ILOG CP Optimizer: Detailed Scheduling Model and OPL Formulation

IBM ILOG Gentilly
9, rue de Verdun
94253 Gentilly

Date: 2011/09/21 11:05:36
Revision: 1.42

Abstract

ILOG OPL Development Studio and ILOG CP Optimizer introduce a set of modelling features for applications dealing with scheduling over time. This report describes the formal semantics of these modelling elements. Each section is dedicated to a concept: it first formally describes the semantics of the corresponding concept and then presents the concept formulation in OPL together with some model samples. The detailed scheduling model of ILOG CP Optimizer is also available in C++, Java and C#.

Contents

1 Introduction

ILOG OPL Development Studio[1] and ILOG CP Optimizer introduce a set of modelling features for applications dealing with scheduling over time[2]. Although in OPL and CP Optimizer, time points are represented as integers, the possible very wide range of time points means that time is effectively continuous.

A consequence of scheduling over effectively continuous time is that the evolution of some known quantities over time (for instance the instantaneous efficiency/speed of a resource or the earliness/tardiness cost for finishing an activity at a given date t) needs to be compactly represented in the model. To that end, CP Optimizer provides the notion of *piecewise linear* and *stepwise functions* (See §2).

Most of the scheduling applications consist in scheduling in time some activities, tasks or operations that have a start and an end time. In CP Optimizer, this type of decision variable is captured by the notion of *interval variable* (See §3). Several types of constraints are expressed on and between interval variables:

- to limit the possible positions of an interval variable (forbidden start/end or "extent" values) (See §4),
- to specify precedence relations between two interval variables (See §5),
- to relate the position of an interval variable with one of a set of interval variables (spanning, synchronization, alternative) (See §6).

An important characteristic of scheduling problems is that time-intervals may be optional and whether to execute a time-interval or not be a decision variable. In CP Optimizer, this is captured by the notion of boolean *presence* status associated with each interval variable. Logical relations can be expressed between the presence of interval variables, for instance to state that whenever interval a is present then interval b must also be present (See §7). More details about the benefits of a representation based on optional interval variables are available in [3].

Another aspect of scheduling is the allocation of scarce resources to time-intervals. The evolution of a resource over time can be modelled by three types of features:

- The evolution of a disjunctive resource over time can be described by the sequence of intervals that represent the activities executing on the resource. For that, CP Optimizer introduces the notion of an *interval sequence variable.* Constraints and expressions are available to control the sequencing of a set of interval variables (See §9).

[1]A trial version of ILOG OPL Development Studio supporting the complete set of features described in this report is available on `http://www.ilog.com/products/oplstudio/trial.cfm`.

[2]This technical report focuses on the modelling features for scheduling problems. Of course, classical integer variables, constraints and expressions (including numerical expressions) are also available in ILOG OPL Development Studio and ILOG CP Optimizer and they can be mixed with the modelling features presented in this report.

- The evolution of a cumulative resource often needs a description of how the resource accumulated usage evolves over time. For that purpose, CP Optimizer provides the concept of *cumul function expression* that can be used to constrain the evolution of the resource usage over time (See §10).
- The evolution of a resource of infinite capacity, the state of which can vary over time is captured in CP Optimizer by the notion of *state function*. The dynamic evolution of a state function can be controlled thanks to the notion of transition distance and constraints are available for specifying conditions on the state function that must be satisfied during fixed or variable intervals (See §11).

Some classical cost functions in scheduling are earliness/tardiness costs, makespan, activities execution/non-execution costs. CP Optimizer generalizes these classical cost functions and provides a set of basic expressions that can be combined together to express a large spectrum of scheduling cost functions that can be efficiently exploited by the CP Optimizer search (See §8).

Each section of this report is dedicated to a concept. Each section first describes the semantics of the corresponding concept then presents the concept formulation in OPL and concludes with some code samples. The notations used in this document are summarized in appendix A.

Note that, besides OPL, all the modeling concepts presented in this report are also available in C++, in Java and in C#. For more details on the CP Optimizer search algorithm and an analysis of its performance over a set of well established scheduling benchmarks, please refer to [2].

A trial version of ILOG OPL Development Studio is available, see [1].

2 Piecewise Linear and Stepwise Functions

2.1 Semantics

2.1.1 Piecewise Linear Functions

In CP Optimizer, piecewise linear functions are typically used to model a known function of time, for instance the cost incurred for completing an activity after a known date t (See §8).

A **piecewise linear function** $F(t)$ is defined by a tuple $F = \mathit{piecewise}(S, T, t_0, v_0)$ where:

- S is a vector of $n+1$ "change" values $s_i \in \mathbb{R}$ for F
- T is a vector of n values $t_i \in \mathbb{R}$ for F such that
  $\forall i \in [1, n-1],\ t_i \leq t_{i+1} \wedge \forall i \in [1, n-2],\ t_i < t_{i+2}$
- (t_0, v_0) is a reference point of the function

The function is defined everywhere on $(-\infty, +\infty)$ as follows:

- $F(t_0) = v_0$

- the slope $F'(t)$ of F on the segment $(-\infty, t_1)$ is equal to s_1
- the slope $F'(t)$ of F on the segment $[t_n, +\infty)$ is equal to s_{n+1}
- $\forall i \in [1, n-1]$:
  - if $t_i \neq t_{i+1}$, then the slope $F'(t)$ of F on the segment $[t_i, t_{i+1})$ is equal to s_{i+1}. That is, $\forall t \in [t_i, t_{i+1})$, $F(t) = F(t_i) + s_{i+1}(t - t_i)$. Note that the point t_{i+1} is not included in the segment.
  - if $t_i = t_{i+1}$, then F is discontinuous at point t_i. s_{i+1} represents the height of the step (positive or negative) at t_i. The value of $F(t_i)$ is equal to the value of the leftmost extremity of the segment immediately to the right of the discontinuity. Formally, this is given by:
    * if $i \neq n-1$ then $F(t_i) = F(t_{i+2}) + s_{i+2}(t_i - t_{i+2})$
    * else if $i \neq 1$ then $F(t_i) = F(t_{i+1}) = F(t_{i-1}) + s_i(t_i - t_{i-1}) + s_{i+1}$
    * Otherwise, n = 2, function has form slope / step / slope, and i = 1
      · if $t_0 \geq t_i$ then $F(t_i) = v_0 + s_3(t_i - t_0)$
      · else $F(t_i) = v_0 + s_1(t_i - t_0) + s_2$

2.1.2 Stepwise Functions

In CP Optimizer, stepwise functions are typically used to model the efficiency of a resource over time (See §3.1.2).

A **stepwise function** is a special case of piecewise linear function where all slopes are equal to 0 and the domain and image of F are integer. A stepwise function $F(t)$ is defined by a tuple $F = \mathit{stepwise}(V, T)$ where:

- V is a vector of $n+1$ function values $v_i \in \mathbb{Z}$ for F
- T is a vector of n values $t_i \in \mathbb{Z}$ for F such that $\forall i, t_i < t_{i+1}$

The function is defined everywhere on $(-\infty, +\infty)$ and the function value at a point t is given by:

- $\forall t < t_1, F(t) = v_1$
- $\forall i \in [1, n-1], \forall t \in [t_i, t_{i+1}), F(t) = v_{i+1}$
- $\forall t \geq t_n, F(t) = v_{n+1}$

2.2 OPL Formulation

A piecewise linear function $\mathit{piecewise}(S, T, t_0, v_0)$ can be defined as follows:

```
pwlFunction F = piecewise(i in 1..n){ S[i]->T[i]; S[n+1] } (t0, v0);
```

The function can also be defined in an explicit way:

```
pwlFunction F = piecewise{ S[1]->T[1]; ...; S[n]->T[n]; S[n+1] } (t0, v0);
```

A stepwise function *stepwise*(V, T) can be defined as follows:

```
stepFunction F = stepwise(i in 1..n){ V[i]->T[i];  V[n+1] };
```

The function can also be defined in an explicit way:

```
stepFunction F = stepwise{ V[1]->T[1]; ...; V[n]->T[n]; V[n+1] };
```

Arrays of piecewise linear and stepwise functions are supported:

```
pwlFunction  F[i in ...] = piecewise(...){ ... } (...);
stepFunction F[i in ...] = stepwise (...){ ... };
```

2.3 Examples

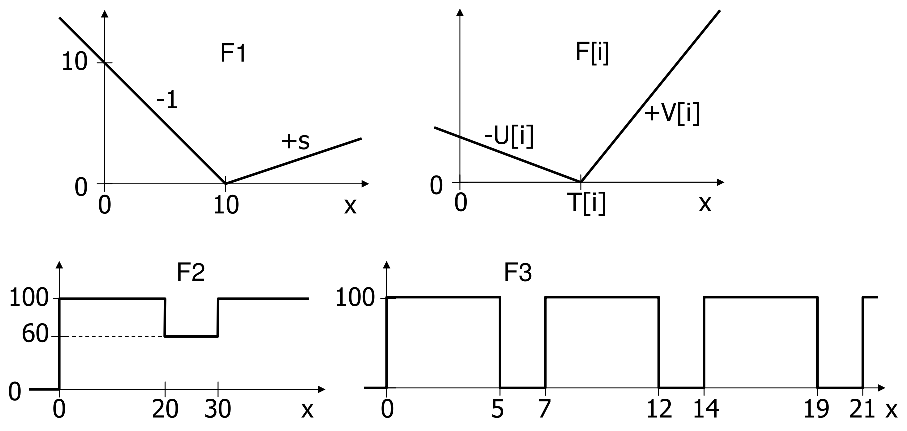

Figure 1: Examples of piecewise linear and stepwise functions

The functions below are depicted on Figure 1.

- A V-shape function with value 0 at $x = 10$, slope -1 before $x = 10$ and slope s after:

  ```
  pwlFunction F1 = piecewise{ -1->10; s } (10, 0);
  ```

- An array of V-shape functions indexed by $i \in [1..n]$ with value 0 at $T[i]$, slope $-U[i]$ before $T[i]$ and slope $V[i]$ after (T, U and V are data integer arrays):

  ```
  pwlFunction F[i in 1..n] = piecewise{ -U[i]->T[i]; V[i] } (T[i],0);
  ```

- A stepwise function with value 0 before 0, 100 on $[0, 20)$, value 60 on $[20, 30)$ and value 100 later on:

  ```
  stepFunction F2 = stepwise{ 0->0; 100->20; 60->30; 100 };
  ```

- A stepwise function with value 0 everywhere except on intervals $[7i, 7i+5)$ for $i \in [0, 51]$ where the value is 100:

  ```
  stepFunction F3 = stepwise(i in 0..51, p in 0..1) {
     100*p -> (7*i)+(5*p) ; 0 };
  ```

3 Interval Variables

3.1 Semantics

3.1.1 Basic Interval Variable

Informally speaking, an interval variable represents an interval of time during which something happens (a task, an activity is carried out) and whose position in time is an unknown of the scheduling problem. An interval is characterized by a start value, an end value and a size. An important additional feature of interval variables is the fact that they can be optional that is, one can decide not to consider them in the solution schedule. This concept is crucial in applications that present at least some of the following features:

- optional activities (operations, tasks) that can be left unperformed (with an impact on the cost) : typical examples are externalized, maintenance or control tasks,
- activities that can execute on a set of alternative resources (machines, manpower) with possibly different characteristics (speed, calendar) and compatibility constraints,
- operations that can be processed in different temporal modes (for instance in series or in parallel),
- alternative modes for executing a given activity, each mode specifying a particular combination of resources,
- alternative processes for executing a given production order, a process being specified as a sequence of operations requiring resources,
- hierarchical description of a project as a work-breakdown structure with tasks decomposed into sub-tasks, part of the project being optional (with an impact on the cost if unperformed), *etc.*

More formally, an **interval variable** a is a variable whose domain $dom(a)$ is a subset of $\{\perp\} \cup \{[s, e) | s, e \in \mathbb{Z}, s \leq e\}$. An interval variable is said to be **fixed** if its domain is reduced to a singleton, i.e., if $\underline{a}$ denotes a fixed interval variable:

- interval is **absent**: $\underline{a} = \perp$; or
- interval is **present**: $\underline{a} = [s, e)$

Absent interval variables have special meaning. Informally speaking, an absent interval variable is not considered by any constraint or expression on interval variables it is involved in. For example, if an absent interval variable is used in a `noOverlap` constraint, the constraint will behave as if the interval was never specified to the constraint. If an absent interval variable a is used in a precedence constraint between interval variables a and b this constraint does not impact interval variable b. Each constraint specifies how it handles absent interval variables.

In this document, the semantics of constraints defined over interval variables is described by the properties that fixed intervals must have in order for the constraint to be true. If a fixed interval $\underline{a}$ is present and such that $\underline{a} = [s, e)$, we will denote $s(\underline{a})$ its integer start value s, $e(\underline{a})$ its integer end value e and $l(\underline{a})$ its positive integer length defined as $e(\underline{a}) - s(\underline{a})$. The presence status $x(\underline{a})$ will be equal to 1. For a fixed interval that is absent, $x(\underline{a}) = 0$ and the start, end and length are undefined.

Until a solution is found it may not be known whether an interval will be present or not. In this case we say that the interval is optional. To be precise, an interval is said to be absent when $dom(a) = \{\perp\}$, present when $\perp \notin dom(a)$ and optional in all other cases.

3.1.2 Intensity and Size

Sometimes the intensity of "work" is not the same during the whole interval. For example let's consider a worker who does not work during weekends (his work intensity during weekends is 0%) and on Friday he works only for half a day (his intensity during Friday is 50%). For this worker, 7 man-days work will span for longer than just 7 days. In this example 7 man-days represent what we call the *size* of the interval: that is, the length of the interval would be if the intensity function was always at 100%.

To model such situations, you can specify a range for the **size** of an interval variable and an integer **stepwise intensity function** F (see §2). For a fixed present interval $\underline{a}$ the following relation will be enforced at any solution between the start, end, size sz of the interval and the integer granularity G (by default, the intensity function is expressed as a percentage so the granularity G is 100):

$$sz(\underline{a}) \leq \int_{s(\underline{a})}^{e(\underline{a})} \frac{F(t)}{G} \mathrm{d}t < sz(\underline{a}) + 1$$

That is, the length of the interval will be at least long enough to cover the work requirements given by the interval size, taking into account the intensity function. However, any over-estimation is always strictly less than one work unit.

If no intensity is specified, it is supposed to be the constant full intensity function ($\forall t, F(t) = 100\%$) so in that case $sz(a) = l(a)$. Note that the size is not defined for absent intervals.

Important: The intensity step function F is a stepwise function with integer values and is not allowed to exceed the granularity (100 by default).

Figure 2 depicts an interval variable of size 14 and an associated intensity function. A valid solution is represented where the interval starts at 10 and ends at 27. In this case,

$$\int_{s(\underline{a})}^{e(\underline{a})} \frac{F(t)}{G} \mathrm{d}t = \frac{1420}{100} = 14.2$$

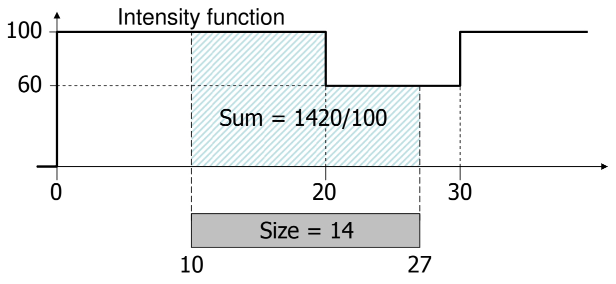

Figure 2: Example of interval intensity

3.2 OPL Formulation

```
dvar interval a [optional[(IsOptional)]]
                [in StartMin..EndMax]
                [size SZ | in SZMin .. SZMax]
                [intensity F]
```

Where

```
int IsOptional, StartMin, EndMax, SZ, SZMin, SZMax;
stepFunction F;
1-(maxint div 2) <= StartMin <= (maxint div 2)-1
1-(maxint div 2) <= EndMax   <= (maxint div 2)-1
               0 <= SZ       <= (maxint div 2)-1
               0 <= SZMin    <= (maxint div 2)-1
               0 <= SZMax    <= (maxint div 2)-1
```

Typically, the problem structure will indicate if an interval is to be optional or not, and in this case the keyword `optional` is used (or not) in the definition of the interval variable. In the case where the optionality depends on input data, you can specify a boolean parameter to the optionality field: `optional(true)` being equivalent to `optional` and `optional(false)` being equivalent to the omission of `optional`.

A window $[StartMin, EndMax]$ can be specified to restrict the position of the interval variable. By default, an interval variable will start after 0 and end before $(maxint\ div\ 2) - 1$. The fixed size or the size range for the interval is specified with the `size` keyword. Note that these bounds are taken into account only when the interval variable is present in the final schedule, that is, they allow specifying conditional bounds on the interval variable *would the interval be present in the final schedule.* For absent intervals, they are just ignored.

3.3 Examples

- An interval variable with unconstrained size (in $[0, \text{(intmax div 2)-1})$):

```
dvar interval a;
```

- An interval variable of size 10:

```
dvar interval a size 10;
```

- An optional interval variable with size 10 that, when present must start at value 20 or later and must end before value 1000:

```
dvar interval a optional in 20..1000 size 10;
```

- An optional interval variable whose size 14 is modulated by an intensity function `F`:

```
dvar interval a optional size 14 intensity F;
```

- An array of `n` optional interval variables of size `PT[i]`:

```
dvar interval A[i in 1..n] optional size PT[i];
```

4 Unary Constraints on Interval Variables

4.1 Semantics

It may be necessary to state that an interval cannot start, cannot end or cannot overlap a set of fixed dates. CP Optimizer provides the following constraints for modelling it. Let $\underline{a}$ denote a fixed interval and F an integer stepwise function.

- **Forbidden start constraint**. The constraint $forbidStart(\underline{a}, F)$, states that whenever the interval is present, it cannot start at a value t where $F(t) = 0$.
- **Forbidden end constraint**. The constraint $forbidEnd(\underline{a}, F)$, states that whenever the interval is present, it cannot end at a value t where $F(t-1) = 0$.
- **Forbidden extent constraint**. The constraint $forbidExtent(\underline{a}, F)$, states that whenever the interval is present, it cannot overlap a point t where $F(t) = 0$.

More formally:

$$\begin{aligned}
forbidStart(\underline{a}, F) &\Leftrightarrow \big(\underline{a} = \perp\big) \vee \big(F(s(\underline{a})) \neq 0\big) \\
forbidEnd(\underline{a}, F) &\Leftrightarrow \big(\underline{a} = \perp\big) \vee \big(F(e(\underline{a}) - 1) \neq 0\big) \\
forbidExtent(\underline{a}, F) &\Leftrightarrow \big(\underline{a} = \perp\big) \vee \big(\forall t \in [s(\underline{a}), e(\underline{a})), F(t) \neq 0\big)
\end{aligned}$$

4.2 OPL Formulation

The constructs below are constraints:

```
forbidStart (a, F);
forbidEnd   (a, F);
forbidExtent(a, F);
```

Where:

```
dvar interval a;
stepFunction F;
```

Important: Constraints `forbidStart`, `forbidEnd` and `forbidExtent` cannot be used in meta-constraints.

4.3 Examples

The constraint below will prevent interval variable a to start in any interval $[7i, 7i + 5)$ for $i \in [0, 51]$. Typically, it represents an activity that cannot start during week-ends:

```
stepFunction F = stepwise(i in 0..51, p in 0..1) {
     100*p -> (7*i)+(5*p) ; 0 };
dvar interval a size 14;

constraints {
   forbidStart(a, F);
}
```

5 Precedence Constraints between Interval Variables

5.1 Semantics

This section describes common constraints in scheduling, namely, **precedence constraints**. Informally saying, these constraints restrict the relative position of interval variables in a solution. For instance a precedence constraint can model the fact that an activity `a` must end before activity `b` starts (optionally with some minimum delay `z`). If one or both of the interval variables of the precedence constraint is absent, then the precedence is systematically considered to be true and thus, it does not impact the schedule.

More formally, the semantics of the relation $TC(\underline{a}, \underline{b}, z)$ on a pair of fixed intervals $\underline{a}$, $\underline{b}$ and for a value z depending on the constraint type TC is given on Table 1.

Relation	**Semantics**
$startBeforeStart$	$x(\underline{a}) \wedge x(\underline{b}) \Rightarrow s(\underline{a}) + z \leq s(\underline{b})$
$startBeforeEnd$	$x(\underline{a}) \wedge x(\underline{b}) \Rightarrow s(\underline{a}) + z \leq e(\underline{b})$
$endBeforeStart$	$x(\underline{a}) \wedge x(\underline{b}) \Rightarrow e(\underline{a}) + z \leq s(\underline{b})$
$endBeforeEnd$	$x(\underline{a}) \wedge x(\underline{b}) \Rightarrow e(\underline{a}) + z \leq e(\underline{b})$
$startAtStart$	$x(\underline{a}) \wedge x(\underline{b}) \Rightarrow s(\underline{a}) + z = s(\underline{b})$
$startAtEnd$	$x(\underline{a}) \wedge x(\underline{b}) \Rightarrow s(\underline{a}) + z = e(\underline{b})$
$endAtStart$	$x(\underline{a}) \wedge x(\underline{b}) \Rightarrow e(\underline{a}) + z = s(\underline{b})$
$endAtEnd$	$x(\underline{a}) \wedge x(\underline{b}) \Rightarrow e(\underline{a}) + z = e(\underline{b})$

Table 1: Precedence relations semantics

5.2 OPL Formulation

The constructs below are constraints:

```
startBeforeStart (a,b[,z]);
startBeforeEnd   (a,b[,z]);
endBeforeStart   (a,b[,z]);
endBeforeEnd     (a,b[,z]);
startAtStart     (a,b[,z]);
startAtEnd       (a,b[,z]);
endAtStart       (a,b[,z]);
endAtEnd         (a,b[,z]);
```

Where:

```
dexpr int z;
dvar interval a;
dvar interval b;
```

Important: In the current version of CP Optimizer, the above constraints cannot be used in meta-constraints.

5.3 Examples

- For the most common case of a precedence relation between the end of an interval `a` and the start of an interval `b`, one simply writes:

  ```
  endBeforeStart(a,b);
  ```

- Interval variable `d` cannot start sooner than 5 units after start of interval variable `c`:

  ```
  startBeforeStart(c,d,5);
  ```

6 Constraints on Groups of Interval Variables

6.1 Semantics

This section describes three constraints over groups of intervals. Their main purpose is to allow hierarchical creation of the model by "encapsulating" a group of interval variables by one "high level" interval. Here is an informal definition of these constraints:

- **Span constraint**. The constraint $\mathit{span}(a, \{b_1, .., b_n\})$ states that the interval a spans over all present intervals from the set $\{b_1, .., b_n\}$. That is: interval a starts together with the first present interval from $\{b_1, .., b_n\}$ and ends together with the last one.
- **Alternative constraint**. The constraint $\mathit{alternative}(a, \{b_1, .., b_n\})$ models an exclusive alternative between $\{b_1, .., b_n\}$. If interval a is present then exactly one of intervals $\{b_1, .., b_n\}$ is present and a starts and ends together with this chosen one. The alternative constraint can also be specified a non-negative integer cardinality c, $\mathit{alternative}(a, \{b_1, .., b_n\}, c)$, in this case, it is not 1 but c interval variables that will be selected among the set $\{b_1, .., b_n\}$ and those c selected intervals will have to start and end together with interval variable a.
- **Synchronize constraint**. The constraint $\mathit{synchronize}(a, \{b_1, .., b_n\})$ makes intervals $b_1 \dots b_n$ start and end together with interval a (if a is present).

More formally, let $\underline{a}, \underline{b_1}, ..., \underline{b_i}, ..., \underline{b_n}$ be a set of fixed intervals and $\underline{c}$ a fixed integer.
The **span constraint** $\mathit{span}(\underline{a}, \{\underline{b_1}, ..., \underline{b_i}, ..., \underline{b_n}\})$ holds if and only if:

$$\begin{aligned} \neg x(\underline{a}) \quad &\Leftrightarrow \quad \forall i \in [1, n], \neg x(\underline{b_i}) \\ x(\underline{a}) \quad &\Leftrightarrow \quad \begin{cases} \exists i \in [1, n], x(\underline{b_i}) \\ s(\underline{a}) = \min_{i \in [1,n], x(\underline{b_i})} s(\underline{b_i}) \\ e(\underline{a}) = \max_{i \in [1,n], x(\underline{b_i})} e(\underline{b_i}) \end{cases} \end{aligned}$$

The **alternative interval constraint** $\mathit{alternative}(\underline{a}, \{\underline{b_1}, ..., \underline{b_i}, ..., \underline{b_n}\}, \underline{c})$ holds if and only if:

$$\begin{aligned} \neg x(\underline{a}) \quad &\Leftrightarrow \quad \forall i \in [1, n], \neg x(\underline{b_i}) \\ x(\underline{a}) \quad &\Leftrightarrow \quad \exists K \subset [1, n] \begin{cases} K \neq \emptyset \wedge |K| = \underline{c} \\ \forall k \in K, x(\underline{b_k}) \wedge \big(s(\underline{a}) = s(\underline{b_k})\big) \wedge \big(e(\underline{a}) = e(\underline{b_k})\big) \\ \forall j \in [1, n] \setminus K, \neg x(\underline{b_j}) \end{cases} \end{aligned}$$

When parameter c is omitted, we assume $c = 1$. Note that when $c \leq 0$, as there cannot be any subset $K \subset [1, n]$ such that $K \neq \emptyset$ and $|K| = c$, it means that necessarily, interval a is absent. And the contrapositive: if interval a is present, then necessarily, for the constraint to be satisfied it must be that $c \geq 1$.

The **synchronization constraint** $\mathit{synchronize}(\underline{a}, \{\underline{b_1}, ..., \underline{b_i}, ..., \underline{b_n}\})$ holds if and only if:

$$x(\underline{a}) \Rightarrow \forall i \in [1, n] \,|\, x(\underline{b_i}), \big(s(\underline{a}) = s(\underline{b_i})\big) \wedge \big(e(\underline{a}) = e(\underline{b_i})\big)$$

6.2 OPL Formulation

The constructs below are constraints:

```
span(a, B);
alternative(a, B[, c]);
synchronize(a, B);
```

Where

```
dvar interval a;
dvar interval B[];
dexpr int c;
```

Important: Constraints `span`, `alternative` and `synchronize` cannot be used in meta-constraints.

6.3 Examples

- The code below defines two interval variables `a1` and `a2` that represent the execution of two tasks on some processors. There are two available processors `p1` and `p2`. The possibility to execute task `ai` on processor `pj` is represented by an optional interval `aip[j]`. The duration of the tasks depends on the processor they execute on as defined by a table `p[i,j]`. Task `a1` must be performed before `a2` and the result of `a1` is used as input to `a2`. The communication delay between two processors `j` and `k` are given by a table `d[j,k]`.

  ```
  dvar interval a1;
  dvar interval a1p[i in 1..2] optional size p[1][i];
  dvar interval a2;
  dvar interval a2p[i in 1..2] optional size p[2][i];

  subject to {
    alternative(a1, a1p);
    alternative(a2, a2p);
    forall (j in 1..2, k in 1..2)
      endBeforeStart(a1p[j],a2p[k],d[j][k]);
  }
  ```

- Note that a given interval can be at the same time member of an alternative (playing the role of a b_i) and master of another alternative constraint (playing the role of a). This modelling pattern is even very useful in some applications as illustrated by the code sample below. Suppose n activities A_i to be performed. An activity A_i must be performed by a worker (to be selected among a set of w alternative workers) and at a position (to be selected among a set of p alternative positions). Some constraints depend on the triplet activity/worker/position (in the code sample, this is just a simple case where the duration of the activity depend on the worker and on the position as specified in a matrix D). Some constraints hold on the workers (e.g. a worker can do only one activity at a time) whereas other constraints hold on the positions (e.g. no more than Q activities executing simultaneously at a given position). In the sample, interval $P[i][j]$ represents the execution of activity $A[i]$ at position j whereas interval $W[i][k]$ represents the execution of activity $A[i]$ by worker k.

```
dvar interval A[i in 1..n];
dvar interval X[i in 1..n][j in 1..p][k in 1..w] optional size D[i][j][k];
dvar interval P[1..n][1..p] optional;
dvar interval W[1..n][1..w] optional;
constraints {
  // Two-level alternatives
  forall(i in 1..n) {
    alternative(A[i], all(j in 1..p) P[i][j]);
    alternative(A[i], all(k in 1..w) W[i][k]);
    forall(j in 1..p)
      alternative(P[i][j], all(k in 1..w) X[i][j][k]);
    forall(k in 1..w)
      alternative(W[i][k], all(j in 1..p) X[i][j][k]);
  }
  // Constraints on locations (see section 10)
  forall(j in 1..p)
    sum(i in 1..n) pulse(P[i][j],1) <= Q;
  // Constraints on workers (see section 9)
  forall(k in 1..w)
    noOverlap(all (i in 1..n) W[i][k]);
}
```

- The code below defines a task `task` of size 2 that requires 3 workers among a candidate set of `m=10` workers. Each of the 3 workers allocated to the task will be used from the start to the end time of the task. Each worker `i` is available only during a specified time window. A solution to the problem consists in finding a time interval of size 2 that is contained in at least 3 worker time windows, for instance the interval [13,15].

```
int m=10;
tuple Window { int start; int end; }
Window w[1..m] = [<0,9>,<10,22>,<11,23>,<15,17>,<3,8>,<23,25>,<13,15>,
                  <22,23>,<18,20>,<18,20>];
```

```
dvar interval task size 2;
dvar interval taskWorker[i in 1..m] optional in w[i].start..w[i].end;

constraints {
  alternative(task, taskWorker, 3);
}
```

7 Logical Constraints between Interval Variables

7.1 Semantics

Presence status of interval variables can be further restricted by logical constraints. The **presence constraint** *presenceOf*(a) states that a given interval variable must be present. Of course, this constraint may be used in meta-constraints, for example there may be two optional intervals a and b, but if interval a is present then b must be present too. This can be modelled by the constraint *presenceOf*(a) $\Rightarrow$ *presenceOf*(b).

The semantics of the presence constraint on a fixed interval $\underline{a}$ is just:

$$presenceOf(\underline{a}) \Leftrightarrow x(\underline{a})$$

7.2 OPL Formulation

The following constraint can be expressed on an interval, it states that the interval variable must be present. Such a constraint can be used in meta-constraints, and its truth value can be used in an arithmetical expression.

```
presenceOf(a);
```

where

```
dvar interval a;
```

7.3 Examples

The code below defines a chain of optional activities `a1` ... `an` such that whenever an activity `ai` is abandoned (absent), all the remaining activities in the chain are also absent.

```
dvar interval a[i in 1..n] optional size d[i];

subject to {
  forall (i in 1..n-1) {
    endBeforeStart(a[i],a[i+1]);
    presenceOf(a[i+1]) => presenceOf(a[i]);
  }
}
```

8 Expressions over Interval Variables

8.1 Semantics

This section shows how to create numerical expressions from interval variables. These expressions can be used to:

- Define a term for the cost function,
- Connect interval variables to integer and floating point experessions.

Integer expressions *startOf*, *endOf*, *lengthOf*, *sizeOf* provide an access to the different attributes of an interval variable. However special care must be taken for optional intervals: in this case an integer value $absVal$ must be specified which represents the value of the expression when the interval is absent. If this value is omitted, it is supposed to be 0.

Numerical expressions (*startEval*, *endEval*, *lengthEval*, *sizeEval*) allow to evaluate a piecewise linear function (See §2.1.1) on a given bound of an interval. Like for the above expressions, a numerical value $absVal$ can be specified that represents the value of the expression when the interval is absent. If this value is omitted, it is supposed to be 0.

Let $\underline{a}$ denote a fixed interval variable. The semantics of these expressions is given on Table 2.

Expression	**Semantics**
$\mathit{startOf}(\underline{a}, \mathit{absVal})$	$\mathit{expr} = \begin{cases} s(\underline{a}) & \text{if } x(\underline{a}) \\ \mathit{absVal} & \text{otherwise} \end{cases}$
$\mathit{endOf}(\underline{a}, \mathit{absVal})$	$\mathit{expr} = \begin{cases} e(\underline{a}) & \text{if } x(\underline{a}) \\ \mathit{absVal} & \text{otherwise} \end{cases}$
$\mathit{lengthOf}(\underline{a}, \mathit{absVal})$	$\mathit{expr} = \begin{cases} l(\underline{a}) & \text{if } x(\underline{a}) \\ \mathit{absVal} & \text{otherwise} \end{cases}$
$\mathit{sizeOf}(\underline{a}, \mathit{absVal})$	$\mathit{expr} = \begin{cases} sz(\underline{a}) & \text{if } x(\underline{a}) \\ \mathit{absVal} & \text{otherwise} \end{cases}$
$\mathit{startEval}(\underline{a}, F, \mathit{absVal})$	$\mathit{expr} = \begin{cases} F(s(\underline{a})) & \text{if } x(\underline{a}) \\ \mathit{absVal} & \text{otherwise} \end{cases}$
$\mathit{endEval}(\underline{a}, F, \mathit{absVal})$	$\mathit{expr} = \begin{cases} F(e(\underline{a})) & \text{if } x(\underline{a}) \\ \mathit{absVal} & \text{otherwise} \end{cases}$
$\mathit{lengthEval}(\underline{a}, F, \mathit{absVal})$	$\mathit{expr} = \begin{cases} F(l(\underline{a})) & \text{if } x(\underline{a}) \\ \mathit{absVal} & \text{otherwise} \end{cases}$
$\mathit{sizeEval}(\underline{a}, F, \mathit{absVal})$	$\mathit{expr} = \begin{cases} F(sz(\underline{a})) & \text{if } x(\underline{a}) \\ \mathit{absVal} & \text{otherwise} \end{cases}$

Table 2: Expression semantics

8.2 OPL Formulation

The constructs below are integer expressions that can be used to link interval and integer variables. The value of the expression will be `absVal` if `a` is absent. If `absVal` is omitted, it is assumed to be zero.

```
dexpr int startOf  (a[,absVal]);
dexpr int endOf    (a[,absVal]);
dexpr int lengthOf (a[,absVal]);
dexpr int sizeOf   (a[,absVal]);
```

Where:

```
dvar interval a;
int absVal;
```

The constructs below are numerical expressions:

```
dexpr float startEval  (a,f[,absVal]);
dexpr float endEval    (a,f[,absVal]);
dexpr float lengthEval (a,f[,absVal]);
dexpr float sizeEval   (a,f[,absVal]);
```

Where:

```
pwlFunction f;
dvar interval a;
float absVal;
```

8.3 Examples

- Here is an illustration of the usage of a functional expression for modelling a V-shape earliness/tardiness cost:

  ```
  pwlFunction ETCost = piecewise { -1->10; 1 } (10, 0);
  dvar interval a[i in 1..n] size d[i];

  minimize sum(i in 1..n) endEval(a[i], ETCost);
  subject to {
    ...
  }
  ```

- Here is a similar example where each activity $a[i]$ specifies its individual prefered end date `dd[i]`:

```
dvar interval a[i in 1..n] size d[i];

minimize sum(i in 1..n) endEval(a[i], piecewise{-1->dd[i];1}(dd[i],0));
subject to {
  ...
}
```

9 Interval Variables Sequencing

9.1 Semantics

9.1.1 Interval Sequence Variables

An *interval sequence variable* is defined on a set of interval variables A. Informally speaking, the value of an interval sequence variable represents a total ordering of the interval variables of A. Note that any absent interval variables are not considered in the ordering.

More formally, an **interval sequence variable** p on a set of interval variables A represents a decision variable whose possible values are all the permutations of the intervals of A. Let $\underline{A}$ be a set of fixed intervals and n denote the cardinality of $\underline{A}$. A permutation π of $\underline{A}$ is a function $\pi : \underline{A} \rightarrow \{\perp\} \cup [1, n]$ such that if we denote $\mathit{length}(\pi) = |\{\underline{a} \in \underline{A}, x(\underline{a})\}|$ the number of present intervals:

1. $\forall \underline{a} \in \underline{A}, x(\underline{a}) \Leftrightarrow (\pi(\underline{a}) \neq \perp)$
2. $\forall \underline{a} \in \underline{A}, x(\underline{a}) \Rightarrow \big(\pi(\underline{a}) \leq \mathit{length}(\pi)\big)$
3. $\forall \underline{a}, \underline{b} \in \underline{A}, \big(x(\underline{a}) \wedge x(\underline{b}) \wedge \underline{a} \neq \underline{b}\big) \Rightarrow \big(\pi(\underline{a}) \neq \pi(\underline{b})\big)$

Note that the sequence alone does not enforce any constraint on the relative position of intervals end-points. For instance, an interval variable a could be sequenced before an interval variable b in a sequence p without any impact on the relative position between the start/end points of a and b (a could still be fixed to start after the end of b). This is because different semantics can be used to define how a sequence constrains the positions of intervals. We will see later how the *noOverlap* constraint implements one of these possible semantics.

The following constraints are available on sequence variables:

- *first*(p, a) states that if interval a is present then, it will be the first interval of the sequence p.
- *last*(p, a) states that if interval a is present then, it will be the last interval of the sequence p.
- *before*(p, a, b) states that if both intervals a and b are present then a will appear before b in the sequence p.

Relation	Semantics
$\mathit{first}(\pi,\underline{a})$	$x(\underline{a}) \Rightarrow \big(\pi(\underline{a}) = 1\big)$
$\mathit{last}(\pi,\underline{a})$	$x(\underline{a}) \Rightarrow \big(\pi(\underline{a}) = \mathit{length}(s)\big)$
$\mathit{before}(\pi,\underline{a},\underline{b})$	$\big(x(\underline{a}) \wedge x(\underline{b})\big) \Rightarrow \big(\pi(\underline{a}) < \pi(\underline{b})\big)$
$\mathit{prev}(\pi,\underline{a},\underline{b})$	$\big(x(\underline{a}) \wedge x(\underline{b})\big) \Rightarrow \big(\pi(\underline{b}) = \pi(\underline{a}) + 1\big)$

Table 3: Sequence relation semantics

- $\mathit{prev}(p, a, b)$ states that if both intervals a and b are present then a will be just before b in the sequence p, that is, it will appear before b and no other interval will be sequenced between a and b in the sequence.

The formal semantics of these basic constraints are given on Table 3.

The sequence variable also allows associating a fixed non-negative integer type with each interval variable in the sequence. In particular, these integers are used by the *noOverlap* constraint (see §9.1.2) and the *typeOfNext/Prev* integer expressions (see §9.1.3). $T(p, a)$ denotes the fixed non-negative integer type of interval variable a in the sequence variable p.

9.1.2 No-overlap Constraint

The **noOverlap** constraint on an interval sequence variable p states that the sequence defines a chain of non-overlapping intervals, any interval in the chain being constrained to end before the start of the next interval in the chain. This constraint is typically useful for modelling disjunctive resources.

More formally, the condition for a permutation value π for sequence p to satisfy the *noOverlap* constraints is defined as:

$$\mathit{noOverlap}(\pi) \;\Leftrightarrow\; \forall \underline{a}, \underline{b} \in \underline{A}, \neg x(\underline{a}) \vee \neg x(\underline{b}) \vee \Big(\big(\pi(\underline{a}) < \pi(\underline{b})\big) \Rightarrow \big(e(\underline{a}) \le s(\underline{b})\big)\Big)$$

A sequence variable together with a no-overlap constraint using it are illustrated on Figure 3.

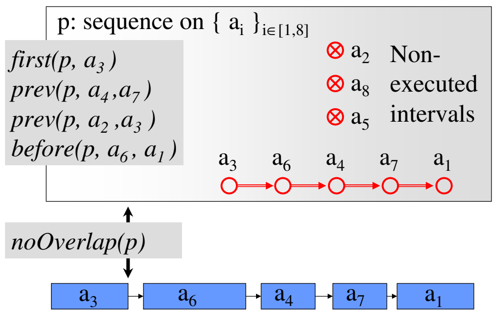

Figure 3: Example of sequence variable and no-overlap constraint

If a transition distance matrix `M` is specified, it defines the minimal non-negative distance that must separate consecutive intervals in the sequence. Two versions of the constraint are available: one that enforces the transition distance between each interval and its next interval in the sequence (*Next*) and the other that enforces the transition distance between each interval and all its successors in the sequence (*After*).

More formally, if `T(p,a)` denotes the non-negative integer type of interval a in the sequence variable p:

$$\mathit{noOverlap}(\pi, M, \mathit{Next}) \Leftrightarrow \forall \underline{a}, \underline{b} \in \underline{A}, \neg x(\underline{a}) \vee \neg x(\underline{b}) \vee \\ \big(\big(\pi(\underline{b}) = \pi(\underline{a}) + 1\big) \Rightarrow \big(e(\underline{a}) + M[T(\pi, \underline{a}), T(\pi, \underline{b})] \leq s(\underline{b})\big)\big)$$

$$\mathit{noOverlap}(\pi, M, \mathit{After}) \Leftrightarrow \forall \underline{a}, \underline{b} \in \underline{A}, \neg x(\underline{a}) \vee \neg x(\underline{b}) \vee \\ \big(\big(\pi(\underline{a}) < \pi(\underline{b})\big) \Rightarrow \big(e(\underline{a}) + M[T(\pi, \underline{a}), T(\pi, \underline{b})] \leq s(\underline{b})\big)\big)$$

Note that if the transition matrix `M` satisfies the triangular inequality, the semantics of the two versions of the constraint $\mathit{noOverlap}(\pi, M, \mathit{Next})$ and $\mathit{noOverlap}(\pi, M, \mathit{After})$ is the same. If `M` does not satisfy the triangular inequality, constraint $\mathit{noOverlap}(\pi, M, \mathit{After})$ is stronger than $\mathit{noOverlap}(\pi, M, \mathit{Next})$.

9.1.3 Integer expressions for getting the type of next/prev interval in a sequence

Two integer expressions are available to get the type of the interval that is next (resp. previous) to a given interval variable a in a sequence p. When interval a is absent or is the last (resp. first) interval of the sequence, specific values can be provided for these integer expressions.

More formally, let π the permutation value of a fixed sequence variable p and $\underline{a}$ the value of a fixed interval variable a in the fixed sequence variable p.

If $\underline{a}$ is present and is not the last interval in π, we will denote $N(\pi, \underline{a})$ the unique interval that is next to $\underline{\text{a}}$ in π.

$$\text{if } x(\underline{a}) \wedge \big(\pi(\underline{a}) < \mathit{length}(\pi)\big) : N(\pi, \underline{a}) = \underline{b} \text{ such that } \pi(\underline{b}) = \pi(\underline{a}) + 1$$

$$\mathit{typeOfNext}(\pi, \underline{a}, \mathit{nVal}, \mathit{absVal}) = \begin{cases} \mathit{absVal} & \text{if } \neg x(\underline{a}) \\ \mathit{nVal} & \text{if } x(\underline{a}) \wedge \big(\pi(\underline{a}) = \mathit{length}(\pi)\big) \\ T(\pi, N(\pi, \underline{a})) & \text{otherwise} \end{cases}$$

If $\underline{a}$ is present and is not the first interval in π, we will denote $P(\pi, \underline{a})$ the unique interval that is previous to $\underline{\text{a}}$ in π.

$$\text{if } x(\underline{a}) \wedge \big(\pi(\underline{a}) > 1\big) : P(\pi, \underline{a}) = \underline{b} \text{ such that } \pi(\underline{a}) = \pi(\underline{b}) + 1$$

$$\mathit{typeOfPrev}(\pi, \underline{a}, \mathit{pVal}, \mathit{absVal}) = \begin{cases} \mathit{absVal} & \text{if } \neg x(\underline{a}) \\ \mathit{pVal} & \text{if } x(\underline{a}) \wedge \big(\pi(\underline{a}) = 1\big) \\ T(\pi, P(\pi, \underline{a})) & \text{otherwise} \end{cases}$$

9.1.4 Integer expressions for getting the bounds of next/previous interval in a sequence

Integer expressions *startOfNext*, *endOfNext*, *lengthOfNext*, *sizeOfNext*, *startOfPrev*, *endOfPrev*, *lengthOfPrev*, *sizeOfPrev* provide an access to the different attributes of the interval that is next (resp. previous) to a given interval variable a in a sequence p. When interval a is absent or is the last (resp. first) interval of the sequence, specific values can be provided for these integer expressions.

The semantics of these expressions is given on Table 4, using the notations introduced in section §9.1.3.

Expression	**Semantics**
$\mathit{startOfNext}(\pi, \underline{a}, \mathit{nVal}, \mathit{absVal})$	$\mathit{expr} = \begin{cases} \mathit{absVal} & \text{if } \neg x(\underline{a}) \\ \mathit{nVal} & \text{if } x(\underline{a}) \wedge \big(\pi(\underline{a}) = \mathit{length}(\pi)\big) \\ s(N(\pi, \underline{a})) & \text{otherwise} \end{cases}$
$\mathit{endOfNext}(\pi, \underline{a}, \mathit{nVal}, \mathit{absVal})$	$\mathit{expr} = \begin{cases} \mathit{absVal} & \text{if } \neg x(\underline{a}) \\ \mathit{nVal} & \text{if } x(\underline{a}) \wedge \big(\pi(\underline{a}) = \mathit{length}(\pi)\big) \\ e(N(\pi, \underline{a})) & \text{otherwise} \end{cases}$
$\mathit{lengthOfNext}(\pi, \underline{a}, \mathit{nVal}, \mathit{absVal})$	$\mathit{expr} = \begin{cases} \mathit{absVal} & \text{if } \neg x(\underline{a}) \\ \mathit{nVal} & \text{if } x(\underline{a}) \wedge \big(\pi(\underline{a}) = \mathit{length}(\pi)\big) \\ l(N(\pi, \underline{a})) & \text{otherwise} \end{cases}$
$\mathit{sizeOfNext}(\pi, \underline{a}, \mathit{nVal}, \mathit{absVal})$	$\mathit{expr} = \begin{cases} \mathit{absVal} & \text{if } \neg x(\underline{a}) \\ \mathit{nVal} & \text{if } x(\underline{a}) \wedge \big(\pi(\underline{a}) = \mathit{length}(\pi)\big) \\ sz(N(\pi, \underline{a})) & \text{otherwise} \end{cases}$
$\mathit{startOfPrev}(\pi, \underline{a}, \mathit{pVal}, \mathit{absVal})$	$\mathit{expr} = \begin{cases} \mathit{absVal} & \text{if } \neg x(\underline{a}) \\ \mathit{pVal} & \text{if } x(\underline{a}) \wedge \big(\pi(\underline{a}) = 1\big) \\ s(P(\pi, \underline{a})) & \text{otherwise} \end{cases}$
$\mathit{endOfPrev}(\pi, \underline{a}, \mathit{pVal}, \mathit{absVal})$	$\mathit{expr} = \begin{cases} \mathit{absVal} & \text{if } \neg x(\underline{a}) \\ \mathit{pVal} & \text{if } x(\underline{a}) \wedge \big(\pi(\underline{a}) = 1\big) \\ e(P(\pi, \underline{a})) & \text{otherwise} \end{cases}$
$\mathit{lengthOfPrev}(\pi, \underline{a}, \mathit{pVal}, \mathit{absVal})$	$\mathit{expr} = \begin{cases} \mathit{absVal} & \text{if } \neg x(\underline{a}) \\ \mathit{pVal} & \text{if } x(\underline{a}) \wedge \big(\pi(\underline{a}) = 1\big) \\ l(P(\pi, \underline{a})) & \text{otherwise} \end{cases}$
$\mathit{sizeOfPrev}(\pi, \underline{a}, \mathit{pVal}, \mathit{absVal})$	$\mathit{expr} = \begin{cases} \mathit{absVal} & \text{if } \neg x(\underline{a}) \\ \mathit{pVal} & \text{if } x(\underline{a}) \wedge \big(\pi(\underline{a}) = 1\big) \\ sz(P(\pi, \underline{a})) & \text{otherwise} \end{cases}$

Table 4: Expression semantics

9.2 OPL Formulation

The construct below defines an interval sequence variable `p` over a set of interval variables `A` given as an array. The optional array of integers `T` specifies the types of interval variables in the sequence, that is, `T[i]` is the integer type of `A[i]` in the sequence.

```
dvar sequence p in A [types T];
```

Where

```
dvar interval A[];
int T[];
```

In OPL, the matrix used by the `noOverlap` constraint needs to be defined as a set of integer triplets. Thus a matrix is a tuple set. For instance:

```
tuple triplet { int id1; int id2; int value; };  // 0<=id1, 0<=id2, 0<=value
{triplet} M = ...;
```

The following constraints are available on interval sequence variables:

```
first (p, a);
last  (p, a);
prev  (p, a, b);
before(p, a, b);
```

Important: In the current version of CP Optimizer, the constraints `first`, `last`, `prev` and `before` cannot be used in meta-constraints.

The constraint(s) below define(s) how a given sequence of intervals influences the values of the interval themselves:

```
noOverlap(p);
noOverlap(p, M [,next]); // With a transition distance
                         // with the next interval if next>0, or
                         // with all successors if next=0
```

Where

```
int next;
```

An additional construction of `noOverlap` is available that shortcuts the creation of the interval sequence variable for simple cases where the sequence is not useful:

```
noOverlap(A); // Same as: dvar sequence p in A;
              //          noOverlap(p);
```

Important: The constraint `noOverlap` cannot be used in meta-constraints.

The following integer expressions are provided:

```
dexpr int t  = typeOfNext  (p, a ,nVal [,absVal]);
dexpr int t  = typeOfPrev  (p, a ,pVal [,absVal]);
dexpr int s  = startOfNext (p, a ,nVal [,absVal]);
dexpr int e  = endOfNext   (p, a ,nVal [,absVal]);
dexpr int l  = lengthOfNext(p, a ,nVal [,absVal]);
dexpr int sz = sizeOfNext  (p, a ,nVal [,absVal]);
dexpr int s  = startOfPrev (p, a ,pVal [,absVal]);
dexpr int e  = endOfPrev   (p, a ,pVal [,absVal]);
dexpr int l  = lengthOfPrev(p, a ,pVal [,absVal]);
dexpr int sz = sizeOfPrev  (p, a ,pVal [,absVal]);
```

Where

```
int nVal;
int pVal;
int absVal;
```

9.3 Examples

- A set of `2*n` activities to be sequenced on a unary resource such that `A[2*i-1]` is constrained to be executed immediately before to `A[2*i]`.

  ```
  dvar interval A[i in 1..2*n] size d[i];
  dvar sequence p in A;

  subject to {
    noOverlap(p);
    forall (i in 1..n)
      prev(p, A[2*i-1], A[2*i]);
  }
  ```

- A set of `n` activities `A[i]` of integer type `T[i]` to be sequences on a machine with a sequence dependent setup time `abs(ti-tj)` to switch from activity type `ti` to activity type `tj`.

  ```
  {int} Types = { T[i] | i in 1..n };
  tuple triplet { int id1; int id2; int value; };
  {triplet} M = { <i,j,ftoi(abs(i-j))> | i in Types, j in Types };

  dvar interval A[i in 1..n] size d[i];
  dvar sequence p in A types T;

  subject to {
    noOverlap(p, M);
  }
  ```

- Here is a complete model of an open-shop scheduling problem:

```
tuple Operation {
  int job; // Job
  int mch; // Machine
  int pt;  // Processing time
}

int NbJobs  = ...;
int NbMchs  = ...;
{Operation} Ops = ...;

dvar interval itvs[o in Ops] size o.pt;
dvar sequence jobs[j in 1..NbJobs] in all(o in Ops:o.job==j) itvs[o];
dvar sequence mchs[m in 1..NbMchs] in all(o in Ops:o.mch==m) itvs[o];

minimize max(o in Ops) endOf(itvs[o]);
subject to {
 forall(j in 1..NbJobs)
    noOverlap(jobs[j]);
 forall(m in 1..NbMchs)
    noOverlap(mchs[m]);
}
```

- Minimization of the total setup cost on a machine.

```
range Types = 1..4;
int Setup[Types,Types]=[
   [2,3,1,5],
   [4,1,3,2],
   [9,2,4,3],
   [1,2,3,1]
];
int InitState = 2; // Machine initial state

range Ops = 1..100;
int OpType [Ops] = ...;
int OpPT   [Ops] = ...;

dvar interval op[t in Ops] size OpPT[t];
dvar sequence mach in all(t in Ops) op[t] types all(t in Ops) OpType[t];

dexpr int setupcost =
  sum(t in Ops) Setup[typeOfPrev(mach,op[t],InitState)][OpType[t]];

minimize setupcost;
subject to {
  noOverlap(mach);
}
```

- It is important to remark that intervals of null length can also be sequenced. This is particularly useful for modelling transition distances between instantaneous time-points. For instance, the sample below models a set of non-overlapping transportation activities $A[i]$ that start at location $l1[i]$ and finish at location $l2[i]$ and such that the transition time between two activities depends both on the finishing position of the predecessor and on the starting position of the successor.

```
tuple triplet { int l1; int l2; int d; };
{ triplet } distance = ;
int l1[1..n] = ;
int l2[1..n] = ;
int dur[1..n] = ...;
dvar interval A[i in 1..n]  size dur[i];
dvar interval P[1..n][1..2] size 0;
dvar sequence Seq
 in    append(all(i in 1..n) P[i][1], all(i in 1..n) P[i][2] )
 types append(all(i in 1..n) l1[i],   all(i in 1..n) l2[i] );
constraints {
  forall (i in 1..n) {
    startAtStart(A[i],P[i][1]);
    endAtEnd(A[i],P[i][2]);
  }
  noOverlap(Seq, distance);
  noOverlap(all(i in 1..n) A[i]);
}
```

10 Cumul Functions

10.1 Semantics

10.1.1 Informal Description

In scheduling problems involving cumulative resources (also known as renewable resources), the cumulated usage of the resource by the activities is usually represented by a function of time. An activity usually increases the cumulated resource usage function at its start time and decreases it when it releases the resource at its end time (pulse function). For resources that can be produced and consumed by activities (for instance the content of an inventory or a tank), the resource level can also be described as a function of time, production activities will increase the resource level whereas consuming activities will decrease it. In these type of problems, the cumulated contribution of activities on the resource can be represented by a function of time and constraints can be posted on this function, for instance a maximal or a safety level.

CP Optimizer introduces the notion of *cumul function expression* which is a function that represents the sum of individual contributions of intervals. A panel of elementary cumul function expressions is available to describe the individual contribution of an interval variable (or a fixed interval of time), they cover the main use-cases mentioned above: *pulse* for usage of a cumulative resource, *step* for resource production/consumption (see §10.1.2). When the elementary cumul function expressions that define a cumul function expression are fixed (and thus, so are their related intervals), the expression is fixed. CP Optimizer provides several constraints over cumul function expressions. These constraints allow restricting the possible values of the function over the complete horizon or over some fixed or variable interval (see §10.1.3). For applications where the actual quantity of resource that is used, produced or consumed by intervals is an unknown of the problem, expressions are available for constraining these quantities (see §10.1.4).

10.1.2 Cumul Function Expressions

Let $\mathcal{F}^+$ denote the set of all functions from $\mathbb{Z}$ to $\mathbb{Z}^+$. A **cumul function expression** f is an expression whose value is a function of $\mathcal{F}^+$ and thus, whose domain $dom(f)$ is a subset of $\mathcal{F}^+$. Let $u, v \in \mathbb{Z}$ and $h, h_{min}, h_{max} \in \mathbb{Z}^+$ and a be an interval variable, we consider the following **elementary cumul function expressions** illustrated in Figure 4: $pulse(u, v, h)$, $step(u, h)$, $pulse(a, h)$, $pulse(a, h_{min}, h_{max})$, $stepAtStart(a, h)$, $stepAtStart(a, h_{min}, h_{max})$, $stepAtEnd(a, h)$, $stepAtEnd(a, h_{min}, h_{max})$.

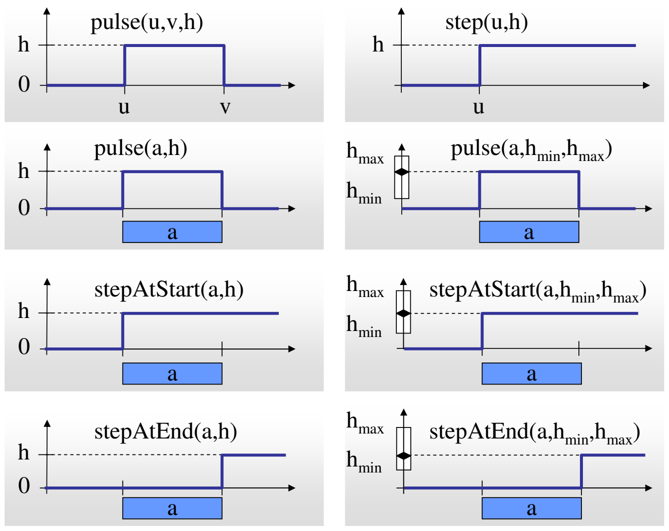

Figure 4: Elementary cumul function expressions

More formally, let $u, v \in \mathbb{Z}$ and $h \in \mathbb{Z}^+$, we define the following particular functions of $\mathcal{F}^+$:

- $\underline{0}$ is the null function that is, the function F such that $\forall t \in \mathbb{Z}, F(t) = 0$
- $\underline{pulse}(u, v, h)$ is the function F such that $F(t) = h$ if $t \in [u, v)$, $F(t) = 0$ otherwise
- $\underline{step}(u, h) = \underline{pulse}(u, +\infty, h)$

The semantics of the elementary function expressions is listed on table 5 together with the formal definition of their domain. The function set 0_a is equal to the singleton $\{\underline{0}\}$ if $\perp \in dom(a)$, that is, if interval variable a is possibly absent and equal to the empty set otherwise.

Function expression	**Domain**
$pulse(u, v, h)$	$\{\ \underline{pulse}(u, v, h)\ \}$ (singleton)
$step(u, h)$	$\{\ \underline{step}(u, h)\ \}$ (singleton)
$pulse(a, h)$	$0_a \cup \{\ \underline{pulse}(s(\underline{a}), e(\underline{a}), h) \mid \underline{a} \in dom(a) \setminus \perp\ \}$
$pulse(a, h_{min}, h_{max})$	$0_a \cup \{\ \underline{pulse}(s(\underline{a}), e(\underline{a}), h) \mid h \in [h_{min}, h_{max}] \wedge \underline{a} \in dom(a) \setminus \perp\ \}$
$stepAtStart(a, h)$	$0_a \cup \{\ \underline{step}(s(\underline{a}), h) \mid \underline{a} \in dom(a) \setminus \perp\ \}$
$stepAtStart(a, h_{min}, h_{max})$	$0_a \cup \{\ \underline{step}(s(\underline{a}), h) \mid h \in [h_{min}, h_{max}] \wedge \underline{a} \in dom(a) \setminus \perp\ \}$
$stepAtEnd(a, h)$	$0_a \cup \{\ \underline{step}(e(\underline{a}), h) \mid \underline{a} \in dom(a) \setminus \perp\ \}$
$stepAtEnd(a, h_{min}, h_{max})$	$0_a \cup \{\ \underline{step}(e(\underline{a}), h) \mid h \in [h_{min}, h_{max}] \wedge \underline{a} \in dom(a) \setminus \perp\ \}$

Table 5: Elementary cumul function expressions

A **cumul function expression** f is an expression built as the algebraical sum of the elementary function expressions of table 5 or their negations. More formally, it is a construct of the form $f = \sum_i \epsilon_i \cdot f_i$ where $\epsilon_i \in \{-1, +1\}$ and f_i is an elementary function expression.

When all elementary function expressions f_i related with a cumul function expression f are fixed (that is, their domain is reduced to a singleton $\underline{f}_i$), the value of the cumul function expression f is fixed and equal to the function $\underline{f} = \sum_i \epsilon_i \cdot \underline{f}_i$.

10.1.3 Constraints on Cumul Function Expressions

The following constraints can be expressed on a cumul function expression f. Let $u, v \in \mathbb{Z}$ and $h_{min}, h_{max} \in \mathbb{Z}^+$ and a be an interval variable:

- $alwaysIn(f, u, v, h_{min}, h_{max})$ means that the values of function f must remain in the range $[h_{min}, h_{max}]$ everywhere on the interval $[u, v)$.
- $alwaysIn(f, a, h_{min}, h_{max})$ means that if interval a is present, the values of function f must remain in the range $[h_{min}, h_{max}]$ between the start and the end of interval variable a.
- $f \leq h_{max}$ means that function f cannot take values greater than h_{max}. It is semantically equivalent to $alwaysIn(f, -\infty, +\infty, 0, h_{max})$.

- $f \geq h_{min}$ means that function f cannot take values lower than h_{min}. It is semantically equivalent to $alwaysIn(f, -\infty, +\infty, h_{min}, +\infty)$.

More formally:

$$\begin{aligned} alwaysIn(\underline{f}, u, v, h_{min}, h_{max}) &\Leftrightarrow \forall t \in [u, v),\ \underline{f}(t) \in [h_{min}, h_{max}] \\ alwaysIn(\underline{f}, \underline{a}, h_{min}, h_{max}) &\Leftrightarrow \forall t \in [s(\underline{a}), e(\underline{a})),\ \underline{f}(t) \in [h_{min}, h_{max}] \end{aligned}$$

10.1.4 Expressions on Cumul Function Expressions

The following elementary cumul function expressions are based on an interval variable a: $pulse(a, h)$, $pulse(a, h_{min}, h_{max})$, $stepAtStart(a, h)$, $stepAtStart(a, h_{min}, h_{max})$, $stepAtEnd(a, h)$, $stepAtEnd(a, h_{min}, h_{max})$.

Some of these expressions define a range $[h_{min}, h_{max}]$ of possible values for the actual height of the function when the interval variable a is present. The actual height is an unknown of the problem. CP Optimizer provides some integer expressions to control this height. These expressions are based on the notion of contribution of a given interval variable a to a (possibly composite) cumul function expression f. This contribution is defined as the sum of all the elementary cumul function expressions based on a in f. This contribution is a discrete function that can change value only at the start and at the end of interval a and is equal to 0 before the start of a.

For instance, let a and b be two interval variables and a cumul function expression f defined by: $f = step(0, 100) + pulse(a, 3) + pulse(a, 2) - stepAtEnd(a, 1) + stepAtStart(b, 2) - stepAtEnd(b, 3)$. The contribution of a to f is the function $pulse(a, 3) + pulse(a, 2) - stepAtEnd(a, 1)$ and the contribution of b to f is the function $stepAtStart(b, 2) - stepAtEnd(b, 3)$.

If interval a is present, the expression $heightAtStart(a, f)$ returns the value of the contribution of a to f evaluated at the start of a that is, it measures the contribution of interval a to cumul function expression f at its start point. Similarly, the expression $heightAtEnd(a, f)$ returns the value of the contribution of a to f evaluated at the end of a that is, it measures the contribution of interval a to cumul function expression f at its end point. An additional integer value $absVal$ can be specified at the construction of the expression in which case, that will be the value returned by the expression when the interval is absent otherwise, if no value is specified, the expression will be equal to 0 when the interval is absent.

In the example above, assuming both interval a and b to be present and with positive length we would get: $heightAtStart(a, f) = 5$, $heightAtEnd(a, f) = -1$, $heightAtStart(b, f) = 2$, $heightAtEnd(b, f) = -1$. Of course, in general when using ranges for the height of elementary cumul function expressions, the $heightAtStart/End$ expressions will not be fixed until all the heights and intervals presence status and length have been fixed by the search.

More formally, if an elementary cumul function expression f_i is based on an interval variable, we denote $ivar(f_i)$ this interval variable. Let f be a cumul function expression defined as $f = \sum_i \epsilon_i \cdot f_i$ where $\epsilon_i \in \{-1, +1\}$ where f_i are elementary cumul function expressions. The contribution of an interval variable a to f is defined as $f|_a = \sum_{i \,|\, ivar(f_i)=a} \epsilon_i \cdot f_i$ and the expressions $heightAtStart/End$ are defined as follows:

- $heightAtStart(\underline{a}, \underline{f}, absVal) = \begin{cases} \underline{f}|_{\underline{a}}(s(\underline{a})) & \text{if } x(\underline{a}) \\ absVal & \text{otherwise} \end{cases}$
- $heightAtEnd(\underline{a}, \underline{f}, absVal) = \begin{cases} \underline{f}|_{\underline{a}}(e(\underline{a})) & \text{if } x(\underline{a}) \\ absVal & \text{otherwise} \end{cases}$

10.2 OPL Formulation

A type of expression `cumulFunction` is introduced. The following functions build elementary cumul function expressions:

```
cumulFunction f = pulse(u,v, h);
cumulFunction f = pulse(a, h);
cumulFunction f = pulse(a, hmin, hmax);
cumulFunction f = step(u, h);
cumulFunction f = stepAtStart(a, h);
cumulFunction f = stepAtStart(a, hmin, hmax);
cumulFunction f = stepAtEnd(a, h);
cumulFunction f = stepAtEnd(a, hmin, hmax);
```

Where

```
int u;
int v;
int h;
int hmin
int hmax;
int absVal;
cumulFunction f;
cumulFunction g;
cumulFunction h;
cumulFunction F[];
cumulFunction G[];
dvar interval a;
```

A composite cumul function expression can be defined as a sum of cumul function expressions, it can be built using a `sum` construct and possibly some `-` operators, for instance:

```
cumulFunction h = sum(i in ...) F[i] - sum(j in ...) G[j];
```

The following are constraints on a cumul function expression `f`:

```
f <= hmax;
hmin <= f;
alwaysIn(f, u, v , hmin, hmax);
alwaysIn(f, a, hmin, hmax);
```

Important: All cumul function expressions `f` involved in a `<=` or an `alwaysIn` constraint are constrained to be non-negative. Constraints `<=` an `alwaysIn` on cumul function expressions cannot be used in meta-constraints.

The following expressions are available on cumul function expressions:

```
dexpr int h = heightAtStart(a,f[,absVal]);
dexpr int h = heightAtEnd(a,f[,absVal]);
```

10.3 Examples

- A set of producing and consuming activities on a tank with a given capacity and safety level.

```
int Horizon     = ...;
int Capacity    = ...;
int SafetyLevel = ...;
int StartLevel  = ...;
int QProd[p in 1..NProd] ...;
int QCons[c in 1..NCons] ...;

dvar interval AProd[p in 1..NProd] ...;
dvar interval ACons[c in 1..NCons] ...;

cumulFunction level =
    step(0, StartLevel)
  + sum (p in 1..NProd) stepAtEnd(AProd[p], QProd[p])
  - sum (c in 1..NCons) stepAtStart(ACons[c], QCons[c]);

subject to {
  alwaysIn(level, 0, Horizon, SafetyLevel, Capacity);
}
```

- Here is a complete model of the classical Resource-Constrained Project Scheduling problem (with renewable resources only):

```
tuple Prec {
  int task1;
  int task2;
}

tuple Req {
  int task;
  int rsrc;
  int qtty;
}

int nbTasks = ...;
int nbRsrcs = ...;
int dur[i in 1..nbTasks] = ...;
int cap[j in 1..nbRsrcs] = ...;
{Prec} precs = ...;
{Req}  reqs  = ...;

dvar interval tasks[i in 1..nbTasks] size dur[i];

minimize max(i in 1..nbTasks) endOf(tasks[i]);
subject to {
 forall(p in precs)
   endBeforeStart(tasks[p.task1],tasks[p.task2]);
 forall(j in 1..nbRsrcs)
   sum (r in reqs: r.rsrc==j) pulse(tasks[r.task],r.qtty) <= cap[j];
}
```

- A set of n activities A_i of length i and using $n - i$ units of a renewable resource have to be scheduled within a fixed horizon. The objective is to minimize the peak usage of the resource. A pulse with variable height is created for modelling the free amount of resource and the objective maximizes this free amount (height of the pulse).

```
int n       = 100;
int horizon = 500;
int capMax  = 500;

dvar interval cover in 0..horizon size horizon;
dvar interval a[i in 1..n] in 0..horizon size i;
cumulFunction free  = pulse(cover, 0, capMax);
cumulFunction level = sum(i in 1..n) pulse(a[i],n-i) + free;
dexpr int peak = capMax - heightAtStart(cover, free);

minimize peak;
subject to {
  level <= capMax;
}
```

- A set of n activities A_i is defined. These activities may overlap (a set of constraints is defined on these activities but this is not the point of the code sample). This code sample defines W, a chain of at most n intervals that exactly represent the set of time-windows during which at least one activity A_i executes. The two `alwaysIn` constraints state that (1) over each interval $W[i]$, there must be at least one activity executing ($1 \leq FA \leq n$) and (2) any date t where an activity $A[i]$ executes must be covered by one (and exactly one) interval W (thus, $FW == 1$). These interval variables W and cumul function expression FW can be useful for instance to model a constraint stating that there cannot be more than a certain number of activities of a given type executing in parallel by computing one function FW per type and constraining the sum of these functions. A solution is illustrated on Figure 5.

```
int horizon = ...;
int n = ...;
dvar interval A[i in 1..n] ...;
cumulFunction FA = sum(i in 1..n) pulse(A[i],1);
dvar interval W[i in 1..n] optional size 1..horizon;
cumulFunction FW = sum(i in 1..n) pulse(W[i],1);
constraints {
  forall(i in 1..n-1) {
    endBeforeStart(W[i],W[i+1],1);
    presenceOf(W[i+1]) => presenceOf(W[i]);
  }
  forall(i in 1..n) {
    alwaysIn(FA, W[i], 1, n);
    alwaysIn(FW, A[i], 1, 1);
  }
  // Constraints on activities A[i]
  // ...
}
```

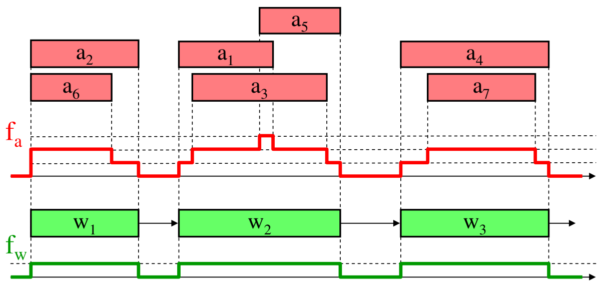

Figure 5: Covering chain

11 State Functions

11.1 Semantics

11.1.1 Informal Description

Some scheduling problems involve reasoning with resources whose state may change over time. The state of the resource can change because of the scheduled activities or because of exogenous events and some activities in the schedule may need a particular condition on the resource state to be true in order to execute. For instance, the temperature of an oven may change due to the execution of an activity that sets the oven temperature to a value v and some cooking activity may require the oven temperature to be maintained at a temperature level v' during its execution. In some applications, the transition between two states is not instantaneous and a transition time is needed for the resource to switch from a state v to a state v'.

CP Optimizer introduces the notion of *state function* which is a function describing the evolution of a given feature of the environment. The possible evolution of this feature is constrained by interval variables of the problem. The main difference between state functions and cumul functions (see §10) is that interval variables have an incremental effect on cumul functions (increasing or decreasing the function value) whereas they have an absolute effect on state functions (requiring the function value to be equal to a particular state or in a set of possible states).

Informally speaking, a state function is a set of non-overlapping intervals over which the function maintains a particular non-negative integer state. In between those intervals, the state of the function is not defined, typically because of an ongoing transition between two states (see 11.1.2). For instance for an oven with 3 possible temperature levels identified by indexes 0, 1 and 2 we could have:

[start=0,	end=100):	state=0,
[start=140,	end=300):	state=1,
[start=320,	end=500):	state=2,
[start=540,	end=600):	state=2, $\cdots$

A set of constraints are available to restrict the evolution of a state function (see 11.1.3). These constraints allows specifying:

- That the state of the function must be defined and should remain equal to a given non-negative state everywhere over a given fixed or variable interval ($alwaysEqual$).
- That the state of the function must be defined and should remain constant (no matter its value) everywhere over a given fixed or variable interval ($alwaysConstant$).
- That intervals requiring the state of the function to be defined cannot overlap a given fixed or variable interval ($alwaysNoState$).
- That everywhere over a given fixed or variable interval, the state of the function, if defined, must remain within a given range of non-negative states $[v_{min}, v_{max}]$ ($alwaysIn$).

Additionally, the constraints *alwaysEqual* and *alwaysConstant* can be complemented to specify that the given fixed or variable interval should have its start and/or end point synchronized with the start and/or end point of the interval of the state function that maintains the required state. This is the notion of *start* and *end alignment* which is particularly useful for modelling parallel batches. For instance in the oven example above, all interval variables that would require an oven temperature of level 1 and specify a start and end alignment, if executed over the interval $[140, 300)$ would have to start exactly at 140 and end at 300. This is depicted on figure 6 where interval variable b_1 is both start and end aligned, a_1 is start aligned only and a_3 is not aligned at all.

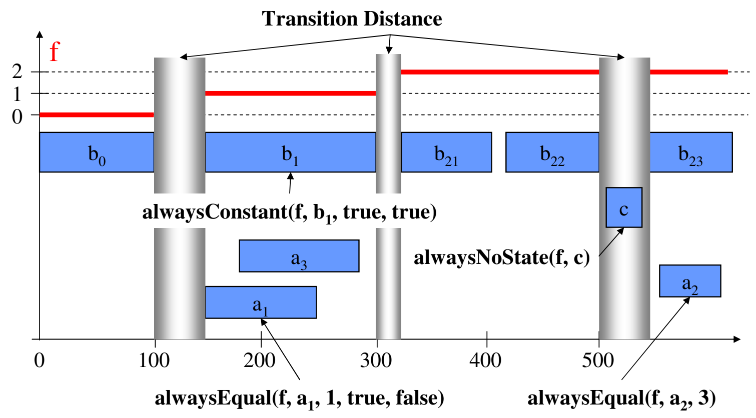

Figure 6: State function and interval variable alignments

11.1.2 State Functions and Transition Distance

A **state function** f is a decision variable whose value is a set of non-overlapping intervals, each interval $[s_i, e_i)$ being associated a non-negative integer value v_i that represents the state of the function over the interval.

Let $\underline{f}$ be a fixed state function, we will denote $\underline{f} = (\ [s_i, e_i) : v_i\ )_{i\in[1,n]}$:

- $\forall i \in [1, n], s_i \in \mathbb{Z}, e_i \in \mathbb{Z}, v_i \in \mathbb{Z}^+, s_i < e_i$
- $\forall i \in [1, n-1], e_i \leq s_{i+1}$

We denote $D(\underline{f}) = \cup_{i\in[1,n]}[s_i, e_i)$ the definition domain of $\underline{f}$, that is, the set of points where the state function is associated a state.

For a fixed state function $\underline{f}$ and a point $t \in D(\underline{f})$, we will denote $[s(\underline{f}, t), e(\underline{f}, t))$ the unique interval of the function that contains t and $\underline{f}(t)$ the value of this interval:

$$[s(\underline{f}, t), e(\underline{f}, t)) = [s_i, e_i), t \in [s_i, e_i)$$
$$\underline{f}(t) = v_i, t \in [s_i, e_i)$$

For instance, in the example of the oven introduced in section 11.1.1, we would have $\underline{f}(200) = 1$, $s(\underline{f}, 200) = 140$ and $e(f, 200) = 300$.

A state function can be associated a **transition distance**. The transition distance defines the minimal distance that must separate two consecutive states in the state function.

More formally, if $M[v, v']$ represent a transition distance matrix between state v and state v', the transition distance means that:

$$\forall i \in [1, n-1], e_i + M[v_i, v_{i+1}] \leq s_{i+1}$$

The transition distance matrix M must satisfy the triangular inequality.

For instance, in the example of the oven, the state function depicted on Figure 6 is consistent with the following transition distance:

$$\mathbf{M} = \begin{pmatrix} 0 & 40 & 50 \\ 30 & 0 & 20 \\ 40 & 30 & 40 \end{pmatrix}$$

Notice that a transition distance between the same states can be non-zero, as it is for state 2 in this example.

11.1.3 Constraints on State Functions

If f is a state function, a an interval variable, v, v_{min}, v_{max} non-negative integers and $algn_s$, $algn_e$ two boolean values:

- The constraint $alwaysEqual(f, a, v, algn_s, algn_e)$ specifies that whenever interval variable a is present, a must be contained in an interval where state function f maintains non-negative state v. If $algn_s$ is true, it means that interval a is start-aligned with f: interval a must start at the beginning of the interval where f is maintained in state s. If $algn_e$ is true, it means that interval a is end-aligned with f: interval a must end at the end of the interval where f is maintained in state s. More formally:

  (a) $[s(\underline{a}), e(\underline{a})) \subset [s(\underline{f}, s(\underline{a})), e(\underline{f}, s(\underline{a})))$

  (b) $algn_s \Rightarrow s(\underline{a}) = s(\underline{f}, s(\underline{a}))$

  (c) $algn_e \Rightarrow e(\underline{a}) = e(\underline{f}, e(\underline{a}))$

- The constraint $alwaysConstant(f, a, algn_s, algn_e)$ specifies that whenever a is present, state function f must be defined everywhere between the start and the end of interval a and be constant over this interval. More formally:
  $\exists v \in Z^+, alwaysEqual(f, a, v, algn_s, algn_e)$

- The constraint $alwaysNoState(f, a)$ specifies that whenever a is present, state function f cannot provide any valid state between the start and the end of interval a. As a consequence, any interval constrained with $alwaysEqual$ or $alwaysConstant$ on this function and thus requiring the function to be defined cannot overlap interval a. More formally, the constraint means that $[s(\underline{a}), e(\underline{a})) \cap D(\underline{f}) = \emptyset$.

- The constraint $alwaysIn(f, a, v_{min}, v_{max})$ where $0 \leq v_{min} \leq v_{max}$ specifies that whenever a is present, everywhere between the start and the end of interval a the state of function f, if defined, must belong to the range $[v_{min}, v_{max}]$. Formally: $\forall t \in [s(\underline{a}), e(\underline{a})) \cap D(\underline{f}), \underline{f}(t) \in [v_{min}, v_{max}]$.

On Figure 6, interval variables $b_0 : [0, 100)$ and $b_1 : [140, 300)$ are start and end aligned and thus, define two segments of the state function (of respective state 0 and 1). A transition distance 40 applies in between those states. Interval variable b_{21} is start aligned and interval b_{22} is end aligned both of state 2. As the transition distance $2 \rightarrow 2$ is greater than $s(b_{22}) - e(b_{21})$, the state function is aligned on $[s(b_{21}), e(b_{21}) = [320, 500)$. Interval variable c is constrained to be scheduled in an interval where the function is not defined. Finally, interval variables a_1, a_2 and a_3 require a particular state of the function, possibly with some alignment constraint as for a_1.

11.2 OPL Formulation

A type `stateFunction` is introduced. The following statement builds a state function:

```
stateFunction f [with M];
```

Where the optional argument `M` is a transition matrix that needs to be defined as a set of integer triplets (just like for the `noOverlap` constraint, see 9.2). Thus this matrix is a tuple set. For instance:

```
tuple triplet { int v1; int v2; int dist; };
{ triplet } M = ...;
```

The following are constraints over a state function `f`:

```
alwaysEqual    (f,s,e,v[,aligns,aligne]);
alwaysEqual    (f,a,v[,aligns,aligne]);
alwaysConstant (f,s,e [,aligns,aligne]);
alwaysConstant (f,a[,aligns,aligne]);
alwaysNoState  (f,s,e);
alwaysNoState  (f,a);
alwaysIn       (f,s,e,vmin,vmax);
alwaysIn       (f,a,vmin,vmax);
```

Where

```
int s;
int e;            // s <= e
int v;            // 0 <= v
int vmin;         // 0 <= vmin
int vmax;         // vmin <= vmax
dvar interval a;
bool aligns;
bool aligne;
```

Important: Constraints `alwaysEqual`, `alwaysConstant`, `alwaysNoState` and `alwaysIn` on state functions cannot be used in meta-constraints.

11.3 Examples

- A machine can be equipped with a tool among a set of n possible tools. Each operation o executed on the machine needs a specific tool $RequiredTool[o]$. The machine can process several operations simultaneously provided these operations are compatible with the tool currently installed on the machine. Changing the tool installed on the machine needs some constant set-up time which is supposed to be independent from the tools.

```
int nbTools   = ...;
int nbOps     = ...;
int setupTime = ...;
range Tools      = 1..nbTools;
range Operations = 1..nbOps;
int Duration     [Operations] = ...;
int RequiredTool [Operations] = ...;

dvar interval op[o in Operations] size Duration[o];

tuple triplet { int tl1; int tl2; int value; };
{triplet} Transition = { <tl1,tl2,setupTime> | tl1, tl2 in Tools };
stateFunction machineTool with Transition;

constraints {
  forall(o in Operations) {
     alwaysEqual(machineTool, op[o], RequiredTool[o]);
  }
}
```

- A limited capacity oven is used to treat items in parallel batches. Items requiring incompatible temperatures cannot be treated in the same batch. Some set-up time is needed to warm-up and cool-down the oven depending on the original and target temperature levels. The oven temperature is modelled as a state function. In a solution, the intervals of the state function represent the different batches. The treatment of each item is modelled by an interval variable. Because of parallel batching, items treated in the same batch need to be synchronized: they have the same start and end time, this is modelled using the notion of start and end alignment.

```
int MaxItemsInOven = ...;
int NbItems = ...;
range Items = 1..NbItems;
```

```
int DurationMin[Items] = ...;
int DurationMax[Items] = ...;
int Temperature[Items] = ...;
tuple triplet { int t1; int t2; int value; };
{triplet} Transition = ...;

dvar interval treat[i in Items] size DurationMin[i]..DurationMax[i];

stateFunction ovenTemperature with Transition;
cumulFunction itemsInOven = sum(i in Items) pulse(treat[i], 1);

constraints {
  itemsInOven <= MaxItemsInOven;
  forall(i in Items)
    alwaysEqual(ovenTemperature, treat[i], Temperature[i], 1, 1);
}
```

Now suppose that a maintenance task is to be performed on the oven and that, during this maintenance task, the oven temperature should not reach high values (say, greater than level 4). This could be modelled by the additional constraint:

```
dvar interval maintenance ...;

constraints {
  // ...
  alwaysIn(ovenTemperature, maintenance, 0, 4);
}
```

12 Isomorphism constraint

12.1 Semantics

This section describes the **isomorphism** constraint between two sets of interval variables $A = \{a_0, ..., a_{n-1}\}$ and $B = \{b_0, ..., b_{m-1}\}$. This constraint states that, in a solution, there is a 1-to-1 correspondance between the present intervals of A and the present intervals B; two intervals in correspondance have the same start and end values. The isomorphism constraint is useful to enforce some patterns of constraints on a set of interval variables (see example blelow).

More formaly, let $\mathbf{P}(A) = \{a \in A, x(\underline{a}) = 1\}$ and $\mathbf{P}(B) = \{b \in B, x(\underline{b}) = 1\}$. The **isomorphism constraint** $isomorphic(A, B)$ holds if and only if:

$$\exists f : \mathbf{P}(A) \rightarrow \mathbf{P}(B), \qquad \text{f is bijective } \wedge$$
$$\forall a \in \mathbf{P}(A) \begin{cases} s(f(\underline{a})) = s(\underline{a}) \\ e(f(\underline{a})) = e(\underline{a}) \end{cases}$$

The constraint can be passed an additional set of $|A|$ integer variables $V = \{v_0, .., v_{|A|-1}\}$ with domain in $[0, |B| - 1] \cup \{\epsilon\}$ that describes the bijective function f at the solution: the constraint $isomorphic(A, B, F)$ holds if and only if:

$$\forall i \in [0, |A| - 1], \begin{cases} \underline{v_i} \neq \epsilon \Leftrightarrow a_i \in \mathbf{P}(A) \Leftrightarrow f(\underline{a_i}) = b_{\underline{v_i}} \\ \underline{v_i} = \epsilon \Leftrightarrow \begin{cases} \epsilon \notin [0, |B| - 1] \Rightarrow a_i \notin \mathbf{P}(A) \\ \epsilon \in [0, |B| - 1] \Rightarrow a_i \notin \mathbf{P}(A) \vee f(\underline{a_i}) = b_\epsilon \end{cases} \end{cases}$$

In others words, V is an indexer that gives the position of a present interval in A, as an ordred set, of its correspondant present interval in in B, as an ordered set. The position of an absent interval of A is, by convention, set to ϵ.

The isomorphism constraint is illustrated on Figure 7.

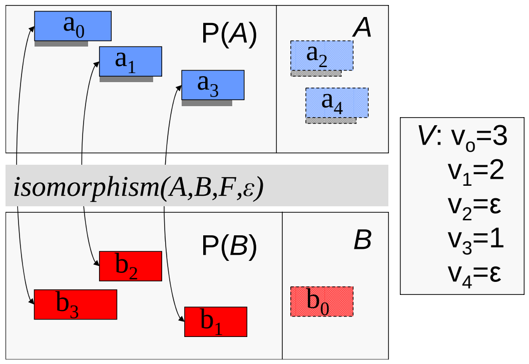

Figure 7: Example of isomorphism constraint

12.2 OPL Formulation

The construct below is a constraint:

```
isomorphism(A, B [, F, eps]);
```

Where

```
dvar interval A[]; // Array of size n
dvar interval B[]; // Array of size n
dvar int V[];      // Array of size n
int  eps;          // Integer Escape Value
```

Important: The constraint `isomorphic` cannot be used in meta-constraints.

12.3 Example

- A set of n activities a[1],...,a[n] is to be sequenced on a worker, each activity a[i] has a nominal duration NDur[i]. There is a position-based learning effect: as the worker is executing the activities, he is learning so that if activity a[i] is executed in position j in the sequence, its actual duration will be $\max(NDur[i], 2 \times NDur[i] \times \alpha^{j-1})$ where $\alpha \in [0,1]$ is the learning index. Other components of the problem is that each task cannot start before its release date, can be delayed from its tardiness date against a linear cost and is eventually abandonned for a cost of $NDur[i]$.

  Besides the n real activities a[i] modeled as interval variables, the model below introduces a chain of slots s[j] on the worker that represents the sequence of activities of the worker. Slot intervals s[j] will give the position of the jth activity executed by the worker. The mapping between the two sets of intervals is realized thanks to an isomorphism constraint.

```
int n =...;                                // Number of activities
float alpha =...;                          // Learning factor in [0, 1]
int NDur[i in 1..n] = ...;                 // Nominal duration
int Rd[i in 1..n] = ...;                   // start date release date
int Dd[i in 1..n] = ...;                   // end date tardy date

dvar interval a[i in 1..n] optional;       // Real activities
dvar interval s[j in 1..n] optional;       // Consecutive slots on machine
dvar int p[i in 1..n] in 0..n-1;           // Mapping slot and activity


minimize sum(i in 1..n) maxl(0, endOf(a[i], 2*Dd) - Dd);
subject to {
  noOverlap(a);
  isomorphic(s,a,f,0);                     // 0: position for absent activity
  forall (i in 1..n-1) {
    endBeforeStart(a[i + 1], a[i]);
    presenceOf(a[i + 1]) <= presenceOf(a[i]);
  }
  forall (i in 1..n) {
    startOf(a[i], Rd[i]) >= Rd[i];
    sizeOf(a[i], NDur[i])== maxl(NDur[i],
                                ftoi(round(2*NDur[i]*pow(alpha,p[i])));
}
```

13 ILOG Script for OPL

13.1 CP Parameters for scheduling

The following parameters are available to control the inference level in the CP Optimizer engine. As for other inference level parameters, their possible values are `Low`, `Basic`, `Medium` and `Extended`. The global default value in CP Optimizer is `Basic`.

- `PrecedenceInferenceLevel` controls the inference level of precedence constraints (See §5)
- `IntervalSequenceInferenceLevel` controls the internal inference level of sequence variables (See §9.1.1)
- `NoOverlapInferenceLevel` controls the inference level of no-overlap constraints (See §9.1.2)
- `CumulFunctionInferenceLevel` controls the inference level of constraints defined on cumul function expressions (See §10)
- `StateFunctionInferenceLevel` controls the inference level of constraints defined on state functions (See §11)
- `TemporalRelaxation` controls the usage of a temporal relaxation internal to engine. This parameter can take values "On" or "Off", with "On" being the default, meaning the relaxation is used in the engine when needed. For some models, using the relaxation becomes inefficient, and you may deactivate the use of the temporal relaxation using value "Off". This is an **advanced** parameter.

13.2 CP Search Phases for scheduling

Two types of search phases are available for scheduling: search phases on interval variables and search phases on sequence variables.

- A search phase on interval variables works on a unique interval variable or on an array of interval variables. During this phase CP Optimizer fixes the value of the specified interval variable(s): each interval variable will be assigned a presence status and for each present interval, a start and an end value.

  This search phase fixes the start and end values of interval variables in an unidirectional manner, starting to fix first the intervals that will be assigned a small start or end value.

  ```
  searchPhase(a);
  searchPhase(A);
  ```

  Where

```
dvar interval a;
dvar interval A[];
```

For instance, this code sample will specify a search that first fixes all interval variables in array `A1` before the ones in array `A2`.

```
dvar interval A1[...] ...;
dvar interval A2[...] ...;

execute {
  var f = cp.factory;
  cp.setSearchPhases(f.searchPhase(A1),
                     f.searchPhase(A2));
}
```

- A search phase on sequence variables works on a unique sequence variable or on an array of sequence variables. During this phase CP Optimizer fixes the value of the specified sequence variable(s): each sequence variable will be assigned a totally ordered sequence of present interval variables. Note that this search phase also fixes the presence statuses of the intervals involved in the sequence variables. This phase does not fix the start and end values of interval variables.

  It is recommended to use this search phase only if the possible range for start and end values of all interval variables is limited (for example by some known horizon that limits their maximal values).

  ```
  searchPhase(p);
  searchPhase(P);
  ```

  Where

  ```
  dvar sequence p;
  dvar sequence P[];
  ```

13.3 Accessing solutions in postprocessing

The value of an interval variable `a` in a solution can be accessed using the following instructions:

- `a.present` returns a boolean describing whether or not interval `a` is present in the solution.

- for a present interval, `a.start`, `a.end` and `a.size` respectively return the start, end and size value of interval `a` in the solution.

For instance the following postprocessing code will display the values of a set of interval variables in a solution.

```
dvar interval a[i in 1..n] ...;

// Model
// ...

// Postprocessing
execute {
 for(var i=1; i<=n; i++) {
    if (a[i].present) {
      writeln("Activity ", i, " executed on [", a[i].start, ",", a[i].end, ")");
    } else {
      writeln("Activity ", i, " is not executed");
    }
  }
}
```

An interval variable `a` in a solution can be displayed using `writeln(a)`. This instruction displays a vector: `< a.present a.start a.end a.size >`.

The value of a sequence variable `p` in a solution can be accessed using the following instructions:

- `p.first()` and `p.last()` respectively return the interval variable that corresponds to the first (resp. last) interval of the sequence. In case the sequence is empty the returned value is `null`.
- `p.next(a)` and `p.prev(a)` respectively return the interval variable sequenced just after `a` (resp. just before `a`) in the sequence. In case `a` is the last (resp. first) interval in the sequence, the returned value is `null`.

For instance, the following postprocessing code displays in chronological order the start end end dates of the intervals in a sequence variable.

```
dvar interval a[i in 1..n] size i;
dvar sequence s in all(i in 1..n) a[i];

// Model
// ...

// Postprocessing
execute {
  for(var x=s.first(); x!=null; x=s.next(x)) {
    writeln("[", x.start, ",", x.end, ")");
  }
}
```

A sequence variable p in a solution can be displayed using writeln(p). This instruction displays the set of present interval variables in the sequence following the total order specified by the sequence.

In a solution, a cumul function f is a non-negative step function. The value of a cumul function f in a solution can be accessed using the following instructions:

- numberOfSegments(f) returns an integer value that corresponds to the number of segments of cumul function f in the solution.
- segmentStart(f,i) returns the start of the i^{th} segment of cumul function f in the solution.
- segmentEnd(f,i) returns the end of the i^{th} segment of cumul function f in the solution.
- segmentValue(f,i) returns the value of the i^{th} segment of cumul function f in the solution.
- cumulFunctionValue(f,t) returns the value of cumul function f at point t in the solution.

For instance, the following postprocessing code displays the steps of a cumul function in a solution.

```
dvar interval a[i in 1..n] size i;
cumulFunction f = sum(i in 1..n) pulse(a[i],i);

// Model
// ...

// Postprocessing
execute {
  for(var i=0; i<Opl.numberOfSegments(f); i++) {
    writeln("Step[", i, "]: value=", Opl.segmentValue(f,i),
            " on [", Opl.segmentStart(f,i), "," , Opl.segmentEnd(f,i), ")");
  }
}
```

A Notations

The main notations are summarized on Table 6. Vectors are denoted by capital letters, for instance Y. The size of a vector Y is denoted $|Y|$. If $n = |Y|$, the vector is denoted $Y = (y_1, ..., y_n)$.

Notation	Type	Description
a, b	interval variable	
$x(a)$	boolean expression	presence status of a
$s(a)$, $e(a)$, $l(a)$	integer expression	start, end, length of a
$sz(a)$	integer expression	size of a
A, B	vector of interval variables	
p	interval sequence variable	
f, g, h	cumul function expression state function	
$\underline{a}$, $\underline{b}$	**fixed** interval	
$x(\underline{a})$	boolean	presence status of $\underline{a}$
$s(\underline{a})$, $e(\underline{a})$, $l(\underline{a})$	integer	start, end, length of $\underline{a}$
$sz(\underline{a})$	integer	size of $\underline{a}$
π	interval variable permutation	
z	integer (expression)	delay of a precedence
t	integer	date
T	vector of integers	array of dates
S	vector of floats	array of slopes
v, w	integer or float	miscellaneous integer or float value
V, W	vector of integers or floats	vector of miscellaneous values
$align$	boolean	alignment flag
F,G,H	piecewise linear function $\mathbb{R} \to \mathbb{R}$ vector of cumul function expressions	
$T(p, a)$	integer	type of interval a in sequence p
M	integer matrix	transition distance matrix
$\mathcal{F}^+$	set of functions	set of all functions $f : \mathbb{Z} \to \mathbb{Z}^+$

Table 6: Main notations

References

[1] ILOG. *ILOG OPL Development Studio 6.1. Manual*, September 2008. Trial version: http://www.ilog.com/products/oplstudio/.

[2] P. Laborie and D. Godard. Self-Adapting Large Neighborhood Search: Application to single-mode scheduling problems. In *Proc. MISTA-07*, 2007.

[3] P. Laborie and J. Rogerie. Reasoning with Conditional Time-intervals. In *Proc. FLAIRS-08*, 2008.