Chapter 6

Interval-Based Language for Modeling Scheduling Problems: An extension to Constraint Programming

Philippe Laborie and Jérôme Rogerie and Paul Shaw and Petr Vilím and Ferenc Katai

Abstract IBM ILOG CP Optimizer (CPO) provides a scheduling language supported by a robust and efficient automatic search. This paper summes up the major language constructs and shed some lights on their propagations. Among the main constructs it introduces the notion of interval variable which enables reasoning on conditional time-intervals representing activities or tasks that may or may not be executed in the final schedule. In Constraint-Based Scheduling, those problems are usually handled by defining new global constraints over classical integer variables. This dual perspective facilitates an easy modelling process while ensuring a strong constraint propagation and an efficient search in the engine. The approach forms the foundations of the new generation of scheduling model and algorithms provided in CPO. Small examples are provided at the end of the language construct, however at the end of the paper three larger/real life examples recently studied in the scheduling literature are presented along some computational results illustrating both the expressivity of the modelling language and the robustness of the automatic search. Interestingly all three problems can easily be modelled with the language in only a few dozen lines (the complete models are provided) and on average the automatic search outperforms existing problem specific approaches.

Philippe Laborie
IBM, 9 rue de Verdun, F-94253 Gentilly Cedex, France, e-mail: laborie@fr.ibm.com

Jérôme Rogerie
IBM, 9 rue de Verdun, F-94253 Gentilly Cedex, France, e-mail: rogerie@fr.ibm.com

Paul Shaw
IBM, 9 rue de Verdun, F-94253 Gentilly Cedex, France, e-mail: paul.shaw@fr.ibm.com

Petr Vilím
IBM, 9 rue de Verdun, F-94253 Gentilly Cedex, France, e-mail: petr_vilim@cz.ibm.com

Ferenc Katai
IBM, 9 rue de Verdun, F-94253 Gentilly Cedex, France, e-mail: ferenc.katai@fr.ibm.com

6.1 Introduction

Many scheduling problems involve reasoning about activities or processes that may or may not be executed in the final schedule. This is particularly true as scheduling is evolving in the direction of AI Planning whose core problem is precisely the selection of the set of activities to be executed. Indeed, most industrial scheduling applications 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 be executed 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*.

Modelling and solving these types of problems is an active topic in Constraint-Based Scheduling. Most of the current approaches are based on defining additional decision variables that represent the existence of an activity in the schedule [3, 2] or the index variable of the alternative resource/mode allocated to an activity [16, 14] and proposing new global constraints and associated propagation algorithms: *XorNode, PEX* in [3], DTP_{FD} in [14], *P/A Graphs* in [2], *alternative resource constraints* in [16].

CPO uses a different approach based on the idea that optional activities should be considered as first class citizen *variables* in the representation (we call them *interval variables*) and that the constraint propagation engine is extended to handle this new type of decision variable. Roughly speaking, it is the dual view compared with existing approaches: instead of defining new constraints over classical integer variables to handle optional activities, we introduce them as new variables in the engine. As we will see in the sequel of this paper, doing this offers several advantages:

- Modelling is easy because the notion of optionality is intrinsic to the concept of *interval variable*: there is no need for additional variables and complex meta-constraints,
- The model is very expressive and separates the temporal aspects from the logical ones,

- Constraint propagation is strong because the conditional domain maintained in *interval variables* naturally allows conjunctive reasoning between constraints,
- Most of the global constraints in Constraint-Based Scheduling can be extended to efficiently propagate on *interval variables*.

Sections 6.2-6.4 focus on the notion of *interval variable* and on the basic temporal and logical constraints between them. This concept forms the foundations of the new generation of scheduling model and algorithms embedded in CPO [9]. Additional constraints available in CPO for modelling resources are presented in sections 6.5-6.7. The modelling language is a direct mapping of those concepts, it is illustrated on three scheduling problems in section 6.9.

6.2 Conditional Interval Model

6.2.1 Usage and rationale

Interval variables and constraints over them make it easy to capture the structure of complex scheduling problems (hierarchical description of the work-breakdown structure of a project, representation of optional activities, alternatives of modes, recipes or processes, etc.) in a well-defined CP paradigm.

The integer expressions are provided to constrain the different components of an interval variable (start, end, length). For instance the expression startOf(a,dv) returns the start of interval variable a when it is present and integer value dv if it is absent. Those expressions make it possible to mix interval variables with integer variables, global constraints and expressions.

6.2.2 Interval Variables

Note that in this article, if x denotes a decision variable of the problem, we denote with an underline $\underline{x}$ a fixed decision variable, that is, a decision variable whose domain is reduced to a singleton.

A **interval variable** a is a variable whose domain $dom(a)$ is a subset of $\{\perp\} \cup \{[s,e)|s,e \in \mathbb{Z}, s \leq e\}$[1]. 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:

- interval is **absent**: $\underline{a} = \perp$; or

[1] Note that there is a small abuse of notation here as we allow $s = e$. This can be used to represent a zero length interval at value s even if in this case the interval $[s,e)$ itself is empty.

- interval is **present**: $\underline{a} = [s, e)$ with $s \leq e$. In this case, s and e are respectively the **start** and **end** of the interval and $l = e - s$ its **length**.

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 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.

The semantics of constraints defined over interval variables is described by the properties that fixed intervals must have in order the constraint to be true. If a fixed interval $\underline{a}$ is present and such that $\underline{a} = [u, v)$, we will denote $s(\underline{a})$ its integer start date u, $e(\underline{a})$ its integer end date v and $d(\underline{a})$ its positive integer length $v - u$. 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 meaningless.

By default interval variables are supposed to be present but they can be specified as being **optional** meaning that $\perp$ is part of the domain of the variable and thus, it is a decision of the problem to have the interval present or absent in the solution. Optional interval variables provide a powerful concept for efficiently reasoning with optional or alternative activities.

An example of declaration of a two dimensional array of interval variables in OPL can be found in Model 1, line 9. An example of declaration of an array of optional interval variables can be found in Model 2, line 7.

The following constraints on interval variables are introduced to model the basic structure of scheduling problems. Let a, a_i and b denote interval variables and z an integer variable:

- A **presence constraint** presenceOf(a) states that interval a is present, that is $a \neq \perp$. This constraint can be composed, for instance presenceOf$(a) \Rightarrow$ presenceOf(b) means that the presence of a implies the presence of b.
- A **precedence constraint** ($e.g.$ endBeforeStart(a, b, z)) specifies a precedence between interval end-points with an integer or variable minimal distance z provided both intervals a and b are present.
- A **span constraint** span$(a, \{a_1, ..., a_n\})$ states that if a is present, it starts together with the first present interval in $\{a_1, ..., a_n\}$ and ends together with the last one. Interval a is absent if and only if all the a_i are absent.
- An **alternative constraint** alternative$(a, \{a_1, ..., a_n\})$ models an exclusive alternative between $\{a_1, ..., a_n\}$: if interval a is present then exactly one of intervals $\{a_1, ..., a_n\}$ is present and a starts and ends together with this chosen one. Interval a is absent if and only if all the a_i are absent.

6.2.3 Presence Constraints

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

$$\text{presenceOf}(\underline{a}) \Leftrightarrow \big(x(\underline{a}) = 1\big)$$

In the basic model described in this section, we only consider unary and binary logical constraints between presence statuses, that is, constraints of the form of 2-SAT clauses over presence statuses: $[\neg]\,\text{presenceOf}(a)$ or $[\neg]\,\text{presenceOf}(a) \vee [\neg]\,\text{presenceOf}(b)$. For example if a and b are two conditional intervals such that when interval a is present then b must be present too, it can be modelled by the constraint $\neg\,\text{presenceOf}(a) \vee \text{presenceOf}(b)$.

An example of declaration of this type of logical constraints in OPL can be found in Model 3, line 32.

6.2.4 Precedence Constraints

The temporal constraint network consists of a Simple Temporal Network (STN) extended to the presence statuses. For instance, a precedence relation endBeforeStart(a,b) states that *if both intervals a and b are present* then the end of a must occur before the start of b.

The semantics of the relation $PC(\underline{a},\underline{b},z)$ on a pair of fixed intervals $\underline{a}$, $\underline{b}$ and for a delay value z depending on the precedence relation type PC is given on Table 6.1.

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 6.1 Precedence constraints semantics

Note that in general, the delay z specified in a precedence constraint can be a variable of the problem rather than a fixed value. For simplicity, we assume in this paper that it is always a fixed value.

An example of declaration of precedence constraints in OPL can be found in Model 1, line 16.

6.2.5 Intensity functions

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 last 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%. In CP Optimizer, this notion is captured by an **integer step function** that describes the instantaneous *intensity* - expressed as a percentage - of a work over time. An interval variable is associated with an **intensity function** and a **size**. The intensity function F specifies the instantaneous ratio between size and length. If an interval variable a is present, the intensity function enforces the following relation:

$$100 \times size(a) \leq \int_{start(a)}^{end(a)} F(t)\, dt < 100 \times (size(a)+1)$$

By default, the intensity function of an interval variable is a flat function equal to 100%. In this case, the concepts of *size* and *length* are identical.

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

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

Integer expressions are provided to constrain the different components of an interval variable (start, end, length, size). For instance the expression startOf(a,dv) returns the start of interval variable a when a is present and returns integer value dv if a is absent (by default if argument dv is omitted it assumes $dv = 0$). Those expressions make it possible to mix interval variables with more traditional integer constraints and expressions.

6.2.6 Interval Composition Constraints

This section describes two constraints over a group of intervals and allow a hierarchical definition of the model by encapsulating a group of intervals into one high-level interval. Both constraints are hybrid in the sense that they combine logical and temporal aspects. Here is an informal definition of these constraints:

- **Span constraint**. The constraint span$(a_0, \{a_1, ..., a_n\})$ states that, if the interval a_0 is present, it spans over all present intervals from the set $\{a_1, .., a_n\}$. That is, interval a_0 starts together with the first present interval from $\{a_1, .., a_n\}$ and ends together with the last one. If interval a_0 is absent then none of intervals $\{a_1, .., a_n\}$ is present.
- **Alternative constraint**. The constraint alternative$(a_0, \{a_1, .., a_n\})$ models an exclusive alternative between $\{a_1, .., a_n\}$. If interval a_0 is present then exactly one of intervals $\{a_1, .., a_n\}$ is present and a_0 starts and ends together with this chosen one. If interval a_0 is absent then none of intervals $\{a_1, .., a_n\}$ is present.

More formally, let $\underline{a_0}, \underline{a_1}, ..., \underline{a_n}$ be a set of fixed interval variables.

The **span constraint** span$(\underline{a_0}, \{\underline{a_1}, ..., \underline{a_n}\})$ holds if and only if:

$$\neg x(\underline{a_0}) \Leftrightarrow \forall i \in [1,n], \neg x(\underline{a_i})$$

$$x(\underline{a_0}) \Leftrightarrow \begin{cases} \exists i \in [1,n], x(\underline{a_i}) \\ s(\underline{a_0}) = \min_{i \in [1,n], x(\underline{a_i})} s(\underline{a_i}) \\ e(\underline{a_0}) = \max_{i \in [1,n], x(\underline{a_i})} e(\underline{a_i}) \end{cases}$$

An example of declaration of span constraint in OPL can be found in Model 3, line 37.

The **alternative intervals constraint** alternative$(\underline{a_0}, \{\underline{a_1}, ..., \underline{a_n}\})$ holds if and only if:

$$\neg x(\underline{a_0}) \Leftrightarrow \forall i \in [1,n], \neg x(\underline{a_i})$$

$$x(\underline{a_0}) \Leftrightarrow \exists k \in [1,n] \begin{cases} x(\underline{a_k}) \\ s(\underline{a_0}) = s(\underline{a_k}) \\ e(\underline{a_0}) = e(\underline{a_k}) \\ \forall j \neq k, \neg x(\underline{a_j}) \end{cases}$$

An example of declaration of alternative constraint in OPL can be found in Model 2, line 12.

6.3 Complexity

Although both the logical constraint network (2-SAT) and the temporal constraint network (STN) are polynomially solvable frameworks, finding a solution to the basic model described above that combines the two frameworks is NP-Complete (even without alternative and span constraints). The proof is a direct consequence of the fact the model allows the expression of the temporal disjunction between two intervals a and b. Indeed, such a temporal disjunction can be modelled using 4 additional conditional intervals a_1, a_2, b_1 and b_2 with the following constraint set:

$$\begin{aligned}
\text{startAtStart}(a,a_1);\quad & \text{startAtStart}(a,a_2);\\
\text{startAtStart}(b,b_1);\quad & \text{startAtStart}(b,b_2);\\
\text{endAtEnd}(a,a_1);\quad & \text{endAtEnd}(a,a_2);\\
\text{endAtEnd}(b,b_1);\quad & \text{endAtEnd}(b,b_2);\\
\text{endBeforeStart}(a_1,b_1);\quad & \text{endBeforeStart}(b_2,a_2);\\
\neg\,\text{presenceOf}(a_1)\vee\neg\,\text{presenceOf}(a_2);\quad & \text{presenceOf}(a_1)\vee\text{presenceOf}(a_2);\\
\neg\,\text{presenceOf}(b_1)\vee\neg\,\text{presenceOf}(b_2);\quad & \text{presenceOf}(b_1)\vee\text{presenceOf}(b_2);\\
\neg\,\text{presenceOf}(a_1)\vee\text{presenceOf}(b_1);\quad & \text{presenceOf}(a_1)\vee\neg\,\text{presenceOf}(b_1);
\end{aligned}$$

6.4 Graphical Conventions and Basic Examples

The following sections illustrate some examples of models. We are using the following graphical conventions:

- Interval variables are represented by a box. When necessary, the interval length is specified inside the box. A dotted box represents an optional interval variable.
- Temporal constraints are represented by plain arrows. Depending on the type of temporal constraint, the end-points of the arrow are connected with the appropriate time-points of the interval variables.
- Logical constraint are represented by dotted arrows and denote an implication relation (for instance $\text{presenceOf}(a) \Rightarrow \text{presenceOf}(b)$). In case the arrow starts or ends at a cross, it means the corresponding end-point of the implication is the negation of the presence status (e.g. $\text{presenceOf}(a) \Rightarrow \neg\,\text{presenceOf}(b)$).
- Span constraints are represented by a box (the spanning interval variable) containing the set of spanned interval variables.
- Alternative constraints are represented by a multi-edge labelled by XOR.

6.4.1 Alternative Modes with Compatibility Constraints

Suppose an activity a can be executed in n possible modes $\{a_i\}_{i\in[1,n]}$ with duration da_i for mode a_i and an activity b can be executed in m possible modes $\{b_j\}_{j\in[1,m]}$ with duration db_j for mode b_j. Furthermore, there are some mode incompatibility constraints (i,j) specifying that mode a_i for a is incompatible with mode b_j for b. This model is represented on Figure 6.1, incompatibilities (i,j) are modelled by implications $\text{presenceOf}(a_i) \Rightarrow \neg\,\text{presenceOf}(b_j)$ $(\neg\,\text{presenceOf}(a_i) \vee \neg\,\text{presenceOf}(b_j))$. Of course, in practical applications, interval variables a_i and b_j will require some conjunction of resources but this is out of the scope of the present section. If binary logical constraints are insufficient to model the compatibility rules, the presence status of the interval $\text{presenceOf}(a)$ can also be used in standard constraint programming constructs such as n-ary logical or arithmetic expressions or table constraints [4].

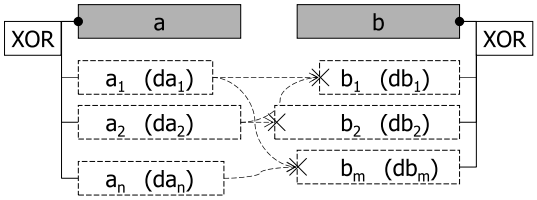

Fig. 6.1 A model for alternative modes

6.4.2 Series/Parallel Alternative

Figure 6.2 depicts a model where a job is composed of two operations a and b that can be executed either in series or in parallel (in this case, both operations are constrained to start at the same date). This is modelled by two alternatives $\text{alternative}(a,\{a_1,a_2\})$ and $\text{alternative}(b,\{b_1,b_2\})$ with (a_1,b_1) describing the serial and (a_2,b_2) describing the parallel execution. Logical constraints $\text{presenceOf}(a_i) \Leftrightarrow \text{presenceOf}(b_i)$ are added to ensure a consistent selection of the alternative.

A similar pattern can be used for any disjunction of a combination of temporal constraints on a pair of time interval variables.

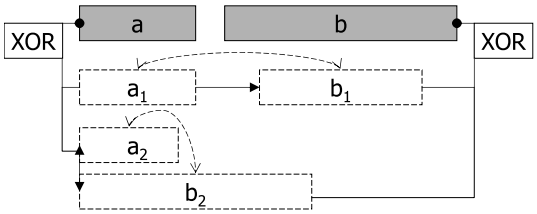

Fig. 6.2 A model for a series/parallel alternative

6.4.3 Alternative Recipes

Figure 6.3 describes a set of 3 alternative recipes. The global process a is modelled as an alternative of the 3 recipes r_1, r_2 and r_3. Each recipe is a spanning interval variable that spans the internal operations of the recipe. Implication constraints between a recipe and some of its internal operations (for instance presenceOf$(r_3) \Rightarrow$ presenceOf(o_{31})) mean that operation is not optional in the recipe. Note that the opposite implications (for instance presenceOf$(o_{31}) \Rightarrow$ presenceOf(r_3)) are part of the span constraint. This pattern can be extended to a hierarchy of spanning tasks which is very convenient for modelling complex work-breakdown structures in project scheduling.

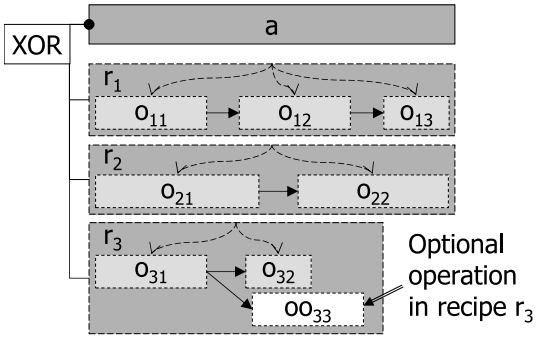

Fig. 6.3 A model for alternative recipes

6.4.4 Temporal Disjunction

The graphical representation of the model for temporal disjunction which shows that the basic framework is NP-Complete is depicted on Figure 6.4.

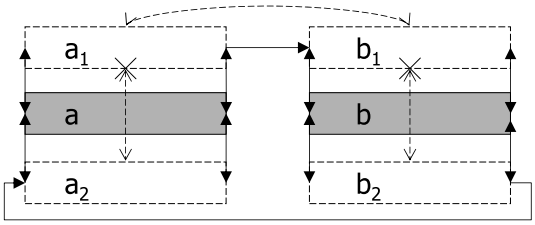

Fig. 6.4 A model for temporal disjunction

6.5 Sequence Variables

6.5.1 Usage and Rationale

Many scheduling problems involve disjunctive resources which can only perform one activity at a time (typical examples are workers, machines or vehicles). From the point of view of the resource, a solution is a sequence of activities to be processed. Besides the fact that activities in the sequence do not overlap in time, common additional constraints on such resources are setup times or constraints on the relative position of activities in the sequence.

To capture this idea we introduce the notion of *sequence variable*, a new type of decision variable whose value is a permutation of a set of interval variables. Constraints on interval variables are provided for ruling out illegal permutations (sequencing constraints) or for stating a particular relation between the order of intervals in the permutation and the relative position of their start and end values (no-overlap constraint).

6.5.2 Formal Semantics

6.5.2.1 Sequence Variable.

A **sequence variable** p is defined on a set of interval variables A. Informally speaking, a value of p is a permutation of all present intervals of A. Let $n = |A|$ and $\underline{A}$ be an instantiation of the intervals of A. A permutation π of $\underline{A}$ is a function $\pi : \underline{A} \to [0,n]$ such that, if we denote $\text{length}(\pi) = |\{\underline{a} \in \underline{A}, x(\underline{a})\}|$ the number of present intervals:

1. $\forall \underline{a} \in \underline{A}, (\underline{a} = \bot) \Leftrightarrow (\pi(\underline{a}) = 0)$
2. $\forall \underline{a} \in \underline{A}, \pi(\underline{a}) \leq \text{length}(\pi)$
3. $\forall \underline{a}, \underline{b} \in \underline{A}, \pi(\underline{a}) = \pi(\underline{b}) \Rightarrow (\underline{a} = \underline{b} = \bot) \vee (a = b)$

For instance, if $A = \{a,b\}$ is a set of two interval variables with a being present and b optional, the domain of the sequence p defined on A consists of 3 values: $\{(a \to 1, b \to 0), (a \to 1, b \to 2), (a \to 2, b \to 1)\}$ or in short $\{(a), (a,b), (b,a)\}$.

6.5.2.2 Sequencing Constraints.

The sequencing constraints below are available:

- first(p,a) states that if interval a is present then, it will be the first interval of the sequence p: $(\underline{a} \neq \perp) \Rightarrow (\pi(\underline{a}) = 1)$.
- last(p,a) states that if interval a is present then, it will be the last interval of the sequence p: $(\underline{a} \neq \perp) \Rightarrow (\pi(\underline{a}) = \text{length}(\pi))$.
- before(p,a,b) states that if both intervals a and b are present then a will appear before b in the sequence p: $(\underline{a} \neq \perp) \wedge (\underline{b} \neq \perp) \Rightarrow (\pi(\underline{a}) < \pi(\underline{b}))$.
- 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 p: $(\underline{a} \neq \perp) \wedge (\underline{b} \neq \perp) \Rightarrow (\pi(\underline{a}) + 1 = \pi(\underline{b}))$.

In the previous example, a constraint prev(p,a,b) would rule out value (b,a) as an illegal value of sequence variable p.

6.5.2.3 Transition Distance.

Let $m \in \mathbb{Z}^+$, a **transition distance** is a function $M : [1,m] \times [1,m] \to \mathbb{Z}^+$. Transition distances are typically used to express a minimal delay that must elapse between two successive non-overlapping intervals.

6.5.2.4 No-overlap Constraint.

Note that the sequencing constraints presented above do not have any impact on the start and end values of intervals, they only constrain the possible values of the sequence variable. The **no-overlap 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. A set of non-negative integer types $T(p,a)$ can be associated to each interval of a sequence variable. If a transition distance M is specified, it defines the minimal non-negative distance that must separate every two intervals in the sequence. More formally, let p be a sequence and let $T(p,a)$ be the type of interval a in sequence variable p, the condition for a permutation value π to satisfy the *no-overlap* constraint on p with transition distance M is defined as:

$$\text{noOverlap}(\pi, M) \Leftrightarrow \forall \underline{a}, \underline{b} \in \underline{A},$$
$$0 < \pi(\underline{a}) < \pi(\underline{b}) \;\Leftrightarrow\; e(\underline{a}) + M[T(p, \underline{a}), T(p, \underline{b})] \leq s(\underline{b})$$

Figure 6.5 illustrates the value of a sequence variable with a set of constraints it satisfies.

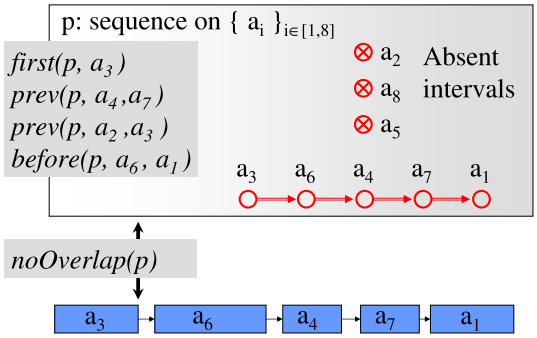

Fig. 6.5 Example of sequence variables and constraints

A simple example of declaration of no-overlap constraint in OPL can be found in Model 1, line 19. A slightly more complex example with transition distance is available in Model 3, line 41.

6.6 Cumul Function Expressions

6.6.1 Usage and Rationale

In scheduling problems involving cumulative resources, the cumulated usage of the resource by the activities is usually represented by a function of time. An activity increases the cumulated resource usage function at its start time and decreases it when it releases the resource at its end time. 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 problem classes, constraints are imposed on the evolution of these functions of time, for instance a maximal capacity or a minimum safety level. CP Optimizer introduces the notion of a *cumul function expression* which is a constrained expression that represents the sum of individual contributions of intervals.[2] A set of elementary

[2] In the rest of the paper, we often drop "expression" from "cumul function expression" to increase readability.

cumul functions is available to describe the individual contribution of an interval variable or a fixed interval of time. These elementary functions cover the use-cases mentioned above: *pulse* for usage of a cumulative resource, and *step* for resource production/consumption. When the elementary cumul functions that define a cumul function are fixed (and thus, so are their related intervals), the cumul function itself is fixed and its value is a stepwise integer function. Several constraints are provided over cumul functions. These constraints allow restricting the possible values of the function over the complete horizon or over some fixed or variable interval.

6.6.2 Formal Semantics

Let $\mathscr{F}^+$ 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 $\mathscr{F}^+$. Let $u, v \in \mathbb{Z}$ and $h, h_{min}, h_{max} \in \mathbb{Z}^+$ and a be an interval variable, we consider **elementary cumul functions** illustrated in Figure 6.6.

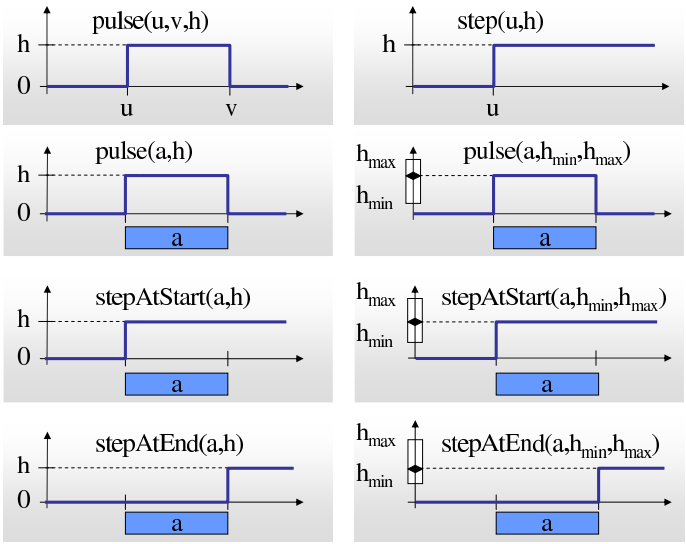

Fig. 6.6 Elementary cumul function expressions

Whenever the interval variable of an elementary cumul function is absent, the function is the zero function. A **cumul function** f is an expression built as the algebraic sum of the elementary functions of Figure 6.6 or their negations. More formally, it is a construct of the form $f = \sum_i \varepsilon_i \cdot f_i$ where $\varepsilon_i \in \{-1, +1\}$ and f_i is an elementary cumul function.

The following constraints can be expressed on a cumul function f to restrict its possible values:

- 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.
- $f \leq h$: function f cannot take values greater than h.
- $f \geq h$: function f cannot take values lower than h.

An integer expression is introduced to get the total contribution of an interval variable a to a cumul function f at its start: heightAtStart(a,f,dh) with a default value dh in case a is absent. A similar expression exists for the end point. These expressions are useful to constrain the variable height of an elementary cumul function specified as a range $[h_{min},h_{max}]$ using classical constraints on integer expressions.

6.6.3 Example

The constraints below model (1) a set of n activities $\{a_i\}$ such that no more than 3 activities in the set can overlap and (2) a chain of optional interval variables w_j that represent the distinct time-windows during which at least one activity a_i must execute. The constraints on interval variable status ensure that only the first intervals in the chain are present and the two alwaysIn constraints state the synchronization relation between intervals a_i and intervals w_j. A solution is illustrated on Figure 6.7.

$$
\begin{aligned}
f_a = \sum_{i=1}^{n} \text{pulse}(a_i,1); \quad f_w &= \sum_{j=1}^{n} \text{pulse}(w_j,1); \\
f_a &\leq 3; \\
\forall j \in [1,n-1] \quad &\begin{cases} \text{presenceOf}(w_{j+1}) \Rightarrow \text{presenceOf}(w_j); \\ \text{endBeforeStart}(w_j,w_{j+1}); \end{cases} \\
\forall i \in [1,n] \quad &\begin{cases} \text{alwaysIn}(f_a,w_i,1,n); \\ \text{alwaysIn}(f_w,a_i,1,1); \end{cases}
\end{aligned}
$$

An example of constrained cumul function expression in OPL can be found in Model 2, line 14.

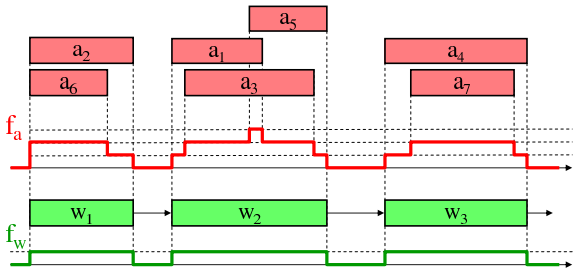

Fig. 6.7 Covering chain

6.7 State Function Variables

6.7.1 Usage and Rationale

In the same way as the value of an integer variable may represent an ordinal integer, functions over ordinal integers are useful in scheduling to describe the time evolution of a state variable. Typical examples are the time evolution of an oven's temperature, of the type of raw material present in a tank or of the tool installed on a machine. To that end, we introduce the notion of a *state function variable* and a set of constraints similar to the alwaysIn constraints on cumul functions to constrain the values of the state function.

A state function is a set of non-overlapping segments[3] over which the function maintains a constant non-negative integer state. In between those segments, the state of the function is not defined. For instance for an oven with 3 possible temperature levels identified by indices 0, 1 and 2 we could have the following time evolution (see also Figure 6.8):

$$
\begin{aligned}
&[\text{start} = 0, \quad \text{end} = 100): \text{state} = 0,\\
&[\text{start} = 140, \text{end} = 300): \text{state} = 1,\\
&[\text{start} = 320, \text{end} = 500): \text{state} = 2,\\
&[\text{start} = 540, \text{end} = 600): \text{state} = 2, \cdots
\end{aligned}
$$

[3] A segment is an interval of integer. In the description of state functions we do not use the term *interval* to avoid confusion with interval variables.

6.7.2 Formal Semantics

6.7.2.1 State Function Variable.

A **state function variable** f is a variable whose value is a set of non-overlapping segments, each segment $[s_i, e_i)$ (with $s_i < e_i$) is associated with a non-negative integer value v_i that represents the state of the function over the segment. Let $\underline{f}$ be a fixed state function, we will denote $\underline{f} = (\ [s_i, e_i) : v_i\ )_{i\in[1,n]}$. 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 segment of the function that contains t and $\underline{f}(t)$ the value of this segment. For instance, in the oven example we would have $\underline{f}(200) = 1, s(\underline{f},200) = 140$ and $e(f,200) = 300$.

A state function can be endowed with a **transition distance**. The transition distance defines the minimal distance that must separate two consecutive states in the state function. More formally, if $M[v,v']$ is a transition distance matrix between state v and state v', we have: $\forall i \in [1, n-1],\ e_i + M[v_i, v_{i+1}] \leq s_{i+1}$.

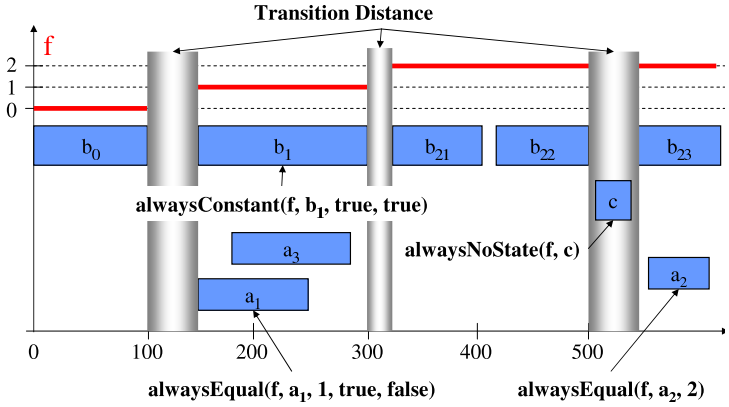

Fig. 6.8 State function

6.7.2.2 Constraints on State Functions

If f is a state function of definition domain $D(f)$, a an interval variable, v, $v_{min} \leq v_{max}$ non-negative integers and $algn_s, algn_e$ two boolean values:

- The constraint alwaysConstant$(f, a, algn_s, algn_e)$ specifies that whenever a is present, the function takes a constant value between the start and the end of a.

Boolean parameters $algn$ allow specifying whether or not interval variable a is synchronized with the start (resp. end) of the state function segment:

(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) $algn_e \Rightarrow e(\underline{a}) = e(\underline{f}, e(\underline{a}))$
(d) $\exists v \in \mathbb{Z}^+, \forall t \in [s(\underline{a}), e(\underline{a})), \underline{f}(t) = v$

- The constraint alwaysEqual$(f, a, v, algn_s, algn_e)$ specifies that whenever a is present the state function takes a constant value v over interval a:

(a) alwaysConstant$(\underline{f}, \underline{a}, algn_s, algn_e)$
(b) $v = \underline{f}(s(\underline{a}))$

- The constraint alwaysNoState(f, a) specifies that if a is present, it must not intersect the definition domain of the function, $[s(\underline{a}), e(\underline{a})) \cap D(\underline{f}) = \emptyset$.
- The constraint alwaysIn(f, a, v_{min}, v_{max}) where $0 \leq v_{min} \leq v_{max}$ specifies that whenever a is present, $\forall t \in [s(\underline{a}), e(\underline{a})) \cap D(\underline{f}),\ \underline{f}(t) \in [v_{min}, v_{max}]$.

Those constraints are also available on a fixed interval [start, end) as well as on an interval variable.

On Figure 6.8, 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 $[s(b_{21}), e(b_{22}) = [320, 500)$. Interval variable c is constrained to be scheduled in a segment 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.

6.7.3 Example

The problem is to cook n items with an oven, each item $i \in [1, n]$ being cooked at a specific temperature v_i and for a specific range of duration. Items that are compatible both in temperature and in duration can be batched together and cooked simultaneously. Between two batches, a delay must elapse for cooling, emptying, loading, and heating the oven. For energy saving reasons the maximum reachable temperature is limited by v_{sup} over some time periods. The oven can be modeled as state function with a transition distance M. Each item is an interval variable a_i, possibly optional if the problem is over-constrained so that not all items can be cooked, and states an

alwaysEqual constraint with start and end alignment. Each energy saving window is a fixed interval $[s_j, e_j)_{j\in[1,m]}$ that states an alwaysIn constraint:

$$\forall i \in [1,n], \text{alwaysEqual}(oven, a_i, v_i, true, true);$$
$$\forall j \in [1,m], \text{alwaysIn}(oven, s_j, e_j, 0, v_{sup});$$

6.8 Constraint Propagation and Search

6.8.1 Interval Variables

The domain of a interval variable a is represented by a tuple of ranges:

$$([x_{min}, x_{max}], [s_{min}, s_{max}], [e_{min}, e_{max}], [d_{min}, d_{max}])$$

$[x_{min}, x_{max}] \subseteq [0,1]$ represents the domain of the presence status of a. $x_{min} = x_{max} = 1$ means that a will be present whereas $x_{min} = x_{max} = 0$ means that it will not be. $[s_{min}, s_{max}] \subseteq \mathbb{Z}$ represents the conditional domain of the start time of a, that is, the minimal and maximal start time *would a be present*. Similarly, $[d_{min}, d_{max}] \subseteq \mathbb{Z}$ and $[e_{min}, e_{max}] \subseteq \mathbb{Z}$ respectively denote the conditional domain of the length and end time of a.

The interval variable maintains the internal consistency between the temporal bounds $[s_{min}, s_{max}]$, $[e_{min}, e_{max}]$ and $[d_{min}, d_{max}]$ that are due to the relation $d = e - s$. When the temporal bounds become inconsistent (for instance because $s_{min} > s_{max}$ or because $e_{min} - s_{max} > d_{max}$), the interval presence status is automatically set to false ($x_{max} = 0$). Of course, if presence status of the interval is already true ($x_{min} = 1$), this will trigger a failure. Just like other classical variables in CSPs:

- The domain of interval variables can be accessed thanks to accessors: *isPresent*, *isAbsent*, $get[Start|End|Length][Min|Max]$
- It can be modified thanks to *modifiers*: *setPresent*, *setAbsent*, $set[Start|End|Length][Min|Max]$,
- Events can be attached to the change of the domain so as to trigger constraint propagation. In this context, accessors are also available to access the previous value of the domain which is useful for implementing efficient incremental constraints: $getOld[Start|End|Length][Min|Max]$.

The following sections describe how logical, temporal and hybrid n-ary constraints are handled by the engine.

6.8.2 Logical Network

All 2-SAT logical constraints between interval presence statuses of the form $[\neg]\,\text{presenceOf}(a) \vee [\neg]\,\text{presenceOf}(b)$ are aggregated in a *logical network* similar to the implication graph described in [5]. The objectives of the logical network are:

- The detection of inconsistencies in logical constraints.
- An $O(1)$ access to the logical relation that can be inferred between any two intervals (a,b).
- A traversal of the set of intervals whose presence is implied by (resp. implies) the presence of a given interval variable a.
- The triggering of some events as soon as a new implication relation is inferred between two intervals (a,b) in order to wake up constraint propagation.

The above services are incrementally ensured when new logical constraints are added to the network or when the presence status of a interval is fixed. They are used by the propagation - as for instance in the temporal network presented in next section - and the search algorithms.

Nodes $\{l_i\}_{i\in[1,n]}$ in the graph correspond to presence statuses $\text{presenceOf}(a)$ or their negation $\neg\,\text{presenceOf}(a)$ and an arc $l_i \rightarrow l_j$ corresponds to an implication relation between the corresponding boolean statuses. For instance a constraint $\text{presenceOf}(a) \vee \text{presenceOf}(b)$ would be associated with an arc $\neg\,\text{presenceOf}(a) \rightarrow \text{presenceOf}(b)$. As links $l_i \rightarrow l_j$ and $\neg l_j \rightarrow \neg l_i$ are equivalent, for each interval variable a, only one node has to be considered in the network, either the one corresponding to $\text{presenceOf}(a)$ or the one corresponding to $\neg\,\text{presenceOf}(a)$. Fixed presence statuses are skipped from the network as in this case binary constraints are reduced to unary constraints. Strongly connected components of the implication graph are collapsed into a single node representing the logical equivalence class and the transitive closure of the resulting directed acyclic graph is maintained as new arcs are added. The logical network becomes inconsistent when it allows to infer both relations $l_i \rightarrow \neg l_i$ and $\neg l_i \rightarrow l_i$.

The time and memory complexity of the logical network for performing the transitive closure is quadratic with the length of the implication graph. In usual scheduling problems, this length tends to be small compared with the number of interval variables. Typically, this length is related with the depth of the work-breakdown structure.

6.8.3 Temporal Network

All temporal constraints of the form $[starts|ends][Before|At][Start|End]$ are aggregated in a *temporal network* whose nodes $\{t_i\}_{i\in[1,n]}$ represent the set of interval start and end time-points. If t_i is a time-point in the network, we denote $x(t_i)$ the (vari-

able) boolean presence status of the interval variable of t_i and $d(t_i)$ the (variable) date of the time-point. An arc (t_i, t_j, z_{ij}) in the network denotes a minimal delay z_{ij} between the two time-points t_i and t_j *would both time-points be present*, that is: $x(t_i) \wedge x(t_j) \Rightarrow (d(t_i) + z_{ij} \leq d(t_j))$. It is easy to see that all temporal constraints can be represented by one or two arcs in the temporal network. Furthermore, the length of the interval is also represented by two arcs, one between the start and the end time-point labelled with the minimal length and the other between the end and the start time-point labelled by the opposite of the maximal length. Let $d_{min}(t_i)$ and $d_{max}(t_i)$ denote the current conditional bounds on the date of time-point t_i. Depending on whether t_i denotes the start or the end of a interval variable, these bounds are the s_{min}, s_{max} or the e_{min}, e_{max} values stored in the current domain of the interval variable of t_i.

The main idea of the propagation of the temporal network is that for a given arc (t_i, t_j, z_{ij}), whenever the logical network can infer the relation $x(t_i) \Rightarrow x(t_j)$ the propagation on the conditional bounds of t_i (time-bounds *would* t_i *be present*) can assume that t_j will also be present and thus the arc can propagate the conditional bounds from time-point t_j on t_i: $d_{max}(t_i) \leftarrow \min(d_{max}(t_i), d_{max}(t_j) - z_{ij})$. Similarly, if the relation $x(t_j) \Rightarrow x(t_i)$ can be inferred by the logical network then the other half of the propagation that propagates on time-point t_j can be performed: $d_{min}(t_j) \leftarrow \max(d_{min}(t_j), d_{min}(t_i) + z_{ij})$. This observation is crucial: it allows to propagate on the conditional bounds of time-points even when their presence status is not fixed. Of course, when the presence status of a time-point t_i is fixed to 1, all other time-points t_j are such that $x(t_j) \Rightarrow x(t_i)$ and thus, the bounds of t_i can be propagated on all the other time-points. When all time-points are surely present, this propagation boils down to the classical bound-consistency on STNs. When the two time-points of an arc have equivalent presence status, the arc can be propagated in both directions, this is in particular the case for arcs corresponding to interval lengths. When the time-bounds of the extremities of an arc (t_i, t_j, z_{ij}) become inconsistent, the logical constraint $x(t_i) \Rightarrow \neg x(t_j)$ can be added to the logical network.

Figure 6.9 depicts the problem described in [2] modelled in our framework. If the deadline for finishing the schedule is 70, the propagation will infer that the alternative *BuyTube* cannot be present as there is not enough space between the minimal start time of *GetTube* (1) and its maximal end time (50) to accommodate its duration of 50. Note that if the duration of operation *BuyTube* was lower than 49 but if the sum of the durations of operations *SawTube* and *ClearTube* was greater than 49, then the propagation would infer that the alternative $SawTube \rightarrow ClearTube$ is impossible because these two operations have equivalent presence status and thus, the precedence arc between them can propagate in both directions.

Figure 6.10 illustrates another example of propagation. The model consists of a chain of n identical optional operations $\{o_i\}_{i \in [1,n]}$ of duration 10 that, if present, need to be executed before date 25 and are such that $\text{presenceOf}(o_{i+1}) \Rightarrow \text{presenceOf}(o_i)$. Although initially, all operations are optional, the propagation will infer that only the first two operations can be present and will compute the exact conditional minimal start and end times for the two possibly present operations. We are not aware of

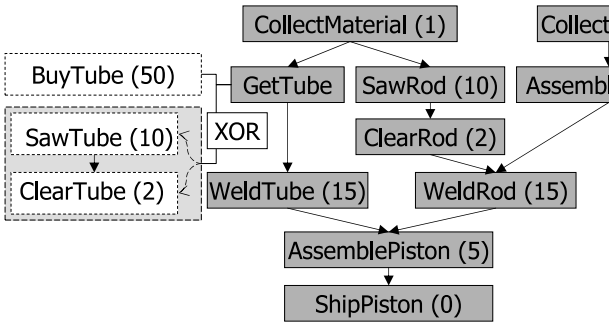

Fig. 6.9 Example of propagation

any other framework that is capable of inferring such type of information on purely optional activities.

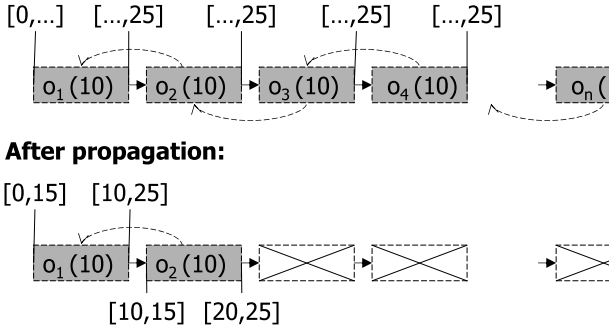

Fig. 6.10 Example of propagation

Most of the classical algorithms for propagating on STNs can be extended to handle conditional time-points. In CP Optimizer, the initial propagation of the temporal network is performed by an improved version of the Bellman-Ford algorithm presented in [7] and the incremental propagation when a time-bound has changed or when a new arc or a new implication relation is detected is performed by an extension of the algorithm for positive cycles detection proposed in [6]. The main difference with the original algorithms is that propagation of the temporal bounds is performed only following those arcs that are allowed to propagate on their target given the implication relations. This propagation allows for instance to infer that a set of optional intervals with equivalent status forming a positive cycle in the temporal network cannot be present.

6.8.4 Interval Composition Constraints

The propagation of both the *span* and *alternative* constraints follow the same pattern. First, the part of the propagation that can be delegated to the logical and temporal networks is transferred to them:

- In the case of a span constraint span$(a_0, \{a_1, ..., a_n\})$, the set of implications presenceOf$(a_i) \Rightarrow$ presenceOf(a_0) are treated by the logical network and the set of arcs $startsBeforeStart(a_0, a_i)$ and $endsBeforeEnd(a_i, a_0)$ by the temporal network.
- In the case of an alternative constraint alternative$(a_0, \{a_1, ..., a_n\})$, the set of implications presenceOf$(a_i) \Rightarrow$ presenceOf(a_0) are treated by the logical network and the set of arcs $startsAtStart(a_0, a_i)$ and $endsAtEnd(a_i, a_0)$ by the temporal network.

The rest of the propagation is performed by the specific constraints themselves:

- The span constraint propagates the fact that when a_0 is present, there must exist at least one present interval variable a_i that starts at the same date as the start of a_0 (and the symmetrical relation for the end).
- The alternative constraint propagates the fact that no two interval variables a_i and a_j can simultaneously execute and maintains the conditional temporal bounds of interval variable a_0 as the constructive disjunction of the conditional temporal bounds of possibly present interval variables a_i. Propagation events on interval variables allow writing efficient incremental propagation for the alternative constraint.

6.8.5 Other Constraints

Classical constraint propagation algorithms have been extended to be able to handle optional interval variables and additional constraints. These algorithms include: time-tabling, precedence graphs or disjunctive constraint [1] and edge-finding variants [20]. For instance, for cumul functions, time-tabling algorithms have been extended to handle alwaysIn constraints on interval variables. For state functions, the time-tabling and disjunctive algorithms have been extended to handle the alignment specifications and the various types of incompatibilities between the alwaysIn, alwaysConstant, alwaysEqual and alwaysNoState constraints.

By default, light propagation algorithms with an average linear complexity are used. A set of inference level parameters is available to the user to perform additional filtering as summarized on Table 6.2.

Model element	Inference level	Filtering algorithms
Sequence variable	**Basic** $\geq$ Medium	Light precedence graph Precedence graph
No-overlap constraint	**Basic** Medium Extended	Timetable + Disjunctive + EF variants
Cumul function expression	**Basic** Medium Extended	Timetable + Disjunctive + EF variants
State function variable	**Basic** $\geq$ Medium	Timetable + Disjunctive

Table 6.2 Constraint propagation algorithms

6.8.6 Search

CP Optimizer implements a robust search algorithm to support the formalism described in this paper. This search was tested on an extensive library of models. It is based on the Self-Adapting Large Neighborhood Search described in [11] that consists of an improvement method that iteratively *unfreezes* and *re-optimizes* a selected fragment of the current solution.

Unfreezing a fragment relies on the notion of Partial Order Schedule [17]. This notion has been generalized to the modeling elements presented in this paper (no-overlap constraint, cumul function expressions and state variables).

The re-optimization of a partially unfrozen solution relies on a tree search using constraint propagation techniques.

6.9 Examples

6.9.1 Flow-Shop with Earliness and Tardiness Costs

6.9.1.1 Problem Description

The first problem studied in the paper is a flow-shop scheduling problem with earliness and tardiness costs on a set of instances provided by Morton and Pentico [15] that have been used in a number of studies including GAs [19] and Large Neighbourhood Search [8]. In this problem, a set of n jobs is to be executed on a set of m machines. Each job i is a chain of exactly m operations, one per machine. All jobs require the machines in the same order that is, the position of an operation in the job determines the machine it will be executed on. Each operation j of a job i is specified by an integer processing time $pt_{i,j}$. Operations cannot be interrupted and each machine can process only one operation at a time. The objective function is

to minimize the total earliness/tardiness cost. Typically, this objective might arise in just-in-time inventory management: a late job has negative consequence on customer satisfaction and time to market, while an early job increases storage costs. Each job i is characterized by its release date rd_i, its due date dd_i and its weight w_i. The first operation of job i cannot start before the release date rd_i. Let C_i be the completion date of the last operation of job i. The earliness/tardiness cost incurred by job i is $w_i \cdot abs(C_i - dd_i)$. In the instances of Morton and Pentico, the total earliness/tardiness cost is normalized by the sum of operation processing times so the global cost to minimize is:

$$\frac{\sum_{i\in[1,n]} \left(w_i \cdot abs(C_i - dd_i)\right)}{W} \text{ where } W = \sum_{i\in[1,n]} \left(w_i \cdot \sum_{j\in[1,m]} pt_{i,j}\right)$$

6.9.1.2 Model

Model 1 - OPL Model for Flow-shop with Earliness and Tardiness Costs

```
 1: using CP;
 2: int n = ...;
 3: int m = ...;
 4: int rd[1..n] = ...;
 5: int dd[1..n] = ...;
 6: float w[1..n] = ...;
 7: int pt[1..n][1..m] = ...;
 8: float W = sum(i in 1..n) (w[i] * sum(j in 1..m) pt[i][j]);
 9: dvar interval op[i in 1..n][j in 1..m] size pt[i][j];
10: dexpr int C[i in 1..n] = endOf(op[i][m]);
11: minimize sum(i in 1..n) w[i]*abs(C[i]-dd[i])/W;
12: subject to {
13:    forall(i in 1..n) {
14:       rd[i] <= startOf(op[i][1]);
15:       forall(j in 1..m-1)
16:          endBeforeStart(op[i][j],op[i][j+1]);
17:    }
18:    forall(j in 1..m)
19:       noOverlap(all(i in 1..n) op[i][j]);
20: }
```

A complete OPL model for this problem is shown in Model 1. The instruction at *line 1* tells the model is a CP model to be solved by CP Optimizer. The section between *line 2* and *line 8* is data reading and data manipulation. The number of jobs n is read from the data file at *line 2* and the number of machines m at *line 3*. A number of arrays are defined to store, for each on the n jobs, the release date (*line 4*), due date (*line 5*), earliness/tardiness cost weight (*line 6*) and, for each machine, the processing time of each operation on the machine (*line 7*). The normalization factor

W is computed at line *line 8*. The model itself is declared between *line 9* and *line 20*. *Line 9* creates a 2-dimensional array of interval variables indexed by the job i and the machine j. Each interval variable represents an operation and is specified with a size corresponding to the operation's processing time. *Line 10* creates one integer expression $C[i]$ for each job i equal to the end of the m^{th} (last) operation of the job. These expressions are used in *line 11* to state the objective function. The constraints are defined between *line 13* and *line 19*. For each job, *line 14* specifies that the first operation of job i cannot start before the job release date whereas precedence constraints between operations of job i are defined at *lines 15-16*. *Lines 18-19* state that for each machine j, the set of operations requiring machine j do not overlap.

6.9.1.3 Experimental Results

Table 6.3 compares the results obtained by the default automatic search of CP Optimizer using the above model (col. CPO) with the best results obtained by various genetic algorithms as reported in [19] (col. GA-best) and the results of the best Large Neighbourhood Search (S-LNS) studied in [8] (col. S-LNS-best). A time limit of one hour was used on a 3GHz processor for CP Optimizer similar to the two hours limit used in [8] on a 1.5GHz processor. The average improvement (using the geometric mean over the ratio $value_{CPO}/value_{Other}$) over the best GA is about 25% whereas the average improvement over the best LNS is more modest (1.7%).

Problem	GA-best	S-LNS-best	*CPO*	Problem	GA-best	S-LNS-best	*CPO*
jb1	0.474	0.191	*0.191*	ljb1	0.279	0.215	*0.215*
jb2	0.499	0.137	*0.137*	ljb2	0.598	0.508	*0.509*
jb4	0.619	0.568	*0.568*	ljb7	0.246	0.110	*0.137*
jb9	0.369	0.333	*0.334*	ljb9	0.739	1.015	*0.744*
jb11	0.262	0.213	*0.213*	ljb10	0.512	0.525	*0.549*
jb12	0.246	0.190	*0.190*	ljb12	0.399	0.605	*0.518*

Table 6.3 Results for Flow-shop Scheduling with Earliness and Tardiness Costs

6.9.2 Satellite Scheduling

6.9.2.1 Problem Description

The second illustrative model is an oversubscribed scheduling problem described in [10]. This model is a generalization of two real-world oversubscribed scheduling domains, the USAF Satellite Control Network (AFSCN) scheduling problem and the USAF Air Mobility Command (AMC) airlift scheduling problem. These two domains share a common core problem structure:

- A problem instance consists of n tasks. In AFSCN, the tasks are communication requests; in AMC they are mission requests.
- A set Res of resources are available for assignment to tasks. Each resource $r \in Res$ has a finite capacity $cap_r \geq 1$. The resources are air wings for AMC and ground stations for AFSCN. The capacity in AMC corresponds to the number of aircraft for a wing; in AFSCN it represents the number of antennas available at the ground station.
- Each task T_i has an associated set Res_i of feasible resources, any of which can be assigned to carry out T_i. Any given task T_i requires 1 unit of capacity (i.e., one aircraft in AMC or one antenna in AFSCN) of the resource $r_j \in Res_i$ that is assigned to perform it. The duration $Dur_{i,j}$ of task T_i depends on the allocated resource r_j.
- Each of the feasible alternative resources $r_j \in Res_i$ specified for a task T_i defines a time window within which the duration of the task needs to be allocated. This time window corresponds to satellite visibility in AFSCN and mission requirements for AMC.
- All tasks are optional; the objective is to minimize the number of unassigned tasks[4].

6.9.2.2 Model

Model 2 - OPL Model for Satellite Scheduling

```
 1: using CP;
 2: tuple Station { string name; key int id; int cap; }
 3: tuple Alternative { string task; int station; int smin; int dur; int emax; }
 4: {Station} Stations = ...;
 5: {Alternative} Alternatives = ...;
 6: {string} Tasks = { a.task | a in Alternatives };
 7: dvar interval task[t in Tasks] optional;
 8: dvar interval alt[a in Alternatives] optional in a.smin..a.emax size a.dur;
 9: maximize sum(t in Tasks) presenceOf(task[t]);
10: subject to {
11:    forall(t in Tasks)
12:       alternative(task[t], all(a in Alternatives: a.task==t) alt[a]);
13:    forall(s in Stations)
14:       sum(a in Alternatives: a.station==s.id) pulse(alt[a],1) <= s.cap;
15: }
```

A complete OPL model for this problem is shown in Model 2 using the AFSCN semantics. The section between *line 2* and *line 6* is data reading and data manipulation.

[4] A second type of model with task priorities is also described in [10]. In the present paper, we focus on the version without task priorities.

A tuple defining ground stations data (with a name, a unique integer identifier and a capacity) is defined at *line 2* and read from the data file at *line 4*. A tuple defining a possible resource assignment for a task (specifying a task, a station, a task minimal start time, a task duration and a task maximal end time) is defined at *line 3* and read from the data file at *line 5*. The set of all tasks *Tasks* is computed at *line 6* as the set of tasks used in at least one possible assignments.

Variables and constraints are defined between *line 7* and *line 15*. *Line 7* defines an array of interval variables indexed by the set of tasks *Tasks*. As tasks are optional and may be left unassigned, each of these interval variable is declared optional so that it can be ignored in the solution schedule. Each of the possible task assignments is defined as an optional interval variable in *line 8*. When present, these interval variables will be of size *dur* and belong to the time window $[smin, emax]$ of the assignment. This is expressed by the `size` and `in` OPL keywords in the interval variable declaration. The objective function is to maximize the number of assigned tasks, that is, the number of present tasks in the schedule; this is specified by a sum of presence constraints at *line 9*.

The constraints *lines 11-12* state that each task, if present, is the alternative among the set of possible assignments for this task, this is modelled by an alternative constraint: if interval $task[t]$ is present, then one and only one of the intervals $alt[a]$ representing a ground station assignment for $task[t]$ will be present and $task[t]$ will start and end together with this selected interval. As specified by the semantics of the alternative constraint, if the task is absent, then all the possible assignments related with this task are absent too. The limited capacity (number of antennas) of ground stations is modelled by *lines 13-14*. For each ground station s, a cumul function is created that represents the time evolution of the number of antennas used by the present assignments on this station s. This is a sum of unit pulse functions $\text{pulse}(alt[a], 1)$. Note that when the assignment $alt[a]$ is absent, the resulting pulse function is the zero function so it does not impact the sum. The resulting sum is constrained to be lower than the maximal capacity cap of the station. An interesting feature of the CP Optimizer model is that it handles optional tasks in a very transparent way: here, the fact that tasks are optional only impacts the declaration of task intervals at *line 7*. The notion of optional interval variable and the handling of absent intervals by the constraints and expressions of the model (here the alternative constraint and the cumul function expressions) allows an elegant modelling of scheduling problems involving optional activities and, more generally, optional and/or alternative tasks, recipes or modes.

6.9.2.3 Experimental Results

Table 6.4 compares the results obtained by the default automatic search of CP Optimizer using the above model (col. CPO) with the TaskSwap (TS) and Squeaky Wheel Optimization (SWO) approaches studied in [10] (col. TS and SWO). Figures represent the average number of unscheduled tasks for each problem set of

the benchmark. The time limit for each instance was fixed to 120s for problem sets $x.1$, 180s for problem sets $x.2$ and 360s for problem sets $x.3$. In average, compared to the best approach described in [10] (SWO), the default automatic search of CP Optimizer assigns 5.3% more tasks.

Problem set	TS	SWO	*CPO*	Problem set	TS	SWO	*CPO*
1.1	30.44	26.60	*27.50*	4.1	3.20	2.00	*1.96*
1.2	114.02	104.72	*98.10*	4.2	13.34	7.90	*7.48*
1.3	87.92	84.52	*86.04*	4.3	16.60	12.46	*9.68*
2.1	11.46	7.80	*7.84*	5.1	3.90	3.80	*3.76*
2.2	45.54	34.26	*30.64*	5.2	32.98	31.98	*31.72*
2.3	33.96	31.18	*32.14*	5.3	46.18	45.22	*44.34*
3.1	2.64	2.32	*2.28*	6.1	1.56	1.28	*1.24*
3.2	15.50	12.82	*11.82*	6.2	11.62	9.56	*8.92*
3.3	32.10	28.58	*24.00*	6.3	25.28	22.60	*19.48*

Table 6.4 Results for Satellite Scheduling

6.9.3 Personal Task Scheduling

6.9.3.1 Problem Description

The third problem treated in this paper is the personal task scheduling problem introduced in [18] and available as an add-on to Google Calendar (`selfplanner.uom.gr/`). It consists of a set of n tasks $\{T_1, ..., T_n\}$. Each task T_i has a duration denoted dur_i. All tasks are considered preemptive, i.e. they can be split into parts that can be scheduled separately. The decision variable p_i denotes the number of parts in which the i^{th} task has been split, where $p_i \geq 1$. T_{ij} denotes the j^{th} part of task T_i, $1 \leq j \leq p_i$. The sum of the durations of all parts of a task T_i must equal its total duration dur_i. For each task T_i, a minimum and maximum allowed duration for its parts, $smin_i$ and $smax_i$, as well as a minimum allowed temporal distance between every pair of its parts, $dmin_i$ are given. Depending on the values of $smax_i$ and $smin_i$ and the overall duration of the task dur_i, implicit constraints are imposed on p_i. For example, if $dur_i < 2 * smin_i$, then necessarily $p_i = 1$ and task T_i is non-preemptive. Each task T_i is associated with a domain $D_i = [s_{i1}, e_{i1}] \cup [s_{i2}, e_{i2}] \cup \cdots \cup [s_{iF_i}, e_{iF_i}]$, consisting of a set of F_i time windows within which all of its parts have to be scheduled. We denote respectively $L_i = s_{i1}$ and $R_i = e_{iF_i}$ the leftmost and rightmost values of domain D_i. A set of m locations is given, $Loc = \{l_1, l_2, \cdots, l_m\}$ as well as a 2-dimensional matrix *Dist* with their temporal distances represented as non-negative integers. Each task T_i has its own spatial reference, $loc_i \in Loc$, denoting the location where the user should be in order to execute each part of the task. A set of ordering constraints, denoted $\prec (T_i, T_j)$ between some pairs of tasks is also defined,

meaning that no part of task T_j can start its execution until all parts of task T_i have finished their execution. Time preferences are expressed for each task T_i. Five types of preference functions are available; they are depicted on Figure 6.11:

f_{-2} Execute as much as possible of task T_i after a date d.

f_{-1} Execute as much as possible of task T_i as late as possible.

f_0 No preference.

f_1 Execute as much as possible of task T_i as early as possible.

f_2 Execute as much as possible of task T_i before a date d.

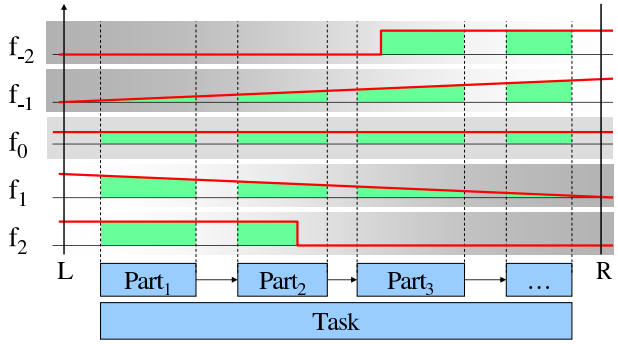

Fig. 6.11 Preference functions

For a given preference function f_i associated with a task T_i that is split into p_i parts $P_{i,1},...,P_{i,p_i}$, the satisfaction related with the execution of task T_i is computed as:

$$\text{satisfaction}(T_i) = \sum_{j=1}^{p_i} \sum_{t \in P_{i,j}} f_i(t)$$

It is to be noted that functions f_i are normalized in the interval [0,1] in such a way that an upper bound for the satisfaction for a task T_i is 1.

6.9.3.2 Model

A complete OPL model for the personal task scheduling problem is shown in Model 3. The section between *line 2* and *line 16* is data reading and data manipulation. A tuple representing a task description is declared at *line 2*, it specifies a unique integer task identifier, the location of the task, the task duration, the minimal and maximal duration of task parts, the minimal delay between two consecutive task parts, an identifier of the type of preference function for the task in $\{-2,-1,0,1,2\}$, the threshold date in case preference function is of type f_{-2} or f_2 and two sets of integers *ds* and *de* respectively representing the start and end dates of the intervals

Model 3 - OPL Model for Personal Task Scheduling

```
1: using CP;
2: tuple Task { key int id; int loc; int dur; int smin; int smax; int dmin; int f; int date; {int} ds;
   {int} de; }
3: {Task} Tasks = ...;
4: tuple Distance { int loc1; int loc2; int dist; };
5: {Distance} Dist = ...;
6: tuple Ordering { int pred; int succ; };
7: {Ordering} Orderings = ...;
8: int L[t in Tasks] = min(x in t.ds) x;
9: int R[t in Tasks] = max(x in t.de) x;
10: int S[t in Tasks] = R[t]-L[t];
11: tuple Part { Task task; int id; }
12: {Part} Parts = { <t,i> | t in Tasks, i in 1 .. t.dur div t.smin };
13: tuple Step { int x; int y; }
14: sorted  {Step} Steps[t in  Tasks] =
15:    {<x,0> | x in t.ds} union {<x,1> | x in t.de};
16: stepFunction holes[t in Tasks] = stepwise(s in Steps[t]) {s.y -> s.x; 0};
17: dvar interval tasks[t in Tasks] in 0..500;
18: dvar interval a[p in Parts] optional size p.task.smin..p.task.smax;
19: dvar sequence seq in all(p in Parts) a[p] types all(p in Parts) p.task.loc;
20: dexpr float satisfaction[t in Tasks] = (t.f==0)? 1 :
21:    (1/t.dur)* sum(p in Parts: p.task==t)
22:       (t.f==-2)? maxl(endOf(a[p]),t.date)-maxl(startOf(a[p]),t.date) :
23:       (t.f==-1)? lengthOf(a[p])*(R[t]-(startOf(a[p])+endOf(a[p])-1)/2)/S[t] :
24:       (t.f== 1)? lengthOf(a[p])*((startOf(a[p])+endOf(a[p])-1)/2-L[t])/S[t] :
25:       (t.f== 2)? minl(endOf(a[p]),t.date)-minl(startOf(a[p]),t.date) : 0;
26: maximize sum(t in Tasks) satisfaction[t];
27: subject to {
28:    forall(p in Parts) {
29:       forbidExtent(a[p], holes[p.task]);
30:       forall(s in Parts: s.task==p.task && s.id==p.id+1) {
31:          endBeforeStart(a[p], a[s], p.task.dmin);
32:          presenceOf(a[s]) => presenceOf(a[p]);
33:       }
34:    }
35:    forall(t in Tasks) {
36:       t.dur == sum(p in Parts: p.task==t) sizeOf(a[p]);
37:       span(tasks[t], all(p in Parts: p.task==t) a[p]);
38:    }
39:    forall(o in Orderings)
40:       endBeforeStart(tasks[<o.pred>], tasks[<o.succ>]);
41:    noOverlap(seq, Dist);
42: }
```

$[s_i, e_i]$ of the task domain. The set of tasks is read from the data file at *line 3*. A triplet representing the temporal distance between two locations is declared at *line 4* and the transition distance matrix represented as a set of such triplets is read from the data file at *line 5*. A tuple storing an ordering constraint is defined on *line 6* and a set of such tuples is read from the data at *line 7*. *Lines 8-10* respectively compute, for each task t the leftmost value, rightmost value and diameter of the task domain. A tuple representing the i^{th} part of a task is defined at *line 11* and the total set of possible parts is computed at *line 12* considering that for each task of duration *dur* and minimal part duration *smin*, the maximal number of parts is $\lfloor dur/smin \rfloor$. *Lines 13-16* define a step function $holes[t]$ for each task t that is equal to 1 in the domain of t and to 0 everywhere else.

Variables and constraints are defined between *lines 17 and 42*. An array of interval variables, one interval $task[t]$ for each task t, is declared at *line 17*; each task is constrained to end before the schedule horizon (500 in the benchmark). *Line 18* defines an optional interval variable for each possible task part with a minimal and a maximal size given by *smin* and *smax*. A sequence variable is created at *line 19* on the set of all parts p, each part being associated with an integer type in the sequence corresponding to the location of the part. The satisfaction expression for each task t is modelled on *lines 20-25* depending on the preference function type; it uses the OPL conditional expression `c?e1:e2` where c is a boolean condition and $e1$ is the returned expression if c is true and $e2$ the returned expression if c is false. The normalization factors are the ones used in [18][5]. The objective function, as defined on *line 26* is to maximize the sum of all tasks satisfaction.

The constraints on *line 29* forbid any part of a task t to overlap a point where the step function $holes[t]$ is zero; this will constrain each task part to be executed in its domain. Constraints on *lines 31-32* state that the set of parts of a given task t forms a chain of optional intervals with minimum separation time *dmin* among which only the first ones will be executed, that is, each part $a[p]$ if present is constrained to be executed before its successor part $a[s]$ and the presence of part $a[s]$ implies the presence of part $a[p]$. Constraints on *line 36* state that the total duration of the part of a task must equal the specified task duration *dur*. Note that when part $a[p]$ is absent, by default the value of sizeOf$(a[p])$ is 0. *Line 37* constrains each task t to span its parts, that is to start at the start of first part and to end with the end of the last executed part. Ordering constraints are declared on *line 40* whereas *line 41* states that task parts cannot overlap and that they must satisfy the minimal transition distance between task locations defined by the set of triplets *Dist*.

[5] The objective expression being quite complex, we used the solution checker provided with the instances to check that the constraints and objective function of our model are equivalent to the ones used in [18].

6.9.3.3 Experimental Results

Table 6.5 compares the results obtained by the default automatic search of CP Optimizer using the above model (col. CPO) and a time limit of 60s for each problem with the Squeaky Wheel Optimization (SWO) approach implemented in SelfPlanner [18] (col. SWO). CP Optimizer finds a solution to more problems than the approach described in [18]: the SWO could not find any solution for the problems with 55 tasks whereas the automatic search of CP Optimizer solves 70% of them. Furthermore, SWO could not find any solution to 4 of the smaller problems with 50 tasks whereas CP Optimizer solves them all but for problem 50-2. On problems where SWO finds a solution, the average task satisfaction (average of the ratio between the total satisfaction and the number of tasks) is 78% whereas it is 87.8% with CP Optimizer. It represents an improvement of about 12.5% in solution quality.

#	SWO	*CPO*	#	SWO	*CPO*	#	SWO	*CPO*	#	SWO	*CPO*
15-1	12.95	*14.66*	30-6	28.09	*29.28*	40-1	24.72	*28.95*	45-6	32.70	*37.35*
15-2	12.25	*13.16*	30-7	23.80	*24.20*	40-2	23.48	*32.07*	45-7	32.40	*35.77*
15-3	13.71	*13.90*	30-8	24.06	*26.89*	40-3	33.57	*37.74*	45-8	31.79	*35.23*
15-4	11.57	*12.55*	30-9	23.42	*24.86*	40-4	31.46	*35.45*	45-9	35.79	*38.86*
15-5	12.64	*14.67*	30-10	22.04	*27.18*	40-5	28.05	*34.21*	45-10	32.78	*40.68*
15-6	14.30	*14.63*	35-1	28.80	*31.56*	40-6	29.46	*34.01*	50-1	42.04	*43.53*
15-7	13.08	*14.46*	35-2	29.17	*32.33*	40-7	33.13	*37.51*	50-2	×	×
15-8	11.46	*12.37*	35-3	27.84	*28.58*	40-8	29.72	*34.90*	50-3	×	*37.17*
15-9	11.44	*11.61*	35-4	26.64	*29.67*	40-9	33.03	*36.89*	50-4	×	*36.52*
15-10	12.07	*13.51*	35-5	25.15	*32.13*	40-10	30.28	*34.19*	50-5	34.25	*43.55*
30-1	24.17	*29.13*	35-6	26.12	*29.49*	45-1	37.42	*42.90*	50-6	38.32	*41.87*
30-2	24.69	*27.55*	35-7	29.28	*31.69*	45-2	33.97	*39.71*	50-7	32.59	*42.48*
30-3	25.61	*26.53*	35-8	25.71	*30.07*	45-3	35.44	*39.40*	50-8	34.70	*43.67*
30-4	27.13	*28.49*	35-9	23.74	*29.60*	45-4	33.02	*37.41*	50-9	×	*42.75*
30-5	23.89	*26.46*	35-10	30.70	*33.41*	45-5	30.83	*36.65*	50-10	37.46	*41.84*
55-1	×	*36.84*	55-4	×	*40.36*	55-7	×	×	55-10	×	×
55-2	×	*38.56*	55-5	×	*42.70*	55-8	×	*45.27*			
55-3	×	×	55-6	×	*35.92*	55-9	×	*42.14*			

Table 6.5 Results for Personal Task Scheduling

6.10 Conclusion

The algebraic model presented in this paper has been implemented in CP Optimizer and is available in C++, Java, C# as well as in OPL [12], [13]. In complement of the notion of *optional interval variable* that considerably simplifies the modelling of complex scheduling structures (optional activities, alternative modes or recipes), a set of global variables and expressions are used for each aspect of a scheduling prob-

lem: *sequence variables* for interval sequencing, *cumul function expressions* for cumulative reasoning and *state function variables* for representing the time evolution of a state variable. A powerful set of constraints on these variables and expressions is provided. As all these constraints handle the optional status of interval variables, they can be posted even on optional or alternative parts of the schedule to effectively prune part of the search space.

The clear separation between (1) the structure of scheduling problems captured with composition constraints on optional interval variables such as *span* and *alternative* and (2) the resource constraints expressed as mathematical concepts such as sequences or functions results in very simple, elegant and concise models. For instance, the model for the classical Multi-Mode Resource-Constrained Project Scheduling Problem with both renewable and non-renewable resources is less than 50 lines long in OPL including data manipulation.

The automatic search algorithm has shown to be robust and efficient for solving a large panel of models as shown in [11].

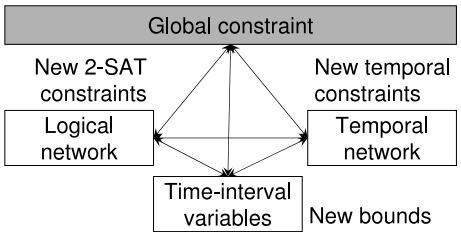

Fig. 6.12 General approach for propagation

As illustrated on Figure 6.12, coupling global constraints with logical and temporal network information allows performing stronger conjunctive deductions on the bounds of interval variables and inferring new logical and temporal relations. Future work will consist in enhancing the current global constraints following this general pattern.

References

1. Baptiste, P., Le Pape, C., Nuijten, W.: Constraint-Based Scheduling. Applying Constraint Programming to Scheduling Problems. Kluwer Academics (2001)
2. Barták, R., Čepek, O.: Temporal networks with alternatives: Complexity and model. In: Proc. FLAIRS-2007 (2007)
3. Beck, J.C., Fox, M.S.: Scheduling alternative activities. In: Proc. AAAI-99 (1999)
4. Bessière, C., Régin, J.C.: Arc consistency for general constraint networks: preliminary results. In: Proc. IJCAI'97, pp. 398–404 (1997)
5. Brafman, R.I.: A simplifier for propositional formulas with many binary clauses. In: Proc. IJCAI-01, pp. 515–522 (2001)

6. Cesta, A., Oddi, A.: Gaining efficiency and flexibility in the simple temporal problem. In: Proc. TIME-96 (1996)
7. Cherkassky, B., Goldberg, A., Radzic, T.: Shortest paths algorithms: Theory and experimental evaluation. Mathematical Programming **73**, 129–174 (1996)
8. Danna, E., Perron, L.: Structured vs. unstructured large neighborhood search: A case study on job-shop scheduling problems with earliness and tardiness costs. In: Proc. CP 2003, pp. 817–821 (2003)
9. ILOG: ILOG CP Optimizer (2008). URL `Web-page:http://www.ilog.com/products/cpoptimizer/`. Http://www.ilog.com/products/cpoptimizer/
10. Kramer, L.A., Barbulescu, L.V., Smith, S.F.: Understanding Performance Tradeoffs in Algorithms for Solving Oversubscribed Scheduling. In: Proc. 22nd AAAI Conference on Artificial Intelligence (AAAI-07), pp. 1019–1024 (2007)
11. Laborie, P., Godard, D.: Self-Adapting Large Neighborhood Search: Application to single-mode scheduling problems. In: Proc. MISTA-07 (2007)
12. Laborie, P., Rogerie, J.: Reasoning with Conditional Time-intervals. In: Proc. FLAIRS-08 (2008)
13. Laborie, P., Rogerie, J., Shaw, P., Vilím, P.: Reasoning with Conditional Time-intervals, Part II: an Algebraical Model for Resources. In: Proc. 22th International FLAIRS Conference (FLAIRS 2009) (2009)
14. Moffitt, M.D., Peintner, B., Pollack, M.E.: Augmenting disjunctive temporal problems with finite-domain constraints. In: Proc. AAAI-2005 (2005)
15. Morton, T., Pentico, D.: Heuristic Scheduling Systems. Wiley (1993)
16. Nuijten, W., Bousonville, T., Focacci, F., Godard, D., Le Pape, C.: Towards an industrial manufacturing scheduling problem and test bed. In: Proc. PMS-2004, pp. 162–165 (2004)
17. Policella, N., Cesta, A., Oddi, A., Smith, S.: Generating robust schedules through temporal flexibility. In: Proc. ICAPS 04. Whistler, Canada (2004)
18. Refanidis, I.: Managing personal tasks with time constraints and preferences. In: Proc. 17th International Conference on Automated Planning and Scheduling Systems (ICAPS-07) (2007)
19. Vázquez, M., Whitley, L.D.: A comparison of genetic algorithms for the dynamic job shop scheduling problem. In: Proc. GECCO 2000 (2000). URL `citeseer.ist.psu.edu/528131.html`
20. Vilím, P.: Global constraints in scheduling. Ph.D. thesis, Charles University in Prague, Faculty of Mathematics and Physics (2007)