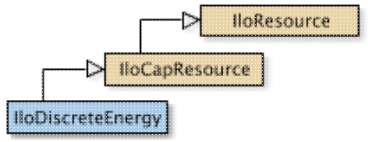
This member function returns a pointer to the implementation object of the invoking handle.

```
public void remove(const IloActivity act) const
```

This member function removes the activity `act` from the set of activities to be covered.

Class IloDiscreteEnergy

Definition file: ilsched/iloenergy.h
Include file: <ilsched/iloscheduler.h>



Discrete Energy Resource.

An instance of the class `IloDiscreteEnergy` represents a resource available as a certain amount of *energy* (for example, in watt-hours, in human-months) over certain time buckets (for example, minutes, hours, months, years). The available energy of a time bucket is used by the activities executed on that time bucket, and as a consequence, *energy capacity constraints* use the energy of the discrete energy resource.

For example, let's assume that each unit of time corresponds to an hour, and that we have defined a discrete energy resource that has a time step of 24 (corresponding to a day), and energy 10. Then if we have an activity of duration 3 (hours) that requires the resource with capacity 2 (machines), it uses energy of 6 (machine-hours). Thus, if this activity is scheduled on the first day, the remaining energy for that first day is 4 (machine-hours).

An instance of the class `IloDiscreteEnergy` uses the concept of energy, differing from the class `IloDiscreteResource`, which uses the concept of instantaneous capacity. However, when the time step of the resource is 1 (one), the energy over an interval corresponds to the instantaneous capacity, and thus in that case, there is no difference between the two classes. The time buckets of the discrete energy resource are defined by the capacity enforcement intervals parameter.

Parameter classes

Minimal and maximal capacity: (class `IloNumToNumStepFunction`)

These parameters describe the minimal and maximal energy per time bucket over time. They are directly modified by the member functions `IloDiscreteEnergy::setEnergyMin` and `IloDiscreteEnergy::setEnergyMax`. `IloNumToNumStepFunction` is documented in the extensions section of the *IBM ILOG Concert Technology Reference Manual*.

Refer to Scheduler Overview for more information on how to share parameters among resources, and how the direct modification of parameters through the resource API may affect them. Also see Resource Usage Profiles.

See Also: IloResource, IloCapResource, IloEnforcementLevel

Constructor Summary	
public	`IloDiscreteEnergy()`
public	`IloDiscreteEnergy(IloDiscreteEnergyI * impl)`
public	`IloDiscreteEnergy(const IloEnv env, IloNum timeStep, IloNum capacity, const char * name=0)`

Method Summary	
public IloNum	`getEnergyMax(IloNum time) const`
public IloNum	`getEnergyMaxMax(IloNum timeMin, IloNum timeMax) const`
public IloNum	`getEnergyMaxMin(IloNum timeMin, IloNum timeMax) const`
public IloNum	`getEnergyMin(IloNum time) const`
public IloNum	`getEnergyMinMax(IloNum timeMin, IloNum timeMax) const`

public IloNum	getEnergyMinMin(IloNum timeMin, IloNum timeMax) const
public IloDiscreteEnergyI *	getImpl() const
public void	setEnergyMax(IloNum timeMin, IloNum timeMax, IloNum energy) const
public void	setEnergyMaxParam(const IloNumToNumStepFunction tfp) const
public void	setEnergyMin(IloNum timeMin, IloNum timeMax, IloNum energy) const
public void	setEnergyMinParam(const IloNumToNumStepFunction tfp) const

Inherited Methods from `IloCapResource`
addMaxTextureIgnoreInterval, addMaxTextureIgnoreInterval, addMaxTextureIgnoreIntervalOnDuration, addMaxTexturePeriodicIgnoreInterval, addMinTextureIgnoreInterval, addMinTextureIgnoreInterval, addMinTextureIgnoreIntervalOnDuration, addMinTexturePeriodicIgnoreInterval, emptyMaxTextureIgnoreIntervals, emptyMinTextureIgnoreIntervals, getCapacity, getImpl, getInitialOccupation, getInitialOccupationMax, getInitialOccupationMin, hasInitialOccupation, hasMaxTextureMeasurement, hasMinTextureMeasurement, removeMaxTextureIgnoreInterval, removeMaxTextureIgnoreInterval, removeMaxTextureIgnoreIntervalOnDuration, removeMaxTexturePeriodicIgnoreInterval, removeMinTextureIgnoreInterval, removeMinTextureIgnoreInterval, removeMinTextureIgnoreIntervalOnDuration, removeMinTexturePeriodicIgnoreInterval, setCapacity, setInitialOccupation, setInitialOccupation, setInitialOccupationParam, setInitialOccupationParam, setMaxTextureHeuristicBeta, setMaxTextureParam, setMaxTextureRandomGenerator, setMinTextureHeuristicBeta, setMinTextureParam, setMinTextureRandomGenerator, unsetMaxTextureRandomGenerator, unsetMinTextureRandomGenerator

Inherited Methods from `IloResource`
addCapacityEnforcementInterval, addTransitionTimeEnforcementInterval, areCalendarConstraintsIgnored, areCapacityConstraintsIgnored, arePrecedenceConstraintsIgnored, areSequenceConstraintsIgnored, areTransitionTimeConstraintsIgnored, getCalendar, getCalendarEnforcement, getCapacityEnforcement, getDurationEnforcement, getImpl, getPrecedenceEnforcement, getSequenceEnforcement, getTransitionTimeEnforcement, hasCalendar, ignoreCalendarConstraints, ignoreCapacityConstraints, ignorePrecedenceConstraints, ignoreSequenceConstraints, ignoreTransitionTimeConstraints, isCapacityResource, isContinuousReservoir, isDiscreteEnergy, isDiscreteResource, isKeptOpen, isReservoir, isStateResource, isUnaryResource, keepOpen, removeCapacityEnforcementInterval, removeTransitionTimeEnforcementInterval, setCalendar, setCalendarEnforcement, setCapacityEnforcement, setCapacityEnforcementIntervalsParam, setDurationEnforcement, setPrecedenceEnforcement, setResourceParam, setSequenceEnforcement, setTransitionTimeEnforcement, setTransitionTimeEnforcementIntervalsParam

Constructors

public **IloDiscreteEnergy**()

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

public **IloDiscreteEnergy**(IloDiscreteEnergyI * impl)

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

```
public IloDiscreteEnergy(const IloEnv env, IloNum timeStep, IloNum capacity, const
char * name=0)
```

This constructor creates a new instance of `IloDiscreteEnergy` and adds it to the set of resources managed in the given environment. The argument `timeStep` represents the default size of time buckets. The energy of the resource is limited to `capacity` for each time bucket.

Methods

```
public IloNum getEnergyMax(IloNum time) const
```

This member function returns the maximal energy that can be used at the given `time`.

```
public IloNum getEnergyMaxMax(IloNum timeMin, IloNum timeMax) const
```

This member function returns the maximal energy that can be used throughout the interval `[timeMin, timeMax)` (that is, the maximal value over the interval `[timeMin, timeMax)` of the maximal resource energy).

```
public IloNum getEnergyMaxMin(IloNum timeMin, IloNum timeMax) const
```

This member function returns the maximal value, over the interval `[timeMin, timeMax)`, of the minimal resource energy.

```
public IloNum getEnergyMin(IloNum time) const
```

This member function returns the minimal energy that must be used at the given `time`.

```
public IloNum getEnergyMinMax(IloNum timeMin, IloNum timeMax) const
```

This member function returns the minimal value, over the interval `[timeMin, timeMax)`, of the maximal resource energy.

```
public IloNum getEnergyMinMin(IloNum timeMin, IloNum timeMax) const
```

This member function returns the minimal energy that must be used throughout the interval `[timeMin, timeMax)` (that is, the minimal value over the interval `[timeMin, timeMax)` of the minimal resource energy).

```
public IloDiscreteEnergyI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public void setEnergyMax(IloNum timeMin, IloNum timeMax, IloNum energy) const
```

This member function states that the maximal energy per time bucket required throughout the interval [`timeMin, timeMax`) is at least `energy`.

```
public void setEnergyMaxParam(const IloNumToNumStepFunction tfp) const
```

This member function sets the maximal energy per time bucket over time to be the function defined in `tfp`.

```
public void setEnergyMin(IloNum timeMin, IloNum timeMax, IloNum energy) const
```

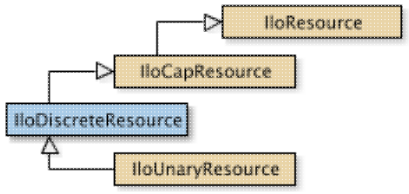
This member function states that the minimal energy per time bucket required throughout the interval [`timeMintimeMax`) is at least `energy`.

```
public void setEnergyMinParam(const IloNumToNumStepFunction tfp) const
```

This member function sets the minimal energy per time bucket over time to be the function defined in `tfp`.

Class IloDiscreteResource

Definition file: ilsched/ilodiscrete.h
Include file: <ilsched/iloscheduler.h>



An instance of the class `IloDiscreteResource` represents a resource of discrete capacity. Capacity can vary over time: at any given time, the capacity represents the number of copies or instances of the resource that are available. For example, the capacity might be the number of milling machines available in a manufacturing shop or the number of bricklayers at work on a construction site. By discrete, we mean that capacity is defined to be a positive integer.

Each activity may require some amount of the resource capacity, for example, one milling machine or three bricklayers. This requirement is represented by resource constraints.

Minimal Capacity

It is possible to constrain the capacity used so that it exceeds some minimal capacity over some interval of time. No solution will be found if at any point in time the minimal capacity exceeds the maximal capacity.

Parameter classes

Minimal and maximal capacity: (class `IloNumToNumStepFunction`)

These parameters describe the minimal and maximal capacities over time. They are directly modified by the member functions `IloDiscreteResource::setCapacityMin` and `IloDiscreteResource::setCapacityMax`.

`IloNumToNumStepFunction` is documented in the extensions section of the *IBM ILOG Concert Technology Reference Manual*.

Refer to Scheduler Overview for more information on how to share parameters among resources, and how the direct modification of parameters through the resource API may affect them.

See Also: IloEnforcementLevel, IloResource, IloCapResource

Constructor Summary	
public	`IloDiscreteResource()`
public	`IloDiscreteResource(IloDiscreteResourceI * impl)`
public	`IloDiscreteResource(const IloEnv env, IloNum capacity, const char * name=0)`

Method Summary	
public IloNum	`getCapacityMax(IloNum time) const`
public IloNum	`getCapacityMaxMax(IloNum timeMin, IloNum timeMax) const`
public IloNum	`getCapacityMaxMin(IloNum timeMin, IloNum timeMax) const`
public IloNum	`getCapacityMin(IloNum time) const`

public IloNum	getCapacityMinMax(IloNum timeMin, IloNum timeMax) const
public IloNum	getCapacityMinMin(IloNum timeMin, IloNum timeMax) const
public IloDiscreteResourceI *	getImpl() const
public void	setCapacityMax(IloNum timeMin, IloNum timeMax, IloNum capacity) const
public void	setCapacityMaxParam(const IloNumToNumStepFunction tfp) const
public void	setCapacityMin(IloNum timeMin, IloNum timeMax, IloNum capacity) const
public void	setCapacityMinParam(const IloNumToNumStepFunction tfp) const

Inherited Methods from `IloCapResource`
addMaxTextureIgnoreInterval, addMaxTextureIgnoreInterval, addMaxTextureIgnoreIntervalOnDuration, addMaxTexturePeriodicIgnoreInterval, addMinTextureIgnoreInterval, addMinTextureIgnoreInterval, addMinTextureIgnoreIntervalOnDuration, addMinTexturePeriodicIgnoreInterval, emptyMaxTextureIgnoreIntervals, emptyMinTextureIgnoreIntervals, getCapacity, getImpl, getInitialOccupation, getInitialOccupationMax, getInitialOccupationMin, hasInitialOccupation, hasMaxTextureMeasurement, hasMinTextureMeasurement, removeMaxTextureIgnoreInterval, removeMaxTextureIgnoreInterval, removeMaxTextureIgnoreIntervalOnDuration, removeMaxTexturePeriodicIgnoreInterval, removeMinTextureIgnoreInterval, removeMinTextureIgnoreInterval, removeMinTextureIgnoreIntervalOnDuration, removeMinTexturePeriodicIgnoreInterval, setCapacity, setInitialOccupation, setInitialOccupation, setInitialOccupationParam, setInitialOccupationParam, setMaxTextureHeuristicBeta, setMaxTextureParam, setMaxTextureRandomGenerator, setMinTextureHeuristicBeta, setMinTextureParam, setMinTextureRandomGenerator, unsetMaxTextureRandomGenerator, unsetMinTextureRandomGenerator

Inherited Methods from `IloResource`
addCapacityEnforcementInterval, addTransitionTimeEnforcementInterval, areCalendarConstraintsIgnored, areCapacityConstraintsIgnored, arePrecedenceConstraintsIgnored, areSequenceConstraintsIgnored, areTransitionTimeConstraintsIgnored, getCalendar, getCalendarEnforcement, getCapacityEnforcement, getDurationEnforcement, getImpl, getPrecedenceEnforcement, getSequenceEnforcement, getTransitionTimeEnforcement, hasCalendar, ignoreCalendarConstraints, ignoreCapacityConstraints, ignorePrecedenceConstraints, ignoreSequenceConstraints, ignoreTransitionTimeConstraints, isCapacityResource, isContinuousReservoir, isDiscreteEnergy, isDiscreteResource, isKeptOpen, isReservoir, isStateResource, isUnaryResource, keepOpen, removeCapacityEnforcementInterval, removeTransitionTimeEnforcementInterval, setCalendar, setCalendarEnforcement, setCapacityEnforcement, setCapacityEnforcementIntervalsParam, setDurationEnforcement, setPrecedenceEnforcement, setResourceParam, setSequenceEnforcement, setTransitionTimeEnforcement, setTransitionTimeEnforcementIntervalsParam

Constructors

public **IloDiscreteResource**()

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

```
public IloDiscreteResource(IloDiscreteResourceI * impl)
```

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

```
public IloDiscreteResource(const IloEnv env, IloNum capacity, const char * name=0)
```

This constructor creates a new instance of `IloDiscreteResource` and adds it to the set of resources managed in the environment. The theoretical capacity of the resource is `capacity`. The argument `name` is the name of the constructed `IloDiscreteResource`.

Methods

```
public IloNum getCapacityMax(IloNum time) const
```

This member function returns the maximal capacity that can be used at a given time.

```
public IloNum getCapacityMaxMax(IloNum timeMin, IloNum timeMax) const
```

This member function returns the maximal value of the maximal resource capacity over the interval [`timeMin, timeMax`).

```
public IloNum getCapacityMaxMin(IloNum timeMin, IloNum timeMax) const
```

This member function returns the maximal value of the minimal resource capacity over the interval [`timeMin, timeMax`).

```
public IloNum getCapacityMin(IloNum time) const
```

This member function returns the minimal capacity that must be used or is actually used at the given time.

```
public IloNum getCapacityMinMax(IloNum timeMin, IloNum timeMax) const
```

This member function returns the minimal value of the maximal resource capacity over the interval [`timeMin, timeMax`).

```
public IloNum getCapacityMinMin(IloNum timeMin, IloNum timeMax) const
```

This member function returns the minimal value of the minimal resource capacity over the interval [`timeMin, timeMax`).

```
public IloDiscreteResourceI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public void setCapacityMax(IloNum timeMin, IloNum timeMax, IloNum capacity) const
```

This member function states that at most `capacity` can be used throughout the interval `[timeMin, timeMax)`.

```
public void setCapacityMaxParam(const IloNumToNumStepFunction tfp) const
```

This member function sets `tfp` as the maximum capacity profile parameter of the invoking resource.

```
public void setCapacityMin(IloNum timeMin, IloNum timeMax, IloNum capacity) const
```

This member function states that at least `capacity` must be used throughout the interval `[timeMin, timeMax)`.

```
public void setCapacityMinParam(const IloNumToNumStepFunction tfp) const
```

This member function sets `tfp` as the minimum capacity profile parameter of the invoking resource.

Class IloGranularFunction

Definition file: ilsched/ilogfbase.h
Include file: <ilsched/iloscheduler.h>

An instance of `IloGranularFunction` holds the description of granular step-wise functions.

The granular function must respect the following properties:

- It is defined over the range `[xmin,xmax)`, and takes only integer values.
- It consists only of steps with non-negative values. These steps are closed on the left and open on the right.
- Its maximum value multiplied by the width `xmax-xmin` of its definition interval must be less than the largest integer value that can be represented by the machine's integer representation.

These properties are checked at extraction time, and an exception will be thrown if necessary.

The positive granularity parameter is optionally used as a scaling factor when computing the integral of the function. This allows limited representation of non-integer function values. This is particularly the case for integral expressions or constraints built with `IloGranularFunction` (see Functional and Integral Constraints on Resources for more information).

When computing the integral of the function over a given interval (for example, the start and end time of an activity), the result is divided by the granularity, and then rounded:

$$\text{Integral Value} = \text{ROUND}\left(\left(\int_{start}^{end} func\right) / \text{granularity}\right)$$

Note that the member function `IloGranularFunction::getValue` does not use the granularity, but returns the actual value stored in the function, without any scaling.

Four rounding modes are available when dividing by `granularity`. Refer to Functional and Integral Constraints on Resources for a detailed description of each rounding mode.

See Also: IloResource, IloGranularFunctionRoundingMode

Constructor Summary	
public	`IloGranularFunction()`
public	`IloGranularFunction(IloGranularFunctionI * impl)`
public	`IloGranularFunction(const IloEnv env, IloNum xmin, IloNum xmax, IloNum granularity=1.0, const char * name=0)`

Method Summary	
public IloNum	`getDefinitionIntervalMax() const`
public IloNum	`getDefinitionIntervalMin() const`
public IloNum	`getGranularity() const`
public IloGranularFunctionI *	`getImpl() const`
public IloGranularFunctionRoundingMode	`getRoundingMode() const`
public IloNum	`getValue(IloNum x) const`

public void	setRoundingMode(IloGranularFunctionRoundingMode rounding=IloGranularFunctionRoundUpward) const
public void	setValue(IloNum x1, IloNum x2, IloNum value) const

Inner Class
IloGranularFunction::Cursor

Constructors

public **IloGranularFunction**()

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

public **IloGranularFunction**(IloGranularFunctionI * impl)

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

public **IloGranularFunction**(const IloEnv env, IloNum xmin, IloNum xmax, IloNum granularity=1.0, const char * name=0)

This constructor creates a new instance of `IloGranularFunction`, with granularity equal to `granularity`. The initial function is on the interval `[xmin, xmax)`, and is set to a constant initial value of `granularity` over this interval.

Methods

public IloNum **getDefinitionIntervalMax**() const

This member function returns the right-most point of the interval of definition of the invoking granular function.

public IloNum **getDefinitionIntervalMin**() const

This member function returns the left-most point of the interval of definition of the invoking granular function.

public IloNum **getGranularity**() const

This member function returns the value of the granularity.

public IloGranularFunctionI * **getImpl**() const

This member function returns a pointer to the implementation object of the invoking handle.

public IloGranularFunctionRoundingMode **getRoundingMode**() const

This member function returns the current rounding mode of the invoking granular function.

```
public IloNum getValue(IloNum x) const
```

This member function returns the current value of the granular function at point `x`. This point must be inside the range `[xmin, xmax)`. Otherwise, an exception is thrown.

```
public void setRoundingMode(IloGranularFunctionRoundingMode
rounding=IloGranularFunctionRoundUpward) const
```

This member function selects the rounding mode that will be used when creating an integral constraint with the invoking granular function.

```
public void setValue(IloNum x1, IloNum x2, IloNum value) const
```

This member function sets the value of the granular function to `value` over the interval `[x1, x2)`. `x1` and `x2` must respect `xmin <= x1 < x2 <= xmax`, or an exception will be thrown.

In addition, the granular function must respect the properties listed in the Introduction to this class.

Class IloPrecedenceConstraint

Definition file: ilsched/iloactivity.h
Include file: <ilsched/iloscheduler.h>

Instances of the class `IloPrecedenceConstraint` are temporal constraints. These temporal constraints express precedence between activities in a schedule. (Other temporal constraints—instances of `IloTimeBoundConstraint`—express constraints on the time interval in which an activity is to be scheduled.)

This class inherits from the IBM ILOG Concert Technology class `IloConstraint`. That class is documented in the *IBM ILOG Concert Technology Reference Manual*.

Instances of this class are created by these member functions:

- `IloActivity::startsAfterStart`
- `IloActivity::startsAfterEnd`
- `IloActivity::endsAfterStart`
- `IloActivity::endsAfterEnd`
- `IloActivity::startsAtStart`
- `IloActivity::startsAtEnd`
- `IloActivity::endsAtStart`
- `IloActivity::endsAtEnd`

For more information, see `IloConstraint` in the IBM ILOG Concert Technology Reference Manual, and Temporal Relations.

See Also: IloActivity, IloActivityConstraintsParam

Constructor Summary	
public	`IloPrecedenceConstraint()`
public	`IloPrecedenceConstraint(IloPrecedenceConstraintI * impl)`

Method Summary	
public IloNum	`getDelay() const`
public IloNumVar	`getDelayVariable() const`
public IloActivity	`getFollowingActivity() const`
public IloPrecedenceConstraintI *	`getImpl() const`
public IloActivity	`getPrecedingActivity() const`
public IloPrecedenceConstraintType	`getType() const`
public IloBool	`hasDelayVariable() const`

Constructors

`public` **`IloPrecedenceConstraint`**`()`

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

`public` **`IloPrecedenceConstraint`**`(IloPrecedenceConstraintI * impl)`

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

Methods

```
public IloNum getDelay() const
```

This member function returns the delay of the invoking precedence constraint.

Example

The statement

```
m.add(act2.startsAfterEnd(act1, delay));
```

posts the constraint that at least the given `delay` must elapse between the end of the preceding activity `act1` and the start of the following activity `act2`.

```
public IloNumVar getDelayVariable() const
```

Returns the variable delay that the constraint was build with. Should not be called if the constraint has been build with a constant delay. Will assert in this case if we are in debug mode.

This member function returns the delay variable of the invoking precedence constraint.

```
public IloActivity getFollowingActivity() const
```

This member function returns the following activity of the invoking precedence constraint.

```
public IloPrecedenceConstraintI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloActivity getPrecedingActivity() const
```

This member function returns the preceding activity of the invoking precedence constraint.

```
public IloPrecedenceConstraintType getType() const
```

This member function returns the type of the invoking precedence constraint.

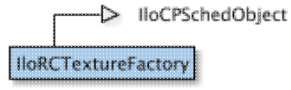
```
public IloBool hasDelayVariable() const
```

IloTrue if the constraint has been build with a variable delay

This member function returns `IloTrue` if the invoking precedence constraint has a delay variable. Otherwise, if the delay is constant, it returns `IloFalse`.

Class IloRCTextureFactory

Definition file: ilsched/ilotextureparami.h
Include file: <ilsched/iloscheduler.h>



RC Texture factory objects in Scheduler Concert Technology depend on the classes `IloRCTextureFactory` and `IloRCTextureFactoryI`. The class `IloRCTextureFactory` is the handle class. An instance of the class `IloRCTextureFactory` contains a data member (the handle pointer) that points to an instance of the class `IloRCTextureFactoryI` (the implementation object). If you define a new class of RC Texture factory with the macro `ILORCTEXTUREFACTORY0`, it will define the implementation class together with the corresponding virtual member function `IloRCTextureFactoryI::extract`, and a member function that returns an instance of the handle class `IloRCTextureFactory`.

Predefined Factories

The following functions, defined using the `ILORCTEXTUREFACTORY0` macro, return instances of RC Texture factory model objects.

```
IloRCTextureFactoryI *IloRCTextureESTFactory(IloEnv env);
```

This function returns a pointer to a factory object which, when extracted, corresponds to an `IlcRCTextureESTFactoryI`.

```
IloRCTextureFactoryI *IloRCTextureProbabilisticFactory(IloEnv env);
```

This function returns a pointer to a factory object which, when extracted, corresponds to an `IlcRCTextureProbabilisticFactoryI`.

```
IloRCTextureFactoryI *IloRCTextureTargetFactory(IloEnv env);
```

This function returns a pointer to a factory object which, when extracted, corresponds to an `IlcRCTextureTargetFactoryI`.

For more information, see Texture Measurements.

See Also: IloRCTextureFactoryI, ILORCTEXTUREFACTORY0, IlcRCTextureFactory

Constructor Summary	
public	`IloRCTextureFactory()`
public	`IloRCTextureFactory(IloRCTextureFactoryI * impl)`

Method Summary	
`public IloRCTextureFactoryI *`	`getImpl() const`

Constructors

`public` **`IloRCTextureFactory`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloRCTextureFactory`**`(IloRCTextureFactoryI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

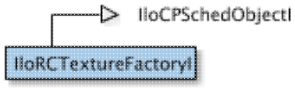
Methods

```
public IloRCTextureFactoryI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

Class IloRCTextureFactoryI

Definition file: ilsched/ilotextureparami.h
Include file: <ilsched/iloscheduler.h>



RC Texture factories in Scheduler Concert Technology depend on the classes `IloRCTextureFactoryI` and `IloRCTextureFactory`. The class `IloRCTextureFactoryI` is the implementation class. If you define a new class of factory with the macro `ILORCTEXTUREFACTORY0`, it will define this implementation class together with the corresponding virtual member function `IloRCTextureFactoryI::extract`, and with a member function that returns an instance of the handle class `IloRCTextureFactory`.

For more information, see Texture Measurements.

See Also: IloRCTextureFactory, ILORCTEXTUREFACTORY0, IlcRCTextureFactory

Method Summary	
`public virtual IlcRCTextureFactoryI *`	`extract(const IloSolver & solver) const`
`protected void`	`use(const IloSolver &, const IloExtractable &) const`

Methods

`public virtual IlcRCTextureFactoryI * **extract**(const IloSolver & solver) const`

This virtual function implements the extraction of the invoking factory into an `IlcRCTextureFactoryI*` by the `solver` given as argument. Note that this member function must be defined by using the macro `ILORCTEXTUREFACTORY0`.

`protected void **use**(const IloSolver &, const IloExtractable &) const`

This member function can only be called from within the member function `IloRCTextureFactoryI::extract` (that is, only in the code of a macro `ILORCTEXTUREFACTORY0`). It states that the invoking factory currently in the process of being extracted by the solver given as argument uses the extractable given as the second argument. As a consequence, the extractable given as the second argument will be immediately extracted by the solver.

Class IloRelocateActivityNHoodI

Definition file: ilsched/ilolnsgoals.h
Include file: <ilsched/iloscheduler.h>

An instance of this class represents an activity neighborhood.

See `IloComparator` and `IloPredicate` in the *IBM ILOG Solver Reference Manual* for more information.

See Also: IloSchedulerLargeNHoodI

Constructor Summary	
public	`IloRelocateActivityNHoodI(IloEnv env, IloComparator< IloActivity > comparator, IloPredicate< IloActivity > predicate, const char * name)`

Method Summary	
`public virtual IloSolution`	`defineSelected(IloSolver solver, IloInt index)`

Inherited Methods from `IloSchedulerLargeNHoodI`
`define, defineRestoreInfo, defineSelected, finalizeDelta, getCurrentSolution, getRestoreActivityDurationPredicate, getRestoreActivityEndPredicate, getRestoreActivityExternalPredicate, getRestoreActivityProcessingTimePredicate, getRestoreActivityStartPredicate, getRestoreExtractablePredicate, getRestoreInfo, getRestoreRCCapacityPredicate, getRestoreRCDirectPredecessorPredicate, getRestoreRCDirectSuccessorPredicate, getRestoreRCNextPredicate, getRestoreRCPrevPredicate, getRestoreRCSelectedPredicate, getRestoreRCSetupPredicate, getRestoreRCTeardownPredicate, isSelected, setRestoreActivityDurationPredicate, setRestoreActivityEndPredicate, setRestoreActivityExternalPredicate, setRestoreActivityProcessingTimePredicate, setRestoreActivityStartPredicate, setRestoreExtractablePredicate, setRestoreRCCapacityPredicate, setRestoreRCDirectPredecessorPredicate, setRestoreRCDirectSuccessorPredicate, setRestoreRCNextPredicate, setRestoreRCPrevPredicate, setRestoreRCSelectedPredicate, setRestoreRCSetupPredicate, setRestoreRCTeardownPredicate`

Constructors

`public` **`IloRelocateActivityNHoodI`**`(IloEnv env, IloComparator< IloActivity > comparator, IloPredicate< IloActivity > predicate, const char * name)`

This constructor creates an activity neighborhood.

The parameter `comparator` is used (if it is not an empty handle) to specify in which order the activities should be considered. When applied the comparator receives as argument the neighborhood.

The parameter `predicate` is used to specify which activities to consider. The size of this neighborhood is the number of activities in the current solution for which this predicate returns `IloTrue`. In case the predicate is an empty handle, the size of this neighborhood is the number of activities in the current solution.

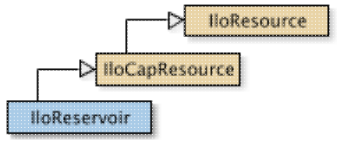
Methods

```
public virtual IloSolution defineSelected(IloSolver solver, IloInt index)
```

This pure virtual member function returns the set of decision variables, or instances of `IloExtractable`, on which to focus the search.

Class IloReservoir

Definition file: ilsched/iloreservoir.h
Include file: <ilsched/iloscheduler.h>



An instance of the class `IloReservoir` represents a resource for which activities can both provide capacity and also require capacity. If a maximal capacity of the reservoir is defined, then this maximal capacity, or level, will never be exceeded.

When the model of your problem represents an ongoing process, you may be faced with the fact that a reservoir level already exists. You can simply pass an initial level like that to the constructor of `IloReservoir` or use the `IloReservoir::setInitialLevel(IloNum)` member function.

The level of a reservoir can vary over time. You can define temporary maximal and minimal levels by using member functions of `IloReservoir`.

Parameter classes

Minimal and maximal capacity: (class `IloNumToNumStepFunction`)

These parameters describe the minimal and maximal levels over time. They are directly modified by the member functions `IloReservoir::setLevelMin` and `IloReservoir::setLevelMax`. `IloNumToNumStepFunction` is documented in the extensions section of the *IBM ILOG Concert Technology Reference Manual*.

Refer to Scheduler Overview for more information on how to share parameters among resources, and how the direct modification of parameters through the resource API may affect them.

See Also: IloEnforcementLevel, IloCapResource, IloResourceConstraint

Constructor Summary	
public	`IloReservoir()`
public	`IloReservoir(IloReservoirI * impl)`
public	`IloReservoir(const IloEnv env, IloNum capacity=IloMaxCapacityReservoir, IloNum initialLevel=0, const char * name=0)`

Method Summary	
public IloReservoirI *	`getImpl() const`
public IloNum	`getInitialLevel() const`
public IloNum	`getLevelMax(IloNum time) const`
public IloNum	`getLevelMaxMax(IloNum timeMin, IloNum timeMax) const`
public IloNum	`getLevelMaxMin(IloNum timeMin, IloNum timeMax) const`
public IloNum	`getLevelMin(IloNum time) const`
public IloNum	`getLevelMinMax(IloNum timeMin, IloNum timeMax) const`
public IloNum	`getLevelMinMin(IloNum timeMin, IloNum timeMax) const`
public void	`setInitialLevel(IloNum level) const`

public void	setLevelMax(IloNum timeMin, IloNum timeMax, IloNum level) const
public void	setLevelMaxParam(const IloNumToNumStepFunction tfp) const
public void	setLevelMin(IloNum timeMin, IloNum timeMax, IloNum level) const
public void	setLevelMinParam(const IloNumToNumStepFunction tfp) const

Inherited Methods from `IloCapResource`
addMaxTextureIgnoreInterval, addMaxTextureIgnoreInterval, addMaxTextureIgnoreIntervalOnDuration, addMaxTexturePeriodicIgnoreInterval, addMinTextureIgnoreInterval, addMinTextureIgnoreInterval, addMinTextureIgnoreIntervalOnDuration, addMinTexturePeriodicIgnoreInterval, emptyMaxTextureIgnoreIntervals, emptyMinTextureIgnoreIntervals, getCapacity, getImpl, getInitialOccupation, getInitialOccupationMax, getInitialOccupationMin, hasInitialOccupation, hasMaxTextureMeasurement, hasMinTextureMeasurement, removeMaxTextureIgnoreInterval, removeMaxTextureIgnoreInterval, removeMaxTextureIgnoreIntervalOnDuration, removeMaxTexturePeriodicIgnoreInterval, removeMinTextureIgnoreInterval, removeMinTextureIgnoreInterval, removeMinTextureIgnoreIntervalOnDuration, removeMinTexturePeriodicIgnoreInterval, setCapacity, setInitialOccupation, setInitialOccupation, setInitialOccupationParam, setInitialOccupationParam, setMaxTextureHeuristicBeta, setMaxTextureParam, setMaxTextureRandomGenerator, setMinTextureHeuristicBeta, setMinTextureParam, setMinTextureRandomGenerator, unsetMaxTextureRandomGenerator, unsetMinTextureRandomGenerator

Inherited Methods from `IloResource`
addCapacityEnforcementInterval, addTransitionTimeEnforcementInterval, areCalendarConstraintsIgnored, areCapacityConstraintsIgnored, arePrecedenceConstraintsIgnored, areSequenceConstraintsIgnored, areTransitionTimeConstraintsIgnored, getCalendar, getCalendarEnforcement, getCapacityEnforcement, getDurationEnforcement, getImpl, getPrecedenceEnforcement, getSequenceEnforcement, getTransitionTimeEnforcement, hasCalendar, ignoreCalendarConstraints, ignoreCapacityConstraints, ignorePrecedenceConstraints, ignoreSequenceConstraints, ignoreTransitionTimeConstraints, isCapacityResource, isContinuousReservoir, isDiscreteEnergy, isDiscreteResource, isKeptOpen, isReservoir, isStateResource, isUnaryResource, keepOpen, removeCapacityEnforcementInterval, removeTransitionTimeEnforcementInterval, setCalendar, setCalendarEnforcement, setCapacityEnforcement, setCapacityEnforcementIntervalsParam, setDurationEnforcement, setPrecedenceEnforcement, setResourceParam, setSequenceEnforcement, setTransitionTimeEnforcement, setTransitionTimeEnforcementIntervalsParam

Constructors

```
public IloReservoir()
```

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

```
public IloReservoir(IloReservoirI * impl)
```

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

```
public IloReservoir(const IloEnv env, IloNum capacity=IloMaxCapacityReservoir,
IloNum initialLevel=0, const char * name=0)
```

This constructor creates a new instance of `IloReservoir` and adds it to the set of resources managed in the given environment. The `capacity` argument expresses the capacity of the new reservoir. The capacity may be consumed by certain activities and produced by others. The argument `initialLevel` defines an initial amount in the reservoir at the time origin of the schedule environment. By default, the reservoir is assumed to be empty at the time origin; that is, the initial level is 0 (zero). If the argument `name` is defined, it is assigned as the name of the newly created reservoir.

Methods

```
public IloReservoirI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloNum getInitialLevel() const
```

This member function returns the initial level of the reservoir.

```
public IloNum getLevelMax(IloNum time) const
```

This member function returns the maximal reservoir level that is present at the given `time`.

```
public IloNum getLevelMaxMax(IloNum timeMin, IloNum timeMax) const
```

This member function returns the maximal value of the reservoir level over the interval [`timeMin, timeMax)` (that is, the maximal value over the interval [`timeMin, timeMax)` of the maximal reservoir level).

```
public IloNum getLevelMaxMin(IloNum timeMin, IloNum timeMax) const
```

This member function returns the maximal value of the minimal reservoir level, over the interval [`timeMin, timeMax)`.

```
public IloNum getLevelMin(IloNum time) const
```

This member function returns the minimal level of the invoking reservoir present at the given `time`.

```
public IloNum getLevelMinMax(IloNum timeMin, IloNum timeMax) const
```

This member function returns the minimal value over the interval [`timeMin, timeMax)` of the maximal reservoir level.

```
public IloNum getLevelMinMin(IloNum timeMin, IloNum timeMax) const
```

This member function returns the minimal reservoir level throughout the interval [`timeMin, timeMax)` (that is, the minimal value over the interval [`timeMin, timeMax)` of the minimal reservoir capacity).

```
public void setInitialLevel(IloNum level) const
```

This member function sets the initial level of the reservoir.

```
public void setLevelMax(IloNum timeMin, IloNum timeMax, IloNum level) const
```

This member function states that the level of the reservoir can be at most `level` throughout the interval [`timeMin, timeMax`).

```
public void setLevelMaxParam(const IloNumToNumStepFunction tfp) const
```

This member function sets the maximal reservoir level over time to be the function defined in `tfp`.

```
public void setLevelMin(IloNum timeMin, IloNum timeMax, IloNum level) const
```

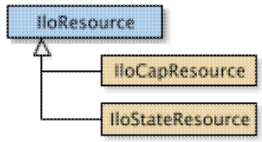
This member function states that the level of the reservoir must be at least `level` throughout the interval [`timeMin timeMax`).

```
public void setLevelMinParam(const IloNumToNumStepFunction tfp) const
```

This member function sets the minimal reservoir level over time to be the function defined in `tfp`.

Class IloResource

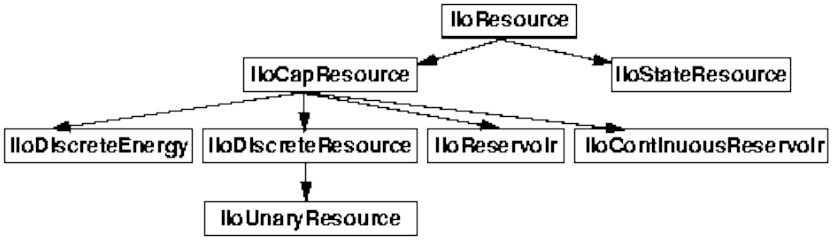
Definition file: ilsched/iloresource.h
Include file: <ilsched/iloscheduler.h>



A resource is represented by an instance of the abstract class `IloResource`. Activities in a schedule may require or provide resources.

This class inherits from the IBM® ILOG® Concert Technology class `IloExtractable`. That class is documented in the *IBM ILOG Concert Technology Reference Manual*.

There are several predefined subclasses of `IloResource`.



Parameter Classes

Calendar: (class `IloCalendar`)

This parameter attached to the resource allows expressing complex behavior as resource breaks and shifts, and efficiency along the schedule. It is modified by the member function `IloResource::setCalendar`.

Resource parameter: (class `IloResourceParam`)

This parameter describes the basic resource parameter of the resource. It is directly modified by the following member functions: `IloResource::ignoreCalendarConstraints`, `IloResource::ignoreCapacityConstraints`, `IloResource::ignorePrecedenceConstraints`, `IloResource::ignoreSequenceConstraints`, `IloResource::ignoreTransitionTimeConstraints`, `IloResource::setBreaksEnforcement`, `IloResource::setCapacityEnforcement`, `IloResource::setPrecedenceEnforcement`, `IloResource::setSequenceEnforcement`, `IloResource::setTransitionTimeEnforcement`, and `IloResource::keepOpen`.

Capacity enforcement intervals: (class `IloIntervalList`)

This parameter describes the set of time intervals on which the resource usage must be enforced. It is directly modified by the following member functions: `IloResource::addCapacityEnforcementInterval` and `IloResource::removeCapacityEnforcementInterval`.

Refer to Scheduler Overview for more information on how to share parameters among resources and how the direct modification of parameters through the resource API may affect them.

For more information, see Calendars, and Resource Usage Profiles .

See Also: IloAltResSet, IloCapResource, IloDiscreteEnergy, IloDiscreteResource, IloEnforcementLevel, IloReservoir, IloStateResource, IloResourceParam, IloResourceConstraint, IloTransitionTime, IloUnaryResource

Constructor Summary	
public	IloResource()
public	IloResource(IloResourceI * impl)

Method Summary	
public void	addCapacityEnforcementInterval(IloNum tmin, IloNum tmax, IloNum step=1) const
public void	addTransitionTimeEnforcementInterval(IloNum tmin, IloNum tmax) const
public IloBool	areCalendarConstraintsIgnored() const
public IloBool	areCapacityConstraintsIgnored() const
public IloBool	arePrecedenceConstraintsIgnored() const
public IloBool	areSequenceConstraintsIgnored() const
public IloBool	areTransitionTimeConstraintsIgnored() const
public IloCalendar	getCalendar() const
public IloEnforcementLevel	getCalendarEnforcement() const
public IloEnforcementLevel	getCapacityEnforcement() const
public IloEnforcementLevel	getDurationEnforcement() const
public IloResourceI *	getImpl() const
public IloEnforcementLevel	getPrecedenceEnforcement() const
public IloEnforcementLevel	getSequenceEnforcement() const
public IloEnforcementLevel	getTransitionTimeEnforcement() const
public IloBool	hasCalendar() const
public void	ignoreCalendarConstraints(IloBool ignored=IloTrue) const
public void	ignoreCapacityConstraints(IloBool ignored=IloTrue) const
public void	ignorePrecedenceConstraints(IloBool ignored=IloTrue) const
public void	ignoreSequenceConstraints(IloBool ignored=IloTrue) const
public void	ignoreTransitionTimeConstraints(IloBool ignored=IloTrue) const
public IloBool	isCapacityResource() const
public IloBool	isContinuousReservoir() const
public IloBool	isDiscreteEnergy() const
public IloBool	isDiscreteResource() const
public IloBool	isKeptOpen() const
public IloBool	isReservoir() const
public IloBool	isStateResource() const
public IloBool	isUnaryResource() const
public void	keepOpen(IloBool open=IloTrue) const
public void	removeCapacityEnforcementInterval(IloNum tmin, IloNum tmax) const

public void	removeTransitionTimeEnforcementInterval(IloNum tmin, IloNum tmax) const
public void	setCalendar(IloCalendar calendar) const
public void	setCalendarEnforcement(IloEnforcementLevel level) const
public void	setCapacityEnforcement(IloEnforcementLevel level) const
public void	setCapacityEnforcementIntervalsParam(const IloIntervalList it) const
public void	setDurationEnforcement(IloEnforcementLevel level) const
public void	setPrecedenceEnforcement(IloEnforcementLevel level) const
public void	setResourceParam(const IloResourceParam params) const
public void	setSequenceEnforcement(IloEnforcementLevel level) const
public void	setTransitionTimeEnforcement(IloEnforcementLevel level) const
public void	setTransitionTimeEnforcementIntervalsParam(const IloIntervalList it) const

Constructors

public **IloResource**()

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

public **IloResource**(IloResourceI * impl)

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

Methods

public void **addCapacityEnforcementInterval**(IloNum tmin, IloNum tmax, IloNum step=1) const

This member function specifies a new time interval [tmin,tmax) during which the usage of the invoking resources will be enforced. The step represents the time precision that must be considered when enforcing the usage of the invoking resource on that interval.

public void **addTransitionTimeEnforcementInterval**(IloNum tmin, IloNum tmax) const

This member function specifies a new time interval [tmin, tmax) during which the transition times on the invoking resources will be enforced.

public IloBool **areCalendarConstraintsIgnored**() const

This member function returns `IloTrue` if the calendar constraints of the invoking resource are to be ignored. Otherwise, it returns `IloFalse`.

```
public IloBool areCapacityConstraintsIgnored() const
```

This member function returns `IloTrue` if the usage of the invoking resource is to be ignored. Otherwise, it returns `IloFalse`.

```
public IloBool arePrecedenceConstraintsIgnored() const
```

This member function returns `IloTrue` if the precedence constraints of the invoking resource are to be ignored. Otherwise, it returns `IloFalse`.

```
public IloBool areSequenceConstraintsIgnored() const
```

This member function returns `IloTrue` if the sequence constraints of the invoking resource are to be ignored. Otherwise, it returns `IloFalse`.

```
public IloBool areTransitionTimeConstraintsIgnored() const
```

This member function returns `IloTrue` if the transition time on the invoking resource is to be ignored. Otherwise, it returns `IloFalse`.

```
public IloCalendar getCalendar() const
```

This member function returns the calendar attached to the invoking resource, if such an object exists.

```
public IloEnforcementLevel getCalendarEnforcement() const
```

This member function returns the enforcement level for the calendar of the invoking resource. Also see `IloResource::setCalendarEnforcement`.

```
public IloEnforcementLevel getCapacityEnforcement() const
```

This member function returns the capacity enforcement level of the invoking resource. See also: `IloResource::setCapacityEnforcement`.

```
public IloEnforcementLevel getDurationEnforcement() const
```

This member function returns the duration enforcement level of the invoking resource. See also: `IloResource::setDurationEnforcement`.

```
public IloResourceI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloEnforcementLevel getPrecedenceEnforcement() const
```

This member function returns the enforcement level for precedence relations of the invoking resource. See also: `IloResource::setPrecedenceEnforcement`.

```
public IloEnforcementLevel getSequenceEnforcement() const
```

This member function returns the enforcement level for sequencing relations of the invoking resource. See also: `IloResource::setSequenceEnforcement`.

```
public IloEnforcementLevel getTransitionTimeEnforcement() const
```

This member function returns the transition time enforcement level of the invoking resource. See also: `IloResource::setTransitionTimeEnforcement`.

```
public IloBool hasCalendar() const
```

This member function returns `IloTrue` if a calendar has been attached to the invoking resource. Otherwise, it returns `IloFalse`.

```
public void ignoreCalendarConstraints(IloBool ignored=IloTrue) const
```

This member function allows specifying if the calendar constraint of the invoking resources will be ignored. If the argument `ignored` is equal to `IloTrue`, it will behave as if no calendar is attached to the invoking resource.

```
public void ignoreCapacityConstraints(IloBool ignored=IloTrue) const
```

This member function allows specifying if the usage of the invoking resource by the activities will be ignored. If the argument `ignored` is equal to `IloTrue`, and the resource is a capacity resource, it will behave as if the capacity constraints on the resource are ignored. If the resource is a state resource, it will behave as if the required state constraints on the resource are ignored.

```
public void ignorePrecedenceConstraints(IloBool ignored=IloTrue) const
```

This member function allows specifying if the precedence relations defined on the invoking resource will be ignored. If the argument `ignored` is equal to `IloTrue`, it will behave as if the precedence constraints are ignored. Precedence relations are expressed with the member function `IloResourceConstraint::setSuccessor`.

```
public void ignoreSequenceConstraints(IloBool ignored=IloTrue) const
```

This member function allows specifying if the sequence relations defined on the invoking resource will be ignored. If the argument `ignored` is equal to `IloTrue`, it will behave as if the sequence constraints are ignored. Sequence relations are expressed with the member functions `IloResourceConstraint::setNext` and `IloResourceConstraint::setNotNext`.

```
public void ignoreTransitionTimeConstraints(IloBool ignored=IloTrue) const
```

This member function allows specifying if the transition times defined on the invoking resource will be ignored. If the argument `ignored` is equal to `IloTrue`, it will behave as if the transition time constraints are ignored.

```
public IloBool isCapacityResource() const
```

This member function distinguishes between the classes of resources available in Scheduler. It returns `IloTrue` if the invoking resource is an instance of the class `IloCapResource`. Otherwise, it returns `IloFalse`.

```
public IloBool isContinuousReservoir() const
```

This member function returns `IloTrue` if the invoking resource is an instance of the class `IloContinuousReservoir`. Otherwise, it returns `IloFalse`.

```
public IloBool isDiscreteEnergy() const
```

This member function returns `IloTrue` if the invoking resource is an instance of the class `IloDiscreteEnergy`. Otherwise, it returns `IloFalse`.

```
public IloBool isDiscreteResource() const
```

This member function returns `IloTrue` if the invoking resource is an instance of the class `IloDiscreteResource`. Otherwise, it returns `IloFalse`.

```
public IloBool isKeptOpen() const
```

This member function returns `IloTrue` if the invoking resource should be kept open. Otherwise, it returns `IloFalse`. See also: `IloResource::keepOpen`.

```
public IloBool isReservoir() const
```

This member function returns `IloTrue` if the invoking resource is an instance of the class `IloReservoir`. Otherwise, it returns `IloFalse`.

```
public IloBool isStateResource() const
```

This member function returns `IloTrue` if the invoking resource is an instance of the class `IloStateResource`. Otherwise, it returns `IloFalse`.

```
public IloBool isUnaryResource() const
```

This member function returns `IloTrue` if the invoking resource is an instance of the class `IloUnaryResource`. Otherwise, it returns `IloFalse`.

```
public void keepOpen(IloBool open=IloTrue) const
```

If the argument `open` is equal to `IloTrue`, this member function states that the invoking resource must be kept open when starting to solve the problem. It means that additional activities requiring or providing the invoking resource may be added during solving. Otherwise, if the argument `open` is equal to `IloFalse`, it states that all the activities requiring or providing the invoking resource will be defined in the model before starting to solve the problem. By default, it is supposed that all the activities requiring or providing the invoking resource are defined in the model before starting to solve the problem.

```
public void removeCapacityEnforcementInterval(IloNum tmin, IloNum tmax) const
```

This member function removes a time interval `[tmin,tmax)` during which the usage of the invoking resources had be enforced. It means that the resource usage does not need to be enforced on the interval `[tmin, tmax)`.

```
public void removeTransitionTimeEnforcementInterval(IloNum tmin, IloNum tmax) const
```

This member function removes a time interval `[tmin, tmax)` during which the transition times on the invoking resources were enforced.

```
public void setCalendar(IloCalendar calendar) const
```

This member function attaches the calendar `calendar` to the invoking resource. Notice that calendars can be shared between resources, and that `setCalendar` does not imply that a local copy of the calendar is made. One should be aware of the fact that any change to a shared calendar, using `IloCalendar` member functions with the calendar that comes from `getCalendar`, applies to all resources sharing the calendar.

```
public void setCalendarEnforcement(IloEnforcementLevel level) const
```

This member function allows specifying an enforcement level for the calendar of the resources. The level of enforcement of calendar constraints describes how the solver will enforce these calendar specifications. The semantics of these levels is solver dependent. The default enforcement level is `IloBasic`.

```
public void setCapacityEnforcement(IloEnforcementLevel level) const
```

This member function allows specifying an enforcement level for the usage of the resources depending on this parameter. The level of enforcement of resource usage describes how the solver will enforce this resource usage. The semantics of these levels is solver dependent. The default enforcement level is `IloBasic`.

```
public void setCapacityEnforcementIntervalsParam(const IloIntervalList it) const
```

This member function sets `it` as the list of capacity enforcement intervals of the invoking resource.

```
public void setDurationEnforcement(IloEnforcementLevel level) const
```

This member function allows specifying an enforcement level for the durations of the activities using the invoking resource. The duration level of enforcement describes how the solver will enforce these durations. The semantics

of these levels are solver dependent. The default enforcement level is `IloBasic`.

```
public void setPrecedenceEnforcement(IloEnforcementLevel level) const
```

This member function allows specifying an enforcement level for the precedence relations defined on the resources depending on this parameter. Precedence relations can be expressed with the member function `IloResourceConstraint::setSuccessor`. The level of enforcement of precedence relations describes how the solver will enforce these constraints. The semantics of these levels is solver dependent. The default enforcement level is `IloBasic`.

```
public void setResourceParam(const IloResourceParam params) const
```

This member function sets `params` as the new resource parameter.

```
public void setSequenceEnforcement(IloEnforcementLevel level) const
```

This member function allows specifying an enforcement level for the sequencing relations defined on the resources depending on this parameter. Sequencing relations are expressed by the member functions `IloResourceConstraint::setNext` and `IloResourceConstraint::setNotNext`. The level of enforcement of sequencing relations describes how the solver will enforce these relations. The semantics of these levels is solver dependent. The default enforcement level is `IloBasic`.

```
public void setTransitionTimeEnforcement(IloEnforcementLevel level) const
```

This member function allows specifying an enforcement level for the transition times defined on the invoking resource. Transition times can be associated with a resource using the class `IloTransitionTime`. The level of enforcement of transition times describes how the solver will enforce these relations. The semantics of these levels is solver dependent. The default enforcement level is `IloBasic`.

```
public void setTransitionTimeEnforcementIntervalsParam(const IloIntervalList it)
const
```

This member function sets `it` as the list of transition time enforcement intervals of the invoking resource.

Class IloResourceConstraint

Definition file: ilsched/iloresconstraint.h
Include file: <ilsched/iloscheduler.h>

Resource constraint.

Instances of the class `IloResourceConstraint` are resource constraints.

This class inherits from the IBM® ILOG® Concert Technology class `IloConstraint`. That class is documented in the *IBM ILOG Concert Technology Reference Manual*.

Instances of this class are created by these member functions:

- `IloActivity::requires`
- `IloActivity::provides`
- `IloActivity::consumes`
- `IloActivity::produces`
- `IloActivity::requiresNot`

The five elements of a resource constraint are an activity, an alternative resource set, a constant or variable demand, a time extent, and a rounding mode.

A resource constraint enforces the fact that on a certain time interval, exactly one resource from a set of alternatives that make up an instance of `IloAltResSet` is used to process the activity. The class `IloActivity` offers a set of member functions to create a resource constraint.

The demand type depends upon the resource. For example, a discrete capacity resource or a reservoir needs a positive numerical demand. The demand type for a state resource is a state or a set of states.

The time extent declares the period of time over which the availability of the resource is affected by the activity. Usually, the default value of the time extent is sufficient, since most activities will “require” a discrete capacity or a state resource from the start to the end of the activity. So in most cases, the time extent does not need to be declared. An activity can either *produce* or *consume* a reservoir.

The rounding mode tells the solver how to manage the use of the time buckets. Depending on the rounding mode, an activity that partially overlaps a time bucket is considered to either require the resource over the entire time bucket, or not require the resource during the time bucket at all.

Note that for resource constraints on continuous reservoirs, the time extent and the rounding mode have no meaning and are not used. This is because the time step of a timetable for a continuous reservoir is 1, so the returned resource constraint has no inward/outward rounding mode. Its time extent, which does not match any case of the enumeration `IloTimeExtent`, is not defined either.

The alternative resource set specifies the set of resources from which one resource must be selected. If the resource constraint has only one possible resource, then it requires an alternative resource set with a single element.

In order to better model interaction between activities and resources, it is possible to attach a specific calendar on a resource constraint. In such a case, the calendar of the resource constraint will subsume the calendar possibly attached to the corresponding resource. That is, even if a calendar is defined on the resource, it is not taken into account when a calendar is specified on the resource constraint.

The member functions of the class `IloResourceConstraint` allow retrieval of this data. In addition, the class `IloResourceConstraint` offers member functions to constrain the relative position of activities on the resource.

For more information, see Temporal Relations.

See Also: IloActivity, IloResource, IloTimeExtent, IloCapResource, IloActivityConstraintsParam, IloReservoir, IloStateResource, IloTransitionCost, IloTransitionTime, IloUnaryResource

Constructor Summary	
public	IloResourceConstraint()
public	IloResourceConstraint(IloResourceConstraintI * impl)

Method Summary	
public IloActivity	getActivity() const
public IloAltResSet	getAltResSet() const
public IloCalendar	getCalendar() const
public IloNum	getCapacity() const
public IloNum	getCapacityMax(const IloResource resource) const
public IloNum	getCapacityMin(const IloResource resource) const
public IloNumVar	getCapacityVariable() const
public IloNum	getDurationMax(const IloResource resource) const
public IloNum	getDurationMin(const IloResource resource) const
public IloNum	getEndMax(const IloResource resource) const
public IloNum	getEndMin(const IloResource resource) const
public IloResourceConstraintI *	getImpl() const
public IloResourceConstraint	getNext() const
public IloNum	getProcessingTimeMax(const IloResource resource) const
public IloNum	getProcessingTimeMin(const IloResource resource) const
public IloResource	getResource() const
public IloShape	getShape() const
public IloNum	getStartMax(const IloResource resource) const
public IloNum	getStartMin(const IloResource resource) const
public IloAny	getState() const
public IloAnySet	getStateSet() const
public IloAnySetVar	getStateSetVariable() const
public IloAnyVar	getStateVariable() const
public IloTimeExtent	getTimeExtent() const
public IloBool	hasAsNext(const IloResourceConstraint ct) const
public IloBool	hasCalendar() const
public IloBool	hasNext() const
public IloBool	hasShape() const

public IloBool	isCapacityConstraint() const
public IloBool	isInwardConstraint() const
public IloBool	isNegativeConstraint() const
public IloBool	isProvidingConstraint() const
public IloBool	isRejected(const IloResource resource) const
public IloBool	isSelected(const IloResource resource) const
public IloBool	isSetup() const
public IloBool	isStateConstraint() const
public IloBool	isStateSetConstraint() const
public IloBool	isSucceededBy(const IloResourceConstraint ct) const
public IloBool	isTeardown() const
public IloBool	isVariableResourceConstraint() const
public IloVariableSlopeShape	makeVariableSlopeShape(IloNumVar slope) const
public void	removeShape() const
public IloResourceSelectionConstraint	select(const IloResource resource) const
public void	setCalendar(IloCalendar calendar) const
public void	setCapacityMax(const IloResource resource, IloNum max) const
public void	setCapacityMin(const IloResource resource, IloNum min) const
public void	setDurationMax(const IloResource resource, IloNum max) const
public void	setDurationMin(const IloResource resource, IloNum min) const
public void	setEndMax(const IloResource resource, IloNum max) const
public void	setEndMin(const IloResource resource, IloNum min) const
public void	setNext(const IloResourceConstraint ct) const
public void	setNotNext(const IloResourceConstraint ct) const
public void	setNotSetup() const
public void	setNotTeardown() const
public void	setProcessingTimeMax(const IloResource resource, IloNum max) const
public void	setProcessingTimeMin(const IloResource resource, IloNum min) const
public void	setRejected(const IloResource resource) const
public void	setSelected(const IloResource resource) const
public void	setSetup() const
public void	setStartMax(const IloResource resource,

	IloNum max) const
public void	setStartMin(const IloResource resource, IloNum min) const
public void	setSuccessor(const IloResourceConstraint ct) const
public void	setTeardown() const
public void	unsetNext(const IloResourceConstraint ct) const
public void	unsetSelection(const IloResource resource) const
public void	unsetSetup() const
public void	unsetSuccessor(const IloResourceConstraint ct) const
public void	unsetTeardown() const

Constructors

public **IloResourceConstraint**()

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

public **IloResourceConstraint**(IloResourceConstraintI * impl)

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

Methods

public IloActivity **getActivity**() const

This member function returns the activity of the invoking resource constraint.

public IloAltResSet **getAltResSet**() const

This member function returns the instance of IloAltResSet associated with the invoking constraint. If the resource constraint was constructed with a single resource, rather than an IloAltResSet, this function returns an IloAltResSet containing only that resource. The alternative resource set, denoted ARS here, is local to the invoking resource constraint. That is, it is not shared with any other resource constraint unless a new resource constraint is subsequently as requiring ARS.

public IloCalendar **getCalendar**() const

This member function returns the calendar attached to the invoking resource constraint, if such an object exists.

public IloNum **getCapacity**() const

This member function returns the required or provided quantity of the invoking resource constraint.

```
public IloNum getCapacityMax(const IloResource resource) const
```

This member function returns the maximal required or provided quantity of the invoking resource constraint assuming `resource` is the selected resource.

```
public IloNum getCapacityMin(const IloResource resource) const
```

This member function returns the minimal required or provided quantity of the invoking resource constraint assuming `resource` is the selected resource.

```
public IloNumVar getCapacityVariable() const
```

This member function returns the variable representing the required or provided quantity of the invoking resource constraint.

```
public IloNum getDurationMax(const IloResource resource) const
```

This member function returns the longest duration of the activity of the invoking resource constraint assuming `resource` is the selected resource.

```
public IloNum getDurationMin(const IloResource resource) const
```

This member function returns the shortest duration of the activity of the invoking resource constraint assuming `resource` is the selected resource.

```
public IloNum getEndMax(const IloResource resource) const
```

This member function returns the latest end time of the activity of the invoking resource constraint assuming `resource` is the selected resource.

```
public IloNum getEndMin(const IloResource resource) const
```

This member function returns the earliest end time of the activity of the invoking resource constraint assuming `resource` is the selected resource.

```
public IloResourceConstraintI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloResourceConstraint getNext() const
```

This member function returns the next resource constraint that affects the availability of the resource after the invoking resource constraint.

```
public IloNum getProcessingTimeMax(const IloResource resource) const
```

This member function returns the longest processing time for the activity of the invoking resource constraint assuming `resource` is the selected resource.

```
public IloNum getProcessingTimeMin(const IloResource resource) const
```

This member function returns the shortest processing time for the activity of the invoking resource constraint assuming `resource` is the selected resource.

```
public IloResource getResource() const
```

This member function returns the resource of the invoking resource constraint. This method throws an exception if the invoking resource constraint has an alternative resource set that does not contain exactly one element.

```
public IloShape getShape() const
```

This function returns the instance of `IloShape` associated with the resource constraint.

```
public IloNum getStartMax(const IloResource resource) const
```

This member function returns the latest start time of the activity of the invoking resource constraint assuming `resource` is the selected resource.

```
public IloNum getStartMin(const IloResource resource) const
```

This member function returns the earliest start time of the activity of the invoking resource constraint assuming `resource` is the selected resource.

```
public IloAny getState() const
```

This member function returns the required state of the invoking resource constraint.

```
public IloAnySet getStateSet() const
```

This member function returns the required set of states of the invoking resource constraint.

```
public IloAnySetVar getStateSetVariable() const
```

This member function returns the variable representing the required set of states of the invoking resource constraint.

```
public IloAnyVar getStateVariable() const
```

This member function returns the variable representing the required state of the invoking resource constraint.

```
public IloTimeExtent getTimeExtent() const
```

This member function returns the time extent of the invoking resource constraint.

```
public IloBool hasAsNext(const IloResourceConstraint ct) const
```

This function returns `IloTrue` if and only if a next relation has been added with the member function `IloResourceConstraint::setNext`.

```
public IloBool hasCalendar() const
```

This member function returns `IloTrue` if a calendar has been attached to the invoking resource constraint. Otherwise, it returns `IloFalse`.

```
public IloBool hasNext() const
```

This function returns `IloTrue` if and only if there exists a unique resource constraint, `ct`, that has been constrained to be next after the invoking resource constraint with the member function `IloResourceConstraint::setNext`.

```
public IloBool hasShape() const
```

This function returns `IlcTrue` if a shape has been associated with the resource constraint

```
public IloBool isCapacityConstraint() const
```

This member function returns `IloTrue` if and only if the invoking resource constraint indicates that a quantity (and thus not a state) is required or provided.

```
public IloBool isInwardConstraint() const
```

This member function returns `IloTrue` if and only if the occupancy of the resource by the invoking constraint is to be rounded inward towards the nearest valid time that corresponds to a time step. This rounding is important only when one of the resource usage enforcement intervals of the resource has a time step greater than 1 (one).

```
public IloBool isNegativeConstraint() const
```

This member function returns `IloTrue` if and only if the invoking constraint was constructed by the member function `IloActivity::requiresNot`

```
public IloBool isProvidingConstraint() const
```

This member function returns `IloTrue` if and only if the invoking constraint was constructed by one of these member functions: `IloActivity::provides` or `IloActivity::produces`.

```
public IloBool isRejected(const IloResource resource) const
```

This member function returns `IloTrue` if `resource` cannot be selected for the activity associated with the invoking resource constraint (see member function `IloResourceConstraint::setRejected`). Otherwise, it returns `IloFalse`.

```
public IloBool isSelected(const IloResource resource) const
```

This member function returns `IloTrue` if resource must be selected by the invoking constraint (see member function `IloResourceConstraint::setRejected`). Otherwise, it returns `IloFalse`.

```
public IloBool isSetup() const
```

This member function returns `IloTrue` if and only if the invoking resource constraint has been constrained to be a setup resource constraint with the member function `IloResourceConstraint::setSetup`.

```
public IloBool isStateConstraint() const
```

This member function returns `IloTrue` if and only if the invoking resource constraint indicates that a single state (and thus not a quantity or one of a set of states) is required.

```
public IloBool isStateSetConstraint() const
```

This member function returns `IloTrue` if and only if the invoking resource constraint indicates that one of a set of states (and thus not a quantity or a single state) is required.

```
public IloBool isSucceededBy(const IloResourceConstraint ct) const
```

This member function returns IloTrue if and only if a successor relation has been added with the member function `IloResourceConstraint::setSuccessor`.

```
public IloBool isTeardown() const
```

This member function returns `IloTrue` if and only if the invoking resource constraint has been constrained to be a teardown resource constraint with the member function `IloResourceConstraint::setTeardown`.

```
public IloBool isVariableResourceConstraint() const
```

This member function returns `IloTrue` if and only if the invoking resource constraint has a variable representing the required quantity or state or provided quantity.

```
public IloVariableSlopeShape makeVariableSlopeShape(IloNumVar slope) const
```

This function associates an instance of `IloVariableSlopeShape` with the resource constraint. Shapes are only available on continuous reservoirs. If the resource constraint already has a shape, that shape is discarded and replaced by the newly created one. An exception will be thrown at extraction time if the minimal value of the slope variable is strictly negative.

See Also: IloShape, IloVariableSlopeShape

```
public void removeShape() const
```

This function removes the instance of `IloShape` associated with the resource constraint.

```
public IloResourceSelectionConstraint select(const IloResource resource) const
```

This member function creates and returns a constraint that specifies that `resource` must be selected for the invoking resource constraint.

The fact that a given resource `r` must not be selected for a resource constraint `rct` can be expressed by the negation of this constraint, as follows:

```
model.add(!rct.select(r))
```

```
public void setCalendar(IloCalendar calendar) const
```

This member function attaches the calendar `calendar` to the invoking resource constraint. Notice that calendars can be shared between resource constraints, and that `setCalendar` does not imply that a local copy of the calendar is made. One should be aware of the fact that any change to a shared calendar (using `IloCalendar` member functions with the calendar that comes from `getCalendar`) applies to all resource constraints sharing the calendar.

```
public void setCapacityMax(const IloResource resource, IloNum max) const
```

This member function states that if `resource` is the selected resource, then the maximal required or provided quantity of the invoking resource constraint is `max`.

```
public void setCapacityMin(const IloResource resource, IloNum min) const
```

This member function states that if `resource` is the selected resource, then the minimal required or provided quantity of the invoking resource constraint is `min`.

```
public void setDurationMax(const IloResource resource, IloNum max) const
```

This member function states that if `resource` is the selected resource, the longest duration of the activity of the invoking resource constraint is `max`.

```
public void setDurationMin(const IloResource resource, IloNum min) const
```

This member function states that if `resource` is the selected resource, then the shortest duration of the activity of the invoking resource constraint is `min`.

```
public void setEndMax(const IloResource resource, IloNum max) const
```

This member function states that if `resource` is the selected resource, then the latest end time of the activity of the invoking resource constraint is `max`.

```
public void setEndMin(const IloResource resource, IloNum min) const
```

This member function states that if `resource` is the selected resource, then the earliest end time of the activity of the invoking resource constraint is `min`.

```
public void setNext(const IloResourceConstraint ct) const
```

This member function states that if the invoking resource constraint and `ct` affect the same resource, then `ct` is next after the invoking resource constraint. There cannot exist any other resource constraint that affects the availability of the resource and starts or finishes between the end of the invoking resource constraint and the start of `ct`.

At extraction time if there is an empty intersection between the alternative resource set for the invoking resource constraint and the alternative resource set for `ct`, this function has no effect, as the condition that the two resource constraints must affect the same resource cannot be true.

```
public void setNotNext(const IloResourceConstraint ct) const
```

This member function states that `ct` is not next after the invoking resource constraint. This means that if the invoking resource constraint and `ct` both affect the availability of the resource, then there must be another resource constraint that affects the availability of the resource and it starts or finishes between the end of the invoking resource constraint and the start of `ct`.

At extraction time if there is an empty intersection between the alternative resource set for the invoking resource constraint and the alternative resource set for `ct`, this function has no effect, as the condition that the two resource constraints must affect the same resource cannot be true.

```
public void setNotSetup() const
```

This member function states that the invoking resource constraint is not a setup resource constraint. This means that if the invoking resource constraint affects the availability of the resource, there must be a different resource constraint previous to it that also affects the availability of the resource.

```
public void setNotTeardown() const
```

This member function states that the invoking resource constraint is not a teardown resource constraint. This means that if the invoking resource affects the availability of the resource, then there must be a different resource constraint that also affects the availability of the resource and occurs after the invoking resource constraint.

```
public void setProcessingTimeMax(const IloResource resource, IloNum max) const
```

This member function states that if `resource` is the selected resource, then the longest processing time for the activity of the invoking resource constraint is `max`.

```
public void setProcessingTimeMin(const IloResource resource, IloNum min) const
```

This member function states that if `resource` is the selected resource, the shortest processing time for the activity of the invoking resource constraint is `max`.

```
public void setRejected(const IloResource resource) const
```

This member function states that it is not possible for `resource` to be selected. If `resource` is not a member of the alternative resource set of the invoking resource constraint, this method has no effect.

```
public void setSelected(const IloResource resource) const
```

This member function states that `resource` must be selected for the activity associated with the invoking constraint. If `resource` is not a member of the alternative resource set of the invoking resource constraint, this method will result in an inconsistent model. Any attempt to solve such a model will fail.

```
public void setSetup() const
```

This member function states that the invoking resource constraint is a setup resource constraint. This means that if the invoking resource constraint affects the availability of the resource, no other resource constraint that affects the availability of the resource can be previous to it.

```
public void setStartMax(const IloResource resource, IloNum max) const
```

This member function states that if `resource` is the selected resource, the latest start time of the activity of the invoking resource constraint is `max`.

```
public void setStartMin(const IloResource resource, IloNum min) const
```

This member function states that if `resource` is the selected resource, the earliest start time of the activity of the invoking resource constraint is `min`.

```
public void setSuccessor(const IloResourceConstraint ct) const
```

This member function states that the invoking resource constraint has the resource constraint `ct` as successor. This means that if the invoking resource constraint and `ct` both affect the availability of the resource, then the activity of `ct` is constrained to execute after the activity of the invoking resource constraint.

At extraction time if there is an empty intersection between the alternative resource set for the invoking resource constraint and the alternative resource set for `ct`, this function has no effect, as the condition that the two resource constraints must affect the same resource cannot be true.

```
public void setTeardown() const
```

This member function states that the invoking resource constraint is a teardown resource constraint. This means that if the invoking resource constraint affects the availability of the resource, then no resource constraint can exist that affects the availability of the resource after the invoking resource constraint.

```
public void unsetNext(const IloResourceConstraint ct) const
```

This member function removes all information regarding the next relation between the invoking resource constraint and `ct`. That is, the information specified by both of the member functions `IloResourceConstraint::setNext` and `IloResourceConstraint::setNotNext` is removed from the model.

```
public void unsetSelection(const IloResource resource) const
```

This member function states that `resource` is a possible resource for the activity associated with the invoking constraint, but not necessarily the one selected. This member function removes the information added to the model by the `IloResourceConstraint::setSelected` and `IloResourceConstraint::setRejected` member functions.

```
public void unsetSetup() const
```

This member function removes all setup information from the invoking resource constraint. That is, this member function removes all information added to the model by the member functions `IloResourceConstraint::setSetup` and `setNotSetup`.

```
public void unsetSuccessor(const IloResourceConstraint ct) const
```

This member function removes all information regarding the successor relation between the invoking resource constraint and `ct`. That is, the information specified by the member function `setSuccessor(ct)` is removed from the model.

```
public void unsetTeardown() const
```

This member function removes all teardown information from the invoking resource constraint. That is, this member function removes all information added to the model by the member functions `IloResourceConstraint::setTeardown` and `setNotTeardown`.

Class IloResourceConstraintIterator

Definition file: ilsched/iloresconstraint.h
Include file: <ilsched/iloscheduler.h>

An instance of this class traverses the set of non-alternative resource constraints defined on an environment.

Note

This class is provided for compatibility with the `IloIterator<IloResourceConstraint>` class of Scheduler 5.0 and Scheduler 5.1. In the current version of the library, `IloIterator<IloResourceConstraint>` traverses all resource constraints (whether or not they are defined on an alternative resource set) defined on an environment.

For more information, see `IloIterator<IloResourceConstraint>` in the Concert Reference Manual.

See Also: IloAltResConstraintIterator

Constructor Summary	
public	IloResourceConstraintIterator(const IloEnv env)

Method Summary	
public IloBool	ok()
public IloResourceConstraint	operator*()
public void	operator++()

Constructors

`public` **`IloResourceConstraintIterator`**`(const IloEnv env)`

This constructor creates an iterator to traverse all the non-alternative resource constraints that are defined on the environment `env`.

Methods

`public IloBool` **`ok`**`()`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all the non-alternative resource constraints have been scanned by the iterator.

`public IloResourceConstraint` **`operator*`**`()`

This operator returns the current instance of `IloResourceConstraint`, the one to which the invoking iterator points. This operator must not be called if the iterator does not point to a valid position, that is, one to which the member function `IloResourceConstraintIterator::ok` returns `IloFalse`.

```
public void operator++()
```

This left-increment operator shifts the current position of the iterator to the next non-alternative instance of `IloResourceConstraint`.

Class IloResourceParam

Definition file: ilsched/iloresourceparam.h
Include file: <ilsched/iloscheduler.h>

Parameters are used to change the default behavior of activities and resources.

Several kinds of constraints can be posted on a resource:

- calendar constraints
- resource usage constraints
- precedence relations between resource constraints on that resource
- sequencing relations among resource constraints on that resource
- transition times.

Instances of `IloResourceParam` are used to specify whether and how these constraints posted on resources must be taken into account during the search.

This class inherits from the IBM® ILOG® Concert Technology class `IloExtractable`. That class is documented in the *IBM ILOG Concert Technology Reference Manual*.

For more information, see Calendars, Parameter Classes, Temporal Relations, Parameters Organized by Function, Resource Usage Profiles, Transition Costs, Transition Times, and Resource Enforcement as Global Constraint Declaration.

See Also: IloResource, IloEnforcementLevel

Constructor Summary	
public	`IloResourceParam()`
public	`IloResourceParam(IloResourceParamI * impl)`
public	`IloResourceParam(const IloEnv env, const char * name=0)`

Method Summary	
public IloBool	`areCalendarConstraintsIgnored() const`
public IloBool	`areCapacityConstraintsIgnored() const`
public IloBool	`arePrecedenceConstraintsIgnored() const`
public IloBool	`areSequenceConstraintsIgnored() const`
public IloBool	`areTransitionTimeConstraintsIgnored() const`
public IloEnforcementLevel	`getCalendarEnforcement() const`
public IloEnforcementLevel	`getCapacityEnforcement() const`
public IloEnforcementLevel	`getDurationEnforcement() const`
public IloResourceParamI *	`getImpl() const`
public IloEnforcementLevel	`getPrecedenceEnforcement() const`
public IloEnforcementLevel	`getSequenceEnforcement() const`
public IloEnforcementLevel	`getTransitionTimeEnforcement() const`
public void	`ignoreCalendarConstraints(IloBool ignore=IloTrue) const`

public void	ignoreCapacityConstraints(IloBool ignore=IloTrue) const
public void	ignorePrecedenceConstraints(IloBool ignore=IloTrue) const
public void	ignoreSequenceConstraints(IloBool ignore=IloTrue) const
public void	ignoreTransitionTimeConstraints(IloBool ignore=IloTrue) const
public IloBool	isKeptOpen() const
public void	keepOpen(IloBool open=IloTrue) const
public void	setCalendarEnforcement(IloEnforcementLevel level) const
public void	setCapacityEnforcement(IloEnforcementLevel level) const
public void	setDurationEnforcement(IloEnforcementLevel level) const
public void	setPrecedenceEnforcement(IloEnforcementLevel level) const
public void	setSequenceEnforcement(IloEnforcementLevel level) const
public void	setTransitionTimeEnforcement(IloEnforcementLevel level) const

Constructors

`public` **`IloResourceParam`**`()`

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

`public` **`IloResourceParam`**`(IloResourceParamI * impl)`

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

`public` **`IloResourceParam`**`(const IloEnv env, const char * name=0)`

This constructor creates a new instance of `IloResourceParam` in the given environment `env`.

Methods

`public IloBool` **`areCalendarConstraintsIgnored`**`() const`

This member function returns `IloTrue` if the calendar of the resources depending on this parameter is to be ignored. Otherwise, it returns `IloFalse`.

`public IloBool` **`areCapacityConstraintsIgnored`**`() const`

This member function returns `IloTrue` if the usage of the resources depending on this parameter is to be ignored. Otherwise, it returns `IloFalse`.

```
public IloBool arePrecedenceConstraintsIgnored() const
```

This member function returns `IloTrue` if the precedence relations defined on the resources depending on this parameter are to be ignored. Otherwise, it returns `IloFalse`. Precedence relations are expressed with the member function `IloResourceConstraint::setSuccessor`.

```
public IloBool areSequenceConstraintsIgnored() const
```

This member function returns `IloTrue` if the sequencing relations defined on the resources depending on this parameter are to be ignored. Otherwise, it returns `IloFalse`. Sequencing relations are expressed with the following member functions: `IloResourceConstraint::setNext`, `IloResourceConstraint::setNotNext`, `IloResourceConstraint::setSetup`, `IloResourceConstraint::setNotSetup`, `IloResourceConstraint::setTeardown`, and `IloResourceConstraint::setNotTeardown`.

```
public IloBool areTransitionTimeConstraintsIgnored() const
```

This member function returns `IloTrue` if the transition time defined on the resources depending on this parameter is to be ignored. Otherwise, it returns `IloFalse`.

```
public IloEnforcementLevel getCalendarEnforcement() const
```

This member function returns the enforcement level for calendar constraint of the invoking parameter. See also: `IloResourceParam::getCalendarEnforcement`

```
public IloEnforcementLevel getCapacityEnforcement() const
```

This member function returns the capacity enforcement level of the invoking parameter. See also: `IloResourceParam::setCapacityEnforcement`

```
public IloEnforcementLevel getDurationEnforcement() const
```

This member function returns the enforcement level on the duration of the invoking parameter. See also: `IloResourceParam::setDurationEnforcement`

```
public IloResourceParamI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloEnforcementLevel getPrecedenceEnforcement() const
```

This member function returns the enforcement level for precedence relations of the invoking parameter. See also: `IloResourceParam::setPrecedenceEnforcement`

```
public IloEnforcementLevel getSequenceEnforcement() const
```

This member function returns the enforcement level for sequencing relations of the invoking parameter. See also: IloResourceParam::setSequenceEnforcement

```
public IloEnforcementLevel getTransitionTimeEnforcement() const
```

This member function returns the enforcement level for transition time relations of the invoking parameter. See also: IloResourceParam::setTransitionTimeEnforcement

```
public void ignoreCalendarConstraints(IloBool ignore=IloTrue) const
```

This member function allows you to specify whether the calendar constraints of the resources depending on this parameter will be ignored. If the argument ignored is equal to IloTrue, it will behave as if no calendar is attached to the resources depending on this parameter.

```
public void ignoreCapacityConstraints(IloBool ignore=IloTrue) const
```

This member function allows you to specify whether the capacity or state constraints on the resources depending on this parameter will be ignored. If the argument ignore is equal to IloTrue, and the resource is a capacity resource, it will behave as if the capacity constraints on the resource are ignored. If the resource is a state resource, it will behave as if the required state constraints on the resource are ignored.

```
public void ignorePrecedenceConstraints(IloBool ignore=IloTrue) const
```

This member function allows you to specify whether the precedence relations defined on the resources depending on this parameter will be ignored. If the argument ignore is equal to IloTrue, it will behave as if the precedence relations are ignored. Precedence relations are expressed with the member function IloResourceConstraint::setSuccessor.

```
public void ignoreSequenceConstraints(IloBool ignore=IloTrue) const
```

This member function allows you to specify whether the sequence relations defined on the resources depending on this parameter will be ignored. If the argument ignore is equal to IloTrue, it will behave as if the sequence relations are ignored. Sequence relations are expressed with the member functions IloResourceConstraint::setNext and IloResourceConstraint::setNotNext.

```
public void ignoreTransitionTimeConstraints(IloBool ignore=IloTrue) const
```

This member function allows you to specify whether the transition times defined on the resources depending on this parameter will be ignored. If the argument ignore is equal to IloTrue, it will behave as if the transition time constraints are ignored.

```
public IloBool isKeptOpen() const
```

This member function returns IloTrue if the resource depending on this parameter should be kept open. Otherwise, it returns IloFalse. See also: IloResourceParam::keepOpen

```
public void keepOpen(IloBool open=IloTrue) const
```

If the argument `open` is equal to `IloTrue`, this member function states that the resources depending on this parameter must be kept open when starting to solve the problem. It means that additional activities requiring or providing these resources may be added during solving. Otherwise, if the argument `open` is equal to `IloFalse`, it states that all the activities requiring or providing the resources depending on this parameter are known before starting to solve the problem. By default, it is supposed that all the activities requiring or providing the resources are known before starting to solve the problem.

```
public void setCalendarEnforcement(IloEnforcementLevel level) const
```

This member function allows specifying an enforcement level for the calendar of the resources. The level of enforcement of calendar constraints describes how the solver will enforce these calendar constraints. The semantics of these levels is solver dependent. The default enforcement level is `IloBasic`. For more information on the enforcement levels and how they are interpreted in Scheduler Engine, see `IloEnforcementLevel` and Resource Enforcement as Global Constraint Declaration.

```
public void setCapacityEnforcement(IloEnforcementLevel level) const
```

This member function allows specifying an enforcement level for the usage of the resources depending on this parameter. The level of enforcement of resource usage describes how the solver will enforce this resource usage. The semantics of these levels is solver dependent. The default enforcement level is `IloBasic`. For more information on the enforcement levels and how they are interpreted in Scheduler Engine, see `IloEnforcementLevel` and *Resource Enforcement as Global Constraint Declaration*.

```
public void setDurationEnforcement(IloEnforcementLevel level) const
```

This member function allows specifying an enforcement level for the duration of activities using the resources depending on this parameter. The level of enforcement describes how the solver will enforce these durations. The semantics of these levels is solver dependent. The default enforcement level is `IloBasic`. For more information on the enforcement levels and how they are interpreted in Scheduler Engine, see `IloEnforcementLevel` and *Resource Enforcement as Global Constraint Declaration*.

```
public void setPrecedenceEnforcement(IloEnforcementLevel level) const
```

This member function allows specifying an enforcement level for the precedence relations defined on the resources depending on this parameter. Precedence relations can be expressed with the member function `IloResourceConstraint::setSuccessor`. The level of enforcement of precedence relations describes how the solver will enforce these constraints. The semantics of these levels is solver dependent. The default enforcement level is `IloBasic`. For more information on the enforcement levels and how they are interpreted in Scheduler Engine, see `IloEnforcementLevel` and *Resource Enforcement as Global Constraint Declaration*.

```
public void setSequenceEnforcement(IloEnforcementLevel level) const
```

This member function allows specifying an enforcement level for the sequencing relations defined on the resources depending on this parameter. Sequencing relations are expressed by the member functions `IloResourceConstraint::setNext` and `IloResourceConstraint::setNotNext`. The level of enforcement of sequencing relations describes how the solver will enforce these relations. The semantics of these levels is solver dependent. The default enforcement level is `IloBasic`. For more information on the enforcement levels and how they are interpreted in Scheduler Engine, see `IloEnforcementLevel` and *Resource Enforcement as Global Constraint Declaration*.

```
public void setTransitionTimeEnforcement(IloEnforcementLevel level) const
```

This member function allows specifying an enforcement level for the transition times defined on the resources depending on this parameter. Transition times can be associated with a resource with the member functions of the class `IloTransitionTime`. The level of enforcement of transition times describes how the solver will enforce these relations. The semantics of these levels is solver dependent. The default enforcement level is `IloBasic`. For more information on the enforcement levels and how they are interpreted in Scheduler Engine, see `IloEnforcementLevel` and *Resource Enforcement as Global Constraint Declaration*.

Class IloResourceValue

Definition file: ilsched/iloresvaluect.h

This class allows associating an integer value (or an integer variable) for each resource in an environment. A default can be specified for all the resources that are not explicitely associated a value.

Constructor Summary	
public	IloResourceValue()
public	IloResourceValue(IloResourceValueI * impl)
public	IloResourceValue(IloEnv env, const char * name=0)

Method Summary	
public IloResourceValueI *	getImpl() const
public IloIntExprArg	operator[](IloResourceConstraint rct)
public void	setDefault(IloInt defaultValue)
public void	setValue(IloResource resource, IloIntVar var)
public void	setValue(IloResource resource, IloInt value)

Constructors

`public` **`IloResourceValue`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloResourceValue`**`(IloResourceValueI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

`public` **`IloResourceValue`**`(IloEnv env, const char * name=0)`

This constructor creates a new instance of `IloResourceValue` to associate an integer value (or integer variable) with each resource in the environment.

Methods

`public IloResourceValueI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public IloIntExprArg` **`operator[]`**`(IloResourceConstraint rct)`

This member function returns an integer expression whose value is the value of the resource that is selected by the resource constraint given as parameter.

`public void` **`setDefault`**`(IloInt defaultValue)`

This member function defines a default value associated with each resource in the environment. It means that the invoking `IloResourceValue` will associate the value `defaultValue` with each resource but those which have explicitely been associated a value thanks to the member functions `setValue`.

```
public void setValue(IloResource resource, IloIntVar var)
```

This member function associates the variable value `var` with resource `resource`.

```
public void setValue(IloResource resource, IloInt value)
```

This member function associates the value `value` with resource `resource`.

Class IloSchedulerEnv

Definition file: ilsched/iloschedenv.h
Include file: <ilsched/iloscheduler.h>

The class `IloSchedulerEnv` is the repository of all the default parameters used when creating new modeling objects. There can be at most one `IloSchedulerEnv` defined on a given instance of `IloEnv`.

Avoiding Overflows

In order to avoid overflows in computations by Scheduler Engine, an efficient policy is to limit the domain of all extracted integer variables to an interval of `[-L,L]`. For example, a variable whose lower bound is 0 and upper bound is `IloInfinity` will be extracted as a variable with the domain `[0,L]`. This limit `L` is set by the member function `IloSchedulerEnv::setIntMaxAtExtraction`. At extraction, this limit is used only for variables that are used by Scheduler objects or constraints, as the start and end times, the capacity variables or the precedence delays. Please note that there is an exception for the cost sum variable (class `IloTransitionCost`). For this variable, the limit `L` is not applied because there is usually no overflow with this variable. The default value of this limit is `IloIntMax/2`, which avoids overflows in most cases.

For more information, see `IloNumToNumStepFunction`, `IloNumToAnySetStepFunction`, and `IloIntervalList` in the extensions section of the *IBM ILOG Concert Technology Reference Manual*, Parameter Classes.

See Also: IloActivityBasicParam, IloActivityBreakParam, IloActivityConstraintsParam, IloActivityOverlapParam, IloActivityShiftParam, IloResourceParam

Constructor Summary	
public	`IloSchedulerEnv(const IloEnv env)`
public	`IloSchedulerEnv(IloSchedulerEnvI * impl)`

Method Summary	
public IloActivityBasicParam	`getActivityBasicParam() const`
public IloActivityBreakParam	`getActivityBreakParam() const`
public IloActivityConstraintsParam	`getActivityConstraintsParam() const`
public IloActivityOverlapParam	`getActivityOverlapParam() const`
public IloActivityShiftParam	`getActivityShiftParam() const`
public IloIntervalList	`getBreakListParam() const`
public IloIntervalList	`getCapacityEnforcementIntervalsParam() const`
public IloNumToNumStepFunction	`getCapacityMaxParam() const`
public IloNumToNumStepFunction	`getCapacityMinParam() const`
public IloEnv	`getEnv() const`
public IloNum	`getHorizon() const`
public IloSchedulerEnvI *	`getImpl() const`
public IloNumToNumStepFunction	`getInitialOccupationParam() const`
public IloInt	`getIntMaxAtExtraction() const`
public IloTextureParam	`getMaxTextureParam() const`
public IloTextureParam	`getMinTextureParam() const`

public IloIntervalList	getMustBeInUseParam() const
public IloNum	getOrigin() const
public IloNumToAnySetStepFunction	getPossibleStatesParam() const
public IloEnforcementLevel	getPrecedenceEnforcement() const
public IloResourceParam	getResourceParam() const
public IloIntervalList	getTransitionTimeEnforcementIntervalsParam() const
public void	setActivityBasicParam(const IloActivityBasicParam param) const
public void	setActivityBreakParam(const IloActivityBreakParam param) const
public void	setActivityConstraintsParam(const IloActivityConstraintsParam param) const
public void	setActivityOverlapParam(const IloActivityOverlapParam param) const
public void	setActivityShiftParam(const IloActivityShiftParam param) const
public void	setBreakListParam(const IloIntervalList param) const
public void	setCapacityEnforcementIntervalsParam(const IloIntervalList param) const
public void	setCapacityMaxParam(const IloNumToNumStepFunction param) const
public void	setCapacityMinParam(const IloNumToNumStepFunction param) const
public void	setHorizon(IloNum horizon) const
public void	setInitialOccupationParam(const IloNumToNumStepFunction param) const
public void	setIntMaxAtExtraction(IloInt max) const
public void	setMaxTextureParam(const IloTextureParam param) const
public void	setMinTextureParam(const IloTextureParam param) const
public void	setMustBeInUseParam(const IloIntervalList param) const
public void	setOrigin(IloNum origin) const
public void	setPossibleStatesParam(const IloNumToAnySetStepFunction param) const
public void	setPrecedenceEnforcement(IloEnforcementLevel level) const
public void	setResourceParam(const IloResourceParam param) const
public void	setTransitionTimeEnforcementIntervalsParam(const IloIntervalList param) const

Constructors

```
public IloSchedulerEnv(const IloEnv env)
```

This constructor creates a new instance of `IloSchedulerEnv` if none currently exists on the given instance of `IloEnv`. If a scheduler environment has already been created on the environment, then the new handle uses it, and points to the same implementation. When created, an instance of `IloSchedulerEnv` will create all the default parameters and initialize them to their default values.

```
public IloSchedulerEnv(IloSchedulerEnvI * impl)
```

This constructor creates an instance of the handle class `IloSchedulerEnv` from the pointer to an instance of the implementation class `IloSchedulerEnvI`.

Methods

```
public IloActivityBasicParam getActivityBasicParam() const
```

This member function returns the default instance of the activity basic parameter. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public IloActivityBreakParam getActivityBreakParam() const
```

This member function returns the default instance of the activity break parameter. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public IloActivityConstraintsParam getActivityConstraintsParam() const
```

This member function returns the default instance of the activity constraints parameter. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public IloActivityOverlapParam getActivityOverlapParam() const
```

This member function returns the default instance of the activity overlap parameter. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public IloActivityShiftParam getActivityShiftParam() const
```

This member function returns the default instance of the activity shift parameter. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public IloIntervalList getBreakListParam() const
```

This member function returns the default instance of the break list parameter. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public IloIntervalList getCapacityEnforcementIntervalsParam() const
```

This member function returns the default instance of the capacity enforcement intervals parameter. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public IloNumToNumStepFunction getCapacityMaxParam() const
```

This member function returns the default instance of the maximal capacity parameter. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public IloNumToNumStepFunction getCapacityMinParam() const
```

This member function returns the default instance of the minimal capacity parameter. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public IloEnv getEnv() const
```

This member function returns the instance of `IloEnv` on which the called object was built.

```
public IloNum getHorizon() const
```

This member function returns the time horizon. The time origin and the time horizon are used by default at extraction to initialize the time window over which resource capacity constraints must be enforced. The origin and horizon are also used for setting earliest start times and latest end times of activities when they are created.

```
public IloSchedulerEnvI * getImpl() const
```

This member function returns a pointer to the implementation object corresponding to the invoking activity (a handle).

```
public IloNumToNumStepFunction getInitialOccupationParam() const
```

This member function returns the default instance of the initial occupation parameter. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public IloInt getIntMaxAtExtraction() const
```

This member function returns the value of the maximum limit of the domain of the integer variables used by Scheduler Engine.

Avoiding Overflows

In order to avoid overflows in computations by Scheduler Engine, an efficient policy is to limit the domain of all extracted integer variables to an interval of `[-L,L]`. For example, a variable whose lower bound is 0 and upper bound is `IloInfinity` will be extracted as a variable with the domain `[0,L]`. This limit `L` is set by the member function `IloSchedulerEnv::setIntMaxAtExtraction`. At extraction, this limit is used only for variables that

are used by Scheduler objects or constraints, as the start and end times, the capacity variables or the precedence delays. Please note that there is an exception for the cost sum variable (class `IloTransitionCost`). For this variable, the limit `L` is not applied because there is usually no overflow with this variable. The default value of this limit is `IloIntMax/2`, which avoids overflows in most cases.

For more information, see Parameters Organized by Function and `IloNumToNumStepFunction` in the *IBM ILOG Concert Technology Reference Manual*.

```
public IloTextureParam getMaxTextureParam() const
```

The member function returns the default instance of the texture parameter on maximum capacity constraints. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public IloTextureParam getMinTextureParam() const
```

The member function returns the default instance of the texture parameter on minimum capacity constraints. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public IloIntervalList getMustBeInUseParam() const
```

This member function returns the default instance of the must be in use parameter. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public IloNum getOrigin() const
```

This member function returns the time origin. The time origin and the time horizon are used by default at extraction to initialize the time window over which resource capacity constraints must be enforced. The origin and horizon are also used for setting earliest start times and latest end times of activities when they are created.

```
public IloNumToAnySetStepFunction getPossibleStatesParam() const
```

This member function returns the default instance of the possible states parameter. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public IloEnforcementLevel getPrecedenceEnforcement() const
```

This member function returns the global precedence enforcement level of the scheduler environment.

```
public IloResourceParam getResourceParam() const
```

This member function returns the default instance of the resource parameter. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public IloIntervalList getTransitionTimeEnforcementIntervalsParam() const
```

This member function returns the default instance of the transition time enforcement intervals parameter. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public void setActivityBasicParam(const IloActivityBasicParam param) const
```

This member function sets `param` as the new default activity basic parameter. Subsequently created modeling objects will point to this parameter instance.

```
public void setActivityBreakParam(const IloActivityBreakParam param) const
```

This member function sets `param` as the new default activity break parameter. Subsequently created modeling objects will point to this parameter instance.

```
public void setActivityConstraintsParam(const IloActivityConstraintsParam param)
const
```

This member function sets `param` as the new default activity constraints parameter. Subsequently created modeling objects will point to this parameter instance.

```
public void setActivityOverlapParam(const IloActivityOverlapParam param) const
```

This member function sets `param` as the new default activity overlap parameter. Subsequently created modeling objects will point to this parameter instance.

```
public void setActivityShiftParam(const IloActivityShiftParam param) const
```

This member function sets `param` as the new default activity shift parameter. Subsequently–created modeling objects will point to this parameter instance.

```
public void setBreakListParam(const IloIntervalList param) const
```

This member function sets `param` as the new default break list parameter. Subsequently created modeling objects will point to this parameter instance.

```
public void setCapacityEnforcementIntervalsParam(const IloIntervalList param) const
```

This member function sets `param` as the new default capacity enforcement intervals parameter. Subsequently created modeling objects will point to this parameter instance.

```
public void setCapacityMaxParam(const IloNumToNumStepFunction param) const
```

This member function sets `param` as the new default maximal capacity parameter. Subsequently created modeling objects will point to this parameter instance.

```
public void setCapacityMinParam(const IloNumToNumStepFunction param) const
```

This member function sets `param` as the new default minimal capacity parameter. Subsequently created modeling objects will point to this parameter instance.

```
public void setHorizon(IloNum horizon) const
```

This member function sets the time horizon to `horizon`. The time origin and the time horizon are used by default at extraction to initialize the time window over which resource capacity constraints must be enforced. The origin and horizon are also used for setting earliest start times and latest end times of activities when they are created.

```
public void setInitialOccupationParam(const IloNumToNumStepFunction param) const
```

This member function sets `param` as the new default initial occupation parameter. Subsequently created modeling objects will point to this parameter instance.

```
public void setIntMaxAtExtraction(IloInt max) const
```

This member function sets `max` as the new value for the maximum limit of the domain of the integer variables used by Scheduler Engine. See IloSchedulerEnv::getIntMaxAtExtraction for more information on avoiding overflows.

```
public void setMaxTextureParam(const IloTextureParam param) const
```

This member function sets the texture parameter on maximum capacity constraints to `param`. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public void setMinTextureParam(const IloTextureParam param) const
```

This member function sets the texture parameter on minimum capacity constraints to `param`. Modeling objects that are created will point to this instance, which can thus be shared between several objects.

```
public void setMustBeInUseParam(const IloIntervalList param) const
```

This member function sets `param` as the new default must be in use parameter. Subsequently created modeling objects will point to this parameter instance.

```
public void setOrigin(IloNum origin) const
```

This member function sets the time origin to `origin`. The time origin and the time horizon are used by default at extraction to initialize the time window over which resource capacity constraints must be enforced. The origin and horizon are also used for setting earliest start times and latest end times of activities when they are created.

```
public void setPossibleStatesParam(const IloNumToAnySetStepFunction param) const
```

This member function sets `param` as the new default possible states parameter. Subsequently created modeling objects will point to this parameter instance.

```
public void setPrecedenceEnforcement(IloEnforcementLevel level) const
```

This member function allows setting `level` as the new global precedence enforcement level of the scheduler environment. See Resource Enforcement as Global Constraint Declaration for a description of how this enforcement level is interpreted at extraction time by the scheduler engine.

```
public void setResourceParam(const IloResourceParam param) const
```

This member function sets `param` as the new default resource parameter. Subsequently created modeling objects will point to this parameter instance.

```
public void setTransitionTimeEnforcementIntervalsParam(const IloIntervalList param)
const
```

This member function sets `param` as the new default transition time enforcement intervals parameter. Subsequently created modeling objects will point to this parameter instance.

Class IloSchedulerLargeNHood

Definition file: ilsched/iloInsgoals.h
Include file: <ilsched/iloscheduler.h>

This class represents a large neighborhood dedicated to scheduling problems.

The current solution is an instance of `IloSchedulerSolution`.

A set of activity predicates is used to specify if the start, the end, the duration, the processing time, and the external variables of an activity should be restored.

Similarly, a set of predicates can be used to specify if the next, the prev, the direct successors, the direct predecessors, the teardown, and the setup of a resource constraint should be restored.

The virtual member function `defineRestoreInfo` is available with a specialized signature for activity and resource constraints.

The virtual member function `finalizeDelta` has been overloaded: it removes from the current solution any link of the types next / prev / direct successors / direct predecessors between a resource constraint that is selected and a resource constraint that is not selected.

See Large Neighborhoods and the *Selectors* concept in the *IBM ILOG Solver Reference Manual* for more information.

See Also: IloSchedulerLargeNHoodI

Method Summary	
public void	defineRestoreInfo(IloSolver, IloSolution)
public IloSolution	defineSelected(IloSolver, IloInt index)
public void	finalizeDelta(IloSolver, IloSolution)
public IloSchedulerSolution	getCurrentSolution() const
public IloPredicate< IloActivity >	getRestoreActivityDurationPredicate() const
public IloPredicate< IloActivity >	getRestoreActivityEndPredicate() const
public IloPredicate< IloActivity >	getRestoreActivityExternalPredicate() const
public IloPredicate< IloActivity >	getRestoreActivityProcessingTimePredicate() const
public IloPredicate< IloActivity >	getRestoreActivityStartPredicate() const
public IloPredicate< IloExtractable >	getRestoreExtractablePredicate() const
public IloPredicate< IloResourceConstraint >	getRestoreRCCapacityPredicate() const
public IloPredicate< IloResourceConstraint >	getRestoreRCDirectPredecessorPredicate() const
public IloPredicate< IloResourceConstraint >	getRestoreRCDirectSuccessorPredicate() const
public IloPredicate< IloResourceConstraint >	getRestoreRCNextPredicate() const
public IloPredicate< IloResourceConstraint >	getRestoreRCPrevPredicate() const

public IloPredicate< IloResourceConstraint >	getRestoreRCSelectedPredicate() const
public IloPredicate< IloResourceConstraint >	getRestoreRCSetupPredicate() const
public IloPredicate< IloResourceConstraint >	getRestoreRCTeardownPredicate() const
public IloBool	isSelected(IloExtractable ext) const
public void	setRestoreActivityDurationPredicate(IloPredicate< IloActivity > predicate)
public void	setRestoreActivityEndPredicate(IloPredicate< IloActivity > predicate)
public void	setRestoreActivityExternalPredicate(IloPredicate< IloActivity > predicate)
public void	setRestoreActivityProcessingTimePredicate(IloPredic IloActivity > predicate)
public void	setRestoreActivityStartPredicate(IloPredicate< IloActivity > predicate)
public void	setRestoreExtractablePredicate(IloPredicate< IloExtractable > predicate)
public void	setRestoreRCCapacityPredicate(IloPredicate< IloResourceConstraint > predicate)
public void	setRestoreRCDirectPredecessorPredicate(IloPredicate IloResourceConstraint > predicate)
public void	setRestoreRCDirectSuccessorPredicate(IloPredicate< IloResourceConstraint > predicate)
public void	setRestoreRCNextPredicate(IloPredicate< IloResourceConstraint > predicate)
public void	setRestoreRCPrevPredicate(IloPredicate< IloResourceConstraint > predicate)
public void	setRestoreRCSelectedPredicate(IloPredicate< IloResourceConstraint > predicate)
public void	setRestoreRCSetupPredicate(IloPredicate< IloResourceConstraint > predicate)
public void	setRestoreRCTeardownPredicate(IloPredicate< IloResourceConstraint > predicate)

Methods

`public void` **`defineRestoreInfo`**`(IloSolver, IloSolution)`

This virtual member function iterates on all extractables and appplies on each extractable the predicate to specify if it must be restored from the current solution.

`public IloSolution` **`defineSelected`**`(IloSolver, IloInt index)`

This pure virtual member function returns the set of decision variables, or instances of `IloExtractable`, on which to focus the search.

`public void` **`finalizeDelta`**`(IloSolver, IloSolution)`

This virtual member function is called to complete the definition of the neighborhood.

```
public IloSchedulerSolution getCurrentSolution() const
```

This member function returns the current solution, which is the one registered by the virtual member function `start`.

```
public IloPredicate< IloActivity > getRestoreActivityDurationPredicate() const
```

This member function returns the predicate used to specify if the duration of an activity in the current solution must be restored.

```
public IloPredicate< IloActivity > getRestoreActivityEndPredicate() const
```

This member function returns the predicate used to specify if the end of an activity in the current solution must be restored.

```
public IloPredicate< IloActivity > getRestoreActivityExternalPredicate() const
```

This member function returns the predicate used to specify if the external variable of an activity in the current solution must be restored.

```
public IloPredicate< IloActivity > getRestoreActivityProcessingTimePredicate()
const
```

This member function returns the predicate used to specify if the processing time of an activity in the current solution must be restored.

```
public IloPredicate< IloActivity > getRestoreActivityStartPredicate() const
```

This member function returns the predicate used to specify if the end of an activity in the current solution must be restored.

```
public IloPredicate< IloExtractable > getRestoreExtractablePredicate() const
```

This member function returns the predicate used to specify which extractables to restore from the current solution.

```
public IloPredicate< IloResourceConstraint > getRestoreRCCapacityPredicate() const
```

This member function returns the predicate used to specify if the capacity of a resource constraint in the current solution must be restored.

```
public IloPredicate< IloResourceConstraint >
getRestoreRCDirectPredecessorPredicate() const
```

This member function returns the predicate used to specify if the direct predecessors of a resource constraint in the current solution must be restored.

```
public IloPredicate< IloResourceConstraint > getRestoreRCDirectSuccessorPredicate()
const
```

This member function returns the predicate used to specify if the direct successors of a resource constraint in the current solution must be restored.

```
public IloPredicate< IloResourceConstraint > getRestoreRCNextPredicate() const
```

This member function returns the predicate used to specify if the next (resource constraint) of a resource constraint in the current solution must be restored.

```
public IloPredicate< IloResourceConstraint > getRestoreRCPrevPredicate() const
```

This member function returns the predicate used to specify if the prev (previous resource constraint) of a resource constraint in the current solution must be restored.

```
public IloPredicate< IloResourceConstraint > getRestoreRCSelectedPredicate() const
```

This member function returns the predicate used to specify if the resource selected of a resource constraint in the current solution must be restored.

```
public IloPredicate< IloResourceConstraint > getRestoreRCSetupPredicate() const
```

This member function returns the predicate used to specify if the setup of a resource constraint in the current solution must be restored.

```
public IloPredicate< IloResourceConstraint > getRestoreRCTeardownPredicate() const
```

This member function returns the predicate used to specify if the teardown of a resource constraint in the current solution must be restored.

```
public IloBool isSelected(IloExtractable ext) const
```

This member function returns `IloTrue` if the extractable is selected. Otherwise, it returns `IloFalse`.

```
public void setRestoreActivityDurationPredicate(IloPredicate< IloActivity >
predicate)
```

This member function sets the predicate used to specify if the duration of an activity in the current solution must be restored. An exception is raised if the predicate is an empty handle.

```
public void setRestoreActivityEndPredicate(IloPredicate< IloActivity > predicate)
```

This member function sets the predicate used to specify if the end of an activity in the current solution must be restored. An exception is raised if the predicate is an empty handle.

```
public void setRestoreActivityExternalPredicate(IloPredicate< IloActivity >
predicate)
```

This member function sets the predicate used to specify if the external variable of an activity in the current solution must be restored. An exception is raised if the predicate is an empty handle.

```
public void setRestoreActivityProcessingTimePredicate(IloPredicate< IloActivity >
predicate)
```

This member function sets the predicate used to specify if the processing time of an activity in the current solution must be restored. An exception is raised if the predicate is an empty handle.

```
public void setRestoreActivityStartPredicate(IloPredicate< IloActivity > predicate)
```

This member function sets the predicate used to specify if the start of an activity in the current solution must be restored. An exception is raised if the predicate is an empty handle.

```
public void setRestoreExtractablePredicate(IloPredicate< IloExtractable >
predicate)
```

This member function sets the predicate used to specify which extractables to restore from the current solution. When applied, the predicate receives the neighborhood as an argument.

```
public void setRestoreRCCapacityPredicate(IloPredicate< IloResourceConstraint >
predicate)
```

This member function sets the predicate used to specify if the capacity of a resource constraint in the current solution must be restored. An exception is raised if the predicate is an empty handle.

```
public void setRestoreRCDirectPredecessorPredicate(IloPredicate<
IloResourceConstraint > predicate)
```

This member function sets the predicate used to specify if the direct predecessors of a resource constraint in the current solution must be restored. An exception is raised if the predicate is an empty handle.

```
public void setRestoreRCDirectSuccessorPredicate(IloPredicate<
IloResourceConstraint > predicate)
```

This member function sets the predicate used to specify if the direct successors of a resource constraint in the current solution must be restored. An exception is raised if the predicate is an empty handle.

```
public void setRestoreRCNextPredicate(IloPredicate< IloResourceConstraint >
```

```
predicate)
```

This member function sets the predicate used to specify if the next (resource constraint) of a resource constraint in the current solution must be restored. An exception is raised if the predicate is an empty handle.

```
public void setRestoreRCPrevPredicate(IloPredicate< IloResourceConstraint >
predicate)
```

This member sets the predicate used to specify if the prev (previous resource constraint) of a resource constraint in the current solution must be restored. An exception is raised if the predicate is an empty handle.

```
public void setRestoreRCSelectedPredicate(IloPredicate< IloResourceConstraint >
predicate)
```

This member function sets the predicate used to specify if the resource selected of a resource constraint in the current solution must be restored. An exception is raised if the predicate is an empty handle.

```
public void setRestoreRCSetupPredicate(IloPredicate< IloResourceConstraint >
predicate)
```

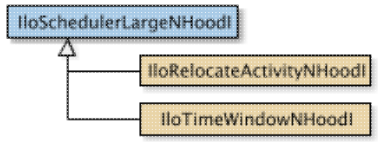
This member function sets the predicate used to specify if the setup of a resource constraint in the current solution must be restored. An exception is raised if the predicate is an empty handle.

```
public void setRestoreRCTeardownPredicate(IloPredicate< IloResourceConstraint >
predicate)
```

This member function sets the predicate used to specify if the teardown of a resource constraint in the current solution must be restored. An exception is raised if the predicate is an empty handle.

Class IloSchedulerLargeNHoodI

Definition file: ilsched/iloInsgoals.h
Include file: <ilsched/iloscheduler.h>



This abstract implementation class is used to define a large neighborhood dedicated to scheduling problems.

The current solution is an instance of `IloSchedulerSolution`.

A set of activity predicates are used to specify if the start, the end, the duration, the processing time, and the external variables of an activity should be restored.

Similarly, a set of predicates can be used to specify if the next, the prev, the direct successors, the direct predecessors, the teardown, and the setup of a resource constraint should be restored.

The virtual member function `defineRestoreInfo` is available with a specialized signature for activity and resource constraints.

The virtual member function `finalizeDelta` has been overloaded: it removes from the current solution any link of the types next / prev / direct successors / direct predecessors between a resource constraint that is selected and a resource constraint that is not selected.

See Large Neighborhoods and the *Selectors* concept in the *IBM ILOG Solver Reference Manual* for more information.

See Also: IloSchedulerLargeNHood

Constructor and Destructor Summary	
public	`IloSchedulerLargeNHoodI(IloEnv env, const char * name)`

Method Summary	
`public virtual IloSolution`	`define(IloSolver solver, IloInt index)`
`public virtual void`	`defineRestoreInfo(IloSolver solver, IloSolution solution)`
`public virtual IloSolution`	`defineSelected(IloSolver solver, IloInt index)`
`public virtual void`	`finalizeDelta(IloSolver solver, IloSolution solution)`
`public IloSchedulerSolution`	`getCurrentSolution() const`
`public IloPredicate< IloActivity >`	`getRestoreActivityDurationPredicate() const`
`public IloPredicate< IloActivity >`	`getRestoreActivityEndPredicate() const`
`public IloPredicate< IloActivity >`	`getRestoreActivityExternalPredicate() const`
`public IloPredicate< IloActivity >`	`getRestoreActivityProcessingTimePredicate() const`
`public IloPredicate< IloActivity >`	`getRestoreActivityStartPredicate() const`

public IloPredicate< IloExtractable >	getRestoreExtractablePredicate() const
public virtual IloInt	getRestoreInfo(IloSolver solver, IloExtractable)
public IloPredicate< IloResourceConstraint >	getRestoreRCCapacityPredicate() const
public IloPredicate< IloResourceConstraint >	getRestoreRCDirectPredecessorPredicate() const
public IloPredicate< IloResourceConstraint >	getRestoreRCDirectSuccessorPredicate() const
public IloPredicate< IloResourceConstraint >	getRestoreRCNextPredicate() const
public IloPredicate< IloResourceConstraint >	getRestoreRCPrevPredicate() const
public IloPredicate< IloResourceConstraint >	getRestoreRCSelectedPredicate() const
public IloPredicate< IloResourceConstraint >	getRestoreRCSetupPredicate() const
public IloPredicate< IloResourceConstraint >	getRestoreRCTeardownPredicate() const
public IloBool	isSelected(IloExtractable) const
public void	setRestoreActivityDurationPredicate(IloPredicate< IloActivity > predicate)
public void	setRestoreActivityEndPredicate(IloPredicate< IloActivity > predicate)
public void	setRestoreActivityExternalPredicate(IloPredicate< IloActivity > predicate)
public void	setRestoreActivityProcessingTimePredicate(IloPredic IloActivity > predicate)
public void	setRestoreActivityStartPredicate(IloPredicate< IloActivity > predicate)
public void	setRestoreExtractablePredicate(IloPredicate< IloExtractable > predicate)
public void	setRestoreRCCapacityPredicate(IloPredicate< IloResourceConstraint > predicate)
public void	setRestoreRCDirectPredecessorPredicate(IloPredicate IloResourceConstraint > predicate)
public void	setRestoreRCDirectSuccessorPredicate(IloPredicate< IloResourceConstraint > predicate)
public void	setRestoreRCNextPredicate(IloPredicate< IloResourceConstraint > predicate)
public void	setRestoreRCPrevPredicate(IloPredicate< IloResourceConstraint > predicate)
public void	setRestoreRCSelectedPredicate(IloPredicate< IloResourceConstraint > predicate)
public void	setRestoreRCSetupPredicate(IloPredicate< IloResourceConstraint > predicate)
public void	setRestoreRCTeardownPredicate(IloPredicate< IloResourceConstraint > predicate)

Constructors and Destructors

public **IloSchedulerLargeNHoodI**(IloEnv env, const char * name)

This constructor creates a neighborhood. The optional argument name, if supplied, is the name of the

neighborhood.

Methods

```
public virtual IloSolution define(IloSolver solver, IloInt index)
```

The virtual member function is overloaded and calls successively the member functions `defineSelected`, `defineRestoreInfo` and `finalizeDelta`.

```
public virtual void defineRestoreInfo(IloSolver solver, IloSolution solution)
```

This virtual member function iterates on all extractables and appplies on each extractable the predicate to specify if it must be restored from the current solution.

```
public virtual IloSolution defineSelected(IloSolver solver, IloInt index)
```

This pure virtual member function returns the set of decision variables, or instances of `IloExtractable`, on which to focus the search.

```
public virtual void finalizeDelta(IloSolver solver, IloSolution solution)
```

This virtual member function is called to complete the definition of the *delta*. It removes all links between two resource constraints of the types prev, next, direct successors and direct predecessors, where one resource constraint is selected and the other is not selected.

```
public IloSchedulerSolution getCurrentSolution() const
```

This member function returns the current solution, that is, the one registered by the virtual member function `start`.

```
public IloPredicate< IloActivity > getRestoreActivityDurationPredicate() const
```

This member function returns the predicate used to specify if the duration of an activity in the current solution must be restored.

```
public IloPredicate< IloActivity > getRestoreActivityEndPredicate() const
```

This member function returns the predicate used to specify if the end of an activity in the current solution must be restored.

```
public IloPredicate< IloActivity > getRestoreActivityExternalPredicate() const
```

This member function returns the predicate used to specify if the external variable of an activity in the current solution must be restored.

```
public IloPredicate< IloActivity > getRestoreActivityProcessingTimePredicate()
const
```

This member function returns the predicate used to specify if the processing time of an activity in the current solution must be restored.

```
public IloPredicate< IloActivity > getRestoreActivityStartPredicate() const
```

This member function returns the predicate used to specify if the start of an activity in the current solution must be restored.

```
public IloPredicate< IloExtractable > getRestoreExtractablePredicate() const
```

This member function returns the predicate used to specify which extractables to restore from the current solution.

```
public virtual IloInt getRestoreInfo(IloSolver solver, IloExtractable)
```

This virtual member function is called to complete the definition of the neighborhood.

```
public IloPredicate< IloResourceConstraint > getRestoreRCCapacityPredicate() const
```

This member function returns the predicate used to specify if the capacity of a resource constraint in the current solution must be restored.

```
public IloPredicate< IloResourceConstraint >
getRestoreRCDirectPredecessorPredicate() const
```

This member function returns the predicate used to specify if the direct predecessors of a resource constraint in the current solution must be restored.

```
public IloPredicate< IloResourceConstraint > getRestoreRCDirectSuccessorPredicate()
const
```

This member function returns the predicate used to specify if the direct successors of a resource constraint in the current solution must be restored.

```
public IloPredicate< IloResourceConstraint > getRestoreRCNextPredicate() const
```

This member function returns the predicate used to specify if the next (resource constraint) of a resource constraint in the current solution must be restored.

```
public IloPredicate< IloResourceConstraint > getRestoreRCPrevPredicate() const
```

This member function returns the predicate used to specify if the previous (resource constraint) of a resource constraint in the current solution must be restored.

```
public IloPredicate< IloResourceConstraint > getRestoreRCSelectedPredicate() const
```

This member function returns the predicate used to specify if the resource selected of a resource constraint in the current solution must be restored.

```
public IloPredicate< IloResourceConstraint > getRestoreRCSetupPredicate() const
```

This member function returns the predicate used to specify if the setup of a resource constraint in the current solution must be restored.

```
public IloPredicate< IloResourceConstraint > getRestoreRCTeardownPredicate() const
```

This member function returns the predicate used to specify if the teardown of a resource constraint in the current solution must be restored.

```
public IloBool isSelected(IloExtractable) const
```

This member function returns `IloTrue` if the extractable is selected. Otherwise, it returns `IloFalse`.

```
public void setRestoreActivityDurationPredicate(IloPredicate< IloActivity >
predicate)
```

This member function sets the predicate used to specify if the duration of an activity in the current solution must be restored. When applied, the predicate receives as argument the neighborhood. An exception is raised if the predicate is an empty handle.

```
public void setRestoreActivityEndPredicate(IloPredicate< IloActivity > predicate)
```

This member function sets the predicate used to specify if the end of an activity in the current solution must be restored. When applied, the predicate receives as argument the neighborhood. An exception is raised if the predicate is an empty handle.

```
public void setRestoreActivityExternalPredicate(IloPredicate< IloActivity >
predicate)
```

This member function sets the predicate used to specify if the external variable of an activity in the current solution must be restored. When applied, the predicate receives as argument the neighborhood. An exception is raised if the predicate is an empty handle.

```
public void setRestoreActivityProcessingTimePredicate(IloPredicate< IloActivity >
predicate)
```

This member function sets the predicate used to specify if the processing time of an activity in the current solution must be restored. When applied, the predicate receives as argument the neighborhood. An exception is raised if the predicate is an empty handle.

```
public void setRestoreActivityStartPredicate(IloPredicate< IloActivity > predicate)
```

This member function sets the predicate used to specify if the start of an activity in the current solution must be restored. An exception is raised if the predicate is an empty handle.

```
public void setRestoreExtractablePredicate(IloPredicate< IloExtractable >
predicate)
```

This member sets the predicate used to specify which extractables to restore from the current solution. When applied, the predicate receives as argument the neighborhood. An error is raised if the predicate is an empty handle.

```
public void setRestoreRCCapacityPredicate(IloPredicate< IloResourceConstraint >
predicate)
```

This member function sets the predicate used to specify if the capacity of a resource constraint in the current solution must be restored. When applied, the predicate receives as argument the neighborhood. An exception is raised if the predicate is an empty handle.

```
public void setRestoreRCDirectPredecessorPredicate(IloPredicate<
IloResourceConstraint > predicate)
```

This member function sets the predicate used to specify if the direct predecessors of a resource constraint in the current solution must be restored. When applied, the predicate receives as argument the neighborhood. An exception is raised if the predicate is an empty handle.

```
public void setRestoreRCDirectSuccessorPredicate(IloPredicate<
IloResourceConstraint > predicate)
```

This member function sets the predicate used to specify if the direct successors of a resource constraint in the current solution must be restored. When applied, the predicate receives as argument the neighborhood. An exception is raised if the predicate is an empty handle.

```
public void setRestoreRCNextPredicate(IloPredicate< IloResourceConstraint >
predicate)
```

This member function sets the predicate used to specify if the next (resource constraint) of a resource constraint in the current solution must be restored. When applied, the predicate receives as argument the neighborhood. An exception is raised if the predicate is an empty handle.

```
public void setRestoreRCPrevPredicate(IloPredicate< IloResourceConstraint >
predicate)
```

This member function sets the predicate used to specify if the prev (previous resource constraint) of a resource constraint in the current solution must be restored. When applied, the predicate receives as argument the neighborhood. An exception is raised if the predicate is an empty handle.

```
public void setRestoreRCSelectedPredicate(IloPredicate< IloResourceConstraint >
predicate)
```

This member function sets the predicate used to specify if the resource selected of a resource constraint in the current solution must be restored. When applied, the predicate receives as argument the neighborhood. An exception is raised if the predicate is an empty handle.

```
public void setRestoreRCSetupPredicate(IloPredicate< IloResourceConstraint >
predicate)
```

This member function sets the predicate used to specify if the setup of a resource constraint in the current solution must be restored. When applied, the predicate receives as argument the neighborhood. An exception is raised if the predicate is an empty handle.

```
public void setRestoreRCTeardownPredicate(IloPredicate< IloResourceConstraint >
predicate)
```

This member function sets the predicate used to specify if the teardown of a resource constraint in the current solution must be restored. When applied, the predicate receives as argument the neighborhood. An exception is raised if the predicate is an empty handle.

Class IloSchedulerSolution

Definition file: ilsched/ilosolution.h
Include file: <ilsched/iloscheduler.h>

The class `IloSchedulerSolution` has two distinct purposes. One use is for the storage of solution data, and the other use is as the central data structure for local search. These two purposes are quite different and so it is useful to clearly distinguish between them.

IloSchedulerSolution for Data Storage

As a general object in which to store information about the state of the solver at the level of scheduling, an `IloSchedulerSolution` object can be used to store and inspect activities, resources and resource constraints. This information can be modified within the `IloSchedulerSolution` object, however, the modifications only affect the data stored in the solution object. If you want the modifications reflected either in the model or in a solver, it is necessary to directly modify the model or the objects extracted in a solver. For example, you can store an instance of an `IloActivity` in a solution object as follows:

```
IloSchedulerSolution sol(env);
IloActivity act(env, 10);
sol.add(act);
```

If we assume that `act` is part of a model that has been extracted by an `IlcScheduler` object, `schedule`, and solved by the associated solver, we can then store the activity in the solution with the command:

```
sol.store(scheduler);
```

This command causes the solver data regarding `act` (and any other extractables added to the solution) to be stored in the solution. In the example of an `IloActivity`, the bounds on the start time, end time, processing time, and duration variables are stored together with any external variable that you may have associated with the activity.

Note that the data that is stored is retrieved from the current state of the solver.

The data in the solution can then be modified via the `IloSchedulerSolution` API. For example, to set the value of the start time *in the solution*, you can write:

```
sol.setStartMin(act, 10);
sol.setStartMax(act, 10);
```

Note that these commands only change the data stored in `sol`. In particular the original model is unchanged and the value of the start time of the `IlcActivity` that the solver has extracted from `act` is unchanged. To change the model, the extractable itself must be changed. For example,

```
act.setLb(sol.getStartMin(act));
act.setUb(sol.getStartMax(act));
```

Similarly, you can write a search goal which takes an `IloSchedulerSolution` object as an argument and then, once inside the search, adds constraints to the solver based on the information stored in the `IloSchedulerSolution`.

IloSchedulerSolution for Local Search

In local search in IBM® ILOG® Scheduler, `IloSchedulerSolution` plays the same role as the `IloSolution` object does in local search in IBM ILOG Solver (see the *IBM ILOG Solver Reference Manual*). That is, an `IloSchedulerSolution` is used to represent a neighbor that is to be explored. Typically, such a neighbor is specified by an `IloSchedulerSolution` which contains only those objects which change in moving from the current solution to the neighbor. The values of those objects in the solution reflect their values in the neighbor that is to be explored. For example, assume you have a solution where four resource constraints on the same

resource are ordered as follows: `A` before `B` before `C` before `D`.

Further assume that one of the neighbors you want to explore is the one where the order of `B` and `C` is reversed. The following code will create an `IloSchedulerSolution` object that represents such a neighbor:

```
IloSchedulerSolution delta(env);
delta.add(A, IloRestoreRCNext);
delta.add(B, IloRestoreRCNext);
delta.add(C, IloRestoreRCNext);
delta.setNext(A, C);
delta.setNext(C, B);
delta.setNext(B, D);
```

Notice that the `IloSchedulerSolution::add` method requires a second argument in this case. This argument specifies what part of the information stored in the solution about the extractable is to be restored by the solver. This information is only relevant in four cases:

1. During local search.
2. If you use the `IloRestoreSolution` goal.
3. If you use the `IloSolution::getConstraint` method.
4. If you use the `IloSolution::restore` method.

In each of these cases, the actual information that is restored (or, in the case of `IloSolution::getConstraint`, added as a constraint) depends on the fields that have been defined to be restored. By default, for `IloActivity` and `IloResourceConstraint` all the stored information is restored. Note that for `IloResource` (and its subclasses) nothing can be restored in this way. This is because, from a local search perspective, the resource does not contain any decision variables.

For more information about local search see the *IBM ILOG Solver Reference Manual* and the *IBM ILOG Solver User Manual*.

Storing Solutions

There are two ways to store solutions. Users can choose the method they prefer and mix methods if desired.

In the first method, all the objects that are to be stored are added to the solution once using `IloSchedulerSolution::add`. Then, when the solver is in the search state that is to be stored, a single call to the function `IloSchedulerSolution::store(const IlcScheduler scheduler)` will store all the added objects that have been extracted by the solver associated with `scheduler`.

In the second method, there is no need to add the objects to the solution. Instead, when the solver is in the search state that is to be stored, a call to `IlcSchedulerSolution::store` can be made for each object that is to be stored. For example, `IloSchedulerSolution::store(const IloActivity activity, IlcScheduler scheduler)` will add `activity` to the solution and then immediately store it.

Sharing Solutions

`IloSchedulerSolution` inherits from class `IloSolution` (see the *IBM ILOG Concert Technology Reference Manual*). This means that all of the actual implementation of the data structure for storing objects is performed by the instance of `IloSolution`. In addition, an instance of `IloSchedulerSolution` can be casted to an instance of `IloSolution` back and forth. This allow to share a single solution instance among different solvers. For example, IBM® ILOG® Dispatcher (see the *IBM ILOG Dispatcher Reference Manual*) contains an `IloRoutingSolution` that also inherits from class `IloSolution`. In a problem that uses both Scheduler and Dispatcher, a single `IloSolution` instance can be shared between the two solvers by casting it to the appropriate class.

Storing Objects

When an object is stored, with either method described earlier, the values of its variables—assuming they are instantiated—are stored in the scheduler solution. If a variable is not bound, the minimal and the maximal value of its domain are stored in most cases.

- When an activity is stored, the start time, end time, duration, processing time, and external variable of the activity are stored. If a variable is not bound, its bounds are stored.
- For a resource constraint on a capacity resource, the minimal and maximal capacity is stored. If the resource constraint has alternatives, the set of resources that are still possible is stored. If the resource constraint is associated with a precedence graph on a resource and if the resource constraint either has no alternatives or if a single resource has been selected, then the next, direct successor, setup, and teardown relations are stored.
- For resources with precedence graph constraints, the order of the resource constraints in its precedence graph is stored.
- For capacity resources with timetable constraints, the minimal and maximal capacity level of the resource is stored. The capacity level of a resource is the amount of capacity that is used by its resource constraints. For example, the capacity level of a discrete resource at a time *t* is the sum of the capacities of all its resource constraints that are associated with activities that cover the time *t*. With such a definition, when the algorithm has found only a partial solution, the minimal (or maximal) capacity level may be less (or greater) than the minimal (or maximal) capacity of the resource. This may be a problem for some applications, so the member function `IloSchedulerSolution::setLevelsStoredWithBounds` is provided to set the resource in a "storage with bounds" mode. In such a mode, the levels are stored so that the minimal level is never less than the minimum capacity and the maximal level is never greater than the maximum capacity.

Consequently, the minimal capacity level of a resource at time *t* is the sum of the minimal capacities of all resource constraints that surely contribute to the resource and that are associated with activities that surely cover time *t*.

If the resource is closed, the maximal capacity level of the resource at time *t* is the sum of the maximal capacities of all resource constraints that possibly contribute to the resource and that are associated with activities that possibly cover time *t*. If the resource is not closed, the maximal capacity level of the resource at time *t* is equal to `IloInfinity`.

Accessing Stored Objects

An instance of the class `IloSchedulerSolution` represents a mapping between stored Scheduler objects and their (possible) values. It is important to note that this mapping is not reversible. This means that the values of the stored objects are preserved when backtracking. The values of the stored objects can be accessed at any time by calling appropriate member functions of the class `IloSchedulerSolution`. There are also iterators that allow access to all the objects stored in a scheduler solution.

Modifying Values of Stored Objects

It is possible to modify the values of the stored objects by calling appropriate member functions of the class `IloSchedulerSolution`.

Restoring a solution

Relations of type next and previous as well as setup and teardown are restored only on unary resources that have a precedence graph (either light or classical).

Successors and predecessors are always restored even if the resource has no precedence graph.

For alternative resource constraints, edges of type next, previous, successor and predecessor are restored only if both resource constraints are or can be on the same resource. When a next, previous, successor or predecessor relation between two resource constraints is restored, it means that whenever both resource constraints are allocated the same resource, the relation holds.

Deleting Solutions

The `IloSchedulerSolution` class provides an `end` function that frees all the memory used to save the data in a solution. Explicitly calling this function is unnecessary as the memory will also be reclaimed when the `IloEnv` is destroyed. However, if the user is creating and manipulating a large number of solutions, it may be desirable to reclaim memory from `IlcSchedulerSolution` instances that are no longer needed. The `end` function has been provided for just such a situation.

For more information, see `IloNumToAnySetStepFunction` in the *IBM ILOG Concert Technology Reference Manual*. Also see Global Constraints.

See Also: IloActivity, IloResource, IloResourceConstraint, IloSchedulerSolution::ActivityIterator, IloSchedulerSolution::ResourceConstraintIterator, IloSchedulerSolution::ResourceIterator

Constructor Summary	
public	IloSchedulerSolution(const IloEnv, const char * name=0)
public	IloSchedulerSolution(const IloSolution)
public	IloSchedulerSolution(const IloModel, const char * name=0)

Method Summary	
public void	add(const IloResource x) const
public void	add(const IloModel x) const
public void	add(const IloResourceConstraint x, IloInt restoreFields=IloRestoreAll) const
public void	add(const IloActivity x, IloInt restoreFields=IloRestoreAll) const
public IloBool	areLevelsStoredWithBounds(const IloResource res) const
public void	copy(IloSchedulerSolution solution) const
public IloNum	getCapacityMax(const IloResourceConstraint rct) const
public IloNum	getCapacityMin(const IloResourceConstraint rct) const
public IloNumToNumSegmentFunction	getContinuousLevelMax(const IloResource res) const
public IloNumToNumSegmentFunction	getContinuousLevelMin(const IloResource res) const
public IloNumToNumStepFunction	getDiscreteLevelMax(const IloResource resource) const
public IloNumToNumStepFunction	getDiscreteLevelMin(const IloResource resource) const
public IloNum	getDurationMax(const IloActivity activity) const
public IloNum	getDurationMin(const IloActivity activity) const
public IloNum	getEndMax(const IloActivity activity) const
public IloNum	getEndMin(const IloActivity activity) const
public IloEnv	getEnv() const
public IloNum	getExternalVariableMax(const IloActivity activity) const
public IloNum	getExternalVariableMin(const IloActivity activity) const
public IloNum	getLevelMax(const IloResource resource, IloNum time) const
public IloNum	getLevelMin(const IloResource resource, IloNum time) const
public IloResourceConstraint	getNextRC(const IloResourceConstraint rct) const

public IloResourceConstraint	getPrevRC(const IloResourceConstraint rct) const
public IloNum	getProcessingTimeMax(const IloActivity activity) const
public IloNum	getProcessingTimeMin(const IloActivity activity) const
public IloInt	getRestorable(IloResourceConstraint rc) const
public IloInt	getRestorable(IloActivity activity) const
public IloResource	getSelected(const IloResourceConstraint rct) const
public IloResourceConstraint	getSetupRC(const IloResource resource) const
public IloNum	getStartMax(const IloActivity activity) const
public IloNum	getStartMin(const IloActivity activity) const
public IloResourceConstraint	getTeardownRC(const IloResource resource) const
public IloBool	hasAsNext(const IloResourceConstraint srct1, const IloResourceConstraint srct2) const
public IloBool	hasCapacityInformation(const IloResource resource) const
public IloBool	hasPredecessors(const IloResourceConstraint rct)
public IloBool	hasSuccessors(const IloResourceConstraint rct)
public IloBool	isResourceSelected(const IloResourceConstraint rct) const
public IloBool	isSetup(const IloResourceConstraint rct) const
public IloBool	isSucceededBy(const IloResourceConstraint srct1, const IloResourceConstraint srct2) const
public IloBool	isTeardown(const IloResourceConstraint rct) const
public IloSchedulerSolution	makeClone(IloEnv e) const
public void	remove(IloModel m) const
public void	setCapacityMax(const IloResourceConstraint ct, IloNum max) const
public void	setCapacityMin(const IloResourceConstraint ct, IloNum min) const
public void	setDurationMax(const IloActivity activity, IloNum max) const
public void	setDurationMin(const IloActivity activity, IloNum min) const
public void	setEndMax(const IloActivity activity, IloNum max) const
public void	setEndMin(const IloActivity activity, IloNum min) const
public void	setExternalVariableMax(const IloActivity activity, IloNum max) const
public void	setExternalVariableMin(const IloActivity activity, IloNum min) const
public void	setLevelMax(const IloResource resource, IloNum start, IloNum end, IloNum max) const
public void	setLevelMin(const IloResource resource, IloNum

	start, IloNum end, IloNum min) const
public void	setLevelsStoredWithBounds(const IloResource res, IloBool withBounds=IloTrue) const
public void	setNext(const IloResourceConstraint srct1, const IloResourceConstraint srct2) const
public void	setNonRestorable(IloResourceConstraint rc) const
public void	setNonRestorable(IloActivity activity) const
public void	setProcessingTimeMax(const IloActivity activity, IloNum max) const
public void	setProcessingTimeMin(const IloActivity activity, IloNum min) const
public void	setRestorable(IloResourceConstraint rc, IloInt storeFields) const
public void	setRestorable(IloActivity activity, IloInt storeFields) const
public void	setSelected(IloResourceConstraint rct, const IloResource resource) const
public void	setSetup(const IloResourceConstraint rct) const
public void	setStartMax(const IloActivity activity, IloNum max) const
public void	setStartMin(const IloActivity activity, IloNum min) const
public void	setSuccessor(const IloResourceConstraint srct1, const IloResourceConstraint srct2) const
public void	setTeardown(const IloResourceConstraint rct) const
public void	store(const IloResourceConstraint ct, const IlcScheduler scheduler) const
public void	store(const IloResource resource, const IlcScheduler scheduler) const
public void	store(const IloActivity activity, const IlcScheduler scheduler) const
public void	store(const IlcScheduler scheduler) const
public void	unsetNext(const IloResourceConstraint rct) const
public void	unsetNext(const IloResourceConstraint srct1, const IloResourceConstraint srct2) const
public void	unsetPrecedences()
public void	unsetPrecedences(const IloResource resource)
public void	unsetPredecessors(const IloResourceConstraint rct) const
public void	unsetPrev(const IloResourceConstraint rct) const
public void	unsetSelected(const IloResourceConstraint rct) const
public void	unsetSequence(const IloResource resource)
public void	unsetSequences()
public void	unsetSetup(const IloResourceConstraint rct) const

public void	unsetSuccessor(const IloResourceConstraint srct1, const IloResourceConstraint srct2) const
public void	unsetSuccessors(const IloResourceConstraint rct) const
public void	unsetTeardown(const IloResourceConstraint rct) const

Inner Enumeration
IloSchedulerSolution::IloResourceConstraintIteratorFilter

Inner Class
IloSchedulerSolution::ActivityIterator
IloSchedulerSolution::ResourceConstraintIterator
IloSchedulerSolution::ResourceIterator

Constructors

`public` **`IloSchedulerSolution`**`(const IloEnv, const char * name=0)`

This constructor creates a new instance of the class `IloSchedulerSolution`. This instance does not contain stored objects. Objects can be stored in this instance by calling the member function `IloSchedulerSolution::store`.

`public` **`IloSchedulerSolution`**`(const IloSolution)`

This constructor creates an instance of `IloSchedulerSolution` that is an interface to the passed `IloSolution` solution. Rather than creating an internal `IloSolution`, an argument is used.

This constructor can be used to share an `IloSolution` instance among a number of solvers, each with a different interface to a solution object.

`public` **`IloSchedulerSolution`**`(const IloModel, const char * name=0)`

This constructor creates a scheduler solution from the environment associated with `model`. It calls `add(IloModel)` and adds all the scheduler extractables (that is, `IloActivity`, `IloResourceConstraint`, `IloResource` and its subclasses) that have been explicity added to the model to the solution. The optional argument `name`, if provided, becomes the name of the underlying `IloSolution` associated with the newly created `IloSchedulerSolution`.

Methods

`public void` **`add`**`(const IloResource x) const`

This member function adds the passed `IloResource` to the list of objects in the solution. The next time `IloSchedulerSolution::store(const IlcScheduler)` is called, the resource will be stored.

`public void` **`add`**`(const IloModel x) const`

This member function adds model x to the invoking scheduler solution. That is, all activities, resource constraints, and resources that have been explicitly added to model are added to the invoking routing solution.

```
public void add(const IloResourceConstraint x, IloInt restoreFields=IloRestoreAll)
const
```

This member function adds x to the list of objects in the solution. The next time IloSchedulerSolution::store(const IlcScheduler) is called, the resource constraint will be stored. The storeFields argument defines the fields that will be restored when applying goal IloRestoreSolution or when calling method IloSolution::restore. Valid values of restoreFields are IloRestoreNothing, IloRestoreAll, and any bitwise-OR combination of IloRestoreRCNext, IloRestoreRCPrev, IloRestoreRCDirectSuccessor, IloRestoreRCDirectPredecessors, IloRestoreRCSetup, IloRestoreRCTeardown, IloRestoreRCCapacity, and IloRestoreRCSelected.

Note

Warning If IloResourceConstraint does not have a resource selected, the precedence relations of type next, previous, direct successor, direct predecessor, setup and teardown will not be stored.

```
public void add(const IloActivity x, IloInt restoreFields=IloRestoreAll) const
```

This member function adds x to the list of objects in the solution. The next time IloScheduler::store(const IlcScheduler) is called, the activity will be stored. The restoreFields argument defines the fields that will be restored in local search, the IloRestoreSolution goal, and via the IloSolution::getConstraint methods. Valid values of storeFields are IloRestoreNothing, IloRestoreAll, and any bitwise-OR combination of IloRestoreActivityStart, IloRestoreActivityEnd, IloRestoreActivityDuration, IloRestoreActivityProcessingTime, IloRestoreActivityExternal.

```
public IloBool areLevelsStoredWithBounds(const IloResource res) const
```

This member function returns IloTrue if the current storage mode of res is “with bounds.” See the section "Storing Objects" at the start of this class for more information.

This member function should be used only if res is stored in the scheduler solution and contains information about capacity levels.

```
public void copy(IloSchedulerSolution solution) const
```

This member function copies solution to the invoking scheduler solution.

```
public IloNum getCapacityMax(const IloResourceConstraint rct) const
```

This member function returns the maximal capacity of the given resource constraint in the invoking scheduler solution.

This member function should be used only if rct is stored in the scheduler solution.

```
public IloNum getCapacityMin(const IloResourceConstraint rct) const
```

This member function returns the minimal capacity of the given resource constraint in the invoking scheduler solution.

This member function should be used only if `rct` is stored in the scheduler solution.

```
public IloNumToNumSegmentFunction getContinuousLevelMax(const IloResource res)
const
```

This member function returns a piecewise linear function describing the stored maximal level of `res` in the invoking scheduler solution. The returned piecewise linear function is a physical representation of the maximal level of `res` in the solution. Therefore, it is possible to modify the stored maximal level by modifying the piecewise linear function; and modifying the stored maximal level of `res` modifies the returned piecewise linear function.

This member function should be used only if `res` is stored in the scheduler solution and contains information about capacity levels.

```
public IloNumToNumSegmentFunction getContinuousLevelMin(const IloResource res)
const
```

This member function returns a piecewise linear function describing the stored minimal level of `res` in the invoking scheduler solution. The returned piecewise linear function is a physical representation of the minimal level of `res` in the solution. Therefore, it is possible to modify the stored minimal level by modifying the piecewise linear function; and modifying the stored minimal level of `res` modifies the returned piecewise linear function.

This member function should be used only if `res` is stored in the scheduler solution and contains information about capacity levels.

```
public IloNumToNumStepFunction getDiscreteLevelMax(const IloResource resource)
const
```

This member function returns a step function describing the stored maximal level of `resource` in the invoking scheduler solution. The returned step function is a physical representation of the maximal level of `resource` in the solution. Therefore, it is possible to modify the stored maximal level by modifying the step function; and modifying the stored maximal level of `resource` modifies the returned step function.

This member function should be used only if `resource` is stored in the scheduler solution and contains information about capacity levels. This member function must not be used if `resource` is a continuous reservoir.

```
public IloNumToNumStepFunction getDiscreteLevelMin(const IloResource resource)
const
```

This member function returns a step function describing the stored minimal level of `resource` in the invoking scheduler solution. The returned step function is a physical representation of the minimal level of `resource` in the solution. Therefore, it is possible to modify the stored minimal level by modifying the step function; and modifying the stored minimal level of `resource` modifies the returned step function.

This member function should be used only if `resource` is stored in the scheduler solution and contains information about capacity levels. This member function must not be used if `resource` is a continuous reservoir.

```
public IloNum getDurationMax(const IloActivity activity) const
```

This member function returns the maximal duration of the given activity in the invoking scheduler solution.

This member function should be used only if `activity` is stored in the scheduler solution.

```
public IloNum getDurationMin(const IloActivity activity) const
```

This member function returns the minimal duration of the given activity in the invoking scheduler solution.

This member function should be used only if `activity` is stored in the scheduler solution.

```
public IloNum getEndMax(const IloActivity activity) const
```

This member function returns the maximal end time of the given activity in the invoking scheduler solution.

This member function should be used only if `activity` is stored in the scheduler solution.

```
public IloNum getEndMin(const IloActivity activity) const
```

This member function returns the minimal end time of the given activity in the invoking scheduler solution.

This member function should be used only if `activity` is stored in the scheduler solution.

```
public IloEnv getEnv() const
```

This member function returns the environment for which the scheduler solution was created.

```
public IloNum getExternalVariableMax(const IloActivity activity) const
```

This member function returns the maximal value for the external variable of the given `activity` in the invoking scheduler solution.

This member function should be used only if `activity` is stored in the scheduler solution.

```
public IloNum getExternalVariableMin(const IloActivity activity) const
```

This member function returns the minimal value for the external variable of the given activity in the invoking scheduler solution.

This member function should be used only if activity is stored in the scheduler solution.

```
public IloNum getLevelMax(const IloResource resource, IloNum time) const
```

This member function returns the maximal level of the given resource in the invoking scheduler solution at the given time.

This member function should be used only if `resource` is stored in the scheduler solution and contains information about capacity levels.

```
public IloNum getLevelMin(const IloResource resource, IloNum time) const
```

This member function returns the minimal level of the given resource in the invoking scheduler solution at the given time.

This member function should be used only if `resource` is stored in the scheduler solution and contains information about capacity levels.

```
public IloResourceConstraint getNextRC(const IloResourceConstraint rct) const
```

This member function returns the next resource constraint of the given resource constraint in the invoking scheduler solution, if such a next constraint exists. In case there are several such constraints, this method returns an empty handle: see `IloSchedulerSolution::ResourceConstraintIterator` to iterate on all the next resource constraints.

This member function should be used only if `rct` is stored in the invoking scheduler solution.

```
public IloResourceConstraint getPrevRC(const IloResourceConstraint rct) const
```

This member function returns the previous resource constraint of the given resource constraint in the invoking scheduler solution, if such a previous constraint exists. In case there are several such constraints, the method returns an empty handle : see `IloSchedulerSolution::ResourceConstraintIterator` to iterate on all previous resource constraints.

This member function should be used only if `rct` is stored in the invoking scheduler solution.

```
public IloNum getProcessingTimeMax(const IloActivity activity) const
```

This member function returns the maximal processing time of the given activity in the invoking scheduler solution.

This member function should be used only if `activity` is stored in the scheduler solution.

```
public IloNum getProcessingTimeMin(const IloActivity activity) const
```

This member function returns the minimal processing time of the given activity in the invoking scheduler solution.

This member function should be used only if `activity` is stored in the scheduler solution.

```
public IloInt getRestorable(IloResourceConstraint rc) const
```

This method returns the value representing the fields of `rc` that will be restored.

```
public IloInt getRestorable(IloActivity activity) const
```

This method returns the value representing the fields of `activity` that will be restored.

```
public IloResource getSelected(const IloResourceConstraint rct) const
```

This member function returns the selected resource of the given resource constraint in the invoking scheduler solution if this selected resource exists.

This member function should be used only if `rct` is stored in the scheduler solution.

```
public IloResourceConstraint getSetupRC(const IloResource resource) const
```

This member function returns the setup resource constraint of the given `resource` in the invoking scheduler solution, if such a setup resource constraint exists. In case there are several such constraints, this method returns an empty handle.

This member function should be used only if `resource` is stored in the invoking scheduler solution.

```
public IloNum getStartMax(const IloActivity activity) const
```

This member function returns the maximal start time of the given activity in the invoking scheduler solution.

This member function should be used only if `activity` is stored in the scheduler solution.

```
public IloNum getStartMin(const IloActivity activity) const
```

This member function returns the minimal start time of the given activity in the invoking scheduler solution.

This member function should be used only if `activity` is stored in the scheduler solution.

```
public IloResourceConstraint getTeardownRC(const IloResource resource) const
```

This member function returns the teardown resource constraint of the given `resource` in the invoking scheduler solution, if such a teardown resource constraint exists. In case there are several such constraints, this method returns an empty handle.

This member function should be used only if `resource` is stored in the invoking scheduler solution.

```
public IloBool hasAsNext(const IloResourceConstraint srct1, const
IloResourceConstraint srct2) const
```

This member function returns `IloTrue` if the resource constraint `srct2` is a next resource constraint of the resource constraint `rct1` in the invoking scheduler solution. Otherwise, it returns `IloFalse`.

```
public IloBool hasCapacityInformation(const IloResource resource) const
```

This member function returns `IloTrue` if information about capacity levels has been stored for the `resource` in the invoking scheduler solution. Otherwise, it returns `IloFalse`.

Capacity information is stored for resources only if sufficient information is available from the solver. In the case of `IlcScheduler`, the timetable constraint must exist on the resources in order for the capacity information to be stored in the solution. See Resource Enforcement as Global Constraint Declaration.

```
public IloBool hasPredecessors(const IloResourceConstraint rct)
```

This member function returns `IloTrue` if the resource constraint `rct` has at least one predecessor in the invoking scheduler solution. Otherwise it returns `IloFalse`.

This member function should be used only if `rct` is stored in the invoking scheduler solution.

```
public IloBool hasSuccessors(const IloResourceConstraint rct)
```

This member function returns `IloTrue` if the resource constraint `rct` has at least one successor in the invoking scheduler solution. Otherwise it returns `IloFalse`.

This member function should be used only if `rct` is stored in the invoking scheduler solution.

```
public IloBool isResourceSelected(const IloResourceConstraint rct) const
```

This member function returns `IloTrue` if the given resource constraint has a selected resource in the invoking scheduler solution. Otherwise, it returns `IloFalse`.

This member function should be used only if `rct` is stored in the scheduler solution.

```
public IloBool isSetup(const IloResourceConstraint rct) const
```

This member function returns `IloTrue` if `rct` is the setup (that is, the first) resource constraint of its resource in the invoking scheduler solution. Otherwise, it returns `IloFalse`.

This member function should be used only if `rct` is stored in the invoking scheduler solution.

```
public IloBool isSucceededBy(const IloResourceConstraint srct1, const
IloResourceConstraint srct2) const
```

This member function returns `IloTrue` if the resource constraint `srct2` succeedes the resource constraint `srct1` in the invoking scheduler solution, that is if a successor relation has been added with the member function `setSuccessor`. Otherwise it returns `IloFalse`.

```
public IloBool isTeardown(const IloResourceConstraint rct) const
```

This member function returns `IloTrue` if `rct` is the teardown (that is, the last) resource constraint of its resource in the invoking scheduler solution. Otherwise, it returns `IloFalse`.

This member function should be used only if `rct` is stored in the invoking scheduler solution.

```
public IloSchedulerSolution makeClone(IloEnv e) const
```

This member function allocates a new solution on `e` and adds to it all objects that were added to the invoking object. The newly created solution is returned.

```
public void remove(IloModel m) const
```

This member function removes all scheduler extractables (such as `IloActivity`, `IloResourceConstraint`, `IloResource` and its subclasses) that have been explicitly added to the model from the invoking scheduler solution.

```
public void setCapacityMax(const IloResourceConstraint ct, IloNum max) const
```

This member function changes the maximal capacity of `ct` in the invoking scheduler solution to `max`.

This member function should be used only if `ct` is stored in the scheduler solution.

```
public void setCapacityMin(const IloResourceConstraint ct, IloNum min) const
```

This member function changes the minimal capacity of `ct` in the invoking scheduler solution to `min`.

This member function should be used only if `ct` is stored in the scheduler solution.

```
public void setDurationMax(const IloActivity activity, IloNum max) const
```

This member function changes the maximal duration of `activity` in the invoking scheduler solution to `max`.

This member function should be used only if `activity` is stored in the scheduler solution.

```
public void setDurationMin(const IloActivity activity, IloNum min) const
```

This member function changes the minimal duration of `activity` in the invoking scheduler solution to `min`.

This member function should be used only if `activity` is stored in the scheduler solution.

```
public void setEndMax(const IloActivity activity, IloNum max) const
```

This member function changes the maximal end time of `activity` in the invoking scheduler solution to `max`.

This member function should be used only if `activity` is stored in the scheduler solution.

```
public void setEndMin(const IloActivity activity, IloNum min) const
```

This member function changes the minimal end time of `activity` in the invoking scheduler solution to `min`.

This member function should be used only if `activity` is stored in the scheduler solution.

```
public void setExternalVariableMax(const IloActivity activity, IloNum max) const
```

This member function changes the maximal value of the external variable of `activity` in the invoking scheduler solution to `max`. This member function should be used only if `activity` is stored in the scheduler solution.

```
public void setExternalVariableMin(const IloActivity activity, IloNum min) const
```

This member function changes the minimal value of the external variable of `activity` in the invoking scheduler solution to `max`.

This member function should be used only if `activity` is stored in the scheduler solution.

```
public void setLevelMax(const IloResource resource, IloNum start, IloNum end,
IloNum max) const
```

This member function changes the maximal level of `resource` in the invoking scheduler solution to `max` during the interval `[start end]`. For this purpose, the internal step function for the stored maximal level of the resource is modified.

This member function should be used only if `resource` is stored in the scheduler solution and contains information about capacity levels.

```
public void setLevelMin(const IloResource resource, IloNum start, IloNum end,
IloNum min) const
```

This member function changes the minimal level of `resource` in the invoking scheduler solution to `min` during the interval `[start end]`. For this purpose, the internal step function for the stored minimal level of the resource is modified.

This member function should be used only if `resource` is stored in the scheduler solution and contains information about capacity levels.

```
public void setLevelsStoredWithBounds(const IloResource res, IloBool
withBounds=IloTrue) const
```

If `withBounds` is equal to `IloTrue`, this member function sets the storage mode of `res` to be “with bounds.” At the construction of the scheduler object, the default mode is “without bounds.” See the section "Storing Objects" at the start of this class for more information.

This member function should be used only if `res` is stored in the scheduler solution and contains information about capacity levels.

```
public void setNext(const IloResourceConstraint srct1, const IloResourceConstraint
srct2) const
```

If the resource constaint `srct1` is stored in the invoking scheduler solution, this member function makes `srct2` the next resource constraint of `srct1` in the invoking scheduler solution. Similarly, if the resource constraint `srct2` is stored in the invoking scheduler solution, this member function makes `srct1` the previous of `srct2`.

Next and previous are restored only on unary resources.

```
public void setNonRestorable(IloResourceConstraint rc) const
```

This member function sets `rc` as non-restorable. No fields of this resource constraint will be restored. This method is equivalent to `setRestorable(rc, IloRestoreNothing);`

```
public void setNonRestorable(IloActivity activity) const
```

This member function sets `activity` as non-restorable. No fields of this activity will be restored. This method is equivalent to `setRestorable(activity, IloRestoreNothing);`

```
public void setProcessingTimeMax(const IloActivity activity, IloNum max) const
```

This member function changes the maximal processing time of `activity` in the invoking scheduler solution to `max`.

This member function should be used only if `activity` is stored in the scheduler solution.

```
public void setProcessingTimeMin(const IloActivity activity, IloNum min) const
```

This member function changes the minimal processing time of `activity` in the invoking scheduler solution to `min`.

This member function should be used only if `activity` is stored in the scheduler solution.

```
public void setRestorable(IloResourceConstraint rc, IloInt storeFields) const
```

This member function sets the data fields that will be restored for `rc` in the context of local search, the `IloRestoreSolution` goal, `IloSolution::getConstraint`, and `IloSolution::restore`. Valid values of `storeFields` are `IloRestoreNothing`, `IloRestoreAll`, and any bitwise-OR combination of `IloRestoreRCNext`, `IloRestoreRCDirectSuccessor`, `IloRestoreRCSetup`, `IloRestoreRCTeardown`, `IloRestoreRCCapacity`, and `IloRestoreRCSelected`.

```
public void setRestorable(IloActivity activity, IloInt storeFields) const
```

This member function sets the data fields that will be restored for `activity` in the context of local search, the `IloRestoreSolution` goal, `IloSolution::getConstraint`, and `IloSolution::restore`. Valid values of `storeFields` are `IloRestoreNothing`, `IloRestoreAll`, and any bitwise-OR combination of `IloRestoreActivityStart`, `IloRestoreActivityEnd`, `IloRestoreActivityDuration`, `IloRestoreActivityProcessingTime`, `IloRestoreActivityExternal`.

```
public void setSelected(IloResourceConstraint rct, const IloResource resource)
const
```

This member function sets the selected resource of `rct` in the invoking scheduler solution to `resource`. If the resource constraint is a setup (resp. teardown) resource constraint and if the resource is stored in the invoking scheduler solution, then the member function `getSetupRC` (resp. `getTeardownRC`) returns the resource constraint `rct`.

This member function should be used only if `rct` is stored in the scheduler solution.

```
public void setSetup(const IloResourceConstraint rct) const
```

This member function makes `rct` a setup resource constraint in the invoking scheduler solution. If the resource constraint has a selected resource and if this selected resource is stored in the invoking scheduler solution, then the member function `getSetupRC` returns the resource constraint `rct`.

This member function should be used only if `rct` is stored in the invoking scheduler solution.

Setups are only restored on unary resources.

```
public void setStartMax(const IloActivity activity, IloNum max) const
```

This member function changes the maximal start time of `activity` in the invoking scheduler solution to `max`.

This member function should be used only if `activity` is stored in the scheduler solution.

```
public void setStartMin(const IloActivity activity, IloNum min) const
```

This member function changes the minimal start time of `activity` in the invoking scheduler solution to `min`.

This member function should be used only if `activity` is stored in the scheduler solution.

```
public void setSuccessor(const IloResourceConstraint srct1, const
IloResourceConstraint srct2) const
```

If the resource constaint `srct1` is stored in the invoking scheduler solution, this member function makes `srct2` a successor of `srct1` in the invoking scheduler solution. Similarly, if the resource constraint `srct2` is stored in the invoking scheduler solution, this member function makes `srct1` a predecessor of `srct2`.

```
public void setTeardown(const IloResourceConstraint rct) const
```

This member function makes `rct` a teardown resource constraint in the invoking scheduler solution. If the resource constraint has a selected resource and if this selected resource is stored in the invoking scheduler solution, then the member functions `getTeardownRC` returns the resource constraint `rct`.

This member function should be used only if `rct` is stored in the invoking scheduler solution.

Teardowns are only restored on unary resources.

```
public void store(const IloResourceConstraint ct, const IlcScheduler scheduler)
const
```

This member function stores `ct` in the invoking scheduler solution. The minimal and maximal capacities of the `ct` are also stored. The data stored is that which is present in the current search state of `scheduler`.

```
public void store(const IloResource resource, const IlcScheduler scheduler) const
```

This member function stores `resource` in the invoking scheduler solution.

If the resource has a precedence graph, then the resource constraints that actually execute on the resource are stored. The resource constraints that are stored are only those that are part of the model: if the solver has created resource constraints during the search, they will *not* be added to the solution. In addition, the precedence

order between those resource constraints (if present) is stored.

```
public void store(const IloActivity activity, const IlcScheduler scheduler) const
```

This member function stores `activity` in the invoking scheduler solution. The minimal and maximal values for start time, end time, processing time, duration, and external variable are stored. The data stored is that which is present in the current search state of `scheduler`.

```
public void store(const IlcScheduler scheduler) const
```

This member function calls the `IloSolution::store(IloAlgorithm)` function on the internal `IloSolution`. That function stores the data (from the algorithm) for each object that:

- has been added to the invoking solution, and
- is used (that is, extracted) by the solver associated with `scheduler`.

The data stored is that which is present in the current search state of `scheduler`.

```
public void unsetNext(const IloResourceConstraint rct) const
```

This member function removes all the next resource constraints of `rct` in the invoking scheduler solution. Consistency is maintained; that is, the resource constraint `rct` is no longer a previous resource constraint of its former next resource constraints.

This member function should be used only if `rct` is stored in the invoking scheduler solution.

```
public void unsetNext(const IloResourceConstraint srct1, const
IloResourceConstraint srct2) const
```

This member function removes the next edge, if it exists, between `srct1` and `srct2` in the invoking schedule solution. Consistency is maintained; that is, `srct1` is no longer a previous resource constraint of `srct2`.

```
public void unsetPrecedences()
```

This member function removes all the successors and all the predecessors of all resource constraints that are stored in the invoking scheduler solution.

```
public void unsetPrecedences(const IloResource resource)
```

This member function removes all the successors and all the predecessors of all resource constraints that have `resource` as the selected resource in the invoking scheduler solution.

This member function should be used only if `rct` is stored in the invoking scheduler solution.

```
public void unsetPredecessors(const IloResourceConstraint rct) const
```

This member function removes all the predecessors of `rct` in the invoking scheduler solution. Consistency is maintained; that is, the resource constraint `rct` is no longer a successor of its former predecessors.

This member function should be used only if `rct` is stored in the invoking scheduler solution.

```
public void unsetPrev(const IloResourceConstraint rct) const
```

This member function removes all the previous resource constraints of `rct` in the invoking scheduler solution. Consistency is maintained; that is, the resource constraint `rct` is no longer a next resource constraint of its former previous resource constraints.

This member function should be used only if `rct` is stored in the invoking scheduler solution.

```
public void unsetSelected(const IloResourceConstraint rct) const
```

This member function removes the selected resource of `rct` in the invoking scheduler solution. If the resource constraint is a setup (resp. teardown) resource constraint and if its former selected resource is stored in the invoking scheduler solution, then the member function `getSetupRC` (resp. `getTeardownRC`) will no longer return the resource constraint `rct`.

This member function should be used only if `rct` is stored in the scheduler solution.

```
public void unsetSequence(const IloResource resource)
```

This member function removes all the next, previous, setup and teardown of all resource constraints that have `resource` as the selected resource in the invoking scheduler solution.

This member function should be used only if `rct` is stored in the invoking scheduler solution.

```
public void unsetSequences()
```

This member function removes all the next, previous, setup and teardown of all resource constraints that are stored in the invoking scheduler solution.

```
public void unsetSetup(const IloResourceConstraint rct) const
```

This member function removes `rct` as a setup resource constraint in the invoking scheduler solution. If the resource constraint has a selected resource and if this selected resource is stored in the invoking scheduler solution, then the member function `getSetupRC` will no longer return the resource constraint `rct`.

This member function should be used only if `rct` is stored in the invoking scheduler solution.

```
public void unsetSuccessor(const IloResourceConstraint srct1, const
IloResourceConstraint srct2) const
```

This member function removes the precedence relation between `srct1` and `srct2` from the invoking scheduler solution, if this edge exists. As a consequence, `srct2` is no longer a successor of `srct1` in the invoking scheduler solution nor is `srct1` a predecessor of `srct2`.

```
public void unsetSuccessors(const IloResourceConstraint rct) const
```

This member function removes all the successors of `rct` in the invoking scheduler solution. Consistency is maintained; that is, the resource constraint `rct` is no longer a predecessor of its former successors.

This member function should be used only if `rct` is stored in the invoking scheduler solution.

```
public void unsetTeardown(const IloResourceConstraint rct) const
```

This member function removes `rct` as a teardown resource constraint in the invoking scheduler solution. If the resource constraint has a selected resource and if this selected resource is stored in the invoking scheduler soluiton, then the member function `getTeardownRC` will no longer return the resource constraint `rct`.

This member function should be used only if `rct` is stored in the invoking scheduler solution.

Inner Enumerations

Enumeration IloResourceConstraintIteratorFilter

Definition file: ilsched/ilosolution.h
Include file: <ilsched/ilosolution.h>

The enumeration `IloResourceConstraintIteratorFilter` can be used to create a `IloSchedulerSolution::ResourceConstraintIterator` that traverses a specifed set of resource constraints such as the predecessors or the successors. The possible values are described below.

`IloPredecessors` indicates that the iterator will traverse the set of resource constraints that are predecessors of a given resource constraint.

`IloSuccessors` indicates that the iterator will traverse the set of resource constraints that are direct successors of a given resource constraint.

`IloPrevious` indicates that the iterator will traverse the set of resource constraints that are previous of a given resource constraint.

`IloNext` indicates that the iterator will traverse the set of resource constraints that are next of a given resource constraint.

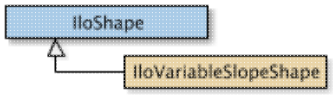
See Also: IloSchedulerSolution::ResourceConstraintIterator

Fields:

```
IloPredecessors = 0
IloSuccessors = 1
IloPrevious = 2
IloNext = 3
```

Class IloShape

Definition file: ilsched/iloresconstrainti.h
Include file: <ilsched/iloscheduler.h>



Instances of `IloShape` are generic objects describing shapes associated with resource constraints on continuous reservoirs.

The concept of *Shape* offers fine control over the production and consumption of a resource constraint on a continuous reservoir. When no shape is activated, the usual behaviour of a resource constraint is to consume or produce at a constant rate, given by the capacity and the duration of the activity. When created, an `IloVariableSlopeShape` can turn this constant rate into a Solver variable, suitable for decision and search.

See Also: IloResourceConstraint, IloReservoir, IloVariableSlopeShape

Method Summary	
`public IloBool`	`hasShape() const`
`public IloBool`	`isVariableSlopeShape() const`

Methods

`public IloBool` **`hasShape`**`() const`

This member function returns `IloTrue` if the invoking handle relates to an active shape.

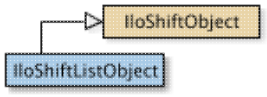
`public IloBool` **`isVariableSlopeShape`**`() const`

This member function returns `IloTrue` if the invoking handle relates to an instance of `IloVariableSlopeShape`. In this case, the handle can be safely down-cast into a `IloVariableSlopeShape` handle, using the corresponding copy-constructor.

See Also: IloVariableSlopeShape

Class IloShiftListObject

Definition file: ilsched/ilocalendar.h



The class `IloShiftListObject` inherits from the class `IloShiftObject`. It allows expressing shifts as a list of forbidden time intervals. Depending on the type of the shift list object, the time restriction can concern the whole activity execution or only its start or its end. Three different types are defined:

- `OnStart`: Shifts only concern the start of the activity. For instance, if the shift is the interval *[a,b)*, then the start of the activity must be strictly smaller than *a* or greater than *b*.
- `OnEnd`: Shifts only concern the end of the activity. For instance, if the shift is the interval *[a,b)*, then the end of the activity must be strictly smaller than *a* or greater than *b*.
- `OnOverlap`: Shifts concern the whole activity. That is the activity cannot overlap shifts. For instance, if the shift is the interval *[a,b)*, then the end of the activity must be strictly smaller than *a* or the start must be greater than *b*.

In addition, each time interval (see `IloIntervalList` in the extensions section of the *IBM ILOG Concert Technology Reference Manual*), can be associated with an integer type. In such cases, activity parameters (see `IloActivityShiftParam`) can specify which types must be ignored.

For more information, see Calendars, and Shift Object Semantic.

Constructor Summary	
public	`IloShiftListObject()`
public	`IloShiftListObject(IloShiftListObjectI * impl)`
public	`IloShiftListObject(const IloEnv env, const char * name=0)`
public	`IloShiftListObject(const IloEnv env, IloIntervalList shiftList, IloShiftListObject::Type type, const char * name=0)`

Method Summary	
public IloShiftListObjectI *	`getImpl() const`
public IloIntervalList	`getReadOnlyShiftListParam() const`
public IloShiftListObject::Type	`getType() const`
public void	`setShiftListParam(IloIntervalList shiftList)`
public void	`setType(IloShiftListObject::Type type)`

Inherited Methods from `IloShiftObject`
`getImpl`

Inner Enumeration
IloShiftListObject::Type

Constructors

`public` **`IloShiftListObject`**`()`

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

```
public IloShiftListObject(IloShiftListObjectI * impl)
```

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

```
public IloShiftListObject(const IloEnv env, const char * name=0)
```

This constructor creates a new instance of `IloShiftListObject`. By default, the shift list is empty. Its name is set to `name`

```
public IloShiftListObject(const IloEnv env, IloIntervalList shiftList,
IloShiftListObject::Type type, const char * name=0)
```

This constructor creates a new instance of `IloShiftListObject`. The shift list is set to `shiftList` and the type is set to `type`. Its name is set to `name`

Methods

```
public IloShiftListObjectI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloIntervalList getReadOnlyShiftListParam() const
```

This member function returns the time interval list that represents the set of forbidden dates of the invoking shift object.

```
public IloShiftListObject::Type getType() const
```

This member function returns the type that defines the behavior of the shifts during the search (see `IloShiftListObject::Type`).

```
public void setShiftListParam(IloIntervalList shiftList)
```

This member function sets `shiftList` as the new shift list of the invoking shift object.

```
public void setType(IloShiftListObject::Type type)
```

This member function sets `type` as the new type of the invoking shift object.

Inner Enumerations

Enumeration Type

Definition file: ilsched/ilocalendar.h

The `Type` of `IloShiftListObject` allows you to define the behavior during the search regarding the variables of concerned activities. The possible types are:

- `OnStart`: Shifts only affect the start of the activity. For instance, if the shift is the interval *[a,b)*, then the start of the activity must be strictly smaller than *a* or greater than *b*.
- `OnEnd`: Shifts only affect the end of the activity. For instance, if the shift is the interval *[a,b)*, then the end of the activity must be strictly smaller than *a* or greater than *b*.
- `OnOverlap`: Shifts affect the whole activity. That is, the activity cannot overlap shifts. For instance, if the shift is the interval *[a,b)*, then the end of the activity must be strictly smaller than *a*, or the start must be greater than *b*.

Fields:

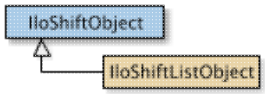
```
OnStart = 0
```

```
OnEnd = 1
```

```
OnOverlap = 2
```

Class IloShiftObject

Definition file: ilsched/ilocalendar.h



The class `IloShiftObject` allows definition of shifts which constrain the possible date execution of concerned activities. Shifts are forbidden time intervals, not necessarily defined in extension (see `ILCUSERSHIFTOBJECT`), which restrict possible starts, ends or whole executions of activities.

To express shifts it is possible to enumerate all intervals with an `IloShiftListObject`, or to write an intention definition using `ILCUSERSHIFTOBJECT`.

For more information, see Calendars and Shift Object Semantic.

Constructor Summary	
`public`	`IloShiftObject()`
`public`	`IloShiftObject(IloShiftObjectI * impl)`

Method Summary	
`public IloShiftObjectI *`	`getImpl() const`

Constructors

`public` **`IloShiftObject`**`()`

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

`public` **`IloShiftObject`**`(IloShiftObjectI * impl)`

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

Methods

`public IloShiftObjectI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object of the invoking handle.

Class IloStateResource

Definition file: ilsched/ilostate.h
Include file: <ilsched/iloscheduler.h>

An instance of the class `IloStateResource` represents a resource of infinite capacity, the state of which can vary over time. Each activity may, throughout its execution, require a state resource to be in a given state (or in any of a given set of states). Consequently, two activities may not overlap if they require incompatible states during their execution.

Parameter Classes

Possible states: (class `IloNumToAnySetStepFunction`)

This parameter describes the possible states of the state resource over time. It is directly modified by the following member functions: `IloStateResource::addPossibleStates`, `IloStateResource::addPossibleState`, `IloStateResource::removePossibleStates`, and `IloStateResource::removePossibleState`.

Must be in use intervals: (class `IloIntervalList`)

This parameter describes the set of time intervals during which the resource needs to be used by some activities. It is directly modified by the following member functions: `IloStateResource::setMustBeInUse`, and `IloStateResource::unsetMustBeInUse`.

Refer to Scheduler Overview for more information on how to share parameters among resources, and how the direct modification of parameters through the resource API may affect them. Also see `IloNumToAnySetStepFunction` and `IloIntervalList` in the extensions section of the *IBM ILOG Concert Technology Reference Manual*.

See Also: IloEnforcementLevel, IloResource, IloResourceConstraint

Constructor Summary	
public	`IloStateResource()`
public	`IloStateResource(IloStateResourceI * impl)`
public	`IloStateResource(const IloEnv env, const IloAnySet states, const char * name=0)`

Method Summary	
public void	`addPossibleState(IloNum timeMin, IloNum timeMax, IloAny state) const`
public void	`addPossibleState(IloAny state) const`
public void	`addPossibleStates(IloNum timeMin, IloNum timeMax, const IloAnySet states) const`
public void	`addPossibleStates(const IloAnySet states) const`
public IloStateResourceI *	`getImpl() const`
public IloAnySet	`getPossibleStates(IloNum time) const`
public IloAnySet	`getPossibleStates() const`
public IloBool	

	isAlwaysPossibleState(IloNum timeMin, IloNum timeMax, IloAny state) const
public IloBool	isEverPossibleState(IloNum timeMin, IloNum timeMax, IloAny state) const
public IloBool	isPossibleState(IloNum time, IloAny state) const
public void	removePossibleState(IloNum timeMin, IloNum timeMax, IloAny state) const
public void	removePossibleState(IloAny state) const
public void	removePossibleStates(IloNum timeMin, IloNum timeMax, const IloAnySet states) const
public void	removePossibleStates(const IloAnySet states) const
public void	setMustBeInUse(IloNum timeMin, IloNum timeMax) const
public void	setMustBeInUseParam(const IloIntervalList intervals) const
public void	setPossibleStatesParam(const IloNumToAnySetStepFunction states) const
public void	unsetMustBeInUse(IloNum timeMin, IloNum timeMax) const

Inherited Methods from `IloResource`
addCapacityEnforcementInterval, addTransitionTimeEnforcementInterval, areCalendarConstraintsIgnored, areCapacityConstraintsIgnored, arePrecedenceConstraintsIgnored, areSequenceConstraintsIgnored, areTransitionTimeConstraintsIgnored, getCalendar, getCalendarEnforcement, getCapacityEnforcement, getDurationEnforcement, getImpl, getPrecedenceEnforcement, getSequenceEnforcement, getTransitionTimeEnforcement, hasCalendar, ignoreCalendarConstraints, ignoreCapacityConstraints, ignorePrecedenceConstraints, ignoreSequenceConstraints, ignoreTransitionTimeConstraints, isCapacityResource, isContinuousReservoir, isDiscreteEnergy, isDiscreteResource, isKeptOpen, isReservoir, isStateResource, isUnaryResource, keepOpen, removeCapacityEnforcementInterval, removeTransitionTimeEnforcementInterval, setCalendar, setCalendarEnforcement, setCapacityEnforcement, setCapacityEnforcementIntervalsParam, setDurationEnforcement, setPrecedenceEnforcement, setResourceParam, setSequenceEnforcement, setTransitionTimeEnforcement, setTransitionTimeEnforcementIntervalsParam

Constructors

`public` **`IloStateResource`**`()`

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

`public` **`IloStateResource`**`(IloStateResourceI * impl)`

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

`public` **`IloStateResource`**`(const IloEnv env, const IloAnySet states, const char * name=0)`

This constructor creates a new instance of `IloStateResource` and adds it to the set of resources managed in the given environment. The argument `states` is the set of pointers that can be accepted as possible states for the resource. If the argument `name` is defined, it is assigned as the name of the newly created state resource.

Methods

`public void` **`addPossibleState`**`(IloNum timeMin, IloNum timeMax, IloAny state) const`

This member function adds a given `state` to the set of possible states of the invoking resource over the interval `[timeMin, timeMax)`.

`public void` **`addPossibleState`**`(IloAny state) const`

This member function adds a `state` to the set of possible states of the invoking resource.

`public void` **`addPossibleStates`**`(IloNum timeMin, IloNum timeMax, const IloAnySet states) const`

This member function adds a set of `states` to the set of possible states of the invoking resource over the interval `[timeMin, timeMax)`.

`public void` **`addPossibleStates`**`(const IloAnySet states) const`

This member function adds a set of `states` to the set of possible states of the invoking resource.

`public IloStateResourceI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public IloAnySet` **`getPossibleStates`**`(IloNum time) const`

This member function returns the set of possible states of the invoking resource at a given `time`.

`public IloAnySet` **`getPossibleStates`**`() const`

This member function returns the set of possible states of the invoking resource.

`public IloBool` **`isAlwaysPossibleState`**`(IloNum timeMin, IloNum timeMax, IloAny state) const`

This member function returns `IloTrue` if and only if it is possible that the invoking resource is in the given `state` *over the entire* interval `[timeMin, timeMax)`. Otherwise, it returns `IloFalse`.

`public IloBool` **`isEverPossibleState`**`(IloNum timeMin, IloNum timeMax, IloAny state) const`

This member function returns `IloTrue` if and only if it is possible that the invoking resource is in the given `state` *at some point* in the interval `[timeMin, timeMax)`. Otherwise, it returns `IloFalse`.

```
public IloBool isPossibleState(IloNum time, IloAny state) const
```

This member function returns `IloTrue` if and only if it is possible that the invoking resource is in the given `state` at the given `time`. Otherwise, it returns `IloFalse`.

```
public void removePossibleState(IloNum timeMin, IloNum timeMax, IloAny state) const
```

This member function states that the invoking resource must not be in the given `state` at any time in the interval `[timeMin, timeMax)`.

```
public void removePossibleState(IloAny state) const
```

This member function removes a `state` from the set of possible states of the invoking resource.

```
public void removePossibleStates(IloNum timeMin, IloNum timeMax, const IloAnySet
states) const
```

This member function states that the invoking resource must not be in any of the given `states` at any time in the interval `[timeMin, timeMax)`.

```
public void removePossibleStates(const IloAnySet states) const
```

This member function removes a set of `states` from the set of possible states of the invoking resource.

```
public void setMustBeInUse(IloNum timeMin, IloNum timeMax) const
```

This member function states that the invoking resource must be in use and that it cannot be idle over the interval `[timeMin, timeMax)`.

```
public void setMustBeInUseParam(const IloIntervalList intervals) const
```

This member function sets the argument `intervals` as the list of time intervals during which the invoking resource must be in use.

```
public void setPossibleStatesParam(const IloNumToAnySetStepFunction states) const
```

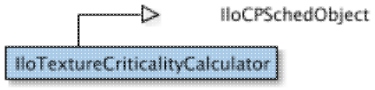
This member function sets the argument `states` as the function that describes the possible states of the invoking resource over time.

```
public void unsetMustBeInUse(IloNum timeMin, IloNum timeMax) const
```

This member function states that the invoking resource does not need to be in use during the interval `[timeMin, timeMax)`.

Class IloTextureCriticalityCalculator

Definition file: ilsched/ilotextureparami.h
Include file: <ilsched/iloscheduler.h>



A texture criticality calculator object in Scheduler Concert Technology is a modeling object that, when extracted, corresponds to a specific `IlcTextureCriticalityCalculator` object. The modeling object itself can be used to specify the criticality calculation that will be performed on resource capacity constraints in the solver.

In Scheduler Concert Technology, texture criticality calculator objects depend on the classes `IloTextureCriticalityCalculator` and `IloTextureCriticalityCalculatorI`. The class `IloTextureCriticalityCalculator` is the handle class. An instance of the class `IloTextureCriticalityCalculator` contains a data member (the handle pointer) that points to an instance of the class `IloTextureCriticalityCalculatorI` (the implementation object). If you define a new class of texture criticality calculator with the macro `ILOTEXTURECRITICALITYCALCULATOR0`, it will define the implementation class together with the corresponding virtual member function `IloTextureCriticalityCalculatorI::extract`, and a member function that returns an instance of the handle class `IloTextureCriticalityCalculator`.

For more information, see Texture Measurements.

Predefined Texture Criticality Calculators

The following functions, defined using the macro `ILOTEXTURECRITICALITYCALCULATOR0`, return instances of texture criticality calculator model objects.

```
IloTextureCriticalityCalculator IloProbabilisticCriticalityCalculator (IloEnv env);
```

This function returns a pointer to a texture criticality calculator object which, when extracted, corresponds to an `IlcProbabilisticCriticalityCalculatorI`.

```
IloTextureCriticalityCalculator IloRelativeDemandCriticalityCalculator (IloEnv env);
```

This function returns a pointer to a texture criticality calculator object which, when extracted, corresponds to an `IlcRelativeDemandCriticalityCalculatorI`.

See Also: IlcTextureCriticalityCalculator, ILOTEXTURECRITICALITYCALCULATOR0, IloTextureCriticalityCalculatorI, IlcProbabilisticCriticalityCalculatorI, IlcRelativeDemandCriticalityCalculatorI

Constructor Summary	
public	`IloTextureCriticalityCalculator()`
public	`IloTextureCriticalityCalculator(IloTextureCriticalityCalculatorI * impl)`

Method Summary	
`public IloTextureCriticalityCalculatorI *`	`getImpl() const`

Constructors

`public` **`IloTextureCriticalityCalculator`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

```
public IloTextureCriticalityCalculator(IloTextureCriticalityCalculatorI * impl)
```

This constructor creates a handle object from a pointer to an implementation object.

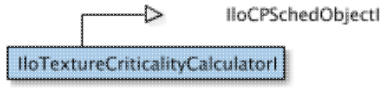
Methods

```
public IloTextureCriticalityCalculatorI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

Class IloTextureCriticalityCalculatorI

Definition file: ilsched/ilotextureparami.h
Include file: <ilsched/iloscheduler.h>



Texture criticality calculator objects in Scheduler Concert Technology depend on the classes `IloTextureCriticalityCalculatorI` and `IloTextureCriticalityCalculator`. The class `IloTextureCriticalityCalculatorI` is the implementation class. If you define a new class of factory with the macro `ILOTEXTURECRITICALITYCALCULATOR0`, it will define this implementation class together with the corresponding virtual member function `IloTextureCriticalityCalculatorI::extract`, and with a member function that returns an instance of the handle class `IloTextureCriticalityCalculator`.

For more information, see Texture Measurements.

See Also: IloTextureCriticalityCalculator, ILOTEXTURECRITICALITYCALCULATOR0, IlcTextureCriticalityCalculator

Method Summary	
`public virtual IlcTextureCriticalityCalculatorI *`	`extract(const IloSolver & solver) const`
`protected void`	`use(const IloSolver &, const IloExtractable &) const`

Methods

```
public virtual IlcTextureCriticalityCalculatorI * extract(const IloSolver & solver)
const
```

This virtual function implements the extraction of the invoking texture criticality calculator into an `IlcTextureCriticalityCalculatorI*` by the `solver` given as argument. Note that this member function must be defined by using the macro `ILOTEXTURECRITICALITYCALCULATOR0`.

```
protected void use(const IloSolver &, const IloExtractable &) const
```

This member function can only be called from within the member function `IloTextureCriticalityCalculatorI::extract` (that is, only in the code of a macro `ILOTEXTURECRITICALITYCALCULATOR0`). It states that the invoking texture criticality calculator currently in the process of being extracted by the solver given as argument uses the extractable given as the second argument. As a consequence, the extractable given as the second argument will be immediately extracted by the solver.

Class IloTextureParam

Definition file: ilsched/ilotextureparam.h
Include file: <ilsched/iloscheduler.h>

Parameters are used to change the default behavior and characteristics of activities and resources. An instance of `IloTextureParam` modifies the existence and type of texture measurement that is created on a resource. By default, no texture measurement is created.

Texture measurements are measurements of some aspect of the search state and can be used to guide heuristics. See Texture Measurements.

For more information, see Texture Measurements.

See Also: IloDiscreteResource, IlcResourceTexture, IloTextureSuccessorGoal, IloTextureAltSuccessorGoal

Constructor Summary	
public	`IloTextureParam()`
public	`IloTextureParam(IloTextureParamI * impl)`
public	`IloTextureParam(const IloEnv env, const char * name=0)`

Method Summary	
public void	`addIgnoreInterval(const IloIntervalList) const`
public void	`addIgnoreInterval(IloNum start, IloNum end) const`
public void	`addIgnoreIntervalOnDuration(IloNum start, IloNum duration) const`
public void	`addPeriodicIgnoreInterval(IloNum start, IloNum duration, IloNum period, IloNum end) const`
public void	`emptyIgnoreIntervals() const`
public IloIntervalList	`getIgnoreIntervals() const`
public IloTextureParamI *	`getImpl() const`
public void	`removeIgnoreInterval(const IloIntervalList) const`
public void	`removeIgnoreInterval(IloNum start, IloNum end) const`
public void	`removeIgnoreIntervalOnDuration(IloNum start, IloNum duration) const`
public void	`removePeriodicIgnoreInterval(IloNum start, IloNum duration, IloNum period, IloNum end) const`
public void	`setCriticalityCalculator(IloTextureCriticalityCalculator)`
public void	`setHeuristicBeta(IloNum) const`
public void	`setRandomGenerator(IloRandom) const`
public void	`setRCTextureFactory(IloRCTextureFactory)`
public void	`unsetRandomGenerator() const`

Constructors

`public` **`IloTextureParam`**`()`

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

`public` **`IloTextureParam`**`(IloTextureParamI * impl)`

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

`public` **`IloTextureParam`**`(const IloEnv env, const char * name=0)`

This constructor creates a new instance of `IloTextureParam`, with the default value that no texture measurement will be created.

Methods

`public void` **`addIgnoreInterval`**`(const IloIntervalList) const`

This member function adds an interval list to the list of intervals that are ignored by the texture measurement. The new intervals are merged with existing intervals that they overlap, if any.

`public void` **`addIgnoreInterval`**`(IloNum start, IloNum end) const`

This member function adds an interval to the list of intervals that are ignored by the texture measurement. The ignored interval is `[start, end)`. The new interval is merged with existing intervals that it overlaps, if any.

`public void` **`addIgnoreIntervalOnDuration`**`(IloNum start, IloNum duration) const`

This member function adds an interval to the list of intervals that are ignored by the texture measurement. The ignored interval is `[start, start+duration)`. The new interval is merged with existing intervals that it overlaps, if any.

`public void` **`addPeriodicIgnoreInterval`**`(IloNum start, IloNum duration, IloNum period, IloNum end) const`

This member function adds a set of intervals to the list of intervals that are ignored by the texture measurement. For every `i >= 0` such that `start + i * period < end`, an interval of `[start + i * period, start + duration + i * period)` is added. Adding a new interval that overlaps with an already existing interval results in the merging of the intervals.

`public void` **`emptyIgnoreIntervals`**`() const`

This member function removes all the intervals from the ignored intervals of the invoking parameter.

`public IloIntervalList` **`getIgnoreIntervals`**`() const`

This member function returns a list of intervals that will be ignored by the texture measurements. No texture calculations will take place on these intervals and the criticality value for all time points within the intervals is 0.

```
public IloTextureParamI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public void removeIgnoreInterval(const IloIntervalList) const
```

This member function removes all intervals ignored by the texture measurement during the intervals in the interval list.

```
public void removeIgnoreInterval(IloNum start, IloNum end) const
```

This member function removes all intervals ignored by the texture measurement between `start` and `end`. If `start` is inside an existing interval `[start1, end1)`, that is, `start1 < start < end1`, this results in an ignored interval `[start1, start)`. If `end` is inside an interval `[start2, end2)` this results in an ignored interval `[end, end2)`.

```
public void removeIgnoreIntervalOnDuration(IloNum start, IloNum duration) const
```

This member function removes all ignored intervals on the invoking parameter between `start` and `start+duration`.

```
public void removePeriodicIgnoreInterval(IloNum start, IloNum duration, IloNum
period, IloNum end) const
```

This member function removes ignored intervals from the invoking parameter. More precisely, for every `i >= 0` such that `start + i * period < end`, this function removes all intervals between `start + i * period` and `start + duration + i * period`.

```
public void setCriticalityCalculator(IloTextureCriticalityCalculator)
```

This member function sets the texture criticality calculator object given as argument as the criticality calculator of the invoking texture parameter.

```
public void setHeuristicBeta(IloNum) const
```

This member function sets the beta value to be used with the random number generator. If no random number generator is used, this function does nothing. For details on the use of the beta argument, see `IlcResourceTexture::setRandomGenerator`.

```
public void setRandomGenerator(IloRandom) const
```

This member function sets the random number generator to be used in choosing the critical time point on the texture measurement. By default, no random number generator is used.

```
public void setRCTextureFactory(IloRCTextureFactory)
```

This member function sets the texture factory object given as argument as the texture factory of the invoking texture parameter.

```
public void unsetRandomGenerator() const
```

This member function removes the random number generator, meaning that no random numbers will be used in choosing the critical time point on the texture measurement.

Class IloTimeBoundConstraint

Definition file: ilsched/iloactivity.h
Include file: <ilsched/iloscheduler.h>

Instances of the class `IloTimeBoundConstraint` are temporal constraints. These temporal constraints express constraints on the time interval in which an activity is to be scheduled. (Other temporal constraints — instances of `IloPrecedenceConstraint`— express precedence between activities in a schedule.)

This class inherits from the Concert Technology class `IloConstraint`, which is documented in the *IBM ILOG Concert Technology Reference Manual*.

Instances of this class are created by these member functions:

- `IloActivity::startsBefore`
- `IloActivity::endsBefore`
- `IloActivity::startsAt`
- `IloActivity::endsAt`
- `IloActivity::startsAfter`
- `IloActivity::endsAfter`.

For more information, see `IloConstraint` in the IBM ILOG Concert Technology Reference Manual, and Temporal Relations.

See Also: IloActivity, IloActivityConstraintsParam

Constructor Summary	
`public`	`IloTimeBoundConstraint()`
`public`	`IloTimeBoundConstraint(IloTimeBoundConstraintI * impl)`

Method Summary	
`public IloActivity`	`getActivity() const`
`public IloTimeBoundConstraintI *`	`getImpl() const`
`public IloNum`	`getTimeBound() const`
`public IloNumVar`	`getTimeBoundVariable() const`
`public IloTimeBoundConstraintType`	`getType() const`
`public IloBool`	`hasTimeBoundVariable() const`

Constructors

`public` **`IloTimeBoundConstraint`**`()`

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

`public` **`IloTimeBoundConstraint`**`(IloTimeBoundConstraintI * impl)`

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

Methods

```
public IloActivity getActivity() const
```

This member function returns the activity of the invoking time-bound constraint.

```
public IloTimeBoundConstraintI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloNum getTimeBound() const
```

This member function returns the time-bound of the invoking time-bound constraint.

```
public IloNumVar getTimeBoundVariable() const
```

This member function returns the time-bound variable of the invoking time-bound constraint.

```
public IloTimeBoundConstraintType getType() const
```

This member function returns the type of the invoking time-bound constraint.

```
public IloBool hasTimeBoundVariable() const
```

This member function returns `IloTrue` if the invoking time-bound constraint has a time-bound variable. Otherwise, it returns `IloFalse`.

Class IloTimeWindowNHoodI::IloTimeWindow

Definition file: ilsched/ilolnsgoals.h
Include file: <ilsched/iloscheduler.h>

The embedded class `IloTimeWindow` represents a time interval [start..end).

Method Summary	
public void	display(ostream & stream) const
public IloInt	getDuration() const
public IloInt	getEnd() const
public IloInt	getStart() const
public IloBool	intersects(const IloTimeWindow & tw) const
public IloBool	operator==(const IloTimeWindow & tw) const

Methods

`public void` **`display`**`(ostream & stream) const`

This member function displays the time interval of [*start..end*) where *start* is the start of the time time interval and *end* is the end of the time interval.

`public IloInt` **`getDuration`**`() const`

This member function returns the duration of the time interval.

`public IloInt` **`getEnd`**`() const`

This member function returns the end of the time interval.

`public IloInt` **`getStart`**`() const`

This member function returns the start of the time interval.

`public IloBool` **`intersects`**`(const IloTimeWindow & tw) const`

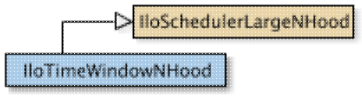
This member function returns `IloTrue` if the time window provided as argument intersects with this time window.

`public IloBool` **`operator==`**`(const IloTimeWindow & tw) const`

This member function returns `IloTrue` if the time window provided as argument starts and ends at the same time as this time window.

Class IloTimeWindowNHood

Definition file: ilsched/iloInsgoals.h
Include file: <ilsched/iloscheduler.h>



An instance of this class represents a time window neighborhood.

The size of this neighborhood is the number of time windows created internally and depends on two parameters (integers) provided to the constructor: *windowSize* and *windowStep*.

The parameter `windowSize` represents the number of activities in any time window.

The parameter `windowStep` represents the number of activities skipped to move to the next time window.

For example, suppose there are 20 activities and `windowSize` equals 10 and `windowStep` is 5. Three time windows are then created, with the first one containing the first 10 activities with index in [0..10), the second one containing the activities with index in [5..15), and the last one containing the activities with index from [15..20).

New member functions are available to check if an activity or a resource constraint is before or after the selected activities and resource constraints.

Default behavior is to only restore resource constraints that are not selected (not in the current time window). This behavior can be changed by specifying appropriate predicates (for example, see member function `IloSchedulerLargeNHoodI::setRestoreActivityStartPredicate`.

See Also: IloSchedulerLargeNHood, IloTimeWindowNHoodI

Constructor Summary	
public	`IloTimeWindowNHood(IloEnv env, IloInt windowSize, IloInt windowStep, const char * name=0)`
public	`IloTimeWindowNHood(IloEnv env, IloInt windowSize, IloInt windowStep, IloComparator< IloTimeWindowNHoodI::IloTimeWindow > comparator, const char * name=0)`
public	`IloTimeWindowNHood(IloEnv env, IloInt windowSize, IloInt windowStep, IloComparator< IloTimeWindowNHoodI::IloTimeWindow > comparator, IloPredicate< IloActivity > predicate, const char * name=0)`

Method Summary	
`public IloBool`	`isAfterSelected(IloResourceConstraint rc) const`
`public IloBool`	`isAfterSelected(IloActivity activity) const`
`public IloBool`	`isBeforeSelected(IloResourceConstraint rc) const`
`public IloBool`	`isBeforeSelected(IloActivity activity) const`

Inherited Methods from `IloSchedulerLargeNHood`
`defineRestoreInfo, defineSelected, finalizeDelta, getCurrentSolution, getRestoreActivityDurationPredicate, getRestoreActivityEndPredicate, getRestoreActivityExternalPredicate, getRestoreActivityProcessingTimePredicate, getRestoreActivityStartPredicate, getRestoreExtractablePredicate, getRestoreRCCapacityPredicate, getRestoreRCDirectPredecessorPredicate, getRestoreRCDirectSuccessorPredicate, getRestoreRCNextPredicate,`

```
getRestoreRCPrevPredicate, getRestoreRCSelectedPredicate,
getRestoreRCSetupPredicate, getRestoreRCTeardownPredicate, isSelected,
setRestoreActivityDurationPredicate, setRestoreActivityEndPredicate,
setRestoreActivityExternalPredicate, setRestoreActivityProcessingTimePredicate,
setRestoreActivityStartPredicate, setRestoreExtractablePredicate,
setRestoreRCCapacityPredicate, setRestoreRCDirectPredecessorPredicate,
setRestoreRCDirectSuccessorPredicate, setRestoreRCNextPredicate,
setRestoreRCPrevPredicate, setRestoreRCSelectedPredicate,
setRestoreRCSetupPredicate, setRestoreRCTeardownPredicate
```

Constructors

```
public IloTimeWindowNHood(IloEnv env, IloInt windowSize, IloInt windowStep, const
char * name=0)
```

This constructor creates a time window neighborhood. The parameter `windowSize` specifies the number of activities of any time window considered, and the parameter `windowStep` specifies the number of activities to skip to move to the next time window.

An error is raised if `windowStep` and `windowSize` are not positive numbers, or if `windowStep` is greater than `windowSize`.

```
public IloTimeWindowNHood(IloEnv env, IloInt windowSize, IloInt windowStep,
IloComparator< IloTimeWindowNHoodI::IloTimeWindow > comparator, const char *
name=0)
```

This constructor creates a time window neighborhood. The parameter `windowSize` specifies the number of activities of any time window considered, and the parameter `windowStep` specifies the number of activities to skip to move to the next time window.

The parameter `comparator` is used to specify in which order the time windows should be considered.

```
public IloTimeWindowNHood(IloEnv env, IloInt windowSize, IloInt windowStep,
IloComparator< IloTimeWindowNHoodI::IloTimeWindow > comparator, IloPredicate<
IloActivity > predicate, const char * name=0)
```

This constructor creates a time window neighborhood. The parameter `windowSize` specifies the number of activities of any time window considered, and the parameter `windowStep` specifies the number of activities to skip to move to the next time window.

The parameter `comparator` is used to specify in which order the time windows should be considered.

The parameter `predicate` is used to specify the activities in the current solution to use to build the time windows.

Methods

```
public IloBool isAfterSelected(IloResourceConstraint rc) const
```

This member function returns `IloTrue` if the start in the current solution of the activity associated with the resource constraint is greater than the end of the time interval.

```
public IloBool isAfterSelected(IloActivity activity) const
```

This member function returns `IloTrue` if the end in the current solution of the activity is greater than or equal to the end of the time interval.

```
public IloBool isBeforeSelected(IloResourceConstraint rc) const
```

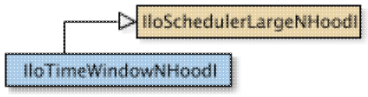
This member function returns `IloTrue` if the start in the current solution of the activity associated with the resource contraint is less than the start of the time interval.

```
public IloBool isBeforeSelected(IloActivity activity) const
```

This member function returns `IloTrue` if the start in the current solution of the activity is less than the start of the time interval.

Class IloTimeWindowNHoodI

Definition file: ilsched/iloInsgoals.h
Include file: <ilsched/iloscheduler.h>



An instance of this class represents a time window neighborhood.

The size of this neighborhood is the number of time windows created internally and depends on two parameters (integers) provided to the constructor: *windowSize* and *windowStep*.

The parameter `windowSize` represents the number of activities in any time window.

The parameter `windowStep` represents the number of activities skipped to move to the next time window.

For example, suppose there are 20 activities and `windowSize` equals 10 and `windowStep` is 5. Three time windows are then created, with the first one containing the first 10 activities with index in [0..10), the second one containing the activities with index in [5..15), and the last one containing the activities with index from [15..20).

New member functions are available to check if an activity or a resource constraint is before or after the selected activities and resource constraints.

Default behavior is to only restore resource constraints that are not selected (not in the current time window). This behavior can be changed by specifying appropriate predicates (for example, see the member function `IloSchedulerLargeNHoodI::setRestoreActivityStartPredicate`.

See Also: IloSchedulerLargeNHoodI, IloTimeWindowNHood

Constructor and Destructor Summary	
public	`IloTimeWindowNHoodI(IloEnv env, IloInt windowSize, IloInt windowStep, IloComparator< IloTimeWindowNHoodI::IloTimeWindow > comparator, IloPredicate< IloActivity > predicate, const char * name=0)`

Method Summary	
`public virtual IloSolution`	`defineSelected(IloSolver solver, IloInt index)`
`public IloBool`	`isAfterSelected(IloResourceConstraint rc) const`
`public IloBool`	`isAfterSelected(IloActivity activity) const`
`public IloBool`	`isBeforeSelected(IloResourceConstraint rc) const`
`public IloBool`	`isBeforeSelected(IloActivity activity) const`

Inherited Methods from `IloSchedulerLargeNHoodI`
`define, defineRestoreInfo, defineSelected, finalizeDelta, getCurrentSolution, getRestoreActivityDurationPredicate, getRestoreActivityEndPredicate, getRestoreActivityExternalPredicate, getRestoreActivityProcessingTimePredicate, getRestoreActivityStartPredicate, getRestoreExtractablePredicate, getRestoreInfo, getRestoreRCCapacityPredicate, getRestoreRCDirectPredecessorPredicate, getRestoreRCDirectSuccessorPredicate, getRestoreRCNextPredicate, getRestoreRCPrevPredicate, getRestoreRCSelectedPredicate, getRestoreRCSetupPredicate, getRestoreRCTeardownPredicate, isSelected, setRestoreActivityDurationPredicate, setRestoreActivityEndPredicate, setRestoreActivityExternalPredicate, setRestoreActivityProcessingTimePredicate,`

setRestoreActivityStartPredicate, setRestoreExtractablePredicate, setRestoreRCCapacityPredicate, setRestoreRCDirectPredecessorPredicate, setRestoreRCDirectSuccessorPredicate, setRestoreRCNextPredicate, setRestoreRCPrevPredicate, setRestoreRCSelectedPredicate, setRestoreRCSetupPredicate, setRestoreRCTeardownPredicate

Inner Class
IloTimeWindowNHoodI::IloTimeWindow

Constructors and Destructors

```
public IloTimeWindowNHoodI(IloEnv env, IloInt windowSize, IloInt windowStep,
IloComparator< IloTimeWindowNHoodI::IloTimeWindow > comparator, IloPredicate<
IloActivity > predicate, const char * name=0)
```

This constructor creates a time window neighborhood. The parameter `windowSize` specifies the number of activities of any time window considered, and the parameter `windowStep` specifies the number of activities to skip to move to the next time window.

The parameter `comparator` is used (if not an empty handle) to specify in which order the time windows should be considered.

The parameter `predicate` is used (if not an empty handle) to specify the activities in the current solution to use to build the time windows.

Parameters `windowSize` and `windowStep` must be positive integers. An error is raised if the parameter `windowStep` is greater than parameter `windowSize`.

Methods

```
public virtual IloSolution defineSelected(IloSolver solver, IloInt index)
```

This pure virtual member function returns the set of decision variables, or instances of `IloExtractable`, on which to focus the search.

```
public IloBool isAfterSelected(IloResourceConstraint rc) const
```

This member function returns `IloTrue` if the start in the current solution of the activity associated with the resource constraint is greater than the end of the time interval.

```
public IloBool isAfterSelected(IloActivity activity) const
```

This member function returns `IloTrue` if the end in the current solution of the activity is greater than or equal to the end of the time interval.

```
public IloBool isBeforeSelected(IloResourceConstraint rc) const
```

This member function returns `IloTrue` if the start in the current solution of the activity associated to the resource contraint is less than the start of the time interval.

```
public IloBool isBeforeSelected(IloActivity activity) const
```

This member function returns `IloTrue` if the start in the current solution of the activity is less than the start of the time interval.

Class IloTransitionCost

Definition file: ilsched/ilotransition.h
Include file: <ilsched/iloscheduler.h>

The class `IloTransitionCost` allows definition of a transition cost for a unary resource. More precisely, an instance of `IloTransitionCost` allows association of an instance of either `IloTransitionParam` or a user-defined `IloTransitionCostObject` with a unary resource.

If the `IloTransitionCost` is defined with an instance of `IloTransitionParam`, the transition costs on that unary resource will then be computed by using the `IloTransitionParam` and the transition types of activities.

If the `IloTransitionCost` is defined with an instance of `IloTransitionCostObject`, the transition costs on that unary resource will then be computed by using the object extracted from the `IloTransitionCostObject` and the virtual transition costs it defines.

Note that several transition costs can be defined on the same unary resource.

The type of the transition cost is set to *Next* whenever one of these member functions is called: `IloTransitionCost::getNextCostExpr`, or `IloTransitionCost::getSetupCostExpr`.

The type of the transition cost is set to *Prev* whenever either of these member functions is called: `IloTransitionCost::getPrevCostExpr`, or `IloTransitionCost::getTeardownCostExpr`.

This class inherits from the IBM ILOG Concert Technology class `IloExtractable`. That class is documented in the *IBM ILOG Concert Technology Reference Manual*.

For more information, see Transition Costs.

See Also: IloUnaryResource, IloTransitionParam, IloResourceParam, IloResourceConstraint, IloTransitionCostObject

Constructor Summary	
public	`IloTransitionCost()`
public	`IloTransitionCost(IloTransitionCostI * impl)`
public	`IloTransitionCost(const IloUnaryResource resource, const IloTransitionParam param, const char * name=0)`
public	`IloTransitionCost(const IloUnaryResource resource, const IloTransitionParam param, IloBool isNext, const char * name=0)`
public	`IloTransitionCost(const IloUnaryResource resource, const IloTransitionCostObject tobj, const char * name=0)`
public	`IloTransitionCost(const IloUnaryResource resource, const IloTransitionCostObject tobj, IloBool isNext, const char * name=0)`

Method Summary	
public IloNum	`getCost(const IloResourceConstraint rct1x, const IloResourceConstraint rct2x) const`
public IloNumVar	`getCostSumVar() const`
public IloTransitionCostI *	`getImpl() const`
public IloIntExprArg	`getNextCostExpr(const IloResourceConstraint rct) const`

public IloNum	getNextCostMax(IloResourceConstraint rct) const
public IloNum	getNextCostMin(IloResourceConstraint rct) const
public IloIntExprArg	getPrevCostExpr(const IloResourceConstraint rct) const
public IloNum	getPrevCostMax(IloResourceConstraint rct) const
public IloNum	getPrevCostMin(IloResourceConstraint rct) const
public IloNum	getSetupCost(const IloResourceConstraint rct) const
public IloIntExprArg	getSetupCostExpr() const
public IloNum	getSetupCostMax() const
public IloNum	getSetupCostMin() const
public IloNum	getTeardownCost(IloResourceConstraint rct) const
public IloIntExprArg	getTeardownCostExpr() const
public IloNum	getTeardownCostMax() const
public IloNum	getTeardownCostMin() const
public IloTransitionCostObject	getTransitionCostObject() const
public IloBool	hasCostSumVar() const
public IloBool	isNextTransitionCost() const
public IloBool	isPrevTransitionCost() const
public void	setCostSumVar(const IloNumVar sum) const
public void	setNextCostMax(IloResourceConstraint rct, IloNum value)
public void	setNextCostMin(IloResourceConstraint rct, IloNum value)
public void	setPrevCostMax(IloResourceConstraint rct, IloNum value)
public void	setPrevCostMin(IloResourceConstraint rct, IloNum value)
public void	setSetupCostMax(IloInt value) const
public void	setSetupCostMin(IloInt value) const
public void	setTeardownCostMax(IloInt value) const
public void	setTeardownCostMin(IloInt value) const
public void	setTransitionParam(const IloTransitionParam param) const

Constructors

public **IloTransitionCost**()

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

public **IloTransitionCost**(IloTransitionCostI * impl)

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

```
public IloTransitionCost(const IloUnaryResource resource, const IloTransitionParam
param, const char * name=0)
```

This constructor creates a transition cost for the unary resource `resource` and adds it to the set of transition costs managed by the resource. Transition costs on this resource will be computed using the transition parameter `param`.

```
public IloTransitionCost(const IloUnaryResource resource, const IloTransitionParam
param, IloBool isNext, const char * name=0)
```

This constructor creates a transition cost for the unary resource `resource` and adds it to the set of transition costs managed by the resource. Transition costs on this resource will be computed by using the transition parameter `param`. If `isNext` equals `IloTrue`, the transition cost will be of type *Next*; that is, the member functions `IloTransitionCost::getPrevCostExpr` and `IloTransitionCost::getTeardownCostExpr` will not be available. If `isNext` equals `IloFalse`, the transition cost will be of type *Prev*; that is, the member functions `IloTransitionCost::getNextCostExpr` and `IloTransitionCost::getSetupCostExpr` will not be available.

```
public IloTransitionCost(const IloUnaryResource resource, const
IloTransitionCostObject tobj, const char * name=0)
```

This constructor creates a transition cost for the unary resource `resource` and adds it to the set of transition costs managed by the resource. Transition costs on this resource will be computed using the transition cost object `tobj`.

```
public IloTransitionCost(const IloUnaryResource resource, const
IloTransitionCostObject tobj, IloBool isNext, const char * name=0)
```

This constructor creates a transition cost for the unary resource `resource` and adds it to the set of transition costs managed by the resource. Transition costs on this resource will be computed by using the transition cost object `tobj`. If `isNext` equals `IloTrue`, the transition cost will be of type *Next*; that is, the member functions `IloTransitionCost::getPrevCostExpr` and `IloTransitionCost::getTeardownCostExpr` will not be available. If `isNext` equals `IloFalse`, the transition cost will be of type *Prev*; that is, the member functions `IloTransitionCost::getNextCostExpr` and `IloTransitionCost::getSetupCostExpr` will not be available.

Methods

```
public IloNum getCost(const IloResourceConstraint rct1x, const
IloResourceConstraint rct2x) const
```

This member function returns the transition cost between the two resource constraints given as arguments. The transition cost is computed by using the transition type of the activities and the current transition parameter. More precisely, if activity of `rct1x` is of type `t1`, and activity `rct2x` is of type `t2`, then this member function will return the value corresponding to line `t1` and column `t2` on the transition parameter. In other words, it will return the value that was set by using `IloTransitionParam::setValue(t1,t2)`. This member function will throw an exception if the transition cost is defined with a transition cost object, as the transition function will be created only at extraction time.

```
public IloNumVar getCostSumVar() const
```

This member function returns the cost sum variable. The cost sum variable represents the sum of the transition costs (including setup and transition costs) of all the resource constraints on the resource of the invoking transition cost.

```
public IloTransitionCostI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloIntExprArg getNextCostExpr(const IloResourceConstraint rct) const
```

This member function returns an expression that represents the cost between `rct` and the resource constraint that will be next to `rct`. If `rct` is the last resource constraint on the resource, then this cost is the teardown cost of `rct`. `rct` must be a resource constraint of the resource of the invoking transition cost. The transition cost is set to type *Next*.

```
public IloNum getNextCostMax(IloResourceConstraint rct) const
```

This member function returns the maximum transition cost between `rct` and the resource constraint that will be next to `rct`. If `rct` is the last resource constraint on the resource, then this cost is the maximum teardown cost of `rct`. `rct` must be a resource constraint of the resource of the invoking transition cost.

```
public IloNum getNextCostMin(IloResourceConstraint rct) const
```

This member function returns the minimimum transition cost between `rct` and the resource constraint that will be next to `rct`. If `rct` is the last resource constraint on the resource, then this cost is the minimum teardown cost of `rct`. `rct` must be a resource constraint of the resource of the invoking transition cost.

```
public IloIntExprArg getPrevCostExpr(const IloResourceConstraint rct) const
```

This member function returns an expression that represents the cost between `rct` and the resource constraint that will be previous to `rct`. If `rct` is the first resource constraint on the resource, then this cost is the setup cost of `rct`. `rct` must be a resource constraint of the resource of the invoking transition cost. The transition cost is set to type *Prev*.

```
public IloNum getPrevCostMax(IloResourceConstraint rct) const
```

This member function returns the maximum transition cost between `rct` and the resource constraint that will be previous to `rct`. If `rct` is the first resource constraint on the resource, then this cost is the maximum setup cost of `rct`. `rct` must be a resource constraint of the resource of the invoking transition cost.

```
public IloNum getPrevCostMin(IloResourceConstraint rct) const
```

This member function returns the minimum transition cost between `rct` and the resource constraint that will be previous to `rct`. If `rct` is the first resource constraint on the resource, then this cost is the minimum setup cost of `rct`. `rct` must be a resource constraint of the resource of the invoking transition cost.

```
public IloNum getSetupCost(const IloResourceConstraint rct) const
```

This member function returns the setup cost of the resource constraint `rct`. The setup cost is computed by using the transition type of the activity and the current transition parameter. This member function will throw an exception if the transition cost is defined with a transition cost object, as the transition function will be created only at extraction time.

```
public IloIntExprArg getSetupCostExpr() const
```

This member function returns an expression that represents the setup cost for the invoking transition cost. That is, the setup cost of the first resource constraint to be processed on the resource is returned. The transition cost is set to type *Next*.

```
public IloNum getSetupCostMax() const
```

This member function returns the maximum setup cost of the invoking transition cost. That is, the maximum setup cost of the first resource constraint to be processed on the resource is returned.

```
public IloNum getSetupCostMin() const
```

This member function returns the minimum setup cost of the invoking transition cost. That is, the minimum setup cost of the first resource constraint to be processed on the resource is returned.

```
public IloNum getTeardownCost(IloResourceConstraint rct) const
```

This member function returns the teardown cost of the resource constraint `rct`. The teardown cost is computed by using the transition type of the activity and the current transition parameter. This member function will throw an exception if the transition cost is defined with a transition cost object, as the transition function will be created only at extraction time.

```
public IloIntExprArg getTeardownCostExpr() const
```

This member function returns an expression that represents the teardown cost for the invoking transition cost. That is, the teardown cost of the last resource constraint to be processed on the resource is returned. The transition cost is set to type *Prev*.

```
public IloNum getTeardownCostMax() const
```

This member function returns the maximum teardown cost of the invoking transition cost. That is, the maximum teardown cost of the last resource constraint to be processed on the resource is returned.

```
public IloNum getTeardownCostMin() const
```

This member function returns the minimum teardown cost of the invoking transition cost. That is, the minimum teardown cost of the last resource constraint to be processed on the resource is returned.

```
public IloTransitionCostObject getTransitionCostObject() const
```

In case the invoking transition cost has been created with a user defined transition cost object, this member function will return this transition cost object. Otherwise, it will return an empty handle.

```
public IloBool hasCostSumVar() const
```

This member function returns `IloTrue` if the invoking transition cost has been created as a cost sum variable. The cost sum variable represents the sum of the transition costs (including setup and transition costs) of all the resource constraints on the resource of the invoking transition cost. A cost sum variable is automatically created when calling the member function `IloTransitionCost::getCostSumVar` or `IloTransitionCost::setCostSumVar`.

```
public IloBool isNextTransitionCost() const
```

This member function returns `IloTrue` if the invoking transition cost is of type next. Otherwise it returns false.

```
public IloBool isPrevTransitionCost() const
```

This member function returns `IloTrue` if the invoking transition cost is of type next. Otherwise it returns false.

```
public void setCostSumVar(const IloNumVar sum) const
```

This member function sets `sum` as the cost sum variable of the invoking transition cost. The cost sum variable represents the sum of the transition costs (including setup and transition costs) of all the resource constraints on the resource of the invoking transition cost.

```
public void setNextCostMax(IloResourceConstraint rct, IloNum value)
```

This member function sets the maximum transition cost between `rct` and the resource constraint that will be next to `rct`. If `rct` is the last resource constraint on the resource, then this cost is the maximum teardown cost of `rct`. `rct` must be a resource constraint of the resource of the invoking transition cost. The transition cost is set to type *Next*.

```
public void setNextCostMin(IloResourceConstraint rct, IloNum value)
```

This member function sets the minimimum transition cost between `rct` and the resource constraint that will be next to `rct`. If `rct` is the last resource constraint on the resource, then this cost is the minimum teardown cost of `rct`. `rct` must be a resource constraint of the resource of the invoking transition cost. The transition cost is set to type *Next*.

```
public void setPrevCostMax(IloResourceConstraint rct, IloNum value)
```

This member function sets the maximum transition cost between `rct` and the resource constraint that will be previous to `rct`. If `rct` is the first resource constraint on the resource, then this cost is the maximum setup cost of `rct`. `rct` must be a resource constraint of the resource of the invoking transition cost. The transition cost is set to type *Prev*.

```
public void setPrevCostMin(IloResourceConstraint rct, IloNum value)
```

This member function sets the minimum transition cost between `rct` and the resource constraint that will be previous to `rct`. If `rct` is the first resource constraint on the resource, then this cost is the minimum setup cost of `rct`. `rct` must be a resource constraint of the resource of the invoking transition cost. The transition cost is set to type *Prev*.

```
public void setSetupCostMax(IloInt value) const
```

This member function sets the maximum setup cost of the invoking transition cost. That is, the maximum setup cost of the first resource constraint to be processed on the resource is returned. The transition cost is set to type *Next*.

```
public void setSetupCostMin(IloInt value) const
```

This member function sets the minimum setup cost of the invoking transition cost. That is, the minimum setup cost of the first resource constraint to be processed on the resource is returned. The transition cost is set to type *Next*.

```
public void setTeardownCostMax(IloInt value) const
```

This member function sets the maximum teardown cost of the invoking transition cost. That is, the maximum teardown cost of the last resource constraint to be processed on the resource is returned. The transition cost is set to type *Prev*.

```
public void setTeardownCostMin(IloInt value) const
```

This member function sets the minimum teardown cost of the invoking transition cost. That is, the minimum teardown cost of the last resource constraint to be processed on the resource is returned. The transition cost is set to type *Prev*.

```
public void setTransitionParam(const IloTransitionParam param) const
```

This member function allows changing the transition parameter used to compute the transition cost.

Class IloTransitionCostObject

Definition file: ilsched/ilotransition.h
Include file: <ilsched/iloscheduler.h>

Transition cost objects in Scheduler Concert Technology depend on the classes `IloTransitionCostObjectI` and `IloTransitionCostObject`. The class `IloTransitionCostObject` is the handle class. An instance of the class `IloTransitionCostObject` contains a data member (the handle pointer) that points to an instance of the class `IloTransitionCostObjectI` (the implementation object). If you define a new class of transition cost object with the macro `ILOTRANSITIONCOSTOBJECT0`, it will define the implementation class together with the corresponding virtual member function `IloTransitionCostObjectI::extract`, and a member function that returns an instance of the handle class `IloTransitionCostObject`.

For more information, see Transition Costs.

See Also: IloTransitionCostObjectI, ILOTRANSITIONCOSTOBJECT0, IloTransitionCost

Constructor Summary	
`public`	`IloTransitionCostObject()`
`public`	`IloTransitionCostObject(IloTransitionCostObjectI * impl)`

Method Summary	
`public IloTransitionCostObjectI *`	`getImpl() const`
`public void`	`setChanged()`

Constructors

`public` **`IloTransitionCostObject`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloTransitionCostObject`**`(IloTransitionCostObjectI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public IloTransitionCostObjectI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object of the invoking handle.

`public void` **`setChanged`**`()`

This member function states that the invoking transition cost object has been changed and therefore needs to be re-extracted.

Class IloTransitionCostObjectI

Definition file: ilsched/ilotransition.h
Include file: <ilsched/iloscheduler.h>

Transition cost objects in Scheduler Concert Technology depend on the classes `IloTransitionCostObjectI` and `IloTransitionCostObject`. The class `IloTransitionCostObjectI` is the implementation class. If you define a new class of transition cost object with the macro `ILOTRANSITIONCOSTOBJECT0`, it will define this implementation class together with the corresponding virtual member function `IloTransitionCostObjectI::extract`, and with a member function that returns an instance of the handle class `IloTransitionCostObject`.

For more information, see Transition Costs.

See Also: IloTransitionCostObject, ILOTRANSITIONCOSTOBJECT0, IloTransitionCost

Method Summary	
`public virtual IlcTransitionCostObjectI *`	`extract(const IloSolver & solver) const`
`protected void`	`use(const IloSolver &, const IloExtractable &) const`

Methods

```
public virtual IlcTransitionCostObjectI * extract(const IloSolver & solver) const
```

This virtual function implements the extraction of the invoking transition cost object into an `IlcTransitionCostObjectI*` by the `solver` given as argument. Note that this member function must be defined by using the macro `ILOTRANSITIONCOSTOBJECT0`.

```
protected void use(const IloSolver &, const IloExtractable &) const
```

This member function can only be called from within the member function `IloTransitionCostObjectI::extract` (that is, only in the code of a macro `ILOTRANSITIONCOSTOBJECT0`). It states that the invoking transition cost object currently in the process of being extracted by the solver given as argument uses the extractable given as the second argument. As a consequence, the extractable given as the second argument will be immediately extracted by the solver currently performing the extraction of the invoking transition cost, which must be given as first argument to this member function.

Class IloTransitionParam

Definition file: ilsched/ilotransition.h
Include file: <ilsched/iloscheduler.h>

The class `IloTransitionParam` allows representation of transition, setup and teardown times and/or costs on a resource.

In the class `IloTransitionParam`, setup and teardown times and/or costs are stored as an array of non-negative numbers indexed by the transition type of activities. By default, the setup and teardown values are considered to be zero.

Transition times and/or costs between resource constraints are stored as a square table of non-negative numbers, indexed by the transition type of activities. By default, it is initially filled with zeros. The table may or may not be symmetric, that is, the transition time and/or cost may or may not be different if an activity follows or precedes another one. If a table is declared as symmetric, only the required triangular half of the table is allocated and only that half needs to be filled. The index of the line of the table is the transition type of the preceding activity. The index of the column of the table is the transition type of the following activity.

Instances of `IloTransitionParam` are used to build transition times on resources (see also `IloTransitionTime`) and transition costs on unary resources (see also `IloTransitionCost`).

For more information, see Transition Costs, Transition Times, Parameter Classes.

See Also: IloActivity, IloTransitionCost, IloTransitionTime

Constructor Summary	
public	`IloTransitionParam()`
public	`IloTransitionParam(IloTransitionParamI * impl)`
public	`IloTransitionParam(const IloEnv env, IloInt size, IloBool isSymmetric=IloFalse, const char * name=0)`
public	`IloTransitionParam(const IloEnv env, IloInt size, IloNum ** ttable, const char * name=0)`
public	`IloTransitionParam(const IloEnv env, IloInt size, IloNum ** ttable, const IloNumArray setups, const IloNumArray teardowns, const char * name=0)`
public	`IloTransitionParam(const IloEnv env, IloInt size, IloNum ** ttable, const IloNumArray setupOrTeardowns, IloBool setup=IloTrue, const char * name=0)`

Method Summary	
public IloBool	`checkTriangularInequality() const`
public IloTransitionParamI *	`getImpl() const`
public IloNum	`getSetup(IloInt line) const`
public IloInt	`getSize() const`
public IloNum	`getTeardown(IloInt line) const`
public IloNum	`getValue(IloInt line, IloInt column) const`
public IloBool	`isSymmetric() const`
public void	`setSetup(IloInt line, IloNum value) const`

public void	setSetupArray(const IloNumArray setups) const
public void	setTeardown(IloInt line, IloNum value) const
public void	setTeardownArray(const IloNumArray teardowns) const
public void	setTransitionTable(IloNum ** ttable) const
public void	setValue(IloInt line, IloInt column, IloNum value) const

Constructors

public **IloTransitionParam**()

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

public **IloTransitionParam**(IloTransitionParamI * impl)

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

public **IloTransitionParam**(const IloEnv env, IloInt size, IloBool isSymmetric=IloFalse, const char * name=0)

This constructor creates a new instance of `IloTransitionParam`. The argument `size`, which must be a strictly non-negative integer, gives the number of lines and columns of the transition table as well as the size of the arrays of setup and teardown values. The table and arrays are initially filled with zeroes. The boolean argument `isSymmetric` expresses the fact that the table is symmetric. If `isSymmetric` is `IloTrue`, only half the table needs to be defined. By default, a transition time parameter is not symmetric.

public **IloTransitionParam**(const IloEnv env, IloInt size, IloNum ** ttable, const char * name=0)

This constructor creates a new instance of `IloTransitionParam`. The argument `size`, which must be a strictly non-negative integer, gives the number of lines and columns of the transition table as well as the size of the arrays of setup and teardown values. The table is initially filled with the values of the array of arrays `ttable`. The setup and teardown values are assumed to be zero.

public **IloTransitionParam**(const IloEnv env, IloInt size, IloNum ** ttable, const IloNumArray setups, const IloNumArray teardowns, const char * name=0)

This constructor creates a new instance of `IloTransitionParam`. The argument `size`, which must be a strictly non-negative integer, gives the number of lines and columns of the transition table as well as the size of the arrays of setup and teardown values. The table is initially filled with the values of the array of arrays `ttable`. The setup and teardown values are initialized by copying the numerical arrays `setups` and `teardowns`.

public **IloTransitionParam**(const IloEnv env, IloInt size, IloNum ** ttable, const IloNumArray setupOrTeardowns, IloBool setup=IloTrue, const char * name=0)

This constructor creates a new instance of `IloTransitionParam`. The argument `size`, which must be a strictly non-negative integer, gives the number of lines and columns of the transition table as well as the size of the arrays of setup and teardown values. The table is initially filled with the values of the array of arrays `ttable`. If the value of the argument `setup` is `IloTrue`, the setup values are initialized by copying the numerical array `setupOrTeardowns` and the teardown values are supposed to be equal to zero. If the value of the argument

setup is IloFalse, the teardown values are initialized by copying the numerical array setupOrTeardowns and the setup values are supposed to be equal to zero.

Methods

public IloBool **checkTriangularInequality**() const

This member function returns IloTrue if and only if the invoking transition param satisfies the triangular inequality. Triangular inequality is defined as:

- for all i,j,k in [0,size): value(i,j) <= value(i,k) + value(k,j)
- for all i,j in [0,size): setup(i) <= setup(j) + value(j,i), and
- teardown(i) <= teardown(j) + value(i,j).

public IloTransitionParamI * **getImpl**() const

This member function returns a pointer to the implementation object of the invoking handle.

public IloNum **getSetup**(IloInt line) const

This member function returns the value of the setup for the argument line. An assert will be violated if the argument line is not a non-negative integer strictly smaller than the size of the transition parameter.

public IloInt **getSize**() const

This member function returns the size of the invoking transition parameter.

public IloNum **getTeardown**(IloInt line) const

This member function returns the value of the teardown for the argument line. An assert will be violated if the argument line is not a non-negative integer strictly smaller than the size of the transition parameter.

public IloNum **getValue**(IloInt line, IloInt column) const

This member function returns the value of the table of the invoking transition parameter for the line line and for the column column. The argument line is the transition type of the preceding activity. The argument column is the transition type of the following activity. An assert will be violated if the arguments line and column are not non-negative integers strictly smaller than the size of the transition parameter.

public IloBool **isSymmetric**() const

This member function returns IloTrue if the table of the invoking transition parameter was initially created as a symmetric table. Otherwise, it returns IloFalse.

public void **setSetup**(IloInt line, IloNum value) const

This member function sets value as the new setup value for the index index. An assert will be violated if the argument line is not a non-negative integer strictly smaller than the size of the transition parameter, or if the

argument `value` is not a non-negative number.

```
public void setSetupArray(const IloNumArray setups) const
```

This member function initializes the array of setups with a copy of the numerical array `setups`. An assert will be violated if the size of the array `setups` not the same as the size of the invoking transition parameter.

```
public void setTeardown(IloInt line, IloNum value) const
```

This member function sets `value` as the new teardown value for the argument `line`. An assert will be violated if the argument `line` is not a non-negative integer strictly smaller than the size of the transition parameter or if the argument `value` is not a non-negative number.

```
public void setTeardownArray(const IloNumArray teardowns) const
```

This member function initializes the array of teardowns with a copy of the numerical array `teardowns`. An assert will be violated if the size of the array `teardown` is not the same as the size of the invoking transition parameter.

```
public void setTransitionTable(IloNum ** ttable) const
```

This member function initializes the table of the invoking transition parameter with the values of the array of arrays `ttable`.

```
public void setValue(IloInt line, IloInt column, IloNum value) const
```

This member function sets `value` as the value of the table of the invoking transition parameter for `line` and `column`. The argument `line` is the transition type of the preceding activity. The argument `column` is the transition type of the following activity. An assert will be violated if the arguments `line` and `column` are not non-negative integers strictly smaller than the size of the transition parameter or if the argument `value` is not a non-negative number.

Class IloTransitionTime

Definition file: ilsched/ilotransition.h
Include file: <ilsched/iloscheduler.h>

The class `IloTransitionTime` allows definition of transition time for a resource.

More precisely, an instance of `IloTransitionTime` allows association of an instance of either an `IloTransitionParam` or a user-defined `IloTransitionTimeObject` with a resource.

If the `IloTransitionTime` is defined with an instance of `IloTransitionParam`, the transition times on that resource will be computed by using the `IloTransitionParam` and the transition types of the activities.

If the `IloTransitionTime` is defined with an instance of `IloTransitionTimeObject`, the transition times on that resource will be computed by using the object extracted from the `IloTransitionTimeObject` and the virtual transition times it defines.

Note that at most one transition time can be active on a resource. Any attempt to create a transition time on a resource that has already been associated with a transition time will override the previous transition time.

This class inherits from the IBM® ILOG® Concert Technology class `IloExtractable`. That class is documented in the *IBM ILOG Concert Technology Reference Manual*.

For more information, see Calendars and Transition Times.

See Also: IloResource, IloResourceParam, IloResourceConstraint, IloTransitionParam, IlcTransitionTimeObject

Constructor Summary	
public	`IloTransitionTime()`
public	`IloTransitionTime(IloTransitionTimeI * impl)`
public	`IloTransitionTime(const IloResource resource, const IloTransitionParam param, const char * name=0)`
public	`IloTransitionTime(const IloResource resource, const IloTransitionTimeObject tobj, const char * name=0)`

Method Summary	
public IloTransitionTimeI *	`getImpl() const`
public IloNum	`getTime(const IloResourceConstraint rct1x, const IloResourceConstraint rct2x) const`
public IloTransitionTimeObject	`getTransitionTimeObject() const`
public IloBool	`isSuspended() const`
public void	`setSuspended(IloBool suspended=IloTrue)`
public void	`setTransitionParam(const IloTransitionParam param) const`

Constructors

`public` **`IloTransitionTime`**`()`

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

```
public IloTransitionTime(IloTransitionTimeI * impl)
```

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

```
public IloTransitionTime(const IloResource resource, const IloTransitionParam
param, const char * name=0)
```

This constructor creates a transition time for `resource`. Transition times on `resource` will be computed by using the transition parameter `param`. If the resource `resource` was already associated a transition time, the new transition time overrides the previous one.

```
public IloTransitionTime(const IloResource resource, const IloTransitionTimeObject
tobj, const char * name=0)
```

This constructor creates a transition time for `resource`. Transition times on `resource` will be computed by using the transition time object `tobj`. If `resource` was already associated a transition time, the new transition time overrides the previous one.

Methods

```
public IloTransitionTimeI * getImpl() const
```

This member function returns a pointer to the implementation object of the invoking handle.

```
public IloNum getTime(const IloResourceConstraint rct1x, const
IloResourceConstraint rct2x) const
```

This member function returns the transition time between the two resource constraints given as arguments. The transition time is computed by using the transition type of the activities and the current transition parameter. More precisely, if activity of `rct1x` is of type `t1`, and activity `rct2x` is of type `t2`, then this member function will return the value corresponding to row `t1` and column `t2` on the transition parameter. In other words, it will return the value that was set by using `IloTransitionParam::setValue(t1,t2)`. This member function will throw an exception if the transition time is defined with a transition time object as the transition function will be created only at extraction time.

```
public IloTransitionTimeObject getTransitionTimeObject() const
```

If the invoking transition time has been created with a user-defined transition time object, this member function will return this transition time object. Otherwise, it will return an empty handle.

```
public IloBool isSuspended() const
```

This member function returns `IloTrue` if and only if the invoking transition time has been declared to be suspended by breaks.

```
public void setSuspended(IloBool suspended=IloTrue)
```

This member function allows specifying whether the invoking transition time is suspended by breaks or not. By default, a transition time is not suspended by breaks.

```
public void setTransitionParam(const IloTransitionParam param) const
```

This member function allows changing the transition parameter used to compute the transition times of the resources of the invoking instance of `IloTransitionTime`.

Class IloTransitionTimeObject

Definition file: ilsched/ilotransition.h
Include file: <ilsched/iloscheduler.h>

Transition time objects in Scheduler Concert Technology depend on the classes `IloTransitionTimeObjectI` and `IloTransitionTimeObject`. The class `IloTransitionTimeObject` is the handle class. An instance of the class `IloTransitionTimeObject` contains a data member (the handle pointer) that points to an instance of the class `IloTransitionTimeObjectI` (the implementation object). If you define a new class of transition time object with the macro `ILOTRANSITIONTIMEOBJECT0`, it will define the implementation class together with the corresponding virtual member function `IloTransitionTimeObjectI::extract`, and a member function that returns an instance of the handle class `IloTransitionTimeObject`.

For more information, see Transition Times.

See Also: IloTransitionTimeObjectI, ILOTRANSITIONTIMEOBJECT0, IloTransitionTime

Constructor Summary	
`public`	`IloTransitionTimeObject()`
`public`	`IloTransitionTimeObject(IloTransitionTimeObjectI * impl)`

Method Summary	
`public IloTransitionTimeObjectI *`	`getImpl() const`
`public void`	`setChanged()`

Constructors

`public` **`IloTransitionTimeObject`**`()`

This constructor creates an empty handle. You must initialize it before you use it.

`public` **`IloTransitionTimeObject`**`(IloTransitionTimeObjectI * impl)`

This constructor creates a handle object from a pointer to an implementation object.

Methods

`public IloTransitionTimeObjectI *` **`getImpl`**`() const`

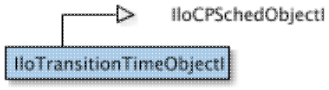
This member function returns a pointer to the implementation object of the invoking handle.

`public void` **`setChanged`**`()`

This member function states that the invoking transition time object has been changed, and therefore needs to be re-extracted.

Class IloTransitionTimeObjectI

Definition file: ilsched/ilotransition.h
Include file: <ilsched/iloscheduler.h>



Transition time objects in Scheduler Concert Technology depend on the classes `IloTransitionTimeObjectI` and `IloTransitionTimeObject`. The class `IloTransitionTimeObjectI` is the implementation class. If you define a new class of transition time object with the macro `ILOTRANSITIONTIMEOBJECT0`, it will define this implementation class together with the corresponding virtual member function `IloTransitionTimeObjectI::extract`, and with a member function that returns an instance of the handle class `IloTransitionTimeObject`.

For more information, see Transition Times.

See Also: IloTransitionTimeObject, ILOTRANSITIONTIMEOBJECT0, IloTransitionTime

Method Summary	
`public virtual IlcTransitionTimeObjectI *`	`extract(const IloSolver & solver) const`
`protected void`	`use(const IloSolver &, const IloExtractable &) const`

Methods

```
public virtual IlcTransitionTimeObjectI * extract(const IloSolver & solver) const
```

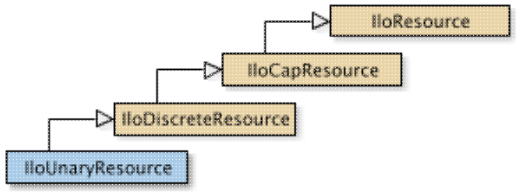
This virtual function implements the extraction of the invoking transition time object into an `IlcTransitionTimeObjectI*` by the `solver` given as argument. Note that this member function must be defined by using the macro `ILOTRANSITIONTIMEOBJECT0`.

```
protected void use(const IloSolver &, const IloExtractable &) const
```

This member function can only be called from within the member function `IloTransitionTimeObjectI::extract` (that is, only in the code of a macro `ILOTRANSITIONTIMEOBJECT0`). It states that the invoking transition time object currently in the process of being extracted by the solver given as argument uses the extractable given as the second argument. As a consequence, the extractable given as the second argument will be immediately extracted by the solver currently performing the extraction of the invoking transition time, which must be given as first argument to this member function.

Class IloUnaryResource

Definition file: ilsched/ilounary.h
Include file: <ilsched/iloscheduler.h>



Unary Resource.

An instance of the class `IloUnaryResource` represents a resource with capacity one.

See Also: IloDiscreteResource, IloEnforcementLevel, IloResourceConstraint, IloTransitionCost

Constructor Summary	
public	IloUnaryResource()
public	IloUnaryResource(IloUnaryResourceI * impl)
public	IloUnaryResource(const IloEnv env, const char * name=0)

Method Summary	
public IloUnaryResourceI *	getImpl() const

Inherited Methods from `IloDiscreteResource`

getCapacityMax, getCapacityMaxMax, getCapacityMaxMin, getCapacityMin, getCapacityMinMax, getCapacityMinMin, getImpl, setCapacityMax, setCapacityMaxParam, setCapacityMin, setCapacityMinParam

Inherited Methods from `IloCapResource`

addMaxTextureIgnoreInterval, addMaxTextureIgnoreInterval, addMaxTextureIgnoreIntervalOnDuration, addMaxTexturePeriodicIgnoreInterval, addMinTextureIgnoreInterval, addMinTextureIgnoreInterval, addMinTextureIgnoreIntervalOnDuration, addMinTexturePeriodicIgnoreInterval, emptyMaxTextureIgnoreIntervals, emptyMinTextureIgnoreIntervals, getCapacity, getImpl, getInitialOccupation, getInitialOccupationMax, getInitialOccupationMin, hasInitialOccupation, hasMaxTextureMeasurement, hasMinTextureMeasurement, removeMaxTextureIgnoreInterval, removeMaxTextureIgnoreInterval, removeMaxTextureIgnoreIntervalOnDuration, removeMaxTexturePeriodicIgnoreInterval, removeMinTextureIgnoreInterval, removeMinTextureIgnoreInterval, removeMinTextureIgnoreIntervalOnDuration, removeMinTexturePeriodicIgnoreInterval, setCapacity, setInitialOccupation, setInitialOccupation, setInitialOccupationParam, setInitialOccupationParam, setMaxTextureHeuristicBeta, setMaxTextureParam, setMaxTextureRandomGenerator, setMinTextureHeuristicBeta, setMinTextureParam, setMinTextureRandomGenerator, unsetMaxTextureRandomGenerator, unsetMinTextureRandomGenerator

Inherited Methods from `IloResource`

addCapacityEnforcementInterval, addTransitionTimeEnforcementInterval, areCalendarConstraintsIgnored, areCapacityConstraintsIgnored, arePrecedenceConstraintsIgnored, areSequenceConstraintsIgnored,

areTransitionTimeConstraintsIgnored, getCalendar, getCalendarEnforcement, getCapacityEnforcement, getDurationEnforcement, getImpl, getPrecedenceEnforcement, getSequenceEnforcement, getTransitionTimeEnforcement, hasCalendar, ignoreCalendarConstraints, ignoreCapacityConstraints, ignorePrecedenceConstraints, ignoreSequenceConstraints, ignoreTransitionTimeConstraints, isCapacityResource, isContinuousReservoir, isDiscreteEnergy, isDiscreteResource, isKeptOpen, isReservoir, isStateResource, isUnaryResource, keepOpen, removeCapacityEnforcementInterval, removeTransitionTimeEnforcementInterval, setCalendar, setCalendarEnforcement, setCapacityEnforcement, setCapacityEnforcementIntervalsParam, setDurationEnforcement, setPrecedenceEnforcement, setResourceParam, setSequenceEnforcement, setTransitionTimeEnforcement, setTransitionTimeEnforcementIntervalsParam

Constructors

`public` **`IloUnaryResource`**`()`

This constructor creates an instance that is empty, that is, one whose handle pointer is null. You must assign it a value before you access it. Any attempt to access it before assignment leads to undefined behaviour.

`public` **`IloUnaryResource`**`(IloUnaryResourceI * impl)`

This constructor creates an instance of the handle class from the pointer to an instance of the implementation class.

`public` **`IloUnaryResource`**`(const IloEnv env, const char * name=0)`

This constructor creates a new instance of `IloUnaryResource` and adds it to the set of resources managed in the given environment. The capacity of the resource is 1 (one). If the argument `name` is defined, it is used as the name of the newly created resource.

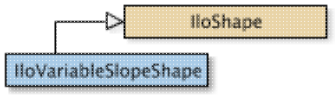
Methods

`public IloUnaryResourceI * ` **`getImpl`**`() const`

This member function returns a pointer to the implementation object of the invoking handle.

Class IloVariableSlopeShape

Definition file: ilsched/iloresconstrainti.h
Include file: <ilsched/iloscheduler.h>



Instances of `IloVariableSlopeShape` provide control of resource constraint rate of production or consumption, through a Solver variable.

For a given resource constraint, the rate of production or consumption is defined as the variable: *Slope = Capacity / Duration*. The corresponding variable can be accessed and set through the member functions `IloVariableSlopeShape::getSlopeVar` and `IloVariableSlopeShape::setSlopeVar`.

Since the slope variable uses floating point representation, the finest precision available on the values of the slope is proportional to *epsilon = 2.21*10^-16*. Hence, any value comprised in the interval *(Slope*(1-epsilon), Slope]* will be considered to verify the constraint *Slope = Capacity / Duration*.

See Also: IloResourceConstraint

Constructor Summary	
`public`	`IloVariableSlopeShape(const IloShape & shape)`

Method Summary	
`public IloNumVar`	`getSlopeVar() const`
`public void`	`setSlopeVar(IloNumVar var) const`

Inherited Methods from `IloShape`
`hasShape, isVariableSlopeShape`

Constructors

`public` **`IloVariableSlopeShape`**`(const IloShape & shape)`

This copy-constructor provides a safe down-cast of a generic instance of `IloShape` into an instance of `IloVariableSlopeShape`. In debug mode, an assertion failure will be raised if the `IloShape` is not a instance of `IloVariableSlopeShape`.

Methods

`public IloNumVar` **`getSlopeVar`**`() const`

This member function returns the variable that parameterizes the slope of the shape.

See Also: IloShape, IloResourceConstraint

`public void` **`setSlopeVar`**`(IloNumVar var) const`

This member function sets the variable that parameterizes the slope of the shape.

See Also: IloShape, IloResourceConstraint

Class IloAltResSet::Iterator

Definition file: ilsched/iloaltresset.h
Include file: <ilsched/iloscheduler.h>

An instance of this class traverses the list of resources in a given instance of `IloAltResSet`.

See Also: IloAltResSet, IloResource

Constructor Summary	
public	Iterator(IloAltResSet set)

Method Summary	
public IloBool	ok() const
public IloResource	operator*()
public Iterator &	operator++()

Constructors

`public` **`Iterator`**`(IloAltResSet set)`

This constructor creates an iterator to traverse all the resources that are stored in a given instance of `IloAltResSet`.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all the resources have been scanned by the iterator.

`public IloResource` **`operator*`**`()`

This operator returns the current instance of `IloResource`, the one to which the invoking iterator points. This operator must not be called if the iterator does not point to a valid position, that is, one to which the member function `Iterator::ok` returns `IloFalse`.

`public Iterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator to the next instance of `IloResource`.

Class IlcResource::ResourceConstraintDeltaIterator

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>

This iterator should be used only in a demon triggered by the event `IlcResource::whenRankedFirstRC` or the event `IlcResource::whenRankedLastRC`. This iterator traverses the set of resource constraints that have been ranked first (or last) since last execution of the event `IlcResource::whenRankedFirstRC` (or `IlcResource::whenRankedLastRC`).

The only possible filters for building this iterator are `RankedFirst` and `RankedLast`. Using another filter will throw an exception at construction time.

See Also: IlcResource::RankFilter

Constructor Summary	
public	`ResourceConstraintDeltaIterator(const IlcResource resource, IlcResource::RankFilter filter)`

Method Summary	
`public IlcBool`	`ok() const`
`public IlcResourceConstraint`	`operator*() const`
`public ResourceConstraintDeltaIterator &`	`operator++()`
`public ResourceConstraintDeltaIterator &`	`operator--()`

Constructors

`public` **`ResourceConstraintDeltaIterator`**`(const IlcResource resource, IlcResource::RankFilter filter)`

This constructor creates an iterator to traverse the delta subset of resource constraints specified by the filter on the unary or state `resource` given as first argument. This constructor should be used only if the ranking information is available on the `resource` (see `IlcResource::hasRankInfo`).

In case the filter is `RankedFirst`, this constructor allows iteration over the new resource constraints that have been ranked first on a resource since last triggering of the event `IlcResource::whenRankedLastRC`. The iterator is initialized at the first newly ranked first resource constraint, and the `ResourceConstraintDeltaIterator::operator++`, member function will traverse the set of newly ranked first resource constraints in chronological order (with respect to the start/end time of activities).

In case the filter is `RankedLast`, this constructor allows iteration over the new resource constraints that have been ranked last on a resource since last triggering of the event `IlcResource::whenRankedLastRC`. The iterator is initialized at the first newly ranked last resource constraints, and the `ResourceConstraintDeltaIterator::operator++`, member function will traverse the set of newly ranked first resource constraints in anti-chronological order (with respect to the start/end time of activities).

Methods

`public IlcBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if the subset of resource constraints has been completely scanned by the iterator.

```
public IlcResourceConstraint operator*() const
```

This operator returns the current instance of `IlcResourceConstraint`, the one to which the invoking iterator points. If the iterator is set past the end position, then this operator returns an empty handle.

```
public ResourceConstraintDeltaIterator & operator++()
```

This left-increment operator shifts the current position of the iterator to the next instance of `IlcResourceConstraint`.

```
public ResourceConstraintDeltaIterator & operator--()
```

This left-increment operator shifts the current position of the iterator to the previous instance of `IlcResourceConstraint`.

Class IlcResource::ResourceConstraintIterator

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>

An instance of this class traverses specific subsets of resource constraints on a given unary or state resource. More precisely, it allows traversing the following items:

- the subset of resource constraints that have been ranked first on the resource (`RankedFirst`).
- the subset of resource constraints that have been ranked last on the resource (`RankedLast`).
- the subset of resource constraints that are still not ranked first or last on the resource (`NotRanked`).
- the subset of resource constraints that are possibly ranked first on the resource (`PossibleFirst`).
- the subset of resource constraints that are possibly ranked last on the resource (`PossibleLast`).

See Also: IlcResource::RankFilter

Constructor Summary	
public	`ResourceConstraintIterator(const IlcResource resource, IlcResource::RankFilter filter)`
public	`ResourceConstraintIterator(const IlcResource resource, IlcResource::RankFilter filter, const IlcResourceConstraint)`

Method Summary	
public IlcBool	`ok() const`
public IlcResourceConstraint	`operator*() const`
public ResourceConstraintIterator &	`operator++()`
public ResourceConstraintIterator &	`operator--()`

Constructors

`public` **`ResourceConstraintIterator`**`(const IlcResource resource, IlcResource::RankFilter filter)`

This constructor creates an iterator to traverse the subset of resource constraints specified by the filter on the unary or state `resource` given as the first argument. In case of a filter `RankedFirst`, the resource constraints are traversed in the chronological order (with respect to the start/end time of activities) with the `ResourceConstraintIterator::operator++` member function. In case of a filter `RankedLast`, the resource constraints are traversed in the anti-chronological order with the `operator++` member function.

This constructor should be used only if the ranking information is available on the `resource` (see `IlcResource::hasRankInfo`).

`public` **`ResourceConstraintIterator`**`(const IlcResource resource, IlcResource::RankFilter filter, const IlcResourceConstraint)`

This constructor creates an iterator to traverse the subset of resource constraints specified by the filter on the unary or state `resource` given as the first argument, and starting at the resource constraint `rct` given as argument. An exception will be thrown in case the ranked status of the resource constraint (ranked first or last, possible first or last) is not compatible with the filter.

This constructor should be used only if the ranking information is available on the `resource` (see `IlcResource::hasRankInfo`).

In case of a filter `RankedFirst`, the resource constraints are traversed in the chronological order (with respect to the start/end time of activities) with the `ResourceConstraintIterator::operator++`, member function. In case of a filter `RankedLast`, the resource constraints are traversed in the anti-chronological order with the `operator++` member function.

Methods

```
public IlcBool ok() const
```

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if the subset of resource constraints has been completely scanned by the iterator.

```
public IlcResourceConstraint operator*() const
```

This operator returns the current instance of `IlcResourceConstraint`, the one to which the invoking iterator points. If the iterator is set past the end position, then this operator returns an empty handle.

```
public ResourceConstraintIterator & operator++()
```

This left-increment operator shifts the current position of the iterator to the next instance of `IlcResourceConstraint`.

```
public ResourceConstraintIterator & operator--()
```

This left-increment operator shifts the current position of the iterator to the previous instance of `IlcResourceConstraint`.

Class IloSchedulerSolution::ResourceConstraintIterator

Definition file: ilsched/ilosolution.h
Include file: <ilsched/iloscheduler.h>

An instance of this class traverses the list of `IloResourceConstraint` instances that have been stored in an `IloSchedulerSolution`.

See Also: IloResourceConstraint, IloSchedulerSolution, IloSchedulerSolution::ResourceIterator, IloSchedulerSolution::ActivityIterator

Constructor and Destructor Summary	
public	`ResourceConstraintIterator(IloSchedulerSolution sol)`
public	`ResourceConstraintIterator(IloSchedulerSolution sol, IloResource r)`
public	`ResourceConstraintIterator(IloSchedulerSolution sol, IloActivity a)`
public	`ResourceConstraintIterator(IloSchedulerSolution sol, IloResourceConstraint ct, IloBool iterateOnSuccessors=IloTrue)`
public	`ResourceConstraintIterator(IloSchedulerSolution sol, IloResourceConstraint ct, IloSchedulerSolution::IloResourceConstraintIteratorFilter filter)`

Method Summary	
public IloBool	`ok() const`
public IloResourceConstraint	`operator*()`
public ResourceConstraintIterator &	`operator++()`

Constructors and Destructors

`public` **`ResourceConstraintIterator`**`(IloSchedulerSolution sol)`

This constructor creates an iterator to traverse all the resource constraints that are stored in the given scheduler solution.

`public` **`ResourceConstraintIterator`**`(IloSchedulerSolution sol, IloResource r)`

This constructor creates an iterator to traverse all the resource constraints that have resource `r` as the selected resource.

`public` **`ResourceConstraintIterator`**`(IloSchedulerSolution sol, IloActivity a)`

This constructor creates an iterator that traverses all the resource constraints on activity `a` that are stored in the solution `sol`.

`public` **`ResourceConstraintIterator`**`(IloSchedulerSolution sol, IloResourceConstraint ct, IloBool iterateOnSuccessors=IloTrue)`

When the boolean `iterateOnSuccessors` is true, this constructor creates an iterator that traverses all the resoure constraints that succeed the resource constraint `ct` in the scheduler solution `sol`. When the boolean `iterateOnSuccessors` is false, this constructor creates an iterator that traverses all the resource constraints that precede the resource constraint `ct` in the scheduler solution `sol`.

```
public ResourceConstraintIterator(IloSchedulerSolution sol, IloResourceConstraint
ct, IloSchedulerSolution::IloResourceConstraintIteratorFilter filter)
```

This constructor creates an iterator that traverses the subset of resource constraints specified by the filter. For instance, to iterate on all successors of `ct` that are stored in the scheduler solution `sol`, use filter value `IloSchedulerSolution::IloSuccessors`. See enum `IloSchedulerSolution::IloResourceConstraintIteratorFilter` for all possible values of the filter.

Methods

```
public IloBool ok() const
```

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all the resources have been scanned by the iterator.

```
public IloResourceConstraint operator*()
```

This operator returns the current instance of `IloResourceConstraint`, the one to which the invoking iterator points. If the iterator is set past the end position, then this operator returns an empty handle.

```
public ResourceConstraintIterator & operator++()
```

This left-increment operator shifts the current position of the iterator to the next instance of `IloResourceConstraint`.

Class IloSchedulerSolution::ResourceIterator

Definition file: ilsched/ilosolution.h
Include file: <ilsched/iloscheduler.h>

An instance of this class traverses the list of `IloResource` instances that have been stored in an `IloSchedulerSolution`.

See Also: IloResource, IloSchedulerSolution, IloSchedulerSolution::ResourceConstraintIterator, IloSchedulerSolution::ActivityIterator

Constructor Summary	
`public`	`ResourceIterator(IloSchedulerSolution sol)`

Method Summary	
`public IloBool`	`ok() const`
`public IloResource`	`operator*()`
`public ResourceIterator &`	`operator++()`

Constructors

`public` **`ResourceIterator`**`(IloSchedulerSolution sol)`

This constructor creates an iterator to traverse all the resources that are stored in the given scheduler solution.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all the resources have been scanned by the iterator.

`public IloResource` **`operator*`**`()`

This operator returns the current instance of `IloResource`, the one to which the invoking iterator points. If the iterator is set past the end position, then this operator returns an empty handle.

`public ResourceIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator to the next instance of `IloResource`.

Class IlcCalendar::ShiftObjectIterator

Definition file: ilsched/sbecprop.h

An instance of this class traverses the list of shift objects that have been added to the corresponding `IlcCalendar`.

Constructor Summary	
`public`	`ShiftObjectIterator(IlcCalendar calendar)`

Method Summary	
`public IlcBool`	`ok() const`
`public IlcShiftObject`	`operator*() const`
`public ShiftObjectIterator &`	`operator++()`

Constructors

`public` **`ShiftObjectIterator`**`(IlcCalendar calendar)`

This constructor creates an iterator to traverse the list of shift objects that have been added to `calendar`.

Methods

`public IlcBool` **`ok`**`() const`

This member function returns `IlcTrue` if the current position of the iterator is a valid one. It returns `IlcFalse` if the list of shift objects has been completely scanned by the iterator.

`public IlcShiftObject` **`operator*`**`() const`

This operator returns the current instance of `IlcShiftObject`, the one to which the invoking iterator points. If the iterator is set past the end position, then this operator returns an empty handle.

`public ShiftObjectIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator to the next instance of `IlcShiftObject`.

Class IloCalendar::ShiftObjectIterator

Definition file: ilsched/ilocalendar.h

An instance of this class traverses the list of shift objects that have been added to the corresponding `IloCalendar`.

Constructor Summary	
`public`	`ShiftObjectIterator(IloCalendar calendar)`

Method Summary	
`public IlcBool`	`ok() const`
`public IloShiftObject`	`operator*() const`
`public ShiftObjectIterator &`	`operator++()`

Constructors

`public` **`ShiftObjectIterator`**`(IloCalendar calendar)`

This constructor creates an iterator to traverse the list of shift objects that have been added to `calendar`.

Methods

`public IlcBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if the list of shift objects has been completely scanned by the iterator.

`public IloShiftObject` **`operator*`**`() const`

This operator returns the current instance of `IloShiftObject`, the one to which the invoking iterator points. If the iterator is set past the end position, then this operator returns an empty handle.

`public ShiftObjectIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator to the next instance of `IloShiftObject`.

Enumeration IlcActivityIteratorFilter

Definition file: ilsched/basic.h
Include file: <ilsched/ilsched.h>

Given an activity `act` on a schedule with a schedule precedence graph, several useful sets of activities can be defined with respect to `act` (set of successors of `act`, set of predecessors of `act`, etc.).

The enumeration `IlcActivityIteratorFilter` can be used to create an activity iterator `IlcActivityIterator` that allows traversing such sets of activities. Each element of the enumeration specifies a particular set to traverse.

`IlcPredecessors` specifies that the iterator traverse the set of activities that are predecessors of a given activity.

`IlcSuccessors` specifies that the iterator traverse the set of activities that are successors of a given activity.

`IlcDirectPredecessors` specifies that the iterator traverse the set of activities that are direct predecessors of a given activity.

`IlcDirectSuccessors` specifies that the iterator traverse the set of activities that are direct successors of a given activity.

`IlcUnranked` specifies that the iterator traverse the set of activities that are unranked with respect to a given activity.

For more information, see Precedence Graph Constraints.

See Also: IlcActivityIterator

Fields:

```
IlcPredecessors = 6

IlcSuccessors = 7

IlcDirectPredecessors = 3

IlcDirectSuccessors = 4

IlcUnranked = 5
```

Enumeration IlcFailReason

Definition file: ilsched/basic.h
Include file: <ilsched/ilsched.h>

This enumeration is used to describe the reason of a failure, if it is known to the Scheduler Engine. See `IlcSchedulerTraceI::getFailReason`.

The values `IlcFailActivityNoStartOverlapVariable` and `IlcFailActivityNoEndOverlapVariable` mean that an activity was to overlap a break at its start (or end, respectively), but no start (or end, respectively) overlap variable was defined.

The values `IlcFailAltResConstraint` and `IlcFailAltResConstraintOpposite` mean that a resource constraint (or its opposite, respectively) was to be propagated, but this led to a failure.

The value `IlcFailAtConstraintOpposite` means that the opposite of an "at constraint" was to be propagated but this led to a failure.

The value `IlcFailBalanceConstraint` means that the balance constraint was propagated and it led to a failure. If the fail reason is associated with a resource constraint `rct`, it means that the balance constraint detected that the resource is over-consumed, or the reservoir overflows or underflows just before or after `rct`. In case this fail reason is not associated with any resource constraint, it means that the balance constraint has detected a global failure.

The values `IlcFailDisjunctive`, `IlcFailEdgeFinder`, `IlcFailBreakConstraint`, `IlcFailCapTimetable`, `IlcFailContinuousTimetable`, `IlcFailPrecedenceConstraint`, `IlcFailStateTimetable`, and `IlcFailTypeTimetable` mean that the corresponding constraint has detected a failure condition.

The values with the prefix `IlcFailPrecedenceGraph` mean that the named modification was to be made, but this led to a failure.

The values `IlcFailResourceConstraint` and `IlcFailResourceConstraintOpposite` mean that a resource constraint (or its opposite, respectively) was to be propagated, but this led to a failure.

The values `IlcFailResourceIntegralConstraint` and `IlcFailResourceFunctionalConstraint` mean that resource integral or resource functional constraint was to be propagated, but this led to a failure.

The value `IlcFailTimeBoundConstraint` means that a time-bound constraint was to be propagated, but this led to a failure.

See Also: IlcSchedulerTraceI

Fields:

`IlcFailReasonUnknown = 0`

`IlcFailDisjunctive`

`IlcFailEdgeFinder`

`IlcFailBreakConstraint`

`IlcFailCapTimetable`

`IlcFailStateTimetable`

`IlcFailTypeTimetable`

`IlcFailContinuousTimetable`

`IlcFailPrecedenceGraphPropagateClose`

IlcFailPrecedenceGraphSetSuccessor

IlcFailPrecedenceGraphSetNext

IlcFailPrecedenceGraphSetNotNext

IlcFailPrecedenceGraphRankFirst

IlcFailPrecedenceGraphRankNotFirst

IlcFailPrecedenceGraphRankLast

IlcFailPrecedenceGraphRankNotLast

IlcFailPrecedenceGraphSetToContribute

IlcFailPrecedenceGraphSetToNotContribute

IlcFailPrecedenceGraphSetToContributeEvt

IlcFailPrecedenceGraphSetToNotContributeEvt

IlcFailPrecedenceGraphSetStrictSuccessorEvt

IlcFailPrecedenceGraphSetSuccessorEvt

IlcFailPrecedenceGraphSetSimultaneousEvt

IlcFailPrecedenceConstraint

IlcFailAtConstraintOpposite

IlcFailTimeBoundConstraint

IlcFailResourceConstraint

IlcFailResourceConstraintOpposite

IlcFailAltResConstraint

IlcFailAltResConstraintOpposite

IlcFailActivityNoStartOverlapVariable

IlcFailActivityNoEndOverlapVariable

IlcFailActivityNoStartProdOverlapVariable

IlcFailActivityNoEndProdOverlapVariable

IlcFailResourceCalendarConstraint

IlcFailBalanceConstraint

IlcFailResourceIntegralConstraint

IlcFailResourceFunctionalConstraint

IlcFailTransitionExpr

IlcFailSchedLast

Enumeration IlcGranularFunctionRoundingMode

Definition file: ilsched/basic.h
Include file: <ilsched/ilsched.h>

This enumeration selects a rounding mode associated to an `IlcGranularFunction` object. This rounding mode is taken into account when using the granular function to create integral constraints.

For more information, see Functional and Integral Constraints on Resources.

See Also: IlcGranularFunction

Fields:

```
IlcGranularFunctionRoundInward = 0x0
```

```
IlcGranularFunctionRoundUpward = 0x1
```

```
IlcGranularFunctionRoundDownward = 0x2
```

```
IlcGranularFunctionRoundOutward = 0x3
```

Enumeration IlcPrecedenceConstraintType

Definition file: ilsched/basic.h
Include file: <ilsched/ilsched.h>

This enumeration describes the type of an instance of `IlcPrecedenceConstraint`.

`IlcStartsAfterStart` signifies that at least a given "delay" must elapse between the beginning of the "preceding" activity and the beginning of the "following" activity.

`IlcStartsAfterEnd` signifies that at least a given "delay" must elapse between the end of the "preceding" activity and the beginning of the "following" activity.

`IlcEndsAfterStart` signifies that at least a given "delay" must elapse between the beginning of the "preceding" activity and the end of the "following" activity.

`IlcEndsAfterEnd` signifies that at least a given "delay" must elapse between the end of the "preceding" activity and the end of the "following" activity.

`IlcStartsAtStart` signifies that exactly a given "delay" must elapse between the beginning of the "preceding" activity and the beginning of the "following" activity.

`IlcEndsAtStart` signifies that exactly a given "delay" must elapse between the beginning of the "preceding" activity and the end of the "following" activity.

`IlcStartsAtEnd` signifies that exactly a given "delay" must elapse between the end of the "preceding" activity and the beginning of the "following" activity.

`IlcEndsAtEnd` signifies that exactly a given "delay" must elapse between the end of the "preceding" activity and the end of the "following" activity.

See Also: IlcPrecedenceConstraint

Fields:

```
IlcStartsAfterStart = 0

IlcStartsAfterEnd = 1

IlcEndsAfterStart = 2

IlcEndsAfterEnd = 3

IlcStartsAtStart = 4

IlcStartsAtEnd = 5

IlcEndsAtStart = 6

IlcEndsAtEnd = 7
```

Enumeration IlcResourceConstraintIteratorFilter

Definition file: ilsched/basic.h
Include file: <ilsched/ilsched.h>

Given a resource constraint `rct` on a resource with a resource precedence graph, several useful sets of resource constraints can be defined with respect to `rct` (set of successors, sets of predecessors, etc.).

The enumeration `IlcResourceConstraintIteratorFilter` can be used to create a resource constraint iterator `IlcResourceConstraintIterator` that traverses such sets of resource constraints. Each element of the enumeration specifies a particular set to traverse.

For more information, see Precedence Graph Constraints.

`IlcDirectPredecessors` indicates that the iterator will traverse the set of resource constraints that are direct predecessors of a given resource constraint.

`IlcDirectSuccessors` indicates that the iterator will traverse the set of resource constraints that are direct successors of a given resource constraint.

`IlcUnranked` indicates that the iterator will traverse the set of resource constraints that are unranked with respect to a given resource constraint.

`IlcPredecessors` indicates that the iterator will traverse the set of resource constraints that are predecessors of a given resource constraint.

`IlcSuccessors` indicates that the iterator will traverse the set of resource constraints that are successors of a given resource constraint.

`IlcPossiblePrevious` indicates that the iterator will traverse the set of resource constraints that are either direct predecessors of or unranked with respect to a given resource constraint.

`IlcPossibleNext` indicates that the iterator will traverse the set of resource constraints that are either direct successors of or unranked with respect to a given resource constraint.

See Also: IlcResourceConstraintIterator

Fields:

```
IlcAllConstraints = 0

IlcActiveConstraints = 1

IlcPostedConstraints = 2

IlcDirectPredecessors = 3

IlcDirectSuccessors = 4

IlcUnranked = 5

IlcPredecessors = 6

IlcSuccessors = 7

IlcPossiblePrevious = 8

IlcPossibleNext = 9

IlcPredSucc = 10

IlcStrictDirectPredecessors = 11

IlcStrictDirectSuccessors = 12
```

```
IlcStrictPredecessors = 13

IlcStrictSuccessors = 14

IlcSimultaneous = 15
```

Enumeration RankFilter

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>

This enumeration allows specifying a subset of resource constraints to traverse with the iterators `IlcResource::ResourceConstraintIterator`, and `IlcResource::ResourceConstraintDeltaIterator`.

`RankedFirst` indicates the subset of resource constraints that have been ranked first on the resource.

`RankedLast` indicates the subset of resource constraints that have been ranked last on the resource.

`NotRanked` indicates the subset of resource constraints that have not yet been ranked first or last on the resource.

`PossibleFirst` indicates the subset of resource constraints that can possibly be ranked first on the resource. This is the set of resource constraints that has not yet been ranked first nor ranked last, nor otherwise determined to be unavailable to be ranked first.

`PossibleLast` indicates the subset of resource constraints that can possibly be ranked last on the resource. This is the set of resource constraints that has not yet been ranked first nor ranked last, nor otherwise determined to be unavailable to be ranked last.

See Also: IlcResource::ResourceConstraintIterator, IlcResource::ResourceConstraintDeltaIterator

Fields:

```
RankedFirst = 1

RankedLast = 2

Ranked = 3

NotRanked = 4

PossibleFirst = 8

PossibleLast = 16
```

Enumeration IlcSchedVariable

Definition file: ilsched/basic.h
Include file: <ilsched/ilsched.h>

This enumeration is used to designate the Solver variable constrained by the constraint `IlcResource::makeIntegralConstraint` or `IlcResource::makeFunctionalConstraint`. For more information, see Functional and Integral Constraints on Resources.

`IlcExternalVariable` specifies that the external variable of each activity linked to the resource via a resource constraint will be constrained. The external variable of an activity can be any Solver variable, associated with an activity via the `IlcActivity::setExternalVar` function.

`IlcProcessingTimeVariable` specifies that the processing time variable of each activity linked to the resource via a resource constraint will be constrained.

`IlcCapacityVariable` specifies that the capacity variable of each resource constraint on the resource will be constrained.

`IlcEnergyVariable` specifies that the energy variable of each resource constraint on the resource will be constrained. The energy of a resource constraint is defined as the product of the capacity variable and the duration of the activity.

`IlcDurationVariable` specifies that the duration variable of each activity linked to the resource via a resource constraint will be constrained.

`IlcStartVariable` specifies that the start variable of each activity linked to the resource via a resource constraint will be constrained.

`IlcEndVariable` specifies that the end variable of each activity linked to the resource via a resource constraint will be constrained.

See Also: IlcResource, IlcGranularFunction

Fields:

```
IlcExternalVariable = 0

IlcProcessingTimeVariable = 1

IlcCapacityVariable = 2

IlcEnergyVariable = 3

IlcDurationVariable = 4

IlcStartVariable = 8

IlcEndVariable = 12
```

Enumeration IlcSchedulerChange

Definition file: ilsched/schedtracei.h
Include file: <ilsched/ilsched.h>

This enumeration is used to describe which scheduler event has occurred. It indicates, for example, the modification of the start of an activity, the modification of a precedence constraint delay, an `IlcResourceConstraint` becomes next of another, and so forth. This enumeration is used in the `IlcSchedulerTraceFilter` functions.

Note that users should not rely on the order of the values listed in the synopsis: It is likely to change between successive releases of IBM® ILOG® Scheduler.

See Also: IlcSchedulerTraceFilter, IlcSchedulerTraceI

Fields:

```
IlcUndefinedSchedulerChange = 0
IlcActivityStart
IlcActivityEnd
IlcActivityProcessingTime
IlcActivityDuration
IlcActivityDurationOfBreaks
IlcActivityStartOverlap
IlcActivityEndOverlap
IlcActivityPostponed
IlcActivityPostponedBackward
IlcResourceConstraintCapacity
IlcResourceConstraintState
IlcResourceConstraintStateSet
IlcResourceConstraintNext
IlcResourceConstraintNextCost
IlcResourceConstraintPrev
IlcResourceConstraintPrevCost
IlcResourceSetup
IlcResourceSetupCost
IlcResourceTeardown
IlcResourceTeardownCost
IlcResourceConstraintNextExpr
IlcResourceConstraintPrevExpr
IlcResourceSetupExpr
IlcResourceTeardownExpr
```

IlcResourceConstraintContribution

IlcAltResConstraintIndex

IlcAltResConstraintCapacity

IlcAltResConstraintContribution

IlcTimeBoundConstraintDate

IlcPrecedenceConstraintDelay

IlcIntTimetableSetMin

IlcIntTimetableSetMax

IlcIntTimetableClose

IlcAnyTimetableMakeCompatible

IlcAnyTimetableSetState

IlcAnyTimetableSetPossibleStates

IlcAnyTimetableRemovePossibleStates

IlcAnyTimetableSetMustBeInUse

IlcAnyTimetableClose

IlcTypeTimetableMakeCompatible

IlcTypeTimetableSetType

IlcTypeTimetableRemovePossibleType

IlcTypeTimetableSetMustBeInUse

IlcActivitySetSuccessor

IlcResourceConstraintSetSuccessor

IlcResourceConstraintSetNext

IlcResourceConstraintSetNotNext

IlcResourceConstraintRankFirst

IlcResourceConstraintRankNotFirst

IlcResourceConstraintRankLast

IlcResourceConstraintRankNotLast

IlcResourceConstraintSetToContribute

IlcResourceConstraintSetToNotContribute

IlcResourceConstraintSetToContributeEvt

IlcResourceConstraintSetToNotContributeEvt

IlcResourceConstraintSetStrictSuccessorEvt

IlcResourceConstraintSetSuccessorEvt

IlcResourceConstraintSetSimultaneousEvt

IlcSchedulerChangeLast

Enumeration Type

Definition file: ilsched/shifts.h

The `Type` of `IlcShiftListObject` allows definition of the behavior during search regarding the variables of concerned activities. The possible types are:

- `OnStart`: Shifts only concern the start of the activity. For instance, if the shift is the interval *[a,b)*, then the start of the activity must be strictly smaller than *a* or greater than *b*.
- `OnEnd`: Shifts only concern the end of the activity. For instance, if the shift is the interval *[a,b)*, then the end of the activity must be strictly smaller than *a* or greater than *b*.
- `OnOverlap`: Shifts concern the whole activity. That is, the activity cannot overlap shifts. For instance, if the shift is the interval *[a,b)*, then the end of the activity must be strictly smaller than *a* or the start must be greater than *b*.

Fields:

```
OnStart = 0

OnEnd = 1

OnOverlap = 2
```

Enumeration IlcSlopeConstraintMode

Definition file: ilsched/basic.h
Include file: <ilsched/ilsched.h>

This enumeration provides values to define the rounding mode of the slope constraint as a parameter of the function `IlcResourceConstraint::setSlope`.

Let *c* be the capacity of the resource constraints, let *s* be its slope and let *d* be the duration of the activity.

When initializing a slope constraint, `IlcAtLeastCapacity` signifies that *d* is constrained to satisfy *c <= s.d < c+1*.

When initializing a slope constraint, `IlcAtMostCapacity` signifies that *d* is constrained to satisfy *c-1 < s.d <= c*.

When initializing a slope constraint, `IlcRoundedCapacity` signifies that *d* is constrained to satisfy *c-1 < s.d < c+1*.

See Also: IlcResourceConstraint

Fields:

```
IlcNoSlopeConstraint = 0
IlcAtLeastCapacity = 1
IlcAtMostCapacity = 2
IlcRoundedCapacity = 3
```

Enumeration IlcSolverChange

Definition file: ilsched/schedtracei.h
Include file: <ilsched/ilsched.h>

This enumeration is used to describe how a Solver variable that is used by a Scheduler object is changed: for example, the minimum value of an integer expression changes, an element is added to the required set of a set variable, and so forth. This enumeration is used with the Scheduler Engine trace mechanism.

See Also: IlcSchedulerTraceFilter, IlcSchedulerTraceI

Fields:

```
IlcUndefinedSolverChange = 0
IlcIntExpSetMin
IlcIntExpSetMax
IlcIntExpSetValue
IlcIntExpRemoveValue
IlcAnyExpSetValue
IlcAnyExpRemoveValue
IlcIntSetVarRemovePossible
IlcIntSetVarAddRequired
IlcIntSetVarSetValue
IlcSolverChangeLast
```

Enumeration IlcTimeBoundConstraintType

Definition file: ilsched/basic.h
Include file: <ilsched/ilsched.h>

This enumeration provides values that are assigned to a data member in the class `IlcTimeBoundConstraint` and its subclasses. That data member expresses time bounds on activities in a schedule.

`IlcStartsBefore` signifies that the latest start time of the activity equals the given time bound.

`IlcEndsBefore` signifies that the latest end time of the activity equals the given time bound.

`IlcStartsAt` signifies that the start time of the activity equals the given time bound.

`IlcEndsAt` signifies that the end time of the activity equals the given time bound.

`IlcStartsAfter` signifies that the earliest start time of the activity equals the given time bound.

`IlcEndsAfter` signifies that the earliest end time of the activity equals the given time bound.

See Also: IlcTimeBoundConstraint

Fields:

```
IlcStartsBefore = 0
IlcEndsBefore = 1
IlcStartsAt = 2
IlcEndsAt = 3
IlcStartsAfter = 4
IlcEndsAfter = 5
```

Enumeration IlcTimeExtent

Definition file: ilsched/basic.h
Include file: <ilsched/ilsched.h>

By default, it is assumed that an activity uses a resource throughout its execution. A non-breakable activity uses the resource from the start time of the activity to the end time of the activity. A breakable activity uses the resource during its processing time with specified breaks. However, it may be possible to specify a time range different from the default start to end range. The enumeration `IlcTimeExtent` is defined for this purpose. Note that, for continuous reservoirs, this notion of time extent is not defined.

`IlcNever` indicates that the activity requires (or provides) the resource at no time.

`IlcAlways` indicates that the activity requires (or provides) the resource at all times (that is, before its start time, from its start time to its end time, and after its end time). It is useful to optimize the maximal available capacity of a resource.

`IlcBeforeStart` indicates that the activity requires (or provides) the resource at all times before its start time.

`IlcAfterStart` indicates that the activity requires (or provides) the resource at all times after its start time (that is, both from its start time to its end time and after its end time). This time extent is useful when an activity consumes a reservoir, for example when part of a budget is spent for the performance of the activity.

`IlcAfterEnd` indicates that the activity requires (or provides) the resource at all times after its end time. This time extent is useful when an activity produces a reservoir, for example, when finished goods are produced in a factory.

`IlcBeforeEnd` indicates that the activity requires (or provides) the resource at all times before its end time (that is, both before its start time and from its start time to its end time). This time extent is useful when some activities require resources that have never been used before, for example when brand-new bank notes are used to test the prototype of an automatic teller machine (ATM).

`IlcBeforeStartAndAfterEnd` indicates that the activity requires (or provides) the resource at all times before its start time and after its end time.

`IlcFromStartToEnd` indicates that the activity requires (or provides) the resource from its start time to its end time. This is the default time extent.

See Also: IlcResourceConstraint

Fields:

```
IlcNever = 0

IlcAlways = 1

IlcBeforeStart = 2

IlcAfterStart = 3

IlcAfterEnd = 4

IlcBeforeEnd = 5

IlcBeforeStartAndAfterEnd = 6

IlcFromStartToEnd = 7
```

Enumeration IloActivitySelector

Definition file: ilsched/iloschedgoals.h
Include file: <ilsched/iloscheduler.h>

This enumeration is used to indicate how activities should be selected by the goals `IloSetTimesForward` and `IloSetTimesBackward`.

See Also: IloActivity

Fields:

`IloSelFirstActMinEndMax = 0`

`IloSelFirstActMinEndMin`

`IloSelLastActMaxStartMin`

`IloSelLastActMaxStartMax`

Enumeration IloEnforcementLevel

Definition file: ilsched/ilobasic.h
Include file: <ilsched/iloscheduler.h>

Several types of global constraints may be expressed on a given resource object. For example, a break list or transition times may be expressed on a given resource.

The enforcement level allows specifying with how much effort a given global constraint on a resource may be expressed.

All levels ensure that any solution found by the scheduler will satisfy the type of constraint associated with the level. Stated otherwise, the enforcement level allows selecting which algorithms are used to enforce the corresponding constraint, but even the lowest enforcement level will ensure that the constraint is satisfied.

The exact semantics of the levels depend on the scheduler.

For more information, see Resource Enforcement as Global Constraint Declaration, and Parameter ClassesParameters Organized by Function.

`IloLow` and `IloMediumLow` represent enforcement levels lower than the default level `IloBasic`. Stating that the enforcement level of a type of constraint is lower than `IloBasic` means that the scheduler will spend less effort at enforcing those constraints than it would do by default.

`IloBasic` is the default enforcement level.

`IloMediumHigh`, `IloHigh` and `IloExtended` correspond to a scale of enforcement levels higher than the default level `IloBasic`. Stating that the enforcement level of a type of constraint is higher than `IloBasic` means that the scheduler will spend more effort at enforcing those constraints than it would do by default.

See Also: IloResource, IloResourceParam

Fields:

```
IloNone = 0
IloLow = 10
IloMediumLow = 20
IloBasic = 30
IloMediumHigh = 40
IloHigh = 50
IloExtended = 60
```

Enumeration IloGranularFunctionRoundingMode

Definition file: ilsched/ilogfbase.h
Include file: <ilsched/iloscheduler.h>

This enumeration selects a rounding mode associated to an `IloGranularFunction` object. This rounding mode is taken into account when using the granular function to create integral constraints. Refer to Functional and Integral Constraints on Resources for a detailed description of each rounding mode.

See Also: IloGranularFunction

Fields:

```
IloGranularFunctionRoundInward = 0x00

IloGranularFunctionRoundUpward = 0x01

IloGranularFunctionRoundDownward = 0x02

IloGranularFunctionRoundOutward = 0x03
```

Enumeration IloPrecedenceConstraintType

Definition file: ilsched/ilobasic.h
Include file: <ilsched/iloscheduler.h>

This enumeration describes the type of an instance of `IloPrecedenceConstraint`.

`IloStartsAfterStart` signifies that at least a given " delay " must elapse between the beginning of the " preceding " activity and the beginning of the " following " activity.

`IloStartsAfterEnd` signifies that at least a given " delay " must elapse between the end of the " preceding " activity and the beginning of the " following " activity.

`IloEndsAfterStart` signifies that at least a given " delay " must elapse between the beginning of the " preceding " activity and the end of the " following " activity.

`IloEndsAfterEnd` signifies that at least a given " delay " must elapse between the end of the " preceding " activity and the end of the " following " activity.

`IloStartsAtStart` signifies that exactly a given " delay " must elapse between the beginning of the " preceding " activity and the beginning of the " following " activity.

`IloStartsAtEnd` signifies that exactly a given " delay " must elapse between the end of the " preceding " activity and the beginning of the " following " activity.

`IloEndsAtStart` signifies that exactly a given " delay " must elapse between the beginning of the " preceding " activity and the end of the " following " activity.

`IloEndsAtEnd` signifies that exactly a given " delay " must elapse between the end of the " preceding " activity and the end of the " following " activity.

See Also: IloPrecedenceConstraint

Fields:

```
IloStartsAfterStart = 0

IloStartsAfterEnd = 1

IloEndsAfterStart = 2

IloEndsAfterEnd = 3

IloStartsAtStart = 4

IloStartsAtEnd = 5

IloEndsAtStart = 6

IloEndsAtEnd = 7
```

Enumeration IloResourceConstraintSelector

Definition file: ilsched/iloschedgoals.h
Include file: <ilsched/iloscheduler.h>

This enumeration is used to indicate how resource constraints should be selected by the goals `IloRankForward` and `IloRankBackward`.

`IloSelFirstRCMinEndMax` selects the resource constraint that is possibly first with the minimal earliest start time, using minimal latest end time to break any ties.

`IloSelFirstRCMinStartMax` selects a resource constraint that is possibly first with the minimal earliest start time, using minimal latest start time to break any ties.

`IloSelLastRCMaxEndMin` selects a resource constraint that is possibly last with the maximal latest end time, using maximal earliest end time to break any ties.

`IloSelLastRCMaxStartMin` selects a resource constraint that is possibly last with the maximal latest end time, using maximal earliest start time to break any ties.

See Also: IloResource, IloResourceConstraint

Fields:

```
IloSelFirstRCMinEndMax = 0
```

```
IloSelFirstRCMinStartMax
```

```
IloSelLastRCMaxEndMin
```

```
IloSelLastRCMaxStartMin
```

Enumeration IloResourceSelector

Definition file: ilsched/iloschedgoals.h
Include file: <ilsched/iloscheduler.h>

This enumeration is used to indicate how resources should be selected by the goals `IloRankForward`, `IloRankBackward`, `IloSequenceForward`, `IloSequenceBackward`, and `IloAssignAlternative`.

Not all the values of the enumeration can be used for selecting any kind of resource. To select an `IloUnaryResource`, use one of `IloSelResMinGlobalSlack`, `IloSelResMinLocalSlack`, `IloSelResSequenceMinGlobalSlack`, or `IloSelResSequenceMinLocalSlack`. To select one resource of an alternative, use one of `IloSelResMinGlobalSlack`, `IloSelResMinLocalSlack`, `IloSelResMinCapacity`, or `IloSelAltRes`. To select a state resource, use `IloSelStateRes`. If a wrong selector is used in a goal, an `IloException` is raised.

`IloSelResMinGlobalSlack` selects the non-ranked unary resource with minimal global slack.

`IloSelResMinLocalSlack` selects the non-ranked unary resource with minimal local slack.

`IloSelResSequenceMinGlobalSlack` selects the non-sequenced unary resource with minimal global slack.

`IloSelResSequenceMinLocalSlack` selects the non-sequenced unary resource with minimal local slack.

`IloSelResMinCapacity` selects the resource with minimal capacity.

`IloSelStateRes` selects an unranked state resource.

`IloSelAltRes` selects a possible resource for an alternative resource constraint.

See Also: IloResource

Fields:

```
IloSelResMinGlobalSlack = 0
IloSelResMinLocalSlack
IloSelResSequenceMinGlobalSlack
IloSelResSequenceMinLocalSlack
IloSelResMinCapacity
IloSelStateRes
IloSelAltRes
```

Enumeration IloSchedVariable

Definition file: ilsched/ilogfbase.h
Include file: <ilsched/iloscheduler.h>

This enumeration is used to designate the Solver variable constrained by `IloResourceIntegralConstraint` and `IloResourceFunctionalConstraint`. See Functional and Integral Constraints on Resources for more information.

`IloExternalVariable` specifies that the external variable of each activity linked to the resource via a resource constraint will be constrained. The external variable of an activity can be any Solver variable, associated with an activity via the `IloActivity::setExternalVariable` function.

`IloProcessingTimeVariable` specifies that the processing time variable of each activity linked to the resource via a resource constraint will be constrained.

`IloCapacityVariable` specifies that the capacity variable of each resource constraint on the resource will be constrained.

`IloEnergyVariable` specifies that the energy of each resource constraint on the resource will be constrained. The energy of a resource constraint is defined as the product of the capacity variable by the duration of the activity.

`IloDurationVariable` specifies that the duration variable of each activity linked to the resource via a resource constraint will be constrained.

`IloStartVariable` specifies that the start variable of each activity linked to the resource via a resource constraint will be constrained.

`IloEndVariable` specifies that the end variable of each activity linked to the resource via a resource constraint will be constrained.

See Also: IloGranularFunction, IloResourceIntegralConstraint, IloResourceFunctionalConstraint

Fields:

```
IloExternalVariable = 0

IloProcessingTimeVariable = 1

IloCapacityVariable = 2

IloEnergyVariable = 3

IloDurationVariable = 4

IloStartVariable = 8

IloEndVariable = 12
```

Enumeration IloResourceConstraintIteratorFilter

Definition file: ilsched/ilosolution.h
Include file: <ilsched/ilosolution.h>

The enumeration `IloResourceConstraintIteratorFilter` can be used to create a `IloSchedulerSolution::ResourceConstraintIterator` that traverses a specifed set of resource constraints such as the predecessors or the successors. The possible values are described below.

`IloPredecessors` indicates that the iterator will traverse the set of resource constraints that are predecessors of a given resource constraint.

`IloSuccessors` indicates that the iterator will traverse the set of resource constraints that are direct successors of a given resource constraint.

`IloPrevious` indicates that the iterator will traverse the set of resource constraints that are previous of a given resource constraint.

`IloNext` indicates that the iterator will traverse the set of resource constraints that are next of a given resource constraint.

See Also: IloSchedulerSolution::ResourceConstraintIterator

Fields:

```
IloPredecessors = 0

IloSuccessors = 1

IloPrevious = 2

IloNext = 3
```

Enumeration IloSequenceIndexSelector

Definition file: ilsched/iloschedgoals.h
Include file: <ilsched/iloscheduler.h>

This enumeration is used to indicate which value to select for the next or prev variable of the resource constraints in the goals `IloSequenceForward` and `IloSequenceBackward`.

`IloSelNextRCMinCostEndMax` selects the resource constraint with the minimal earliest start time. If there is a tie, then selection is determined by minimal transition cost, and then by minimal latest end time.

`IloSelNextRCMinEndMaxCost` selects the resource constraint with the minimal earliest start time. If there is a tie, then selection is determined by minimal latest end time, and then by minimal transition cost.

`IloSelNextRCMinCostEndMin` selects the resource constraint with the minimal earliest start time. If there is a tie, then selection is determined by minimal transition cost, and then by minimal earliest end time.

`IloSelNextRCMinEndMinCost` selects the resource constraint with the minimal earliest start time. If there is a tie, then selection is determined by minimal earliest end time, and then by minimal transition cost.

`IloSelNextRCMinCostStartMax` selects the resource constraint with the minimal earliest start time. If there is a tie, then selection is determined by minimal transition cost, and then by minimal latest start time.

`IloSelNextRCMinStartMaxCost` selects the resource constraint with the minimal earliest start time. If there is a tie, then selection is determined by minimal latest start time, and then by minimal transition cost.

`IloSelPrevRCMaxCostStartMin` selects the resource constraint with the maximal latest end time. If there is a tie, then selection is determined by minimal transition cost, and then by maximal earliest start time.

`IloSelPrevRCMaxStartMinCost` selects the resource constraint with the maximal latest end time. If there is a tie, then selection is determined by maximal earliest start time, and then by minimal transition cost.

`IloSelPrevRCMaxCostStartMax` selects the resource constraint with the maximal latest end time. If there is a tie, then selection is determined by minimal transition cost, and then by maximal latest start time.

`IloSelPrevRCMaxStartMaxCost` selects the resource constraint with the maximal latest end time. If there is a tie, then selection is determined by maximal latest start time, and then by minimal transition cost.

Fields:

`IloSelNextRCMinCostEndMax = 0`

`IloSelNextRCMinEndMaxCost`

`IloSelNextRCMinCostEndMin`

`IloSelNextRCMinEndMinCost`

`IloSelNextRCMinCostStartMax`

`IloSelNextRCMinStartMaxCost`

`IloSelPrevRCMaxCostStartMin`

`IloSelPrevRCMaxStartMinCost`

`IloSelPrevRCMaxCostStartMax`

`IloSelPrevRCMaxStartMaxCost`

Enumeration Type

Definition file: ilsched/ilocalendar.h

The `Type` of `IloShiftListObject` allows you to define the behavior during the search regarding the variables of concerned activities. The possible types are:

- `OnStart`: Shifts only affect the start of the activity. For instance, if the shift is the interval *[a,b)*, then the start of the activity must be strictly smaller than *a* or greater than *b*.
- `OnEnd`: Shifts only affect the end of the activity. For instance, if the shift is the interval *[a,b)*, then the end of the activity must be strictly smaller than *a* or greater than *b*.
- `OnOverlap`: Shifts affect the whole activity. That is, the activity cannot overlap shifts. For instance, if the shift is the interval *[a,b)*, then the end of the activity must be strictly smaller than *a*, or the start must be greater than *b*.

Fields:

```
OnStart = 0

OnEnd = 1

OnOverlap = 2
```

Enumeration IloTimeBoundConstraintType

Definition file: ilsched/ilobasic.h
Include file: <ilsched/iloscheduler.h>

This enumeration provides values that are assigned to a data member in the class `IloTimeBoundConstraint` and its subclasses. That data member expresses time bounds on activities in a schedule.

`IloStartsBefore` signifies that the latest start time of the activity equals the given time bound.

`IloEndsBefore` signifies that the latest end time of the activity equals the given time bound.

`IloStartsAt` signifies that the start time of the activity equals the given time bound.

`IloEndsAt` signifies that the end time of the activity equals the given time bound.

`IloStartsAfter` signifies that the earliest start time of the activity equals the given time bound.

`IloEndsAfter` signifies that the earliest end time of the activity equals the given time bound.

See Also: IloTimeBoundConstraint

Fields:

```
IloStartsBefore = 0
IloEndsBefore = 1
IloStartsAt = 2
IloEndsAt = 3
IloStartsAfter = 4
IloEndsAfter = 5
```

Enumeration IloTimeExtent

Definition file: ilsched/ilobasic.h
Include file: <ilsched/iloscheduler.h>

By default, an activity uses a resource throughout its execution; that is, from the start time of the activity to the end time of the activity. However, it is possible to specify a time range different from the default " start to end " range. The enumeration `IloTimeExtent` is defined for this purpose.

`IloNever` indicates that the activity requires (or provides) the resource at no time.

`IloAlways` indicates that the activity requires (or provides) the resource at all times (that is, before its start time, from its start time to its end time, and after its end time). `IloAlways` is useful for optimizing the maximal available capacity of a resource.

`IloBeforeStart` indicates that the activity requires (or provides) the resource at all times before its start time.

`IloAfterStart` indicates that the activity requires (or provides) the resource at all times after its start time (including after its end time). This time extent is useful when an activity consumes a reservoir; for example, when part of a budget is spent for the performance of the activity.

`IloAfterEnd` indicates that the activity requires (or provides) the resource at all times after its end time. This time extent is useful when an activity produces a reservoir; for example, when finished goods are produced in a factory.

`IloBeforeEnd` indicates that the activity requires (or provides) the resource at all times before its end time (that is, from before its start time up to its end time). This time extent is useful when some activities require resources that have never been used before; for example, when brand-new bank notes are used to test the prototype of an automatic teller machine (ATM).

`IloBeforeStartAndAfterEnd` indicates that the activity requires (or provides) the resource at all times before its start time and after its end time.

`IloFromStartToEnd` indicates that the activity requires (or provides) the resource from its start time to its end time. This is the default time extent.

See Also: IloActivity, IloResourceConstraint

Fields:

```
IloNever = 0

IloAlways = 1

IloBeforeStart = 2

IloAfterStart = 3

IloAfterEnd = 4

IloBeforeEnd = 5

IloBeforeStartAndAfterEnd = 6

IloFromStartToEnd = 7
```

Global function IloSetTimesForward

```
public IloGoal IloSetTimesForward(const IloEnv env, IloActivitySelector
activitySelector=IloSelFirstActMinEndMax)
public IloGoal IloSetTimesForward(const IloEnv env, const IloNumVar criterion,
IloActivitySelector activitySelector=IloSelFirstActMinEndMax)
```

Definition file: ilsched/iloschedgoals.h
Include file: <ilsched/iloscheduler.h>

This function creates and returns a goal that assigns a start time to all activities in the model. If the argument `criterion` is given, then this variable will be bound, if possible, to its minimal value at the end of the search. By default, that is, if no activity selector is given as an argument, the activity selector `IloSelFirstActMinEndMax` selects the next activity.

Note

WARNING In order to ensure a purely anti-chronological scheduling, the supplied activity selector should always choose an unscheduled activity of maximal latest end time. Furthermore, scheduling an activity at time t should not have an impact on the latest end times of later activities.

In particular, one should be careful in using precedence constraints with a negative delay (and similar Solver constraints on start and end variables).

For further details about the interpretation of the `IloSetTimesForward` goal in the Scheduler Engine and about these restrictions, see `IlcSetTimes`.

See Also: IlcSetTimes, IloActivitySelector

Global function IlcActivityStartVarBoundPredicate

```
public IloPredicate< IlcActivity > IlcActivityStartVarBoundPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an activity predicate whose `operator(const IlcActivity& activity)` returns `IlcTrue` if and only if the processing time variable of the activity is bound.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceConstraintSurelyContributesPredicate

```
public IloPredicate< IlcResourceConstraint >
IlcResourceConstraintSurelyContributesPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint predicate whose `operator(const IlcResourceConstraint& rc)` returns `IlcTrue` if and only if `rc` surely affects the availability of the resource. Otherwise, it returns `IlcFalse`. This member function returns `IlcFalse` if the resource constraint `rc` is an empty handle; that is, if it corresponds to a virtual resource constraint (source or sink node). This predicate is implemented using `IlcResourceConstraint::possiblyContributes`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcAltResConstraintNbPossibleEvaluator

```
public IloEvaluator< IlcAltResConstraint >
IlcAltResConstraintNbPossibleEvaluator(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an `IloEvaluator::IlcAltResConstraint` whose `operator(const IlcAltResConstraint& altrc)` method returns the number of resources that are possible for `altrc`. This method uses the method `IlcAltResConstraint::getNumberOfPossible`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcActivityRandomEvaluator

```
public IloEvaluator< IlcActivity > IlcActivityRandomEvaluator(IlcManager,
IloRandom)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an activity evaluator whose `operator(const IlcResource& resource)` returns a random number drawn with uniform probability from the interval `[0..1)`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceConstraintCapacityMinEvaluator

```
public IloEvaluator< IlcResourceConstraint >
IlcResourceConstraintCapacityMinEvaluator(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint evaluator whose `operator(const IlcResourceConstraint& ct)` returns the minimum value of the capacity variable of `ct`. If `ct` is not a variable resource constraint, the function returns the actual capacity required or produced by the resource constraint. Undefined behavior occurs if `ct` is not a capacity resource constraint. The minimum capacity of the virtual source and sink nodes is defined to be 0.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceConstraintNextTransitionCostEvaluator

```
public IloEvaluator< IlcResourceConstraint >
IlcResourceConstraintNextTransitionCostEvaluator(IlcManager,
IlcTransitionCostObject)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint evaluator whose `operator(const IlcResourceConstraint& ct, IlcAny context1)` returns the transition cost between `ct` and a "comparison" resource constraint given in the context.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceConstraintCapacityMaxEvaluator

```
public IloEvaluator< IlcResourceConstraint >
IlcResourceConstraintCapacityMaxEvaluator(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint evaluator whose `operator(const IlcResourceConstraint& ct)` returns the maximum value of the capacity variable of `ct`. If `ct` is not a variable resource constraint, the function returns the actual capacity required or produced by the resource constraint. Undefined behavior occurs if `ct` is not a capacity resource constraint. The maximum capacity of the virtual source and sink nodes is defined to be 0.

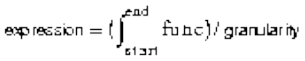
This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcActivityIntegralExp

```
public IlcIntExp IlcActivityIntegralExp(const IlcActivity act, const
IlcGranularFunction func)
```

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h>

This function creates and returns an expression computed as the integral, over the duration of the activity `act`, of the function `func` divided by the granularity, and properly rounded (see `IlcGranularFunction`).



Global function IlcResourceConstraintProvidingConstraintPredicate

```
public IloPredicate< IlcResourceConstraint >
IlcResourceConstraintProvidingConstraintPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint predicate whose `operator(const IlcResourceConstraint& rc)` returns `IlcTrue` if and only if `rc` was constructed by the member function `IlcActivity::provides` or was extracted from an `IloResourceConstraint` that was constructed by the member function `IloActivity::provides` or `IloActivity::produces`. This member function returns `IlcFalse` if the resource constraint `rc` is an empty handle; that is, if it corresponds to a virtual resource constraint (source or sink node). This predicate is implemented using `IlcResourceConstraint::possiblyContributes`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcActivityStartMaxEvaluator

`public IloEvaluator< IlcActivity > **IlcActivityStartMaxEvaluator**(IlcManager)`

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an activity evaluator whose `operator(const IlcActivity& activity)` returns the maximum value of the start time variable of the activity.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IloTextureSuccessorGoal

```
public IloGoal IloTextureSuccessorGoal(const IloEnv env)
public IloGoal IloTextureSuccessorGoal(const IloEnv env, const IloNumVar criterion)
```

Definition file: ilsched/iloschedgoals.h
Include file: <ilsched/iloscheduler.h>

This function creates and returns a goal that adds successor relations between all pairs of resource constraints on all resources in the model that have a texture measurement. If the argument `criterion` is given, then this variable will be bound, if possible, to its minimal value at the end of the search.

Each time the goal is executed, the resource and time point with highest criticality is identified and the set of resource constraints, `S`, that contributed to that criticality are examined. Two resource constraints are identified:

1. The resource constraint, `A`, with the highest contribution to the critical point that is not a successor or predecessor of all other elements of `S`.
2. The resource constraint, `B`, with the highest contribution to the critical point, that is not `A`, nor a successor or predecessor of `A`.

A choice point is created using the `IlcResourceConstraint::setSuccessor` function. The direction of the successor relation (that is, `A.setSuccessor(B)` or `B.setSuccessor(A)`) that is explored first is the one that preserves the most local slack between `A` and `B`. The reverse successor relation is posted on backtracking.

Global function IlcAltResConstraintVariableConstraintPredicate

```
public IloPredicate< IlcAltResConstraint >
IlcAltResConstraintVariableConstraintPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an alternative resource constraint predicate whose `operator(const IlcAltResConstraint& altrc)` returns `IlcTrue` if and only the activity associated with `altrc` has a variable representing the required or provided capacity. This predicate is implemented using `IlcAltResConstraint::isVariableResourceConstraint`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceIsCapacityResourcePredicate

```
public IloPredicate< IlcResource >
IlcResourceIsCapacityResourcePredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource predicate whose `operator(const IlcResource& resource)` returns `IlcTrue` if and only if the resource is an `IlcCapResource`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceConstraintSlopeEvaluator

```
public IloEvaluator< IlcResourceConstraint >
IlcResourceConstraintSlopeEvaluator(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint evaluator whose `operator(const IlcResourceConstraint& ct)` returns the value of the slope of `ct`. Undefined behavior occurs if `ct` has no slope or if its resource is not a continuous reservoir. The slope of the virtual source and sink nodes is defined to be 0.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceIsUnaryResourcePredicate

```
public IloPredicate< IlcResource > IlcResourceIsUnaryResourcePredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource predicate whose `operator(const IlcResource& resource)` returns `IlcTrue` if and only if the resource is an `IlcUnaryResource`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcActivityIsRankedPredicate

`public IloPredicate< IlcActivity > IlcActivityIsRankedPredicate(IlcManager)`

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an activity predicate whose `operator(const IlcActivity& activity)` returns `IlcTrue` if and only if all the other activities in the schedule are constrained to execute either before or after the activity.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcAssign

```
public IlcGoal IlcAssign(IlcAltResConstraint alternatives, IloSelector<
IlcResource, IlcAltResConstraint > possibleSel=0)
```

Definition file: ilsched/srchalt.h
Include file: <ilsched/search.h>

This function returns a goal which assigns a resource as the selected one in `alternatives`. The goal uses the selector `possibleSel` to choose the resource. If no instance of `IloSelector<IlcResource,IlcAltResConstraint>` is given, the goal uses a default selector which tries the resources in the same order as `IlcPossibleAltResIterator`.

See `IloSelector` in the *IBM ILOG Solver Reference Manual* for more information.

Implementation

This function could be defined like this:

```
IlcGoal IlcAssign(IlcAltResConstraint constraint,
                  IloSelector<IlcResource> possibleSel){
   if (!constraint.isResourceSelected()) {
     IlcResource resource;
     if (possibleSel.select(resource, constraint))
       return IlcAnd(IlcTryAssign(resource, constraint),
                     IlcAssign(constraint, possibleSel));
   }
   return 0;
}
```

See Also: IlcAltResConstraint, IlcResource, IlcTryAssign

Global function IlcScheduleOrPostpone

```
public IlcGoal IlcScheduleOrPostpone(IlcActivity activity)
```

Definition file: ilsched/srchgoal.h
Include file: <ilsched/search.h>

This function sets a choice point and then assigns to `activity` its earliest start time. In case of failure, `activity` is postponed by its member function `IlcActivity::postpone`.

See Also: IlcActivity, IlcActivity::postpone, IlcSetTimes

Global function IlcResourceConstraintPossibleLastPredicate

```
public IloPredicate< IlcResourceConstraint >
IlcResourceConstraintPossibleLastPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint predicate whose `operator(const IlcResourceConstraint& rc)` returns `IlcTrue` if and only if `rc` can be ranked last among the non-ranked resource constraints. In particular, it returns `IlcFalse` if `rc` is already ranked or if `rc` represents a virtual source or sink node (empty handle). This predicate is implemented using `IlcResourceConstraint::isPossibleLast`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IloTimeWindowBackwardChronologicalComparator

```
public IloComparator< IloTimeWindowNHoodI::IloTimeWindow >
IloTimeWindowBackwardChronologicalComparator(IloEnv env)
```

Definition file: ilsched/ilolnsgoals.h
Include file: <ilsched/iloscheduler.h>

This function returns a predefined time interval comparator. The comparator compares two instances of the class `IloTimeWindowNHoodI::IloTimeWindow`, `tw1` and `tw2`. The time interval `tw1` is less than time interval `tw2` if the end of `tw1` is greater than the end of `tw2`. If the ends are equal, it compares the starts of both intervals. The comparator is allocated on the memory allocation stack of `env`.

See `IloComparator` in the *IBM ILOG Solver Reference Manual* for more information.

See Also: IloSchedulerLargeNHood

Global function IloTimeWindowBackwardChronologicalComparator

```
public IloComparator< IloTimeWindowNHoodI::IloTimeWindow >
IloTimeWindowBackwardChronologicalComparator(IloSolver solver)
```

Definition file: ilsched/ilolnsgoals.h
Include file: <ilsched/iloscheduler.h>

This function returns a predefined time interval comparator. The comparator compares two instances of the class `IloTimeWindowNHoodI::IloTimeWindow`, `tw1` and `tw2`. The time interval `tw1` is less than time interval `tw2` if the end of `tw1` is greater than the end of `tw2`. If the ends are equal, it compares the starts of both intervals. The comparator is allocated on the memory allocation stack of `solver`.

See `IloComparator` in the *IBM ILOG Solver Reference Manual* for more information.

See Also: IloSchedulerLargeNHood

Global function IlcRank

```
public IlcGoal IlcRank(IlcResource resource, IloSelector< IlcResourceConstraint,
IlcResource > rcSel=0)
public IlcGoal IlcRank(IlcSchedule schedule, IloSelector< IlcResource, IlcSchedule
> rSel=0, IloSelector< IlcResourceConstraint, IlcResource > rcSel=0)
public IlcGoal IlcRank(IlcSchedule schedule, IlcIntVar criterion, IloSelector<
IlcResource, IlcSchedule > rSel=0, IloSelector< IlcResourceConstraint, IlcResource
> rcSel=0)
```

Definition file: ilsched/srchgoal.h
Include file: <ilsched/ilsched.h>

This function creates and returns a goal that ranks all resource constraints on a set of resources in chronological order.

- If its first argument is an instance of `IlcResource`, it considers all the resource constraints on that instance of `IlcResource`.
- If its first argument is an instance of `IlcSchedule`, it considers all the resource constraints on instances of `IlcResource` in that schedule. The resource selector `rSel` selects the next resource. By default, that is, when no resource selector object is given, a resource selector `defaultResSel` defined as follows is used:

```
IloSelector<IlcResource,IlcSchedule> defaultResSel = IlcResourceInScheduleSelector(s);
sel.setPredicate(IlcResourceIsUnaryResourcePredicate(s) &&
                 !IlcResourceRankedPredicate(s));
sel.setComparator(IlcResourceGlobalSlackEvaluator(s));
```

- If the argument `criterion` is given, then this variable will be bound, if possible, to its *minimal* value at the end of the search.

The resource constraint selector `rcSel` selects the next resource constraint to be ranked first on a given resource given as first context to the selection function. By default, that is, when no `IloSelector<IlcResourceConstraint,IlcResource>` is given, a resource constraint selector `defaultRCSel` defined as follows is used:

```
IloSelector<IlcResourceConstraint,IlcResource> defaultRCSel = IlcResourceConstraintInScheduleSelector(s);
IlcTranslator<IlcActivity, IlcResourceConstraint> ac = IlcActivityResourceConstraintTranslator(s);
sel.setPredicate(IlcResourceConstraintPossibleFirstPredicate(s));
sel.setComparator(IlcLexicalComposition(IlcCompareMin(IlcActivityStartMinEvaluator(s)<<ac),
                                        IlcCompareMin(IlcActivityStartMaxEvaluator(s)<<ac)));
```

Note

WARNING This function assumes that all resources that are selected have been closed, that is, that no unknown resource constraints have yet to be posted. If you cannot close all resources, you should use a resource selector that selects only closed resources. In such a case, you should also handle the ranking of the resources that are not closed yourself.

Ranking is only well defined on unary and state resources. Therefore, your resource selector should contain an instance of `IlcResourceIsUnaryResourcePredicate` or `IlcResourceIsStateResourcePredicate` (or both, joined by a logical-OR).

Implementation

Here's how we could define the goal returned by the first version of the function.

```
ILCGOAL2(IlcRankResource, IlcResource, resource,
           IloSelector<IlcResourceConstraint,IlcResource>, rcSel) {
  IlcResourceConstraint constraint;
  if (rcSel.select(constraint, resource))
```

```
    return IlcAnd(IlcTryRankFirst(constraint), this);
  return 0;
}
```

Here's how we could define the goal returned by the second version of the function:

```
ILCGOAL3(IlcRankSchedule, IlcSchedule, schedule,
         IloSelector<IlcResource,IlcSchedule>, rSel,
         IloSelector<IlcResourceConstraint,IlcResource>, rcSel) {
  IlcResource resource;
  if (rSel.select(resource, schedule))
    return IlcAnd(IlcRank(resource, rcSel), this);
  return 0;
}
```

See `IloSelector` in the *IBM ILOG Solver Reference Manual* for more information.

See Also: IlcSchedule, IlcResource, IlcTryRankFirst

Global function IlcResourceConstraintSetupPredicate

```
public IloPredicate< IlcResourceConstraint >
IlcResourceConstraintSetupPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint predicate whose `operator(const IlcResourceConstraint& rc)` returns `IlcTrue` if and only if `rc` is the setup resource constraint. This member function returns `IlcFalse` if `rc` represents a virtual source or sink node (empty handle). This predicate is implemented using `IlcResourceConstraint::isSetup`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceConstraintTeardownPredicate

```
public IloPredicate< IlcResourceConstraint >
IlcResourceConstraintTeardownPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint predicate whose `operator(const IlcResourceConstraint& rc)` returns `IlcTrue` if and only if `rc` is the teardown resource constraint. This member function returns `IlcFalse` if `rc` represents a virtual source or sink node (empty handle). This predicate is implemented using `IlcResourceConstraint::isTeardown`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IloUnionNHood

```
public IloSchedulerLargeNHood IloUnionNHood(IloEnv env, IloSchedulerLargeNHood
nhood1, IloSchedulerLargeNHood nhood2, const char * name=0)
```

Definition file: ilsched/ilolnsgoals.h
Include file: <ilsched/iloscheduler.h>

This function creates a *composed large neighborhood*. The neighborhood formed is the union of the neighborhood of `nhood1` and `nhood2`.

The size of the neighborhood is the product of the sizes of `nhood1` and `nhood2`.

The set of selected extractables of the union for index *i* is the union of the sets of selected extractables for index *i1* for neighborhood *n1* and for index *i2* for neighborhood *n2* such that *i = i1*size(n2) + i2*.

For any extractable, it will be restored if either `nhood1` or `nhood2` specify that it must be restored.

Global function IlcActivityResourceConstraintTranslator

```
public IloTranslator< IlcActivity, IlcResourceConstraint >
IlcActivityResourceConstraintTranslator(IloEnv)
public IloTranslator< IlcActivity, IlcResourceConstraint >
IlcActivityResourceConstraintTranslator(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a translator which implements a function (`IloTranslator::operator`) that, when given a resource constraint, returns the activity of this resource constraint. It is useful for transforming activity predicates and evaluators into the corresponding resource constraints predicates and evaluators.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcActivityAltResConstraintTranslator

```
public IloTranslator< IlcActivity, IlcAltResConstraint >
IlcActivityAltResConstraintTranslator(IloEnv)
public IloTranslator< IlcActivity, IlcAltResConstraint >
IlcActivityAltResConstraintTranslator(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a translator which implements a function (`IloTranslator::operator`) that, when given an alternative resource constraint, returns the activity of this alternative resource constraint. It is useful for transforming activity predicates/selectors into the corresponding alternative resource constraints predicates/selectors.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceConstraintPossibleSetupPredicate

```
public IloPredicate< IlcResourceConstraint >
IlcResourceConstraintPossibleSetupPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint predicate whose `operator(const IlcResourceConstraint& rc)` returns `IlcTrue` if and only if `rc` can be a setup resource constraint. This member function returns `IlcFalse` if `rc` represents a virtual source or sink node (empty handle). This predicate is implemented using `IlcResourceConstraint::isPossibleSetup`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceConstraintPossibleFirstPredicate

```
public IloPredicate< IlcResourceConstraint >
IlcResourceConstraintPossibleFirstPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint predicate whose `operator(const IlcResourceConstraint& rc)` returns `IlcTrue` if and only if `rc` can be ranked first among the non-ranked resource constraints. In particular, it returns `IlcFalse` if `rc` is already ranked or if `rc` represents a virtual source or sink node (empty handle). This predicate is implemented using `IlcResourceConstraint::isPossibleFirst`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcActivityIsBreakablePredicate

`public IloPredicate< IlcActivity > **IlcActivityIsBreakablePredicate**(IlcManager)`

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an activity predicate whose `operator(const IlcActivity& activity)` returns `IlcTrue` if and only if the activity is a breakable activity.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcRCTextureProbabilisticFactory

```
public IlcRCTextureFactory IlcRCTextureProbabilisticFactory(IloSolver solver)
```

Definition file: ilsched/texture.h
Include file: <ilsched/ilsched.h>

This function returns an `IlcRCTextureFactory` instance whose `IlcRCTextureFactory::createRCTexture` method returns an instance of `IlcRCTextureProbabilisticI`.

Global function IlcAssignAlternative

```
public IlcGoal IlcAssignAlternative(IlcAltResSet resources, IloSelector<
IlcResource, IlcAltResConstraint > possibleSel=0, IloSelector< IlcAltResConstraint,
IlcAltResSet > constraintSel=0)
public IlcGoal IlcAssignAlternative(IlcSchedule schedule, IloSelector< IlcResource,
IlcAltResConstraint > possibleSel=0, IloSelector< IlcAltResConstraint, IlcAltResSet
> constraintSel=0)
public IlcGoal IlcAssignAlternative(IlcResource resource, IloSelector<
IlcAltResConstraint, IlcAltResSet > constraintSel=0)
```

Definition file: ilsched/srchalt.h
Include file: <ilsched/search.h>

This function returns a goal that assigns a possible resource as the selected one for a set of constraints.

- If its first argument is an instance of `IlcAltResSet`, it considers all the constraints on that instance of `IlcAltResSet`.
- If its first argument is an instance of `IlcSchedule`, it considers all the constraints on instances of `IlcAltResSet` in that schedule.
- If its first argument is an instance of `IlcResource`, it considers all the constraints on instances of `IlcAltResSet` for which that resource is a possible alternative.

The goal uses the resource selector `possibleSel` to choose the resource and the alternative resource constraint selector `constraintSel` to choose a constraint. If no argument of type `IloSelector<IlcResource,IlcAltResConstraint>` is given, the goal uses a default selector, which tries the resources in the same order as an instance of `IlcPossibleAltResIterator`. If no argument of type `IloSelector<IlcAltResConstraint,IlcAltResSet>` is given, the goal uses a default selector, which tries the posted or metaposted alternative resources constraints in the same order as an instance of `IlcAltResConstraintIterator`.

Implementation

This function can be defined like this:

```
IlcGoal IlcAssignAlternative
        (IlcAltResSet resources,
         IloSelector<IlcResource,IlcAltResConstraint> possibleSel,
         IloSelector<IlcAltResConstraint,IlcAltResSet> constraintSel) {
   IlcAltResConstraint alternative;
   if (constraintSel.select(alternative, resources))
     return IlcAnd(IlcAssign(alternative, possibleSel),
                   IlcAssignAlternative(resources,
                                        possibleSel,
                                        constraintSel));
   return 0;
}
```

See `IloSelector` in the *IBM ILOG Solver Reference Manual* for more information.

See Also: IlcAltResConstraint, IlcAltResSet, IlcAssign, IlcResource, IlcSchedule

Global function IlcResourceConstraintVariableConstraintPredicate

```
public IloPredicate< IlcResourceConstraint >
IlcResourceConstraintVariableConstraintPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint predicate whose `operator(const IlcResourceConstraint& rc)` returns `IlcTrue` if and only if `rc` has a variable representing the required or provided quantity of the required state. This member function returns `IlcFalse` if `rc` represents a virtual source or sink node (empty handle). This predicate is implemented using `IlcResourceConstraint::isCapacityConstraint`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcActivityProcessingTimeMaxEvaluator

```
public IloEvaluator< IlcActivity >
IlcActivityProcessingTimeMaxEvaluator(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an activity evaluator whose `operator(const IlcActivity& activity)` returns the maximum value of the processing time variable of the activity.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceConstraintStateSetConstraintPredicate

```
public IloPredicate< IlcResourceConstraint >
IlcResourceConstraintStateSetConstraintPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint predicate whose `operator(const IlcResourceConstraint& rc)` returns `IlcTrue` if and only if `rc` indicates that one of a set of states of a state resource is required by the activity of the resource constraint. This member function returns `IlcFalse` if `rc` represents a virtual source or sink node (empty handle). This predicate is implemented using `IlcResourceConstraint::isStateConstraint`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcTryAssign

```
public IlcGoal IlcTryAssign(IlcResource resource, IlcAltResConstraint alternatives)
```

Definition file: ilsched/srchalt.h
Include file: <ilsched/search.h>

This function sets a choice point and then assigns the `resource` to be the selected one in `alternatives`. In case of failure, `resource` is set to be not possible for `alternatives`. This function returns the choice point as a goal.

See Also: IlcAltResConstraint, IlcResource

Global function IlcScheduleOrPostponeBackward

```
public IlcGoal IlcScheduleOrPostponeBackward(IlcActivity activity)
```

Definition file: ilsched/srchgoal.h
Include file: <ilsched/search.h>

This function sets a choice point and then assigns `activity` its latest end time. In case of failure, `activity` is postponed backward using the member function `IlcActivity::postponeBackward`.

Note

An activity that is postponed backward will actually be performed earlier than it would have been performed if it the assignment of its latest end time had not failed.

See Also: IlcActivity, IlcSetTimesBackward

Global function IloIntersectNHood

```
public IloSchedulerLargeNHood IloIntersectNHood(IloEnv env, IloSchedulerLargeNHood
nhood1, IloSchedulerLargeNHood nhood2, const char * name=0)
```

Definition file: ilsched/ilolnsgoals.h
Include file: <ilsched/iloscheduler.h>

This function creates a *composed large neighborhood*. The neighborhood formed is the intersection of the neighborhood of `nhood1` and `nhood2`.

The size of the neighborhood is the product of sizes of `nhood1` and `nhood2`.

The set of selected extractables of the intersection for index *i* is the intersection of the sets of selected extractables for index *i1* for neighborhood *nhood1* and for index *i2* for neighborhood *nhood2* such that *i = i1*size(n2) + i2*.

For any extractable, it will be restored if either `nhood1` or `nhood2` specify that it must be restored.

See Large Neighborhoods and the *Selectors* concept in the *IBM ILOG Solver Reference Manual* for more information.

Global function IlcResourceTextureEvaluator

```
public IloEvaluator< IlcResource > IlcResourceTextureEvaluator(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource evaluator whose `operator(const IlcResource& resource)` returns the maximum value of criticality of the texture measurement curve on the resource object to which it is applied. The behavior is undefined if this evaluator is applied to a resource that is not an `IlcDiscreteResource` or if no texture measurement has been created on the resource.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceIsReservoirPredicate

```
public IloPredicate< IlcResource > IlcResourceIsReservoirPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource predicate whose `operator(const IlcResource& resource)` returns `IlcTrue` if and only if the resource is an `IlcReservoir`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceConstraintNegativeConstraintPredicate

```
public IloPredicate< IlcResourceConstraint >
IlcResourceConstraintNegativeConstraintPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint predicate whose `operator(const IlcResourceConstraint& rc)` returns `IlcTrue` if and only if `rc` was constructed by the member function `IlcActivity::requiresNot` or was extracted from an `IloResourceConstraint` that was constructed by the member function `IloActivity::requiresNot`. This member function returns `IlcFalse` if the resource constraint `rc` is an empty handle; that is, if it corresponds to a virtual resource constraint (source or sink node). This predicate is implemented using `IlcResourceConstraint::possiblyContributes`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcRCTextureTargetFactory

```
public IlcRCTextureFactory IlcRCTextureTargetFactory(IloSolver solver)
```

Definition file: ilsched/texture.h
Include file: <ilsched/ilsched.h>

This function returns an `IlcRCTextureFactory` instance whose `IlcRCTextureFactory::createRCTexture` method returns an instance of `IlcRCTextureTargetI`.

Global function IlcAltResConstraintCapacityEvaluator

```
public IloEvaluator< IlcAltResConstraint >
IlcAltResConstraintCapacityEvaluator(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an `IloEvaluator::IlcAltResConstraint` whose `operator(const IlcAltResConstraint& altrc)` method returns the quantity required or provided by the alternative resource constraint `altrc`. This method uses the method `IlcAltResConstraint::getCapacity`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcRelativeDemandCriticalityCalculator

```
public IlcTextureCriticalityCalculator
IlcRelativeDemandCriticalityCalculator(IloSolver solver)
```

Definition file: ilsched/texture.h
Include file: <ilsched/ilsched.h>

This function returns an `IlcTextureCriticalityCalculator` instance that calculates criticality with an instance of the `IlcRelativeDemandCriticalityCalculatorI` implementation class. The new instance is allocated on the solver heap.

Global function IlcProbabilisticCriticalityCalculator

```
public IlcTextureCriticalityCalculator
IlcProbabilisticCriticalityCalculator(IloSolver solver)
```

Definition file: ilsched/texture.h
Include file: <ilsched/ilsched.h>

This function returns an `IlcTextureCriticalityCalculator` instance that calculates criticality with an instance of the `IlcProbabilisticCriticalityCalculatorI` implementation class. The new instance is allocated on the solver heap.

Global function IloResourceIntegralConstraint

```
public IloConstraint IloResourceIntegralConstraint(const IloResource resource,
IloSchedVariable leftVariable, const IloGranularFunction func, IloBool
ignoredSuspensionAtStartEnd=IloTrue)
```

Definition file: ilsched/ilogfbase.h
Include file: <ilsched/iloscheduler.h>

This function creates an integral constraint from the function `func` on all the activities requiring the resource `res`. If the time extent is `IloNever` or `IloAlways`, the resource constraint will be ignored.

For each such activity, the integral of the function `func` is computed over the activity duration, divided by the granularity, and properly rounded (see `IloGranularFunction`). It is then set to be equal to the value of the variable designated by `leftVar`:

$$\text{leftVar} = \left(\int_{\text{start}}^{\text{end}} \text{func}\right) / \text{granularity}$$

Such a function can be used to specify detailed, time-varying constraints for all activities on a resource. For example, such a function can be used to specify precisely how the duration of an activity depends on its start and end times. Whenever the processing time is used (`IloProcessingTimeVariable`), every activity executing on the resource must be breakable, and the granular function `func` must take a value less than or equal to its granularity. Otherwise an error will be raised when starting to solve the problem.

The suspension of activities at the start or end (see `IloActivity::canBeSuspendedAtStart` and `IloActivity::canBeSuspendedAtEnd`) is by default not taken into account. To take forbidden suspensions into account, the argument `ignoreSuspensionAtStartEnd` may be set to `IlcFalse`. Then, the resulting integral constraint will accordingly prevent activities to start/end in intervals where the granular function `func` has zero values.

For more information, see Functional and Integral Constraints on Resources.

See Also: IloGranularFunction, IloSchedVariable

Global function IlcResourceConstraintHasNextPredicate

```
public IloPredicate< IlcResourceConstraint >
IlcResourceConstraintHasNextPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint predicate whose `operator(const IlcResourceConstraint& rc)` returns `IlcTrue` if and only if the immediately following activity of the activity of `rc` is known. In case `rc` represents a virtual node, it is assumed that it represents the virtual source node, and the evaluation of the predicate returns `IlcTrue` if and only if the setup node (that is, the node that directly follows the virtual source node), is known. This predicate is implemented using `IlcResourceConstraint::hasNextRC`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceConstraintStateConstraintPredicate

```
public IloPredicate< IlcResourceConstraint >
IlcResourceConstraintStateConstraintPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint predicate whose `operator(const IlcResourceConstraint& rc)` returns `IlcTrue` if and only if `rc` indicates that a single state of a state resource is required by the activity of the resource constraint. This member function returns `IlcFalse` if `rc` represents a virtual source or sink node (empty handle). This predicate is implemented using `IlcResourceConstraint::isStateConstraint`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceConstraintPossiblePrevVisitor

```
public IloVisitor< IlcResourceConstraint, IlcResourceConstraint >
IlcResourceConstraintPossiblePrevVisitor(IlcManager)
public IloVisitor< IlcResourceConstraint, IlcResourceConstraint >
IlcResourceConstraintPossiblePrevVisitor(IloEnv)
```

Definition file: ilsched/srchseq.h
Include file: <ilsched/ilsched.h>

This function returns a visitor that allows to traverse the set of resource constraints that are possibly previous to the resource constraint given as container. In case the resource constraint given as container is a sequence virtual node (see `IlcResourceConstraint::isVirtualNode`), the visitor traverses the set of resource constraints that are possible teardown resource constraints on the resource.

See the section about Visitors in the concept Selectors of the *IBM ILOG Solver Reference Manual*

Global function IloResourceFunctionalConstraint

```
public IloConstraint IloResourceFunctionalConstraint(const IloResource resource,
IloSchedVariable leftVariable, const IloGranularFunction func, IloSchedVariable
rightVariable=IloDurationVariable)
```

Definition file: ilsched/ilogfbase.h
Include file: <ilsched/iloscheduler.h>

This function creates a functional constraint from the function `func` on all the activities requiring the resource `res`. If the time extent is `IloNever` or `IloAlways`, the resource constraint will be ignored.

For each such activity, the function `func` is evaluated at the value of the variable designated by `rightVar`, and set to be equal to the value of variable designated by `leftVar`:

```
leftVar = func(rightVar)
```

By default, if no `rightVar` is given, `IloDurationVariable` is used. Such a function can be used to specify detailed, time-varying constraints for all activities on a resource. For example, such a function can be used to specify precisely how the duration of an activity depends on its start time. Whenever the processing time is used (`IloProcessingTimeVariable`), every activity executing on the resource must be breakable, and the granular function `func` must take a value less than or equal to its granularity. Otherwise an error will be raised when starting to solve the problem.

Global function IlcResourceConstraintPrevTransitionCostEvaluator

```
public IloEvaluator< IlcResourceConstraint >
IlcResourceConstraintPrevTransitionCostEvaluator(IlcManager,
IlcTransitionCostObject)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint evaluator whose `operator(const IlcResourceConstraint& ct, IlcAny context1)` returns the transition cost between a "comparison" resource constraint given in the context and `ct`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceConstraintSlopeConstraintPredicate

```
public IloPredicate< IlcResourceConstraint >
IlcResourceConstraintSlopeConstraintPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint predicate whose `operator(const IlcResourceConstraint& rc)` returns `IlcTrue` if and only if `rc` has a slope constraint. This member function returns `IlcFalse` if `rc` represents a virtual source or sink node (empty handle). This predicate is implemented using `IlcResourceConstraint::hasSlope`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcTestSequencedResource

```
public void IlcTestSequencedResource(const IlcUnaryResource resource)
```

Definition file: ilsched/srchseq.h
Include file: <ilsched/search.h>

This function checks whether `resource` is sequenced. That is, that its setup and teardown activities are known and that each activity has either a next and a previous activity or is not visited. If an activity does not follow these rules, it is set to “not visited”.

The resource must be closed and have its sequence constraint created.

This function is intended to be used in the search goals `IlcSequence` and `IlcSequenceBackward`, which are used for sequencing resources, to check if the sequence found is a valid path.

For more information, see Sequence Constraint.

See Also: IlcSequenceBackward, IlcUnaryResource, IlcSequence

Global function IlcResourceRandomEvaluator

```
public IloEvaluator< IlcResource > IlcResourceRandomEvaluator(IlcManager,
IloRandom)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource evaluator whose `operator(const IlcResource& resource)` returns a random number drawn with uniform probability from the interval `[0..1)`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcRankBackward

```
public IlcGoal IlcRankBackward(IlcResource resource, IloSelector<
IlcResourceConstraint, IlcResource > rcSel=0)
public IlcGoal IlcRankBackward(IlcSchedule schedule, IloSelector< IlcResource,
IlcSchedule > rSel=0, IloSelector< IlcResourceConstraint, IlcResource > rcSel=0)
public IlcGoal IlcRankBackward(IlcSchedule schedule, IlcIntVar criterion,
IloSelector< IlcResource, IlcSchedule > rSel=0, IloSelector< IlcResourceConstraint,
IlcResource > rcSel=0)
```

Definition file: ilsched/srchgoal.h
Include file: <ilsched/ilsched.h>

This function creates and returns a goal that ranks all resource constraints on a set of resources in anti-chronological order.

- If its first argument is an instance of `IlcResource`, it considers all the resource constraints on that instance of `IlcResource`.
- If its first argument is an instance of `IlcSchedule`, it considers all the resource constraints on instances of `IlcResource` in that schedule. The resource selector `rSel` selects the next resource. By default, that is, when no resource selector object is given, a resource selector `defaultResSel` defined as follows is used:

```
IloSelector<IlcResource,IlcSchedule> defaultResSel = IlcResourceInScheduleSelector(s);
sel.setPredicate(IlcResourceIsUnaryResourcePredicate(s) &&
                 !IlcResourceRankedPredicate(s));
sel.setComparator(IlcResourceGlobalSlackEvaluator(s));
```

- If the argument `criterion` is given, then this variable will be bound, if possible, to its *maximal* value at the end of the search.

The resource constraint selector `rcSel` selects the next resource constraint to be ranked last on a given resource given as first context to the selection function. By default, that is, when no `IloSelector<IlcResourceConstraint>` is given, a resource constraint selector `defaultRCSel` defined as follows is used:

```
IloSelector<IlcResourceConstraint,IlcResource> defaultRCSel = IlcResourceConstraintInScheduleSelector(s);
IlcTranslator<IlcActivity, IlcResourceConstraint> ac = IlcActivityResourceConstraintTranslator(s);
sel.setPredicate(IlcResourceConstraintPossibleLastPredicate(s));
sel.setComparator(IlcLexicalComposition(IlcCompareMax(IlcActivityEndMaxEvaluator(s)<<ac),
                                        IlcCompareMax(IlcActivityEndMinEvaluator(s)<<ac)));
```

Note

WARNING This function assumes that all resources that are selected have been closed, that is, that no unknown resource constraints have yet to be posted. If you cannot close all resources, you should use a resource selector that selects only closed resources. In such a case, you should also handle the ranking of the resources that are not closed yourself.

Ranking is only well defined on unary and state resources. Therefore, your resource selector should contain an instance of `IlcResourceIsUnaryResourcePredicate` or `IlcResourceIsStateResourcePredicate` (or both, joined by a logical-OR).

Implementation

Here's how we could define the goal returned by the first version of the function.

```
ILCGOAL2(IlcRankBackwardResource, IlcResource, resource,
        IloSelector<IlcResourceConstraint,IlcResource>, rcSel) {
  IlcResourceConstraint constraint;
  if (rcSel.select(constraint, resource))
```

```
    return IlcAnd(IlcTryRankLast(constraint), this);
  return 0;
}
```

Here's how we could define the goal returned by the second version of the function:

```
ILCGOAL3(IlcRankBackwardSchedule, IlcSchedule, schedule,
         IloSelector<IlcResource,IlcSchedule>, rSel,
         IloSelector<IlcResourceConstraint,IlcResource>, rcSel) {
  IlcResource resource;
  if (rSel.select(resource, schedule))
    return IlcAnd(IlcRankBackward(resource, rcSel), this);
  return 0;
}
```

See `IloSelector` in the *IBM ILOG Solver Reference Manual* for more information.

See Also: IlcSchedule, IlcResource, IlcTryRankLast

Global function IloSequenceForward

```
public IloGoal IloSequenceForward(const IloEnv env, const IloUnaryResource uRes)
public IloGoal IloSequenceForward(const IloEnv env, const IloNumVar criterion,
IloResourceSelector rSel=IloSelResMinGlobalSlack)
public IloGoal IloSequenceForward(const IloEnv env, IloResourceSelector
rSel=IloSelResMinGlobalSlack)
public IloGoal IloSequenceForward(const IloEnv env, const IloNumVar criterion,
const IloTransitionParam param, IloSequenceIndexSelector
iSel=IloSelNextRCMinCostEndMax, IloResourceSelector rSel=IloSelResMinGlobalSlack)
public IloGoal IloSequenceForward(const IloEnv env, const IloUnaryResource uRes,
const IloTransitionParam param, IloSequenceIndexSelector
iSel=IloSelNextRCMinCostEndMax)
public IloGoal IloSequenceForward(const IloEnv env, const IloTransitionParam param,
IloSequenceIndexSelector iSel=IloSelNextRCMinCostEndMax, IloResourceSelector
rSel=IloSelResMinGlobalSlack)
```

Definition file: ilsched/iloschedgoals.h
Include file: <ilsched/iloscheduler.h>

These functions return instances of `IloGoal` that sequence instances of `IloUnaryResource` from their setup activity to their teardown activity (sequencing forward). The involved resources must be instances of `IloUnaryResource`, and they must not be kept open.

When the argument `uRes` is given, the function returns an instance of `IloGoal` that sequences `uRes`. When there is no resource argument, all the unary resources of the model that are not to be kept open are successively sequenced according to the unary resource selector argument `rSel`. By default, the resource selector is `IloSelResSequenceMinGlobalSlack`. Unary resources that are to be kept open during the search will not be sequenced.

The selector `iSel`, a value from the enumeration `IloSequenceIndexSelector`, selects a possible value for the next resource constraint of each resource constraint. The default selector is `IloSelNextRCMinCostEndMax`.

When the argument `criterion` is used, the goal also assigns it to its smallest consistent value after the sequencing has been completed.

When an `IloSequenceIndexSelector` is used, the estimation of the cost will be determined by the `IloTransitionParam` parameter passed to the constructor. The default value for such an index selector is `IloSelNextRCMinCost`.

See Also: IloSequenceBackward, IloUnaryResource, IloTransitionParam, IloSequenceIndexSelector

Global function IlcTextureSuccessorGoal

```
public IlcGoal IlcTextureSuccessorGoal(IlcSchedule s, IloSelector< IlcResource,
IlcSchedule > rSel=0)
public IlcGoal IlcTextureSuccessorGoal(IlcSchedule s, IlcIntVar criterion,
IloSelector< IlcResource, IlcSchedule > rSel=0)
public IlcGoal IlcTextureSuccessorGoal(IlcUnaryResource res)
```

Definition file: ilsched/txtgoal.h
Include file: <ilsched/ilsched.h>

These functions create and return goals that completely sequence the resource constraints on a unary resource(s). The sequencing is achieved by setting the successor relations on the precedence graph associated with each resource. The sequence between the two activities is the one that results in the greatest remaining pair-wise slack.

The goal created by this function assumes that the precedence graph exists on the resources.

Functions Taking an IlcSchedule

The resource selector indicated by `rSel` selects the next resource. By default, that is, when no resource selector is given, the resources are selected in descending order of texture criticality. The pair of activities to sequence is chosen in descending order of their contribution to the critical time point on the selected resource.

Note that the texture measurements are recalculated during every execution of the goal, and therefore the decision-making focus may opportunistically move from one resource to another without the former resource being completely sequenced.

For more information, see Texture Measurements.

See Also: IlcTextureAltSuccessorGoal, IlcResourceTexture, IlcRCTexture

Global function IlcActivityTransitionTypeEvaluator

```
public IloEvaluator< IlcActivity > IlcActivityTransitionTypeEvaluator(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an activity evaluator whose `operator(const IlcActivity& activity)` returns the transition type of the activity.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcActivityPostponedBackwardPredicate

```
public IloPredicate< IlcActivity >
IlcActivityPostponedBackwardPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an activity predicate whose `operator(const IlcActivity& activity)` returns `IlcTrue` if and only if the activity is postponed backward.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

See Also: IlcSetTimesBackward

Global function IlcResourceGlobalSlackEvaluator

```
public IloEvaluator< IlcResource > IlcResourceGlobalSlackEvaluator(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource evaluator whose `operator(const IlcResource& resource)` returns the global slack of the resource object to which it is applied. The behavior is undefined if this evaluator is applied to a resource that is not an `IlcDiscreteResource`. See `IlcDiscreteResource::getGlobalSlack`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcSetTimesBackward

```
public IlcGoal IlcSetTimesBackward(IlcSchedule schedule, IloSelector< IlcActivity,
IlcSchedule > aSel=0)
public IlcGoal IlcSetTimesBackward(IlcSchedule schedule, IlcIntVar criterion,
IloSelector< IlcActivity, IlcSchedule > aSel=0)
```

Definition file: ilsched/srchgoal.h
Include file: <ilsched/search.h>

This function creates and returns a goal that assigns an end time to all activities managed by `schedule`. If the argument `criterion` is given, then the assignments are made to maximize `criterion`. The activity selector `aSel` selects the next activity. By default, that is, if no activity selector is given as an argument, the activity selector used is defined as follows:

```
IloSelector<IlcActivity,IlcSchedule> aSel = IlcActivityInScheduleSelector(s);
sel.setPredicate(!IlcActivityEndVarBoundPredicate(s) &&
                 !IlcActivityPostponedBackwardPredicate(s));
sel.setComparator(IlcLexicalComposition(IlcCompareMax(IlcActivityEndMaxEvaluator(s)),
                                        IlcCompareMax(IlcActivityStartMinEvaluator(s))));
```

The function is designed to efficiently schedule activities in an anti-chronological order. It considers only solutions that can be produced as follows: in each step, choose an unscheduled activity *A* of maximal latest end time and schedule it as late as possible, as allowed by the previously scheduled activities (which have greater end times than *A*).

Internally, the function uses the schedule-or-postpone-backward method that works as follows.

1. A selected activity is assigned to its latest end time.
2. If that end time leads to a failure, the activity is postponed backward until its latest end time has been removed. This removal can occur as a result of a combination of decisions and propagation.

Before assigning the latest end time to an activity, with this candidate end time *et*, it is determined whether a backward postponed activity *actP* exists that can be or should be scheduled after *et*, that is, whether *actP.getStartMax() >= et* or *actP.getEndMin() >= et*. If this is the case, a fail is generated based on the reasoning that if we have “normal” precedence and resource constraints, the latest end time of *actP* will never be removed and thus *actP* will remain postponed backward and no solution will be found in this branch of the search tree.

As implied by the above description, `IlcSetTimesBackward` can be thought of as a “mirror image” of `IlcSetTimes`: rather than scheduling chronologically according to earliest start times and postponement, it schedules anti-chronologically according to latest end times and “backward” postponement. Consequently, there are also mirror image situations where `IlcSetTimesBackward` performs an incomplete search. To examine this, let's adapt the first example presented in `IlcSetTimes`. Assume we have an activity `B` that can start a maximal 50 units after the start of an activity `A` which can be expressed by a precedence constraint with negative delay.

```
A.startsAfterStart(B, -50)
```

Given a scheduling horizon of 1000, suppose that *A* cannot be scheduled at a later time than 900 since it requires a resource that has a maximal capacity of 0 in the interval [900,1000). Then *B* cannot be scheduled after time 950 even if there are no later activities. `IlcSetTimesBackward` will not find a solution here, because it will attempt to assign *B* to an end time of 1000. When this fails, it will postpone-backward *B* but then realize that there are no other activities that can be schedule after *B* and therefore it concludes that there are no solutions. Note that in this very simple case, it is likely that constraint propagation will discover that 950 is the actual latest end time of *B* and so `IlcSetTimesBackward` will successfully find a solution. However, if this situation is part of a larger, more complex constraint interaction, it cannot be guaranteed that constraint propagation will discover the globally consistent latest end time. In such a situation, then, a solution will be missed.

Similarly, the code presented in `IlcSetTimes` can be easily adapted to produce an example where `IlcSetTimesBackward` concludes that no solutions exist, even when there is one.

In general, `IlcSetTimesBackward` can miss solutions if there are precedence constraints with negative delay, if the processing time of an activity depends on its start or end time, or if reservoirs are used.

Note

WARNING In order to ensure a purely chronological scheduling, the supplied activity selector should always choose an unscheduled activity of minimal earliest start time. Furthermore, scheduling an activity at time t should not have an impact on the earliest start times of activities that have been scheduled earlier.

In particular, care should be taken in using precedence constraints with a negative delay, activities where the processing time depends on the start or end time of the activity, and reservoirs.

See `IloSelector` in the *IBM ILOG Solver Reference Manual* for more information.

See Also: IlcSchedule, IlcSetTimes, IlcScheduleOrPostponeBackward

Global function IlcActivityDurationMinEvaluator

```
public IloEvaluator< IlcActivity > IlcActivityDurationMinEvaluator(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an activity evaluator whose `operator(const IlcActivity& activity)` returns the minimum value of the duration variable of the activity.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceLocalSlackEvaluator

`public IloEvaluator< IlcResource > **IlcResourceLocalSlackEvaluator**(IlcManager)`

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource evaluator whose `operator(const IlcResource& resource)` returns the local slack of the resource object to which it is applied. The behavior is undefined if this evaluator is applied to a resource that is not an `IlcDiscreteResource`. See `IlcDiscreteResource::getGlobalSlack`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcSequenceBackward

```
public IlcGoal IlcSequenceBackward(IlcUnaryResource resource, IloSelector<
IlcResourceConstraint, IlcResourceConstraint > prevSelect=0)
public IlcGoal IlcSequenceBackward(IlcSchedule schedule, IloSelector<
IlcResourceConstraint, IlcResourceConstraint > prevSelect)
public IlcGoal IlcSequenceBackward(IlcSchedule schedule, IloSelector< IlcResource,
IlcSchedule > resSel=0, IloSelector< IlcResourceConstraint, IlcResourceConstraint >
prevSelect=0)
public IlcGoal IlcSequenceBackward(IlcSchedule schedule, IlcIntVar criterion,
IloSelector< IlcResourceConstraint, IlcResourceConstraint > prevSelect)
public IlcGoal IlcSequenceBackward(IlcSchedule schedule, IlcIntVar criterion,
IloSelector< IlcResource, IlcSchedule > resSel=0, IloSelector<
IlcResourceConstraint, IlcResourceConstraint > prevSelect=0)
```

Definition file: ilsched/srchseq.h
Include file: <ilsched/search.h>

These functions return instances of `IlcGoal` that sequence instances of `IlcUnaryResource` from their teardown activity to their setup activity (sequencing in reverse). The involved resources must be instances of `IlcUnaryResource`, closed, and with their precedence graph constraints posted.

When the argument `resource` is given, the function returns an instance of `IlcGoal` that sequences `resource`.

When the argument `schedule` is given, all the closed unary resources of `schedule` with a posted precedence graph constraint are successively sequenced in reverse according to the resource selector argument `resSel`. By default, the resource selector is defined as follows:

```
IloSelector<IlcResource,IlcSchedule> resSel(!IlcResourceSequencedPredicate(solver),
                                IlcResourceGlobalSlackEvaluator(solver));
```

Note

WARNING `IlcSequenceBackward` can only be applied to `IlcUnaryResource` instances. When defining your own selectors make sure that the predicate `IlcResourceSequencedPredicate` (solver) or the predicate `IlcResourceIsUnaryResourcePredicate` (solver) is used.

The resource constraint selector `prevSelect` selects a resource constraint that is a possibly previous to the lastly sequenced backward resource constraint. The instance of `IlcResourceConstraint` in the context of this selector represents the lastly sequenced backward resource constraint. At the begining of the search when no resource constraint has been sequenced backward, the search goal passes the sequence virtual node of the resource as context. See `IlcUnaryResource::getVirtualNodeRC`. When it is not specified, the previous selector object used by default selects the possibly previous resource constraint with the biggest maximal end time, using the biggest minimal start time to break ties.

If the argument `criterion` is given, then this variable will be bound, if possible, to its minimal value at the end of the search.

For more information, see Sequence Constraint.

See Also: IlcSequence, IlcTestSequencedResource, IlcUnaryResource

Global function IlcResourceIsDiscreteResourcePredicate

```
public IloPredicate< IlcResource >
IlcResourceIsDiscreteResourcePredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource predicate whose `operator(const IlcResource& resource)` returns `IlcTrue` if and only if the resource is an `IlcDiscreteResource`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcActivityEndMaxEvaluator

`public IloEvaluator< IlcActivity > IlcActivityEndMaxEvaluator(IlcManager)`

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an activity evaluator whose `operator(const IlcActivity& activity)` returns the maximum value of the end time variable of the activity.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceIsContinuousReservoirPredicate

```
public IloPredicate< IlcResource >
IlcResourceIsContinuousReservoirPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource predicate whose `operator(const IlcResource& resource)` returns `IlcTrue` if and only if the resource is an `IlcContinuousReservoir`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcActivityStartMinEvaluator

`public IloEvaluator< IlcActivity > IlcActivityStartMinEvaluator(IlcManager)`

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an activity evaluator whose `operator(const IlcActivity& activity)` returns the minimum value of the start time variable of the activity.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcMakeTransitionCost

```
public IlcTransitionCostObject IlcMakeTransitionCost(IlcTransitionTable table,
IlcIntArray setups, IlcIntArray teardowns)
public IlcTransitionCostObject IlcMakeTransitionCost(IlcTransitionTable table)
public IlcTransitionCostObject IlcMakeTransitionCost(IlcTransitionTable table,
IlcIntArray setups)
```

Definition file: ilsched/trancost.h
Include file: <ilsched/ilsched.h>

This function creates an instance of `IlcTransitionCostObject`. It uses an instance of `IlcTransitionTable` for the transition cost and `IlcIntArray` for the setup and teardown costs. The table and arrays must be of the same size and filled with positive integers.

The transition, setup, and teardown costs of an instance of `IlcTransitionCostObject` are calculated using the transition type of the activities of the resource constraint. Notice that this implies that the transition type of the activities must be non-negative and strictly smaller than the size of the table.

See Also: IlcActivity::getTransitionType, IlcMakeTransitionTime, IlcTransitionTable, IlcTransitionCostObject

Global function IloRankForward

```
public IloGoal IloRankForward(const IloEnv env, const IloUnaryResource res,
IloResourceConstraintSelector rcSel=IloSelFirstRCMinStartMax)
public IloGoal IloRankForward(const IloEnv env, IloResourceSelector
rSel=IloSelResMinGlobalSlack, IloResourceConstraintSelector
rcSel=IloSelFirstRCMinStartMax)
public IloGoal IloRankForward(const IloEnv env, const IloNumVar criterion,
IloResourceSelector rSel=IloSelResMinGlobalSlack, IloResourceConstraintSelector
rcSel=IloSelFirstRCMinStartMax)
public IloGoal IloRankForward(const IloEnv env, const IloStateResource res,
IloResourceConstraintSelector rcSel=IloSelFirstRCMinStartMax)
```

Definition file: ilsched/iloschedgoals.h
Include file: <ilsched/iloscheduler.h>

Resource Constraints on a Resource

These functions (1st and 4th) create and return a goal that ranks all resource constraints of the unary or state resource `res`. The resource constraint selector indicated by `rcSel` selects the next resource constraint to be ranked first. By default (when no resource constraint selector is given as an argument), the resource constraint selector `IloSelFirstRCMinStartMax` selects the next resource constraint to be ranked first.

Resource Constraints in a Model

This function (2nd) creates and returns a goal that ranks all resource constraints of all unary or state resources in the model. The resource selector indicated by `rSel` selects the next resource. The resource constraint selector `rcSel` selects the next resource constraint to be ranked first.

This function will rank either all the state resources (if `rSel == IloSelStateRes`), or all the unary resources (if `rSel == IloSelResMinGlobalSlack`, or `IloSelResMinLocalSlack`). By default (when no selectors are given as arguments), the resource selector `IloSelResMinGlobalSlack` selects the next resource, and the resource constraint selector `IloSelFirstRCMinStartMax` selects the next resource constraint to be ranked first.

Resource Constraints in a Model with a Criterion

This function (3rd) creates and returns a goal that is a logical AND of the goal described in the previous section with a goal that attempts to instantiate `criterion`. The instantiation goal tries values for `criterion` in ascending order.

> **Note**
>
> **WARNING** This function assumes that all unary or state resources that are selected are not kept open, that is, that there are no unknown resource constraints yet to be added. If you cannot close all unary or state resources, you should use a resource selector that selects only closed resources. In such a case, you should also handle the ranking of the resources that are not closed yourself.

Implementation

For an example of how the goals returned by these functions could be implemented, see IlcRank. Also see Ranking .

See Also: IloStateResource, IloUnaryResource, IlcSchedule, IlcRank, IloResourceConstraintSelector

Global function operator<<

```
public ostream & operator<<(ostream & stream, const IlcActivity & activity)
```

Definition file: ilsched/schedule.h
Include file: <ilsched/ilsched.h> <ilsched/timetabh.h>

```
ostream& operator<< (ostream& stream,
                     const IlcActivity& activity);
ostream& operator<< (ostream& stream,
                     const IlcAltResSet& resource);
ostream& operator<<(ostream& stream,
                    const IlcAnyTimetable& table);
ostream& operator<< (ostream& stream,
                     const IlcIntervalList& bl);
ostream& operator<< (ostream& stream,
                    const IlcIntTimetable& table);
ostream& operator<< (ostream& stream,
                     const IlcResource& resource);
ostream& operator<< (ostream& stream,
                     const IlcSchedule& schedule);
```

This operator directs its output to an output stream (normally, standard output).

The operator uses the virtual member function `display` of an implementation class. For instance, the member function `IlcScheduleI::display` defines how the invoking instance of `IlcSchedule` is printed on the given output stream. To display a schedule, you simply write:

```
solver.out() << schedule;
```

The class `IloSolver` is documented in the *IBM ILOG Solver Reference Manual*.

Implementation

The operator is defined like this:

```
ostream& operator<<(ostream& stream, const IlcSchedule
schedule) {
  schedule.getImpl()->display(stream);
  return stream;
}
```

The definition files for this operator include ilsched/altresh.h, ilsched/breaks.h, and ilsched/schedule.h.

See Also: IlcActivity, IlcAltResSet, IlcIntervalList, IlcResource, IlcSchedule

Global function IloShapeLowerThan

```
public IloConstraint IloShapeLowerThan(IloResourceConstraint resCt1,
IloResourceConstraint resCt2)
```

Definition file: ilsched/iloresconstraint.h

This function returns a constraint acting on each resource constraint passed as an argument. An error will be raised if one the resource constraints does not have a shape specified. The constraint returned will state that the contribution of the first resource constraint's shape must be less than or equal to the contribution of the second shape. This condition must hold at each time point.

See Also: IloResourceConstraint, IloShape

Global function IloRelocateActivityNHood

```
public IloSchedulerLargeNHood IloRelocateActivityNHood(IloEnv env, IloComparator<
IloActivity > comparator=0, IloPredicate< IloActivity > activity=0, const char *
name=0)
```

Definition file: ilsched/ilolnsgoals.h
Include file: <ilsched/iloscheduler.h>

This funtion creates an activity neighborhood.

The optional parameter `comparator` is used to specify in which order the activities should be considered. When applied the comparator receives as argument the neighborhood.

The optional parameter `predicate` is used to specify which activities to consider. The size of this neighborhood is the number of activities in the current solution for which this predicate returns `IloTrue`. If the predicate is an empty handle, the size of this neighborhood is the number of activities in the current solution.

For more information, see Large Neighborhoods. In the *IBM ILOG Solver Reference Manual*, see the *Selectors* concept and `IloComparator` and `IloPredicate`.

See Also: IloSchedulerLargeNHood, IloRelocateActivityNHoodI

Global function IlcFunctionalExp

```
public IlcIntExp IlcFunctionalExp(const IlcGranularFunction func, const IlcIntVar
x)
```

Definition file: ilsched/gfbase.h
Include file: <ilsched/ilsched.h>

This function creates and returns an integer expression constrained to be the value of the function `func` at the value of `x`. The granularity of the function must be equal to 1, otherwise, an error will be raised.

Global function IlcActivityEndVarBoundPredicate

```
public IloPredicate< IlcActivity > IlcActivityEndVarBoundPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an activity predicate whose `operator(const IlcActivity& activity)` returns `IlcTrue` if and only if the end variable of the activity is bound.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IloRankBackward

```
public IloGoal IloRankBackward(const IloEnv env, const IloUnaryResource res,
IloResourceConstraintSelector rcSel=IloSelLastRCMaxEndMin)
public IloGoal IloRankBackward(const IloEnv env, IloResourceSelector
rSel=IloSelResMinGlobalSlack, IloResourceConstraintSelector
rcSel=IloSelLastRCMaxEndMin)
public IloGoal IloRankBackward(const IloEnv env, const IloNumVar criterion,
IloResourceSelector rSel=IloSelResMinGlobalSlack, IloResourceConstraintSelector
rcSel=IloSelLastRCMaxEndMin)
public IloGoal IloRankBackward(const IloEnv env, const IloStateResource res,
IloResourceConstraintSelector rcSel=IloSelLastRCMaxEndMin)
```

Definition file: ilsched/iloschedgoals.h
Include file: <ilsched/iloscheduler.h>

Resource Constraints on a Resource

These functions (1st and 4th) create and return a goal that ranks all resource constraints of the unary or state resource `res`. The resource constraint selector indicated by `rcSel` selects the next resource constraint to be ranked last. By default (when no resource constraint selector is given as an argument), the resource constraint selector `IloSelLastRCMaxEndMin` selects the next resource constraint to be ranked last.

Resource Constraints in a Model

This function (2nd) creates and returns a goal that ranks all resource constraints of all unary or state resources in the model. The resource selector indicated by `rSel` selects the next resource. The resource constraint selector `rcSel` selects the next resource constraint to be ranked last.

This function will rank either all the state resources (if `rSel == IloSelStateRes`), or all the unary resources (if `rSel == IloSelResMinGlobalSlack`, or `IloSelResMinLocalSlack`). By default (when no selectors are given as arguments), the resource selector `IloSelResMinGlobalSlack` selects the next resource, and the resource constraint selector `IloSelLastRCMaxEndMin` selects the next resource constraint to be ranked last.

Resource Constraints in a Model with a Criterion

This function (3rd) creates and returns a goal that is a logical AND of the goal described in the previous section with a goal that attempts to instantiate `criterion`. The instantiation goal tries values for `criterion` in descending order.

> **Note**
>
> **WARNING** This function assumes that all unary or state resources that are selected are not kept open, that is, that there are no unknown resource constraints yet to be added. If you cannot close all unary or state resources, you should use a resource selector that selects only closed resources. In such a case, you should also handle the ranking of the resources that are not closed yourself.

Implementation

For an example of how the goals returned by these functions could be implemented, see `IlcRankBackward`. Also see Ranking .

See Also: IloStateResource, IloUnaryResource, IlcSchedule, IlcRankBackward, IloResourceConstraintSelector

Global function IlcActivityDurationMaxEvaluator

```
public IloEvaluator< IlcActivity > IlcActivityDurationMaxEvaluator(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an activity evaluator whose `operator(const IlcActivity& activity)` returns the maximum value of the duration variable of the activity.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcTextureAltSuccessorGoal

```
public IlcGoal IlcTextureAltSuccessorGoal(IlcSchedule s, IloSelector< IlcResource,
IlcSchedule > rSel=0)
public IlcGoal IlcTextureAltSuccessorGoal(IlcSchedule s, IlcIntVar criterion,
IloSelector< IlcResource, IlcSchedule > rSel=0)
public IlcGoal IlcTextureAltSuccessorGoal(IlcUnaryResource res)
```

Definition file: ilsched/txtgoal.h
Include file: <ilsched/ilsched.h>

These functions create and return goals that interleave the assignment of resources to alternative resource constraints and the sequencing of resource constraints on unary resource(s). Given a resource, either by definition or by a resource selector, the goal identifies the time point on the resource with the highest criticality according to the texture measurements. The resource constraint with the maximum contribution to the resource and time point is then examined. If that resource constraint has no alternatives and if there exists another non-alternative resource constraint with which it can be sequenced, a precedence constraint (using the `setSuccessor` method) is posted. Otherwise, a new constraint is added specifying that the corresponding activity must execute on another resource.

The sequence between the two activities is the one that results in the greatest remaining pair-wise slack.

The goal created by this function assumes that the precedence graph exists on the resources.

Functions taking an IlcSchedule

The resource selector indicated by `rSel` selects the next resource. By default, that is, when no resource selector is given, the resources are selected in descending order of texture criticality. The pair of activities to sequence is chosen in descending order of their contribution to the critical time point on the selected resource.

Note that the texture measurements are recalculated during every execution of the goal, and therefore the decision-making focus may opportunistically move from one resource to another without the former resource being completely sequenced.

For more information, see Texture Measurements.

See Also: IlcTextureSuccessorGoal, IlcResourceTexture, IlcRCTexture

Global function IlcResourceConstraintHasPrevPredicate

```
public IloPredicate< IlcResourceConstraint >
IlcResourceConstraintHasPrevPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint predicate whose `operator(const IlcResourceConstraint& rc)` returns `IlcTrue` if and only if the immediately preceding activity of the activity of `rc` is known. In case `rc` represents a virtual node, it is assumed that it represents the virtual sink node, and the evaluation of the predicate returns `IlcTrue` if and only if the teardown node (that is, the node that directly precedes the virtual sink node), is known. This predicate is implemented using `IlcResourceConstraint::hasPrevRC`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceIsStateResourcePredicate

```
public IloPredicate< IlcResource > IlcResourceIsStateResourcePredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource predicate whose `operator(const IlcResource& resource)` returns `IlcTrue` if and only if the resource is an `IlcStateResource`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcTryRankLast

```
public IlcGoal IlcTryRankLast(IlcResourceConstraint rct)
```

Definition file: ilsched/srchgoal.h
Include file: <ilsched/search.h>

This function sets a choice point and then ranks the resource constraint `rct` to be last on its resource. In case of failure, `rct` is set to be not last on its resource. If the resource is not closed, then ranking `rct` to be not last has no influence on the latest end time of the activity of `rct`.

Ranking facilities are defined only for unary and state resources whose ranking information is available (see `IlcResource::hasRankInfo`). A resource constraint can be ranked if and only if its time extent is `IlcFromStartToEnd`.

For more information, see Ranking .

See Also: IlcRank, IlcRankBackward, IlcResourceConstraint, IlcStateResource, IlcTimeExtent, IlcUnaryResource

Global function IlcMakeTransitionTime

```
public IlcTransitionTimeObject IlcMakeTransitionTime(IlcTransitionTable table,
IlcBool triangularInequality=IlcFalse)
```

Definition file: ilsched/trancost.h
Include file: <ilsched/ilsched.h>

This function creates an instance of `IlcTransitionTimeObject` using an instance of `IlcTransitionTable`. If the value of the argument `triangularInequality` is `IlcTrue`, it means that the transition table satisfies the triangular inequality. This may be useful to improve the performances of the search as it will not be necessary to check it.

The transition times of an instance of `IlcTransitionTimeObject` use the transition type of the activities of the resource constraints. Notice that this implies that the transition type of the activities must be non-negative and strictly smaller than the size of the table.

See Also: IlcActivity::getTransitionType, IlcMakeTransitionCost, IlcTransitionTable, IlcTransitionTimeObject

Global function IlcResourceConstraintPossibleTeardownPredicate

```
public IloPredicate< IlcResourceConstraint >
IlcResourceConstraintPossibleTeardownPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint predicate whose `operator(const IlcResourceConstraint& rc)` returns `IlcTrue` if and only if `rc` can be a teardown resource constraint. This member function returns `IlcFalse` if `rc` represents a virtual source or sink node (empty handle). This predicate is implemented using `IlcResourceConstraint::isPossibleTeardown`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcRCTextureESTFactory

```
public IlcRCTextureFactory IlcRCTextureESTFactory(IloSolver solver)
```

Definition file: ilsched/texture.h
Include file: <ilsched/ilsched.h>

This function returns an `IlcRCTextureFactory` instance whose `IlcRCTextureFactory::createRCTexture` method returns an instance of `IlcRCTextureESTI`.

Global function IlcResourceIsDiscreteEnergyPredicate

```
public IloPredicate< IlcResource > IlcResourceIsDiscreteEnergyPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource predicate whose `operator(const IlcResource& resource)` returns `IlcTrue` if and only if the resource is an `IlcDiscreteEnergy`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcGetThreadId

```
public IlcInt IlcGetThreadId()
```

Definition file: ilsched/workserv.h
Include file: <ilsched/workserv.h>

This function returns an integer that uniquely identifies the thread from which it was called. The returned value is system dependent.

See Also: IlcWorkServer

Global function IlcResourceConstraintPossibleNextVisitor

```
public IloVisitor< IlcResourceConstraint, IlcResourceConstraint >
IlcResourceConstraintPossibleNextVisitor(IlcManager)
public IloVisitor< IlcResourceConstraint, IlcResourceConstraint >
IlcResourceConstraintPossibleNextVisitor(IloEnv)
```

Definition file: ilsched/srchseq.h
Include file: <ilsched/ilsched.h>

This function returns a visitor that allows to traverse the set of resource constraints that are possibly next to the resource constraint given as container. In case the resource constraint given as container is a sequence virtual node (see `IlcResourceConstraint::isVirtualNode`), the visitor traverses the set of resource constraints that are possible setup resource constraints on the resource.

See the section about Visitors in the concept Selectors of the *IBM ILOG Solver Reference Manual*

Global function IlcAltResConstraintResourceSelectedPredicate

```
public IloPredicate< IlcAltResConstraint >
IlcAltResConstraintResourceSelectedPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an alternative resource constraint predicate whose `operator(const IlcAltResConstraint& altrc)` returns `IlcTrue` if and only if a single resource has been selected for the activity corresponding to `altrc`. If there are still multiple resources that can be selected, `IlcFalse` is returned. This predicate is implemented using `IlcAltResConstraint::isResourceSelected`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceRankedPredicate

`public IloPredicate< IlcResource > IlcResourceRankedPredicate(IlcManager)`

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource predicate whose `operator(const IlcResource& resource)` returns `IlcTrue` if and only if ranking is not supported on the resource (if `IlcResource::hasRankInfo` returns `IlcFalse`) or if ranking is supported and the resource constraints on the resource are completely ranked.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IloSequenceBackward

```
public IloGoal IloSequenceBackward(const IloEnv env, const IloUnaryResource uRes)
public IloGoal IloSequenceBackward(const IloEnv env, const IloNumVar criterion,
IloResourceSelector rSel=IloSelResMinGlobalSlack)
public IloGoal IloSequenceBackward(const IloEnv env, IloResourceSelector
rSel=IloSelResMinGlobalSlack)
public IloGoal IloSequenceBackward(const IloEnv env, const IloNumVar criterion,
const IloTransitionParam param, IloSequenceIndexSelector
iSel=IloSelPrevRCMaxCostStartMin, IloResourceSelector rSel=IloSelResMinGlobalSlack)
public IloGoal IloSequenceBackward(const IloEnv env, const IloUnaryResource uRes,
const IloTransitionParam param, IloSequenceIndexSelector
iSel=IloSelPrevRCMaxCostStartMin)
public IloGoal IloSequenceBackward(const IloEnv env, const IloTransitionParam
param, IloSequenceIndexSelector iSel=IloSelPrevRCMaxCostStartMin,
IloResourceSelector rSel=IloSelResMinGlobalSlack)
```

Definition file: ilsched/iloschedgoals.h
Include file: <ilsched/iloscheduler.h>

These functions return instances of `IloGoal` that sequence instances of `IloUnaryResource` from their teardown activity to their setup activity (sequencing in reverse). The involved resources must be instances of `IloUnaryResource`, and they must not be kept open.

When the argument `uRes` is given, the function returns an instance of `IloGoal` that sequences `uRes`. When there is no resource argument, all the unary resources in the model that are not to be kept open are successively sequenced in reverse according to the unary resource selector argument `rSel`. By default, the resource selector is `IloSelResMinGlobalSlack`.

The selector `iSel`, a value from the enumeration `IloSequenceIndexSelector`, selects a possible value of a previous resource constraint of each resource constraint. The default selector is `IloSelPrevRCMaxCostStartMin`.

When the argument `criterion` is used, the goal also assigns it to its smallest consistent value after the sequencing has been completed.

When an `IloSequenceIndexSelector` is used, the estimation of the cost will be determined by the `IloTransitionParam` parameter passed to the constructor. The default value for such an index selector is `IloSelPrevRCMaxCostStartMin`.

See Also: IloSequenceForward, IloUnaryResource, IloTransitionParam, IloSequenceIndexSelector

Global function IlcShapeLowerThan

```
public IlcConstraint IlcShapeLowerThan(IlcResourceConstraint resCt1,
IlcResourceConstraint resCt2)
```

Definition file: ilsched/shaperct.h

This function returns a Solver constraint acting on each resource constraint passed as an argument. An error will be raised if one the resource constraints does not have a shape specified. The constraint returned will state that the contribution of the first resource constraint's shape must be less than or equal to the contribution of the second shape. This condition must hold at each time point.

See Also: IlcResourceConstraint, IlcShape

Global function IlcResourceHasTexturePredicate

`public IloPredicate< IlcResource > IlcResourceHasTexturePredicate(IlcManager)`

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource predicate whose `operator(const IlcResource& resource)` returns `IlcTrue` if and only if the resource has an associated `IlcResourceTexture` instance.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceConstraintRandomEvaluator

```
public IloEvaluator< IlcResourceConstraint >
IlcResourceConstraintRandomEvaluator(IlcManager, IloRandom)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint evaluator whose `operator(const IlcResourceConstraint& ct)` returns a random number drawn with uniform probability from the interval `[0..1)`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceResourceConstraintTranslator

```
public IloTranslator< IlcResource, IlcResourceConstraint >
IlcResourceResourceConstraintTranslator(IloEnv)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a translator which implements a function (`IloTranslator::operator`) that, when given a resource constraint, returns the resource of this resource constraint. It is useful for transforming resource predicates/selectors into the corresponding resource constraints predicates/selectors.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceResourceConstraintTranslator

```
public IloTranslator< IlcResource, IlcResourceConstraint >
IlcResourceResourceConstraintTranslator(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a translator which implements a function (`IloTranslator::operator`) that, when given a resource constraint, returns the resource of this resource constraint. It is useful for transforming resource predicates/selectors into the corresponding resource constraints predicates/selectors.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceSequencedPredicate

`public IloPredicate< IlcResource > IlcResourceSequencedPredicate(IlcManager)`

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource predicate whose `operator(const IlcResource& resource)` returns `IlcTrue` if and only if sequencing is not supported on the resource (if the resource is not a closed unary resource with a posted sequence constraint) or if sequencing is supported and the resource constraints on the resource are completely sequenced.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceConstraintInwardConstraintPredicate

```
public IloPredicate< IlcResourceConstraint >
IlcResourceConstraintInwardConstraintPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint predicate whose `operator(const IlcResourceConstraint& rc)` returns `IlcTrue` if and only if `rc` was constructed by the member function `IlcActivity::requiresNot` or was extracted from an `IloResourceConstraint` that was constructed by the member function `IloActivity::requiresNot`. This member function returns `IlcFalse` if the resource constraint `rc` is an empty handle; that is, if it corresponds to a virtual resource constraint (source or sink node). This predicate is implemented using `IlcResourceConstraint::possiblyContributes`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcActivityEndMinEvaluator

`public IloEvaluator< IlcActivity > IlcActivityEndMinEvaluator(IlcManager)`

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an activity evaluator whose `operator(const IlcActivity& activity)` returns the minimum value of the end time variable of the activity.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function operator<=

```
public IlcConstraint operator<=(IlcIntTimetable, IlcIntToIntStepFunction)
public IlcConstraint operator<(IlcIntTimetable, IlcIntToIntStepFunction)
public IlcConstraint operator>=(IlcIntTimetable, IlcIntToIntStepFunction)
public IlcConstraint operator>(IlcIntTimetable, IlcIntToIntStepFunction)
public IlcConstraint operator==(IlcIntTimetable, IlcIntToIntStepFunction)
public IlcConstraint operator<=(IlcIntToIntStepFunction f, IlcIntTimetable t)
public IlcConstraint operator<(IlcIntToIntStepFunction f, IlcIntTimetable t)
public IlcConstraint operator>=(IlcIntToIntStepFunction f, IlcIntTimetable t)
public IlcConstraint operator>(IlcIntToIntStepFunction f, IlcIntTimetable t)
public IlcConstraint operator==(IlcIntToIntStepFunction f, IlcIntTimetable t)
```

Definition file: ilsched/timetabh.h
Include file: <ilsched/ilsched.h>

These constraint operators create constraints between an integer step function and an integer timetable that constrain the maximal (or minimal) availability profile of the resources.

Note

Any modification of the `IlcIntToIntStepFunction` argument after the constraint has been posted will have no effect on the constraint.

Example

The following code defines an integer step function `capMax` and uses it to constrain the maximal availability profile of a discrete resource `res`.

```
// Must be during search (e.g., inside a goal)
  IloSolver solver = getSolver();
IlcScheduler schedule(solver, 0, 100);

IlcDiscreteResource res(schedule,10);
IlcIntToIntStepFunction capMax(solver,0,100);
capMax.setSteps(IlcIntArray(solver,3,30,50,90),
                IlcIntArray(solver,4,10,8,6,10));
solver.add(res.getTimetable() >= capMax); // ...
```

Global function IloTextureAltSuccessorGoal

```
public IloGoal IloTextureAltSuccessorGoal(const IloEnv env)
public IloGoal IloTextureAltSuccessorGoal(const IloEnv env, const IloNumVar
criterion)
```

Definition file: ilsched/iloschedgoals.h
Include file: <ilsched/iloscheduler.h>

This function creates and returns a goal that adds successor relations between all pairs of resource constraints, and assigns all alternative resource constraints on all resources in the model that have a non-zero texture measurement. This goal can be combined with `IloAssignAlternative` to create a complete goal which includes cases with a texture measurements of zero (`IloTextureAltSuccessorGoal && IloAssignAlternative`).

If the argument `criterion` is given, then this variable will be bound, if possible, to its minimal value at the end of the search.

At each step of the goal the resource `R` and the time point with highest criticality are identified, and the set of resource constraints `S` that contribute to the criticality of `R` at the most critical time point are examined. Three resource constraints are identified:

1. The non-alternative resource constraint, `A`, with the highest contribution to the critical point that is not a successor or predecessor of all other elements of `S`.
2. The non-alternative resource constraint, `B`, with the highest contribution to the critical point that is not `A`, nor a successor or predecessor of `A`.
3. The alternative resource constraint, `C`, with the highest contribution to the critical point of all alternative resource constraints.

The contribution to the critical point by resource constraint `A` is then compared to that of resource constraint `C`. If `A` has the higher contribution, then a choice point based on a successor relation between `A` and `B` is formed. This choice point is exactly the same form as that created by `IloTextureSuccessorGoal`.

If the contribution of `C` is greater than that of `A`, a choice point that removes the resource `R` from the set of possible resources for `C` is created. On backtracking, the choice point enforces that `C` must execute on resource `R`.

Global function IlcResourceHasBreaksPredicate

`public IloPredicate< IlcResource > IlcResourceHasBreaksPredicate(IlcManager)`

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource predicate whose `operator(const IlcResource& resource)` returns `IlcTrue` if and only if the resource has breaks.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcActivityProcessingTimeMinEvaluator

```
public IloEvaluator< IlcActivity >
IlcActivityProcessingTimeMinEvaluator(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an activity evaluator whose `operator(const IlcActivity& activity)` returns the minimum value of the processing time variable of the activity.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IloTimeWindowForwardChronologicalComparator

```
public IloComparator< IloTimeWindowNHoodI::IloTimeWindow >
IloTimeWindowForwardChronologicalComparator(IloSolver solver)
```

Definition file: ilsched/ilolnsgoals.h
Include file: <ilsched/iloscheduler.h>

This function returns a predefined time interval comparator. The comparator compares two instances of the class `IloTimeWindowNHoodI::IloTimeWindow`, `tw1` and `tw2`. The time interval `tw1` is less than time interval `tw2` if the start of `tw1` is less than the start of `tw2`. If the starts are equal, it compares the ends of both intervals. The comparator is allocated on the memory allocation stack of `solver`.

See `IloComparator` in the *IBM ILOG Solver Reference Manual* for more information.

See Also: IloSchedulerLargeNHood

Global function IloTimeWindowForwardChronologicalComparator

```
public IloComparator< IloTimeWindowNHoodI::IloTimeWindow >
IloTimeWindowForwardChronologicalComparator(IloEnv env)
```

Definition file: ilsched/iloInsgoals.h

This function returns a predefined time interval comparator. The comparator compares two instances of the class `IloTimeWindowNHoodI::IloTimeWindow`, `tw1` and `tw2`. The time interval `tw1` is less than time interval `tw2` if the start of `tw1` is less than the start of `tw2`. If the starts are equal, it compares the ends of both intervals. The comparator is allocated on the memory allocation stack of `env`.

See `IloComparator` in the *IBM ILOG Solver Reference Manual* for more information.

See Also: IloSchedulerLargeNHood

Global function IlcResourceCapacityEvaluator

```
public IloEvaluator< IlcResource > IlcResourceCapacityEvaluator(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource evaluator whose `operator(const IlcResource& resource)` returns the theoretical capacity of the `IlcCapResource` object to which it is applied. The behavior is undefined if this evaluator is applied to a resource that is not an `IlcCapResource`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IloAssignAlternative

```
public IloGoal IloAssignAlternative(const IloEnv env, IloResourceSelector
possibleSel=IloSelAltRes)
public IloGoal IloAssignAlternative(const IloEnv env, const IloResource resource)
public IloGoal IloAssignAlternative(const IloEnv env, const IloAltResSet resources,
IloResourceSelector possibleSel=IloSelAltRes)
```

Definition file: ilsched/iloschedgoals.h
Include file: <ilsched/iloscheduler.h>

This function returns a goal that assigns a possible resource as the selected one for an alternative resource constraint.

- If the second argument is an instance of `IloAltResSet`, it considers all the constraints on that instance of `IloAltResSet`.
- If the second argument is an instance of `IloResource`, it considers all the constraints on instances of `IloAltResSet` for which that resource is a possible alternative.
- If the second argument is an `IloResourceSelector`, it considers all the constraints on instances of `IloAltResSet` for which that resource is a possible alternative.

The goal uses the selector `possibleSel` to choose the resource. If no argument of type `IloResourceSelector` is given, the goal uses the default selector, `IloSelAltRes`. The goal selects posted and metaposted alternative resource constraints in arbitrary order. To customize this behavior it is necessary to directly use the Scheduler Engine classes `IlcAssignAlternative` and `IloSelector<IlcAltResConstraint>`.

Implementation

See `IlcAssignAlternative` for an example of how this function can be defined.

See Also: IloAltResSet, IloResource, IlcAssignAlternative

Global function IlcResourceEnergyEvaluator

```
public IloEvaluator< IlcResource > IlcResourceEnergyEvaluator(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource evaluator whose `operator(const IlcResource& resource)` returns the maximum theoretical energy level of the `IlcDiscreteEnergy` object to which it is applied. The behavior is undefined if this evaluator is applied to a resource that is not an `IlcDiscreteEnergy` resource.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceConstraintCapacityConstraintPredicate

```
public IloPredicate< IlcResourceConstraint >
IlcResourceConstraintCapacityConstraintPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint predicate whose `operator(const IlcResourceConstraint& rc)` returns `IlcTrue` if and only if `rc` requires or provides a quantity of a resource (rather than a state). This member function returns `IlcFalse` if `rc` represents a virtual source or sink node (empty handle). This predicate is implemented using `IlcResourceConstraint::isCapacityConstraint`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IloSetTimesBackward

```
public IloGoal IloSetTimesBackward(const IloEnv env, IloActivitySelector
activitySelector=IloSelLastActMaxStartMin)
public IloGoal IloSetTimesBackward(const IloEnv env, const IloNumVar criterion,
IloActivitySelector activitySelector=IloSelLastActMaxStartMin)
```

Definition file: ilsched/iloschedgoals.h
Include file: <ilsched/iloscheduler.h>

This function creates and returns a goal that assigns an end time to all activities in the model. If the argument `criterion` is given, then this variable will be bound, if possible, to its maximal value at the end of the search. By default, that is, if no activity selector is given as an argument, the activity selector `IloSelLastActMaxStartMin` selects the next activity.

Note

WARNING In order to ensure a purely anti-chronological scheduling, the supplied activity selector should always choose an unscheduled activity of maximal latest end time. Furthermore, scheduling an activity at time t should not have an impact on the latest end times of later activities.

In particular, one should be careful in using precedence constraints with a negative delay (and similar Solver constraints on start and end variables).

For further details about the interpretation of the `IloSetTimesBackward` goal in the Scheduler Engine and about these restrictions, see `IlcSetTimesBackward`.

See Also: IlcSetTimesBackward, IloActivitySelector

Global function IlcSequence

```
public IlcGoal IlcSequence(IlcUnaryResource resource, IloSelector<
IlcResourceConstraint, IlcResourceConstraint > nextSelect=0)
public IlcGoal IlcSequence(IlcSchedule schedule, IloSelector<
IlcResourceConstraint, IlcResourceConstraint > nextSelect)
public IlcGoal IlcSequence(IlcSchedule schedule, IloSelector< IlcResource,
IlcSchedule > resSel=0, IloSelector< IlcResourceConstraint, IlcResourceConstraint >
nextSelect=0)
public IlcGoal IlcSequence(IlcSchedule schedule, IlcIntVar criterion, IloSelector<
IlcResourceConstraint, IlcResourceConstraint > nextSelect)
public IlcGoal IlcSequence(IlcSchedule schedule, IlcIntVar criterion, IloSelector<
IlcResource, IlcSchedule > resSel=0, IloSelector< IlcResourceConstraint,
IlcResourceConstraint > nextSelect=0)
```

Definition file: ilsched/srchseq.h
Include file: <ilsched/search.h>

These functions return instances of `IlcGoal` that sequence instances of `IlcUnaryResource` from their setup activity to their teardown activity (sequencing forward). The involved resources must be instances of `IlcUnaryResource`, closed, and with their precedence graph constraint posted.

When the argument `resource` is given, the function returns an instance of `IlcGoal` that sequences `resource`.

When the argument `schedule` is given, all the closed unary resources of `schedule` with a posted precedence graph constraint are successively sequenced according to the resource selector argument `resSel`. By default, the resource selector is defined as follows:

```
IloSelector<IlcResource,IlcSchedule> resSel(!IlcResourceSequencedPredicate(solver),
                                IlcResourceGlobalSlackEvaluator(solver));
```

Note

WARNING `IlcSequence` can only be applied to `IlcUnaryResource` instances. When defining your own selectors make sure that the predicate `IlcResourceSequencedPredicate` (solver) or the predicate `IlcResourceIsUnaryResourcePredicate` (solver) is used.

The resource constraint selector `nextSelect` selects a resource constraint that is possibly next to the lastly sequenced resource constraint. The instance of `IlcResourceConstraint` in the context of this selector represents the lastly sequenced resource constraint. At the begining of the search when no resource constraint has been sequenced, the search goal passes the sequence virtual node of the resource as context. See `IlcUnaryResource::getVirtualNodeRC`. When this selector is not specified, the next selector object used by default selects the possibly next resource constraint with the smallest minimal start time, using the smallest maximal end time to break ties.

If the argument `criterion` is given, then this variable will be bound, if possible, to its minimal value at the end of the search.

For more information, see Sequence Constraint.

See Also: IlcSequenceBackward, IlcTestSequencedResource, IlcUnaryResource

Global function IlcResourceClosedPredicate

```
public IloPredicate< IlcResource > IlcResourceClosedPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource predicate whose `operator(const IlcResource& resource)` returns `IlcTrue` if and only if the resource is closed.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcActivityPostponedPredicate

```
public IloPredicate< IlcActivity > IlcActivityPostponedPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an activity predicate whose `operator(const IlcActivity& activity)` returns `IlcTrue` if and only if the activity is postponed.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

See Also: IlcSetTimes

Global function IlcSetTimes

```
public IlcGoal IlcSetTimes(IlcSchedule schedule, IloSelector< IlcActivity,
IlcSchedule > aSel=0)
public IlcGoal IlcSetTimes(IlcSchedule schedule, IlcIntVar criterion, IloSelector<
IlcActivity, IlcSchedule > aSel=0)
```

Definition file: ilsched/srchgoal.h
Include file: <ilsched/search.h>

This function creates and returns a goal that assigns a start time to all activities managed by `schedule`. If the argument `criterion` is given, then the assignments are made to minimize `criterion`. The activity selector `aSel` selects the next activity. By default, that is, if no activity selector is given as an argument, the activity selector used is defined as follows:

```
IloSelector<IlcActivity,IlcSchedule> aSel = IlcActivityInScheduleSelector(s);
sel.setPredicate(!IlcActivityStartVarBoundPredicate(s) &&
                  !IlcActivityPostponedPredicate(s));
sel.setComparator(IlcLexicalComposition(IlcCompareMin(IlcActivityStartMinEvaluator(s)),
                                        IlcCompareMin(IlcActivityEndMaxEvaluator(s))));
```

The goal is designed to efficiently schedule activities in a chronological order. It considers only solutions that can be produced as follows: in each step, choose an unscheduled activity *A* of minimal earliest start time and schedule it as early as possible, as allowed by the previously scheduled activities (which have smaller start times than *A*).

Internally, the function uses a schedule-or-postpone method that works as follows:

1. A selected activity is assigned to its earliest start time.
2. If that start time leads to a failure, the activity is postponed until its earliest start time has been removed. This removal can occur as a result of a combination of decisions and propagation.

Before assigning the earliest start time to an activity with a candidate start time *st*, it is determined whether an already postponed activity *actP* exists that can be or should be scheduled before *st*, that is, whether *actP.getEndMin() <= st* or *actP.getStartMax() <= st*. If this is the case, a fail is generated based on the reasoning that if we have “normal” precedence and resource constraints, the earliest start time of *actP* will never be removed and thus *actP* will remain postponed and no solution will be found in this branch of the search tree.

This reasoning can result in an incomplete search (that is, solutions may be missed) in some situations. To take a specific example, suppose we have an activity `B` that can start as much as 50 units after the start of an activity `A`. This can be expressed by a precedence constraint with negative delay:

```
A.startsAfterStart(B, -50)
```

Assume that `B` cannot be scheduled at a time earlier than 100 because it requires a resource that has a maximal capacity of 0 in the interval [0,100). `A`, therefore, cannot be scheduled before time 50 even if there are no earlier activities. `IlcSetTimes` will not find a solution here, because it will attempt to assign `A` to a start time of 0. When this fails, it will postpone `A` but then realize that there are no other activities that can be schedule before `A` and therefore it concludes that there are no solutions. Note that in this very simple case, it is likely that constraint propagation will discover that 50 is the actual earliest start time of `A` and so `IlcSetTimes` will successfully find a solution. However, if this situation is part of a larger, more complex constraint interaction, it cannot be guaranteed that constraint propagation will discover the globally consistent earliest start time. In such a situation, then, a solution will be missed.

Another situation where missed solutions can arise is when the processing time of an activity depends on its start or end time. For example, we have three activities, `A`, `B`, and `C`, all of which require the same unary capacity resource. Activities `B` and `C` have processing times of 10. Activity `A` has a processing time of 1 if its start time is in the set {1, 2, 3, 4} and a processing time of 10 otherwise. The scheduling horizon is 25 time units and `B` and `C` can only start at or after time unit 1.

The Scheduler Concert Technology code below and the output demonstrate that such a situation leads to a missed solution when `IlcSetTimes` (extracted from the `IloSetTimesForward` goal) is used.

```
IloEnv env;
IloModel model(env);

IloSchedulerEnv schedEnv(env);
schedEnv.setOrigin(0);
schedEnv.setHorizon(25);

IloNumVar ptA(env, 1, 10, ILOINT);
IloActivity a(env, ptA);
a.setName("A");

IloActivity b(env, 10, "B");
IloActivity c(env, 10, "C");

model.add(b.startsAfter(1));
model.add(c.startsAfter(1));

// processing time is 1 iff start time is in set {1, 2, 3, 4}
// otherwise processing time is 10

IloNumVar stA = a.getStartVariable();
model.add((stA < 1) || (stA > 4) || (ptA == 1));
model.add(((stA >= 1) && (stA <= 4)) || (ptA == 10));

IloUnaryResource r(env);
model.add(a.requires(r));
model.add(b.requires(r));
model.add(c.requires(r));

IloSolver solver(model);
IloBool solved = solver.solve(IloSetTimesForward(env));
solver.out() << "First solve: " << solved << endl;

model.add(a.startsAt(1));
solved = solver.solve(IloSetTimesForward(env));
solver.out() << "Solve after adding a.startsAt(1): "
             << solved << endl;

Output:
First solve: 0
Solve after adding a.startsAt(1): 1
```

What happens in this example is that the `IlcSetTimes` goals tries to assign activity `A` to start time 0. This leads to a dead-end because this implies that all three activities have a processing time of 10 which cannot be accommodated on a unary resource within the 25 time-unit scheduling horizon. Activity `A` is then postponed and the goal attempts to assign `B` to time 1. This fails, as all three activities again would have to have processing times of 10. Finally, the goal tries to schedule activity `C` at time 1, with the same result as when it tried activity `B`. Therefore, the goal concludes, there are no solutions.

Clearly, however, there are solutions, as demonstrated when we assign the start time of `A` to be 1. The processing time of `A` is therefore 1 and the other two activities can then follow within the scheduling horizon.

In general, `IlcSetTimes` may miss solutions if there are precedence constraints with negative delay, if the processing time of an activity depends on its start or end time, or if reservoirs are used.

Note

WARNING In order to ensure a purely chronological scheduling, the supplied activity selector should always choose an unscheduled activity of minimal earliest start time. Furthermore, scheduling an activity at time t should not have an impact on the earliest start times of activities that have been scheduled earlier.

In particular, care should be taken in using precedence constraints with a negative delay, activities where the processing time depends on the start or end time of the activity, and reservoirs.

See `IloSelector` in the *IBM ILOG Solver Reference Manual* for more information.

See Also: IlcSchedule, IlcScheduleOrPostpone, IlcSetTimesBackward

Global function IlcTrySetSuccessor

```
public IlcGoal IlcTrySetSuccessor(IlcResourceConstraint, IlcResourceConstraint)
```

Definition file: ilsched/srchpg.h
Include file: <ilsched/search.h>

This function sets a choice point and then adds the precedence relation `rct1.setSuccessor(rct2)` on the precedence graph of the resource with the method `IlcResourceConstraint::setSuccessor`. In case of failure, the precedence relation `rct2.setSuccessor(rct1)` is added on the precedence graph of the resource.

This goal can be used only when a precedence graph constraint has been posted on the resource.

For more information, see Precedence Graph Constraints.

See Also: IlcResource, IlcResourceConstraint

Global function IlcResourceConstraintVirtualNodePredicate

```
public IloPredicate< IlcResourceConstraint >
IlcResourceConstraintVirtualNodePredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint predicate whose `operator(const IlcResourceConstraint& rc)` returns `IlcTrue` if and only if `rc` represents a sequence virtual node. The sequence virtual node of a unary resource is an automatically created resource constraint that does not affect the availability of the resource and that is used in the sequence goals and selectors to represent the virtual initial (in case of a chronological sequence goal like `IlcSequence`) or final (in case of an antichronological sequence goal like `IlcSequenceBackward`) resource constraint in the sequence of resource constraints of the unary resource .

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcTryRankFirst

```
public IlcGoal IlcTryRankFirst(IlcResourceConstraint rct)
```

Definition file: ilsched/srchgoal.h
Include file: <ilsched/search.h>

This function sets a choice point and then ranks the resource constraint `rct` to be first on its resource. In case of failure, `rct` is set to be not first on its resource. If the resource is not closed, then ranking `rct` to be not first has no influence on the earliest start time of the activity of `rct`.

Ranking facilities are defined only for unary and state resources whose ranking information is available (see `IlcResource::hasRankInfo`). A resource constraint can be ranked if and only if its time extent is `IlcFromStartToEnd`.

For more information, see Ranking .

See Also: IlcRank, IlcResourceConstraint, IlcStateResource, IlcTimeExtent, IlcUnaryResource

Global function IlcActivityProcessingTimeVarBoundPredicate

```
public IloPredicate< IlcActivity >
IlcActivityProcessingTimeVarBoundPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns an activity predicate whose `operator(const IlcActivity& activity)` returns `IlcTrue` if and only if the processing time variable of the activity is bound.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceHasAltResConstraintPredicate

```
public IloPredicate< IlcResource >
IlcResourceHasAltResConstraintPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource predicate whose `operator(const IlcResource& resource)` returns `IlcTrue` if and only if there exists a resource constraint on the resource that is an `IlcAltResConstraint` that has not been assigned to a resource.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Global function IlcResourceConstraintPossiblyContributesPredicate

```
public IloPredicate< IlcResourceConstraint >
IlcResourceConstraintPossiblyContributesPredicate(IlcManager)
```

Definition file: ilsched/selector.h
Include file: <ilsched/ilsched.h>

This function returns a resource constraint predicate whose `operator(const IlcResourceConstraint& rc)` returns `IlcTrue` if and only if `rc` possibly affects the availability of the resource. Otherwise, it returns `IlcFalse`. This member function returns `IlcFalse` if the resource constraint `rc` is an empty handle; that is, if it corresponds to a virtual resource constraint (source or sink node). This predicate is implemented using `IlcResourceConstraint::possiblyContributes`.

This functions exists with either an `IloEnv` or an `IloSolver` as argument.

Macro ILCALTRCDEMON

Definition file: ilsched/altresh.h

ILCALTRCDEMON(DemonClass, ConstraintClass, method)

This macro creates an instance of the class `DemonClass` which is a subclass of `IlcAltRCDemon`. When this demon is triggered, it executes the function `method` of the constraint `ConstraintClass` given as parameter to the macro. The signature of this method must be: `void ConstraintClass::method(IlcAltResConstraint rc, IlcResource resource)`. The argument `resource` is a possible resource whose change in some ranges for the alternative resource constraint is responsible for the triggering of the demon; for example, the start min of the activity if the resource was selected in the alternative resource constraint. Actually, the demon is trigerred each time a change happens in the start, end, duration, processing time, or capacity range.

Once the resource demon class has been defined with the macro `ILCALTRCDEMON(DemonClass, ConstraintClass, method)`, an instance of this demon can be created by passing an instance of `ConstraintClass` as a parameter of the member function `IlcAltResConstraint::whenRange` as follows:

```
ConstraintClass* ct = ...;
IlcAltRCDemon myDemon = DemonClass(ct);
```

Example

The following code defines a demon `AltRCDemonCaller` that prints pieces of information about the resources whose ranges change.

```
class AltRCDemonCallI : public IlcConstraintI {
  IlcAltResConstraint _ct;
public:
  AltRCDemonCallI(IlcManager m, IlcAltResConstraint ct)
    :IlcConstraintI(m), _ct(ct)
    {}
  ~AltRCDemonCallI() {}
  virtual void post();
  virtual void propagate();
  void showInfo(IlcAltResConstraint rc, IlcResource resource);
};

void AltRCDemonCallI::showInfo(IlcAltResConstraint rc, IlcResource resource) {
  IlcAltResSet set = rc.getAltResSet();
  cout << endl << "--------AltRCDemonCallI---------" << endl;
  cout << "IlcAltResConstraint :" << rc << endl;
  cout << "IlcResource : " << resource << endl;
  cout << "Index : " << set.getIndex(resource) << " in " << rc.getIndexVariable() << endl;
  cout << "---------------------------------" << endl;
}

ILCALTRCDEMON(AltRCDemonCaller, AltRCDemonCallI, showInfo);

void AltRCDemonCallI::post() {
  _ct.whenRange(AltRCDemonCaller(this));
}

void AltRCDemonCallI::propagate() {
  cout << endl << "--------AltRCDemonCallI---------" << endl;
  cout << "Demon For " << _ct << endl;
  cout << "---------------------------------" << endl;
}

IloSolver ...;
IlcSchedule schedule(s, 0, 500);
IlcDiscreteResource r1(schedule, 2);
IlcDiscreteResource r2(schedule, 2);
r1.setName("r1");
r2.setName("r2");
IlcAltResSet set(schedule, 2, r1, r2);
IlcActivity a1(schedule, 30);
IlcAltResConstraint rc1 = a1.requires(set, 2);
```

```
s.add(rc1);
s.add(new (s.getHeap()) AltRCDemonCallI(s, rc1));
```

See Also: IlcAltResConstraint, IlcAltRCDemon

Macro ILCRESOURCEDEMON

Definition file: ilsched/schedule.h

ILCRESOURCEDEMON(DemonClass, ConstraintClass, method)

This macro creates an instance of the class `DemonClass` which is a subclass of `IlcResourceDemon`. When this demon is triggered, it executes the function `method` of the constraint `ConstraintClass` given as parameter to the macro. The signature of this method must be: `void ConstraintClass::method(IlcResourceConstraint rct)`. The resource constraint given as argument is the one that is responsible for the triggering of the demon; for example, the resource constraint whose set of successors has changed.

Once the resource demon class has been defined with the macro `ILCRESOURCEDEMON(DemonClass, ConstraintClass, method)`, an instance of this demon can be created by passing an instance of `ConstraintClass` as a parameter as follows:

```
ConstraintClass* ct = ...;
IlcResourceDemon myDemon = DemonClass(ct);
```

Example

The following code defines a couple of demons `PrintDemonSucc` and `PrintDemonPred` that respectively print the new successors and predecessors of any resource constraints whose set of successors or predecessors has changed.

```
 class PrintCtI :public IlcConstraintI {
 public:
 PrintCtI(IloSolver solver)
    :IlcConstraintI(solver){}
 void printNewSuccessors(IlcResourceConstraint changedRct) {
   getSolver().out() << "New successors of " << changedRct << ":" << endl;
   for (IlcResourceConstraintDeltaIterator ite(changedRct, IlcSuccessors); ite.ok(); ++ite) {
    getSolver().out() << "t" << *ite << endl;
   }
 }
 void printNewPredecessors(IlcResourceConstraint changedRct) {
   getSolver().out() << "New predecessors of " << changedRct << ":" << endl;
   for (IlcResourceConstraintDeltaIterator ite(changedRct, IlcPredecessors); ite.ok(); ++ite) {
    getSolver().out() << "t" << *ite << endl;
   }
 }
};

ILCRESOURCEDEMON(PrintDemonSucc, PrintCtI, printNewSuccessors);
ILCRESOURCEDEMON(PrintDemonPred, PrintCtI, printNewPredecessors);

IloSolver solver ...;
IlcResource resource ...;

PrintCtI* printCt = new (solver) PrintCtI(solver);

resource.whenSuccessors(PrintDemonSucc(printCt));
resource.whenPredecessors(PrintDemonPred(printCt));
```

See Also: IlcResource, IlcResourceDemon

Macro ILCSCHEDULEDEMON

Definition file: ilsched/schedule.h

ILCSCHEDULEDEMON(DemonClass, ConstraintClass, method)

This macro creates an instance of the class `DemonClass` which is a subclass of `IlcScheduleDemon`. When this demon is triggered, it executes the function `method` of the constraint `ConstraintClass` given as parameter to the macro. The signature of this method must be: `void ConstraintClass::method(IlcActivity act)`. The activity given as argument is the one that is responsible for the triggering of the demon; for example, the activity whose set of successors has changed.

Once the schedule demon class has been defined with the macro `ILCSCHEDULEDEMON(DemonClass, ConstraintClass, method)`, an instance of this demon can be created by passing an instance of `ConstraintClass` as a parameter as follows:

```
ConstraintClass* ct = ...;
IlcScheduleDemon myDemon = DemonClass(ct);
```

See `ILCRESOURCEDEMON` for an example of code using a similar macro.

See Also: IlcSchedule, IlcScheduleDemon

Macro IlcTransitionCost

Definition file: ilsched/trancost.h

`IlcTransitionCost``(fct)`

This macro defines a function that returns an instance of the class `IlcTransitionCostObject`. Such an object can be passed to the function `IlcUnaryResource::addNextTransitionCost` or `IlcUnaryResource::addPrevTransitionCost`. In that case, the object defines a transition cost function to be used by the sequence constraint on the invoking unary resource.

Notice that this transition cost is constant, that is, only the function `IlcTransitionCostObject::getTransitionCost` is defined and any call to `IlcTransitionCostObject::getTransitionCostMin` or `IlcTransitionCostObject::getTransitionCostMax` raises an error. Also note that the setup and teardown costs are zero.

Example

The call `IlcTransitionCost(functionName)` defines the following function:

```
IlcTransitionCostObject functionNameObject(IloSolver s);
```

The argument `functionName` should be a pointer to a function that takes two instances of the class `IlcResourceConstraint` as its arguments and returns an integer. Here is an example of such a function:

```
IlcInt myTransCostFct(const IlcResourceConstraint rct1,
                      const IlcResourceConstraint rct2)
{
  return IlcAbs(rct1.getActivity().getTransitionType() -
                rct2.getActivity().getTransitionType());
}
```

Now using that function and the call `IlcTransitionCost(myTransCostFct)`, we can define the following function:

```
IlcTransitionCostObject myTransCostFctObject(IloSolver s);
```

This function can be used to define a transition cost for a unary resource `resource`:

```
IlcUnaryResource resource(schedule);
s.add(resource.makeSequenceConstraint());
resource.addNextTransitionCost(IlcTransitionCost(myTransCostFct);
```

See Also: IlcTransitionCostObject, IlcUnaryResource

Macro IlcTransitionTime

Definition file: ilsched/schedule.h

IlcTransitionTime(fct)

This macro defines a function that returns an instance of the class `IlcTransitionTimeObject`. Such an object can be passed to constructors of the classes `IlcUnaryResource` and `IlcStateResource`. In that case, the object defines which transition time function will be used for the resource being constructed.

Example

The call `IlcTransitionTime(functionName)` defines the following function:

```
IlcTransitionTimeObject functionNameObject(IloSolver s0);
```

The argument `functionName` should be a pointer to a function that takes two instances of the class `IlcResourceConstraint` as its arguments and returns an integer.

Here is an example of such a function:

```
IlcInt myTransTimeFct(const IlcResourceConstraint rct1,
                      const IlcResourceConstraint rct2)
{
  return (rct1.getActivity().getDurationMin() +
          rct2.getActivity().getDurationMin());
}
```

Now using that function and the call `IlcTransitionTime(myTransTimeFct)`, you can define the following function:

```
IlcTransitionTimeObject myTransTimeFctObject(IloSolver s0);
```

This last function can be used to define the transition time of a resource, like this:

```
IlcUnaryResource resource(schedule, myTransTimeFctObject(solver));
```

See Also: IlcStateResource, IlcTransitionTimeObject, IlcUnaryResource

Macro ILCUSERSHIFTOBJECT

Definition file: ilsched/shifts.h

ILCUSERSHIFTOBJECT(name, start, end, ilcAct, isMin)

This macro defines a user shift object, subclass of `IlcUserShiftObject`), named `name`. The referenced arguments `start` and `end` define the interval `[start, end)` to be propagated. This time interval corresponds to a possible position for the activity `ilcAct`. When the argument `isMin` is true, the interval `[start, end)` is the first possible position of the activity, that is, `[startMin, endMin)`. Otherwise, the interval is the last possible position, that is, `[startMax, endMax)`. Then the body of the macro has to affect valid values to `start` and `end` arguments.

Notice that the written code must return `IlcTrue` when the computed interval `[start, end)` is valid, and returns `IlcFalse` in case of fail.

For example, the following code defines a shift object that forbide beginning or ending during a week-end:

```
ILCUSERSHIFTOBJECT(userWeekEndShift, start, end, ilcAct, isMin) {
  const IlcInt delta = (isMin) ? 7 : 4;
  if (start % 7 > 4) then start = (start / 7) *7 + delta;
  if (end   % 7 > 4) then end   = (end   / 7) *7 + delta;
  return IlcTrue;
}
```

The use of this macro is the only way to define a new subclass of `IlcShiftObject`.

Since the argument `nameI` (`name` followed by "I") is used to name the use shift object class, it is not possible to use the same `nameI` for other classes.

See Also: IlcCalendar, IlcShiftObject

Macro ILORCTEXTUREFACTORY0

Definition file: ilsched/ilotextureparami.h

```
ILORCTEXTUREFACTORY0(nameI, solver)
ILORCTEXTUREFACTORY1(nameI, solver, t1, a1)
ILORCTEXTUREFACTORY2(nameI, solver, t1, a1, t2, a2)
ILORCTEXTUREFACTORY3(nameI, solver, t1, a1, t2, a2, t3, a3)
ILORCTEXTUREFACTORY4(nameI, solver, t1, a1, t2, a2, t3, a3, t4, a4)
```

This macro defines an RC Texture factory, a subclass of `IloRCTextureFactoryI` named `nameI`. The argument `solver` is the name of the `IloSolver` that performs the extraction. Within the macro, this name denotes the solver currently performing the extraction. The types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type `ti` and a name `ai`. The call to the macro must be followed immediately by the body of the `extract` member function of the factory class being defined. It must return a pointer to an instance of `IlcRCTextureFactoryI` that corresponds to the extracted object. Besides the definition of the class `nameI`, this macro also defines a function named `name` that creates an instance of the class `nameI` and returns an instance of the class `IloRCTextureFactoryI*` that points to it.

The use of this macro is the only way to define a new subclass of `IloRCTextureFactoryI`.

Since the argument `name` is used to name the factory class, it is not possible to use the same name for other classes.

Example

This example shows how to define a factory with two data members.

```
ILORCTEXTUREFACTORY2(MyRCTextureFactory, mySolver,
                     IloInt, iloInt,
                     IloNumVar, iloVar) {
  use(mySolver, iloVar);
  IlcIntVar ilcVar = mySolver.getIntVar(iloVar);
  return new (mySolver.getHeap()) MyIlcRCTextureFactory(mySolver,
                                                        iloInt, ilcVar);
}
```

For more information, see Texture Measurements.

See Also: IloRCTextureFactoryI, IloRCTextureFactory, IlcRCTextureFactoryI, IlcRCTextureFactory

Macro ILOTEXTURECRITICALITYCALCULATOR0

Definition file: ilsched/ilotextureparami.h

```
ILOTEXTURECRITICALITYCALCULATOR0(nameI, solver)
ILOTEXTURECRITICALITYCALCULATOR1(nameI, solver, t1, a1)
ILOTEXTURECRITICALITYCALCULATOR2(nameI, solver, t1, a1, t2, a2)
ILOTEXTURECRITICALITYCALCULATOR3(nameI, solver, t1, a1, t2, a2, t3, a3)
ILOTEXTURECRITICALITYCALCULATOR4(nameI, solver, t1, a1, t2, a2, t3, a3, t4, a4)
```

This macro defines an texture criticality calculator, a subclass of `IloTextureCriticalityCalculatorI` named `nameI`. The argument `solver` is the name of the `IloSolver` that performs the extraction. Within the macro, this name denotes the solver currently performing the extraction. When `n` is greater than 0 (zero), the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type `ti` and a name `ai`. The call to the macro must be followed immediately by the body of the `extract` member function of the criticality calculator class being defined. It must return a pointer to an instance of `IlcTextureCriticalityCalculatorI` that corresponds to the extracted object. Besides the definition of the class `nameI`, this macro also defines a function named `name` that creates an instance of the class `nameI` and returns an instance of the class `IloTextureCriticalityCalculatorI*` that points to it.

The use of this macro is the only way to define a new subclass of `IloTextureCriticalityCalculatorI`.

Since the argument `name` is used to name the texture criticality calculator class, it is not possible to use the same name for other classes.

See the `ILORCTEXTUREFACTORY0` macro for an analogous example of how this macro can be used.

For more information, see Texture Measurements.

See Also: IloTextureCriticalityCalculatorI, IloTextureCriticalityCalculator, IlcTextureCriticalityCalculatorI, IlcTextureCriticalityCalculator

Macro ILOTRANSITIONCOSTOBJECT0

Definition file: ilsched/ilotransition.h

```
ILOTRANSITIONCOSTOBJECT0(_this, solver)
ILOTRANSITIONCOSTOBJECT4(_this, solver, t1, a1, t2, a2, t3, a3, t4, a4)
ILOTRANSITIONCOSTOBJECT3(_this, solver, t1, a1, t2, a2, t3, a3)
ILOTRANSITIONCOSTOBJECT2(_this, solver, t1, a1, t2, a2)
ILOTRANSITIONCOSTOBJECT1(_this, solver, t1, a1)
```

This macro defines a transition cost object, subclass of `IloTransitionCostObjectI`, named `nameI`. The argument `solver` is the name of the `IloSolver` that performs the extraction. Within the code of the macro, this name will denote the solver currently performing the extraction. When `n` is greater than 0 (zero), the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type `ti` and a name `ai`. The call to the macro must be followed immediately by the body of the extract member function of the transition cost object class being defined. It must return a pointer to an instance of `IlcTransitionCostObjectI` that corresponds to the extracted object. Besides the definition of the class `nameI`, this macro also defines a function named `name` that creates an instance of the class `nameI` and that returns an instance of the class `IloTransitionCostObject` that points to it.

The use of this macro is the only way to define a new subclass of `IloTransitionCostObjectI`.

Since the argument `name` is used to name the transition cost object class, it is not possible to use the same name for other classes.

For an example on the use of this macro, see information for the similar macro `ILOTRANSITIONTIMEOBJECT0`.

For more information, see Transition Costs.

See Also: IloTransitionCostObject, IloTransitionCostObjectI, IloTransitionCost

Macro ILOTRANSITIONTIMEOBJECT0

Definition file: ilsched/ilotransition.h

```
ILOTRANSITIONTIMEOBJECT0(_this, solver)
ILOTRANSITIONTIMEOBJECT4(_this, solver, t1, a1, t2, a2, t3, a3, t4, a4)
ILOTRANSITIONTIMEOBJECT3(_this, solver, t1, a1, t2, a2, t3, a3)
ILOTRANSITIONTIMEOBJECT2(_this, solver, t1, a1, t2, a2)
ILOTRANSITIONTIMEOBJECT1(_this, solver, t1, a1)
```

This macro defines a transition time object, a subclass of `IloTransitionTimeObjectI` named `nameI`. The argument `solver` is the name of the `IloSolver` that performs the extraction. Within the macro, this name denotes the solver currently performing the extraction. When n is greater than 0 (zero), the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type `ti` and a name `ai`. The call to the macro must be followed immediately by the body of the extract member function of the transition time object class being defined. It must return a pointer to an instance of `IlcTransitionTimeObjectI` that corresponds to the extracted object. Besides the definition of the class `nameI`, this macro also defines a function named `name` that creates an instance of the class `nameI` and that returns an instance of the class `IloTransitionTimeObject` that points to it.

The use of this macro is the only way to define a new subclass of `IloTransitionTimeObjectI`.

Since the argument `name` is used to name the transition time object class, it is not possible to use the same name for some other classes.

Example

This example shows how to define a transition time object with two data members. The first one is a constant that corresponds to the transition time between any pair of resource constraints when the transition time is considered. The second one is a Concert Technology boolean variable that states whether or not the transition time is to be considered.

```
ILOTRANSITIONTIMEOBJECT2(MyIloTTObject, mySolver,
                         IloInt, delay,
                         IloNumVar, iloVar) {
  use(mySolver, iloVar);
  IlcIntVar ilcVar = mySolver.getIntVar(iloVar);
  return new (mySolver.getHeap()) MyIlcTTObjectI(mySolver, delay, ilcVar);
}
```

Here is how the corresponding `IlcTransitionTimeObject` could be defined:

```
class MyIlcTTObjectI :public IlcTransitionTimeObjectI {
private:
  IlcInt    _delay;
  IlcIntVar _ilcVar;
public:
  MyIlcTTObjectI(IloSolver solver, IlcInt delay, IlcIntVar ilcVar);
  ~MyIlcTTObjectI(){};
  IlcInt getTransitionTime(const IlcResourceConstraint,
                           const IlcResourceConstraint) const {
    if (_ilcVar.getMin() == 1)
      return _delay;
    return 0;
  }
};

MyIlcTTObjectI::MyIlcTTObjectI(IloSolver solver,
     IlcInt delay, IlcIntVar ilcVar)
  :IlcTransitionTimeObjectI(),
   _delay  (delay),
   _ilcVar (ilcVar)
{};
```

The following statement creates an instance of the class `MyIloTTObjectI` and returns a handle that points to it.

```
IloTransitionTimeObject myTTObj = MyIloTTObject(env, delay, iloVar);
```

This transition time could, for instance, be associated with a resource `res` by creating a transition time as follows:

```
IloTransitionTime(res, myTTObj);
```

For more information, see Transition Times.

See Also: IloTransitionTimeObject, IloTransitionTimeObjectI, IloTransitionTime, IlcTransitionTimeObjectI

Typedef IlcSchedulerTraceFilter

Definition file: ilsched/schedtracei.h
Include file: <ilsched/ilsched.h>

```
IlcBool(* IlcSchedulerTraceFilter)(IlcBool, IlcSchedulerChange, IlcSolverChange)
```

An `IlcSchedulerTraceFilter` allows you to specify which kind of events should be handled by an instance of `IlcSchedulerPrintTrace`. It is a pointer to a user-defined function having three arguments and returning a boolean value.

If a filter is set (see `IlcSchedulerPrintTrace::setFilter`), the instance of `IlcSchedulerPrintTrace` will call this function before handling the event. If the function returns `IlcTrue`, then the event will be handled. Otherwise, it is ignored.

```
typedef IlcBool (*IlcSchedulerTraceFilter)(IlcBool, IlcSchedulerChange, IlcSolverChange);
```

`IlcBool` equals `IlcTrue` if the event is a *beginningevent*; that is, the event has yet to happen but *will* happen. If `IlcBool` is equal to `IlcFalse`, it means that the event is an *ending event*, and that the corresponding modifications have been made.

`IlcSchedulerChange` indicates the kind of event that occurs (for example, modification of the start of an activity or modification of the capacity of a resource constraint).

If the change is related to a Solver variable (such as the start of an activity), then `IlcSolverChange` indicates how this object is modified (for example, the minimum is changed). If there is no such Solver variable, then this argument equals `IlcUndefinedSolverChange`.

Example

In this example, we do not want to have printouts after the events. If the event is related to an activity, we only want to know when the start of the activity increases or becomes bound.

In all other cases, the function returns `IlcFalse`, so that the event will not be handled.

```
IlcBool MyFilter(IlcBool isBeginEvent,
                 IlcSchedulerChange change,
                 IlcSolverChange solverChange) {
  if (!isBeginEvent) {
    return IlcFalse;
  }
  switch (change) {
    case IlcActivityStart:
      return ((solverChange == IlcIntExpSetMin) ||
              (solverChange == IlcIntExpSetValue));
    case IlcActivityEnd:
    case IlcActivityProcessingTime:
    case IlcActivityDuration:
    case IlcActivityDurationOfBreaks:
    case IlcActivityStartOverlap:
    case IlcActivityEndOverlap:
      return IlcFalse;
    default:
      return IlcTrue;
  }
}

int main() {

  ...

  IlcSchedulerPrintTrace trace(scheduler);
  trace.setFilter(MyFilter);

  ...
```

```
}
```

See Also: llcSchedulerPrintTrace, llcSchedulerChange, llcSolverChange