# IBM ILOG CP optimizer for scheduling

Philippe Laborie, Jérôme Rogerie, Paul Shaw & Petr Vilím

**Constraints**
An International Journal

ISSN 1383-7133

Constraints
DOI 10.1007/s10601-018-9281-x

Springer

Your article is protected by copyright and all rights are held exclusively by Springer Science+Business Media, LLC, part of Springer Nature. This e-offprint is for personal use only and shall not be self-archived in electronic repositories. If you wish to self-archive your article, please use the accepted manuscript version for posting on your own website. You may further deposit the accepted manuscript version in any repository, provided it is only made publicly available 12 months after official publication or later and provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be accompanied by the following text: "The final publication is available at link.springer.com".

Springer

Author's personal copy

Constraints
https://doi.org/10.1007/s10601-018-9281-x

# IBM ILOG CP optimizer for scheduling

## 20+ years of scheduling with constraints at IBM/ILOG

**Philippe Laborie[1] · Jérôme Rogerie[1] · Paul Shaw[2] · Petr Vilím[3]**

© Springer Science+Business Media, LLC, part of Springer Nature 2018

**Abstract** IBM ILOG CP Optimizer is a generic CP-based system to model and solve scheduling problems. It provides an algebraic language with simple mathematical concepts to capture the temporal dimension of scheduling problems in a combinatorial optimization framework. CP Optimizer implements a model-and-run paradigm that vastly reduces the burden on the user to understand CP or scheduling algorithms: modeling is by far the most important. The automatic search provides good performance out of the box and it is continuously improving. This article gives a detailed overview of CP Optimizer for scheduling: typical applications, modeling concepts, examples, automatic search, tools and performance.

**Keywords** Scheduling · Constraint programming · Modeling · Automatic search

This article belongs to the Topical Collection: *20th Anniversary Issue*

✉ Philippe Laborie
laborie@fr.ibm.com

Jérôme Rogerie
rogerie@fr.ibm.com

Paul Shaw
paul.shaw@fr.ibm.com

Petr Vilím
petr_vilim@cz.ibm.com

[1] IBM France, 9 rue de Verdun, BP 85, 94253 Gentilly, France

[2] IBM France, 1681 Route des Dolines, Les Taissounières, 06560 Valbonne, France

[3] IBM Czechia, V Parku 2294/4, Praha 4 Chodov, 14800, Czech Republic

Published online: 19 March 2018

Author's personal copy

## 1 Introduction

Scheduling is one of the most successful application areas of Constraint Programming (CP) and it was already the case at the time of the first issue of the *Constraints* journal in 1996, 20 years ago. The *cumulative* global constraint was already available in CHIP [1] and our team at ILOG was developing ILOG Scheduler [39] which was then used for years to solve industrial scheduling problems. These two systems can be considered as the root of two different approaches for integrating scheduling in CP.

The first approach complements classical CP on integer variables with a set of global constraints useful for modeling scheduling problems such as the different variants of the *cumulative* or the *disjunctive* constraints in CHIP [1], Choco [43] or Gecode [12]. An advantage of this approach is that it completely fits into the CP box: scheduling is supported by just a number of additional global constraints. A difficulty is that modeling complex scheduling problems in this framework usually involves several global constraints glued together (or to the rest of the model) by reified constraints. This may result in fairly intricate models that are hard to understand and hard to maintain. Moreover, it is difficult to design efficient and robust generic algorithms to solve such models because most of the temporal structure is lost when expressing the scheduling problem using (non-temporal) integer variables.

An alternative approach was taken in ILOG Scheduler in the early '90s with the development of a completely *scheduling-dedicated* modeling language on top of classical integer CP (ILOG Solver) with scheduling concepts such as different types of *activities*, *resources* and *resource constraints*. Over the years, ILOG Scheduler was successfully applied to many industrial scheduling problems but at a certain cost: (1) growing complexity of the modeling language due to successive extensions,[1] (2) lack of usable automatic search and (3) difficulty to efficiently handle some important aspects of scheduling problems such as optional activities or alternative recipes and modes.

In 2007, our team decided to completely redesign our CP tools in light of its more than 15-year-old experience in applying constraint-based scheduling to industrial applications. In particular, we put a strong emphasis on a model-and-run development process and on convergence with mathematical programming (MP) tools as advocated in [44]. IBM ILOG CP Optimizer (CP Optimizer) was born. Since its very first version it was designed with the following requirements in mind:

- It should be accessible not only to CP experts but also to software engineers and to people used to MP;
- It should be simple, non-redundant and use a minimal number of concepts so as to reduce the learning curve for new users;
- It should naturally fit into a combinatorial optimization paradigm with clearly identified decision variables, expressions, constraints and objective function;
- It should be expressive enough to handle complex industrial scheduling applications, which often are over-constrained, involve optional activities, alternative recipes, non-regular objective functions, *etc.*
- It should provide a robust and efficient automatic search algorithm so that, like in MP, the user can focus on the declarative model without needing to write a complex search algorithm.

[1]As an indication of the complexity of the modeling language, the reference manual of the last version of ILOG Scheduler was about 1000 pages long.

Author's personal copy

This article gives a fairly complete overview of CP Optimizer for scheduling as of today. Section 2 summarizes some recent applications using CP Optimizer. Section 3 describes how CP Optimizer extends the classical Constraint Programming (CP) framework by introducing a few new types of decision variables, expressions and constraints for scheduling. Four complete examples are provided in Section 4. Both in Sections 3 and 4 we hope to make it clear that the CP Optimizer scheduling concepts are not just syntactic sugar on top of classical integer CP: although the model examples given in these sections are quite simple, the reader is invited to think how these small problems could be reformulated in other CP frameworks. Section 5 introduces the different ways to parametrize the automatic search. The reader interested in what happens under the hood of the resolution engine will refer to Section 6. Section 7 summarizes some performance results. Section 8 presents the different APIs and languages through which CP Optimizer is available, together with a set of tools provided for speeding up the development of efficient models.

## 2 Applications

It is clearly not possible to provide an exhaustive description of all industrial applications that are using CP Optimizer. Nevertheless, if we focus only on recent articles published in the last 18 months, we can get an idea of the diversity of scheduling problems successfully addressed by CP Optimizer. These problems cover several domains:

- *Manufacturing* is of course an important area of application, both for discrete scheduling, for instance in the context of semiconductor manufacturing [18], or for disjunctive resource sequencing [14].
- *Computer and network scheduling* is a domain with several successful applications of CP Optimizer, in particular for scheduling MapReduce jobs on the cloud [17, 19] or Ethernet communication [9].
- Recent *space* applications range from scheduling satellite observations [10, 34] to scheduling the training of astronauts for ISS missions [38].
- In *automated systems* scheduling, applications of CP Optimizer have been reported for scheduling robots in the context of retirement homes [2, 50] as well as for more general robot planning problems [3] and for planning UAV activities [48].
- In *civil engineering* for scheduling linear construction projects such as highways, bridges or pipelines [47].
- *Transportation* is also an important field with recent applications such as railway scheduling [5], supply-delivery in the petrochemical industry [15], snow plow scheduling [23], bike sharing [22], dredge fleet scheduling [46] or team orienteering problems that have applications in technicians routing and disaster relief routing [13].
- In *port management* for scheduling container terminal operations [25] and in particular for berth allocation [45].
- In *manpower scheduling* for scheduling vacations for a railway operator [24].

Interestingly, several of these very recent use-cases of CP Optimizer were developed by non-CP experts and published in application-oriented conferences and journals. Therefore we think that the main goal of CP Optimizer, which is to make CP Optimizer accessible not only to CP experts, is bearing fruit.

Beside the technical articles published in the academic community, here are some companies that are using CP Optimizer to solve their scheduling problems:

Author's personal copy

- Container terminals and shipping lines: Navis, Yantian International Terminal
- Electronics: KCE Electronics, Nanya Technology Corporation, STMicroelectronics, Taiwan Semiconductor Manufacturing Company
- Robotics: Transcriptic (life sciences)
- Manpower planning: a leader in Integrated Facility Management
- Aerospace: Dassault Aviation, Embraer
- Manufacturing: Ajover (plastic products), Danieli (metallurgy), TAL Group (textile)

## 3 Modeling

CP Optimizer provides a set of modeling features suitable for applications dealing with scheduling over time [21, 35, 37]. Although time points in CP Optimizer are represented as integers, the very wide range of time points means that time is effectively continuous.[2]

Most scheduling applications consist of scheduling activities, tasks, or operations that have a start and an end time. In CP Optimizer, such time intervals are represented by *interval variables* (see Section 3.2). Some of the basic constraints on interval variables allow:

- to limit the possible positions of an interval variable (forbidden start/end or "extent" values) (see Section 3.2.2),
- to specify precedence relations between two interval variables (see Section 3.4)
- to relate the position of an interval variable with one of a set of interval variables (spanning, alternative) (see Section 3.5).

An important characteristic of scheduling problems is that time intervals may be optional. For example, some tasks may be non-mandatory and can be left unperformed. Thus, solving a problem with optional interval variables also means deciding which optional interval variables will be *present* and which interval variables will be kept unperformed (*absent*).

Optionality is represented by a Boolean presence status associated with each interval variable. Logical relations can be expressed between the presence of interval variables. For instance, it is possible to state that whenever interval $a$ is present then interval $b$ must also be present (see Section 3.3).

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

- The evolution of a disjunctive resource over time is described by the sequence of intervals that represent the activities executing on the resource. For that, CP Optimizer introduces an *interval sequence variable*. Constraints and expressions are available to control the sequencing of a set of interval variables (see Section 3.6).
- The evolution of a cumulative resource often needs a description of how the accumulated usage of a resource evolves over time. For that purpose, CP Optimizer provides

[2] In fact, time values could have been implemented as floating point variables in CP Optimizer as (1) they are never used to index any internal data structure and (2) the domain of time variables is never enumerated by the propagation, and only very exceptionally by the search (during optimality proof for some specific cases). In general, unlike time-indexed MIP models, the complexity of solving a CP Optimizer scheduling model does not depend on the choice of the time unit.

Author's personal copy

*cumul function expression* that can be used to constrain the temporal evolution of the resource usage (see Section 3.7).
- The evolution of a resource, the state of which can vary over time, is represented by a *state function*. The dynamic evolution of a state function can be controlled via transition times between states, and constraints are available for specifying conditions on the state function that must be satisfied during fixed or variable intervals (see Section 3.8).

Some classical cost functions in scheduling are earliness/tardiness costs, makespan, and activity 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 wide range of scheduling cost functions that are efficiently exploited by the automatic search (see Section 3.2.3).

Note that even though models are often used to represent scheduling problems with activities and resources, the modeling concepts provided in CP Optimizer are mathematical abstractions of the reality. This is why we decided to use mathematical terminology for these concepts - like *intervals* and *functions* - rather than *activities* and *resources*. Moreover, in many problems there is no one-to-one mapping between activities and interval variables or between resources and functions. For instance:

- An interval variable may be used to represent a period of time when a resource is idle, which is something that hardly qualifies for being called an activity.
- A physical activity can be represented by several interval variables in the case of preemptivity or a stepwise resource usage profile.
- Optional interval variables are often used to represent the *possible* allocation of an activity to a resource, not the activity itself.
- Quite often, a complex physical resource needs the combination of several constructs for its formulation (sequence variable, cumul and/or state function and even sometimes some interval variables).

In this section we use the modeling language OPL [51] to illustrate the different elements of the model, however the same modeling concepts are also available in C++, Python, Java and C# (see Section 8.1).

## 3.1 Constant structures

The description of scheduling problems often involves certain constant data structures, e.g. known *functions* of time or *transition matrices*. This section describes the constant structures provided by CP Optimizer.

### *3.1.1 Piecewise-linear functions*

Piecewise-linear functions are typically used to model a known function of time, such as the cost incurred for completing an activity at a given date t (see Section 3.2.3). No assumption is made on piecewise-linear functions, in particular these functions can be non-convex or non-continuous.

An example of a piecewise-linear function that would typically be used to model an earliness/tardiness cost is shown in Fig. 1. In CP Optimizer (OPL), this function would be defined as:

```
pwlFunction F = piecewise{-0.015->100; 0->200; 1.5->200; 0.01} (100,0);
```

Author's personal copy

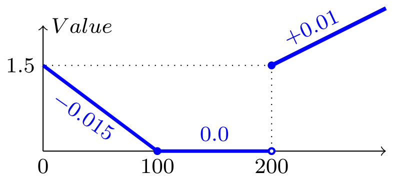

**Fig. 1** Example of a piecewise-linear function $F$

In curly braces we have a number of `slope->terminating-time-value` pairs together with a final slope going off to infinity (the first slope is assumed to come from negative infinity). Finally an anchor point `(t, F(t))` is given to fix the function. Discontinuities are modeled by repeating a terminating time value. In this case, the second "slope" is interpreted as a step. (See the 1.5 unit step at $t = 200$ in the above example.)

If $t$ is a decision variable or an expression, one can directly use the expression $F(t)$ in the model. Note that without piecewise linear functions, $F(t)$ would have to be written as:

$$(t < 100) * (1.5 - 0.015 * t) + (t \geq 200) * (0.01 * t - 0.5)$$

which is not natural and also leads to less propagation because of multiple occurrences of $t$ in the expression.

### *3.1.2 Step functions*

A stepwise function is a special case of piecewise linear function $F$ where all slopes are equal to 0 and the domain and image of $F$ are integer. Stepwise functions are typically used to model the efficiency of a resource over time (see Section 3.2.2). An example of step function is shown in Fig. 2. In OPL, this function would be defined as:

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

### *3.1.3 Transition distances*

Transition distances (or transition matrices) are typically used to model the sequence dependent setup time that must elapse between two consecutive activities on a disjunctive resource depending of the type of the activities (see Section 3.6). Transition distances are specified as a matrix. In OPL this is via a sparse `<row, column, value>` syntax. Any missing entries are considered to be zero.

For instance, the following transition distance in OPL could be used to define a setup-time of 10 between activities of type 0 and activities of type 1.

```
tuple triplet { int type_i; int type_j; int v_ij; };
{triplet} M = { <0,0,0>, <0,1,10>, <1,0,15>, <1,1,5> } ;
```

## 3.2 Interval variables

### *3.2.1 Basic interval variables*

Informally speaking, an interval variable represents an interval of time of interest in the schedule (a task or an activity is carried out, a resource is being used, there is some waiting

Author's personal copy

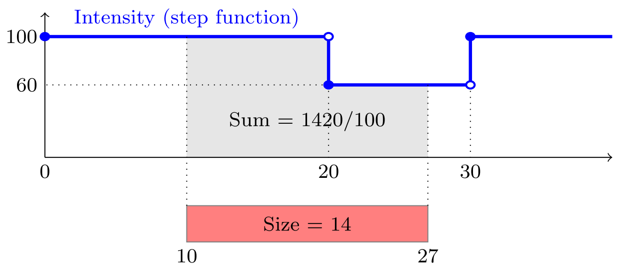

**Fig. 2** Example of interval intensity function

time between two activities, *etc.*) and whose position in time is an unknown of the scheduling problem. An interval is characterized by a start value and an end value. An important additional feature of interval variables is the fact that they can be optional; in other words, 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 where each mode specifying a particular combination of resources,
- alternative processes for executing a given production order where 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 decision variable whose domain $\mathrm{dom}(a)$ is a subset of $\{\perp\} \cup \{[s, e) | s, e \in \mathbb{Z}, s \leq e\}$.[3] As any decision variable, an interval variable is said to be *fixed* if its domain is reduced to a singleton, *i.e.* if $a$ denotes a fixed interval variable then:

- either interval is *absent*: $a = \perp$;
- or interval is *present*: $a = [s, e)$. In this case, $s$ and $e$ are respectively the *start* and *end* of the interval and $l = e - s$ its *length*.

Note than every interval variable has an associated additional attribute which we call its *size* which is never greater than its *length*, but is related to it. We will expand on this in the next section.

Absent interval variables have special meaning. Informally speaking, an absent interval variable is not considered by any constraint in which it is involved. For example, if an absent interval variable is used in a `noOverlap` constraint (see Section 3.6), the constraint will behave as if the interval was never specified to the constraint. If an absent interval variable

[3]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.

Author's personal copy

$a$ is used in a precedence constraint between interval variables $a$ and $b$ (see Section 3.4) 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 $a$ is present and such that $a = [s, e)$, we will denote $s(a)$ its integer start value $s$, $e(a)$ its integer end value $e$ and $l(a)$ its non-negative length defined as $e(a) - s(a)$. The presence status $x(a)$ will be `true`. For a fixed interval variable $a$ that is absent, the presence status $x(a)$ is `false` and the start, end, and length are undefined (and cannot be used).

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 $\mathrm{dom}(a) = \{\perp\}$, present when $\perp \notin \mathrm{dom}(a)$ and optional otherwise.

*Example* The OPL lines below define three interval variables. Variable $a$ defines an interval variable that uses the default domain: the interval must be present, with an unfixed length (possibly 0) and non-negative start and end values. The distinction between *size* and *length* will be explained in the next section but here *size* and *length* are equivalent. Variable $b$ represents a present interval variable with a possible length in the range [0, 10]. Variable $c$ represents an optional interval variable of length 10 with an initial domain that constrains the interval (if present) to start after -1000 and end before 1000. Note that, as illustrated by variable $c$, start and end times can be negative, this is typically useful to represent activities that started in the past in case time 0 represents the present time.

```
dvar interval a;
dvar interval b size 0..10;
dvar interval c optional in -1000..1000 size 10;
```

### 3.2.2 *Intensity, size and forbidden values*

Sometimes the intensity of *work* is not the same during the whole interval. For instance let us 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 represents 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, CP Optimizer provides a range for the size of an interval variable and an integer stepwise intensity function (see Section 3.1.1) that can be used when constructing the interval variable. For a fixed present interval $a$ 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):

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

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 assumed to be the constant full intensity function ($\forall t, F(t) = G$) so in that case $sz(a) = l(a)$. Note that the size (like the start, end and length) is not defined for absent intervals.

Author's personal copy

Figure 2 depicts an interval variable of size 14 and an associated intensity function $F$. A valid solution is represented where the interval starts at 10 and ends at 27, so its length is 17. In this case,

$$\int_{s(a)}^{e(a)} \frac{F(t)}{G}\, dt = \frac{1420}{100} = 14.2$$

Intensity functions, if any, are specified at the creation of the interval variable. For instance the interval variable depicted in Fig. 2 could be created as:

```
stepFunction F = stepwise{100->20; 60->30; 100};
dvar interval a size 14 intensity F;
```

It may be necessary to state that an interval cannot start at, end at, or overlap with a fixed set of dates. CP Optimizer provides a set of constraints for modeling this. Let $a$ denote an interval variable and $F$ an integer step function.

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

The combination of intensity functions and the above constraints make it possible to efficiently model a wide range of resource calendar constraints.

### *3.2.3 Expressions over interval variables*

Numeric expressions defined on interval variables can be used to define a term for the cost function or connect interval variables with other integer and floating point expressions.

Integer expressions `startOf`, `endOf`, `lengthOf`, and `sizeOf` provide access to the different attributes of an interval variable. However special care must be taken for optional intervals: in this case an integer "escape" value must be specified which represents the value of the expression should the interval be absent. If this value is omitted, it is assumed to be 0.

Typically, a makespan minimization objective on $n$ activities modeled as interval variables $act[i]$ can be defined as:

```
minimize max(i in 1..n) endOf(act[i]);
```

Numeric expressions (`startEval`, `endEval`, `lengthEval`, `sizeEval`) specify piecewise-linear functions (see Section 3.1.1) evaluated on a given bound of an interval. Like for the above expressions, a numeric value can be specified that represents the value of the expression when the interval is absent. If this value is omitted, it is assumed to be 0.

Suppose for instance $n$ optional activities modeled as interval variables $act[i]$ such that each activity $i$ has a fixed cost $C[i]$ if not executed and an earliness and/or tardiness cost defined by a piecewise-linear function $F[i]$. The total cost of the schedule can be modeled by a single expression:

```
minimize sum(i in 1..n) endEval(act[i], F[i], C[i]);
```

## 3.3 Logical constraints

The presence constraint `presenceOf`$(a)$ states that a given interval variable $a$ must be present. This constraint may be used in logical constraints.

Author's personal copy

For example, there may be two optional interval variables $a$ and $b$, but if interval $a$ is present then $b$ must be present too. This can be modeled by the constraint:

```
presenceOf(a) => presenceOf(b);
```

The case where interval variables $a$ and $b$ should be either both present or both absent will be modeled as:

```
presenceOf(a) == presenceOf(b);
```

Binary constraints between presence statuses like the two examples above play a central role in many CP Optimizer scheduling models (see for example Section 4.4). As we will see in Section 6.2, binary constraints on presence statuses are aggregated into a logical network and heavily exploited by the propagation algorithms.

## 3.4 Temporal constraints

CP Optimizer's temporal constraint network consists of a Simple Temporal Network [8] extended to the presence statuses. CP Optimizer offers different types of temporal constraints. For instance the very common precedence constraint:

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

states that if both intervals $a$ and $b$ are present then the end of $a$ must occur at least $z$ time units before the start of $b$. So that the semantics of the constraint is $x(a) \wedge x(b) \Rightarrow e(a)+z \leq s(b)$. CP Optimizer offers other precedences, for example `startBeforeEnd(a,b,z)`. Such a precedence can be useful for making sure that $a$ begins before $b$ terminates, but more commonly with a negative value of $z$, for ensuring that at most $-z$ time units elapse between the end of $a$ and the beginning of $b$ (a maximum delay).

Note that the delay $z$ specified in a precedence constraint can be either a fixed value but also a variable or an expression.

## 3.5 Constraints on groups of interval variables

CP Optimizer offers constraints over groups of interval variables. 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 $\texttt{span}(a, \{b_1, \ldots, b_n\})$ states that the interval $a$, if present, spans over all present intervals from the set $\{b_1, \ldots, b_n\}$. In other words, interval $a$ starts together with the first present interval from $\{b_1, \ldots, b_n\}$ and ends together with the last one. Interval $a$ is absent iff all intervals in $b$ are absent.
– Alternative constraint. The constraint $\texttt{alternative}(a, \{b_1, \ldots, b_n\})$ models an exclusive alternative between $\{b_1, \ldots, b_n\}$. If interval $a$ is present then exactly one of intervals $\{b_1, \ldots, b_n\}$ is present and $a$ starts and ends together with this chosen one. The alternative constraint can also specify a non-negative integer cardinality $c$: $\texttt{alternative}(a, \{b_1, \ldots, b_n\}, c)$. In this case exactly $c$ interval variables will be selected from the set $\{b_1, \ldots, b_n\}$ and those $c$ selected intervals will have to start and end together with interval variable $a$. Note that $c$ can be a variable or an expression. Interval $a$ is absent iff all intervals in $b$ are absent.

These constraints make it easy to capture the structure of complex scheduling problems (hierarchical description of the work-breakdown structure of a project, representation of alternative modes/recipes/processes, *etc.*) without using complex expressions of logical connectors.

Author's personal copy

*Example* Work-Breakdown Structure The model in Fig. 3 describes the Work-Breakdown Structure (WBS) of a project. The project consists of $n$ tasks whose decomposition structures are provided. A decomposition (line 2) consists of a parent task and a set of subtasks. A given task may be the parent of several decompositions, meaning that there are alternative ways to decompose the task. In this case the solution must specify which decomposition is selected. A given sub-task may be compulsory in the decomposition it belongs to (this is defined in line 4) meaning that whenever the decomposition is selected the sub-task must be executed. Otherwise the execution of the sub-task is optional. For a given task $i$, line 6 computes the number of alternative decompositions of the task while line 7 computes the number of parent tasks of task $i$ (`nbParents[i]=0` means that task $i$ is a top-level task). Decision variables are defined in lines 9-10: one optional interval variable `task[i]` for each task $i$ and one optional interval variable `dec[d]` for each decomposition $d$. Constraints at lines 14–15 state that top-level tasks that are compulsory should be present. In case the task has some decomposition (thus, it is not a leaf of the WBS), an alternative constraint is posted between the task and its possible alternative decompositions at line 17. Each of the possible decompositions is then constrained to span their subtasks at lines 18–19. Finally, for each sub-task of a decomposition that is compulsory, line 23 posts a constraint stating that whenever the decomposition is selected, the sub-task should be present. The model only describes the constraints related with the WBS: in a real problem one would of course need to add additional constraints like precedence constraints between sub-tasks in a decomposition and constraints relating to resources. These constraints would be posted directly between the optional interval variables `task[i]`. It is important to notice that the complete model would not involve any meta-constraint (except for the binary constraints on presence statuses at line 23 which will be efficiently aggregated and exploited by the engine).

```
 1  using CP;
 2  tuple Dec { int task; {int} subtasks; };
 3  int n = ...;
 4  int compulsory[1..n] = ...;
 5  {Dec} Decs = ...;
 6  int nbDecs[i in 1..n] = card( {d | d in Decs : d.task==i} );
 7  int nbParents[i in 1..n] = card( {d | d in Decs : i in d.subtasks} );
 8
 9  dvar interval task[i in 1..n] optional;
10  dvar interval dec[d in Decs] optional;
11
12  constraints {
13    forall(i in 1..n) {
14      if (nbParents[i]==0 && 0<compulsory[i])
15        presenceOf(task[i]);
16      if (nbDecs[i]>0) {
17        alternative(task[i], all(d in Decs: d.task==i) dec[d]);
18        forall(d in Decs: d.task==i)
19          span(dec[d], all(j in d.subtasks) task[j]);
20      }
21    }
22    forall(d in Decs, j in d.subtasks: 0<compulsory[j])
23      presenceOf(dec[d]) => presenceOf(task[j]);
24  }
```

**Fig. 3** Example of a Work-Breakdown Structure with Alternative Tasks

Author's personal copy

## 3.6 Interval variables sequencing

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 CP Optimizer introduces the notion of a *sequence variable*, a new type of decision variable whose value is a permutation of a set of interval variables. Constraints on sequence 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).

**Sequence Variables** An *interval sequence variable* (or *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 present interval variables of $A$ (the absent intervals of $A$ not being considered in the ordering). More formally, suppose that all interval variables in $A$ are fixed, let $n = |A|$. A permutation $\pi$ of $A$ is a function $\pi : A \rightarrow [0, n]$ such that, if we denote $\text{length}(\pi) = |\{a \in A, x(a)\}|$ the number of present intervals:

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

The domain of a sequence variable $p$ defined on $A$ is the set of all possible permutations $\pi$. 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 permutation values $\{\pi_1, \pi_2, \pi_3\}$ such that $\pi_1(a) = 1, \pi_1(b) = 0, \pi_2(a) = 1, \pi_2(b) = 2, \pi_3(a) = 2, \pi_3(b) = 1$ or in short $\{(a), (a, b), (b, a)\}$.

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

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

**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 non-negative

Author's personal copy

integer type $T(p,a)$ can be associated with each interval $a$ of a sequence variable $p$. If a transition distance $M$ (see Section 3.1.3) 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 $p$, the condition for a permutation value $\pi$ to satisfy the *no-overlap* constraint on $p$ with transition distance $M$ is defined as:

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

A second version of the no-overlap constraint enforces a slightly less strong condition by only working on immediate successors in the permutation:

$$\texttt{noOverlapDirect}(\pi, M) \Leftrightarrow \forall a, b \in A,$$
$$0 < \pi(a), \pi(b) = \pi(a) + 1 \Leftrightarrow e(a) + M[T(p,a),$$
$$T(p,b)] \le s(b)$$

Note that if the transition distance matrix $M$ satisfies the triangular inequality then the two versions of the no-overlap constraint are equivalent.

*Example* Vehicle Routing Problem with Time-Windows Fig. 4 illustrates the use of sequencing and no-overlap constraints on sequence variables to model a Vehicle Routing Problem (VRP) with $n$ visits and $m$ vehicles. Each visit must happen within a specified time-window. We are assuming the duration of a visit $i$ depends on the selected vehicle $j$ and is given by integer parameter $D[i][j]$. Visits 1 to $n$ represent actual visits whereas visit 0 (resp. visit $n+1$) represents the departure from (resp. arrival at) the common depot for all vehicles. Travel times between visits are also vehicle dependent and are given as a transition distance

```
using CP;
int n = ...; // Number of visits
int m = ...; // Number of vehicles
int D[0..n+1][1..m] = ...; // Visit durations
tuple window { int start; int end; };
window W[1..n] = ...; // Visit time-windows
tuple triplet { int id1; int id2; int value; };
{triplet} M[1..m] = ...; // Travel times for each vehicle

dvar interval visit[i in 1..n] in W[i].start..W[i].end;
dvar interval vvisit[i in 0..n+1][j in 1..m] optional size D[i][j];
dvar sequence route[j in 1..m]
  in all(i in 0..n+1) vvisit[i][j] types all(i in 0..n+1) i;

minimize sum(j in 1..m) endOf(vvisit[n+1][j]);
subject to {
  forall(j in 1..m) {
    noOverlap(route[j], M[j], true);
    presenceOf(vvisit[0][j]);
    presenceOf(vvisit[n+1][j]);
    first(route[j], vvisit[0][j]); // Departure from depot
    last(route[j], vvisit[n+1][j]); // Return to depot
  }
  forall(i in 1..n)
    alternative(visit[i], all(j in 1..m) vvisit[i][j]);
}
```

**Fig. 4** Example of a Vehicle Routing Problem with Time-Windows

Author's personal copy

matrix $M[j]$ for each vehicle $j$. Actual visits are represented by present interval variables at line 10. Possible visits performed by a given vehicle $j$ are modeled as optional interval variables at line 11. Note that variables `vvisit[0][j]` and `vvisit[n+1][j]` respectively denote the departure from depot and arrival to depot: these interval variables are enforced to be present at lines 19-20. For each vehicle $j$, a sequence variable `route[j]` is defined at lines 12-13. The sequence variable is defined on all the interval variables of vehicle $j$ and the associated type of `vvisit[i][j]` is the index `i`. These indices are the ones used to index the transition distances `M[j]`. The no-overlap constraint at line 18 states that the visits selected for vehicle $j$ will not overlap in time and that a vehicle-dependent minimal travel time `M[j]` will be enforced between consecutive visits (the `true` argument specifies the use of the noOverlapDirect version of the constraint). Lines 21-22 state that the departure from (resp. return to) the depot is the first (resp. last) interval variable in sequence `route[j]`. The allocation of vehicles to visits is handled by the alternative constraints in line 25. Finally, the objective function of this version of the VRP is to minimize the total usage time of vehicles.

**Same-sequence and same-common-subsequence constraints** The `sameSequence` and `sameCommonSubsequence` constraints are binary constraints on a pair of sequence variables $p$ and $p'$. These constraints state that the relative position of some related subsets of interval variables is the same in both sequences: typically, if $a$ is before $b$ in sequence $p$ then $a'$ (the interval related to $a$) is before $b'$ (related to $b$) in sequence $p'$. Here are two examples of use-cases where these constraints are useful:

– First in/first out and no-bypass constraints. In some physical systems like trains on a single line railway or items on a conveyor belt, bypassing is not possible and items must enter and exit a given section of the system in the same order. If entering and exiting the sections or the junctions of the system is modeled by two related interval variables, those constraints can be modeled by `sameSequence` constraints (or, if the items can follow different paths, `sameCommonSubsequence` for the items that follow the same path) on the sequences of entering and exiting intervals. A classical example of such a constraint is the permutation flow-shop scheduling problem.
– Scenario-Based approaches for scheduling with uncertainties. In presence of uncertainties (for instance, uncertain activity durations) one may be interested in building sequences of activities on disjunctive resources that optimize some robustness or statistical criterion (for instance the expected makespan). A scenario is a sub-model that defines a particular realization of the uncertainties in the environment. As one must find robust sequences that optimize some criterion over all scenarios, the different sequences of a given disjunctive resource across all scenarios are linked with `sameSequence` constraints.

*Example* Stochastic Job-shop Scheduling problem Suppose we want to solve a stochastic job-shop scheduling problem for which the duration of operations is uncertain and given as a set of $s$ scenarios. This model is described in Fig. 5. The constant $dur[k][i][j]$ denotes the duration of the $j^{th}$ operation of job $i$ in scenario $k \in 1..s$. Constant $mch[i][j]$ denotes the machine used by the $j^{th}$ operation of job $i$. The model below formulates the problem to build a robust sequencing on the different machines (the same sequencing should be used across all scenarios) that minimizes the expected makespan. Note the use of `sameSequence` constraint for expressing the fact that operations are scheduled in the same order on the different machines across all scenarios. This is a typical 2-stage stochastic

Author's personal copy

```
using CP;
int n = ...; // Number of jobs
int m = ...; // Number of machines = number of operations per job
int s = ...; // Number of scenarios
int dur[k in 1..s][i in 1..n][j in 1..m] = ...;
int mch[i in 1..n][j in 1..m] = ...;

dvar interval op[k in 1..s][i in 1..n][j in 1..m] size dur[k][i][j];
dvar sequence seq[k in 1..s][l in 1..m]
  in all(i in 1..n, j in 1..m : mch[i][j]==l) op[k][i][j];

minimize sum(k in 1..s) (max(i in 1..n) endOf(op[k][i][m])) / s;
subject to {
  forall(k in 1..s) {
    forall(l in 1..m) {
      noOverlap(seq[k][l]);
      if (1<k)
        sameSequence(seq[1][l],seq[k][l]);
    }
    forall(i in 1..n, j in 2..m)
      endBeforeStart(op[k][i][j-1], op[k][i][j]);
  }
}
```

**Fig. 5** Example of a Stochastic Job-shop Scheduling problem

optimization process for which first stage variables are the sequence variables and second stage variables are the actual start and end dates of operations. The `sameSequence` constraints are used to state that first stage variables should have the same value in all scenarios.

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

*Example* Transition costs A typical use-case of these expressions is to model a transition cost on a disjunctive resource. Suppose the operations on the disjunctive resource are modeled as interval variables `o[i]`, that each operation has a known type `T[i]` and that switching from an operation of type $u$ to an operation of type $v$ incurs a cost `C[u][v]`. Minimizing the total transition cost on the machine could be modeled as:

```
int n = ...; // Number of activities
int m = ...; // Number of types
int T[i in 1..n]= ...; // Type of activity act[i]
float C[0..m][0..m] = ...; // Transition costs
dvar interval o[i in 1..n];
dvar sequence seq in o types T;
minimize sum(i in 1..n) C[T[i]][typeOfNext(seq,o[i],0,0)];
subject to {
  noOverlap(seq);
}
```

Author's personal copy

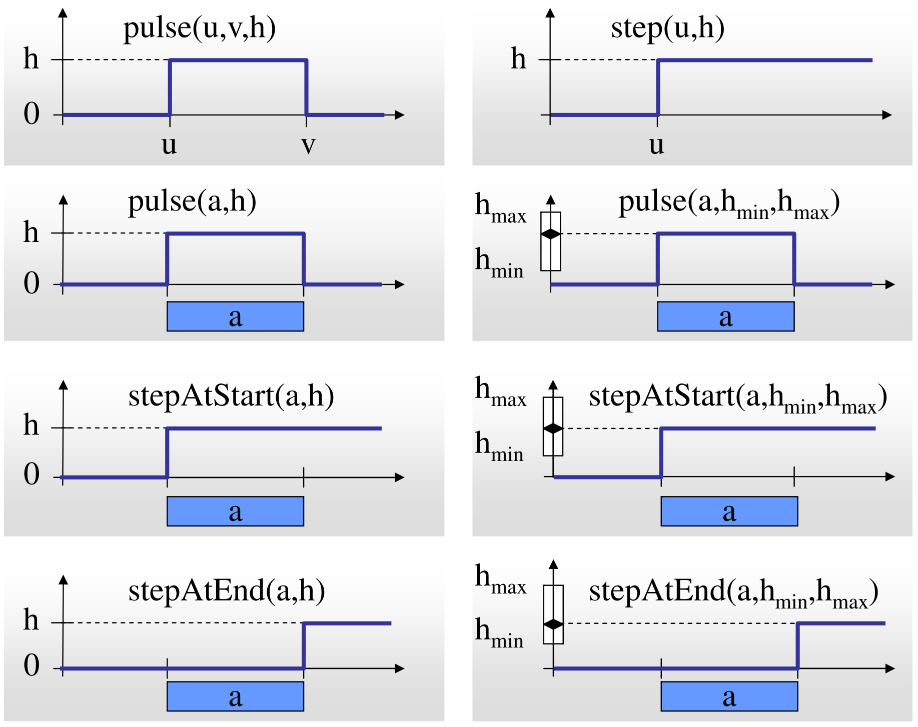

**Fig. 6** Elementary cumul function expressions

## 3.7 Cumul function expressions

CP Optimizer supports cumulative resources by defining functions of time that represent cumulated usage of a resource by activities. Typically an activity increases the function at its start time and decreases it at its end time. For resources that can be produced and consumed by activities (for instance the content of an inventory or a tank) production activities increase the resource level whereas consuming activities 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.[4] A set of elementary 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 can be used to restrict the value of the function over the complete horizon or only over some fixed or variable interval.

A *cumul function expression* $f$ is an expression whose value is a function mapping the integers to the non-negative integers. Assuming $u, v \in \mathbb{Z}$ $h, h_{min}, h_{max} \in \mathbb{Z}^+$ and $a$ is an

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

Author's personal copy

interval variable, we consider **elementary cumul functions** illustrated in Fig. 6. 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 Fig. 6 or their negations. More formally, $f$ 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 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 value 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 value 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 less 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.

*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 $a_i$ executes. The constraints on interval variable status ensure that only the first intervals in the chain $w$ are executed and the two alwaysIn constraints state the synchronization relation between intervals $a_i$ and intervals $w_j$. A solution is illustrated in Fig. 7.

```
dvar interval a[i in 1..n] size ...;
dvar interval w[j in 1..m] optional size 1..H;
cumulFunction fa = sum(i in 1..n) pulse(a[i],1);
cumulFunction fw = sum(j in 1..m) pulse(w[j],1);
constraints {
  fa <= 3;
  forall(j in 2..m) {
    presenceOf(w[j]) => presenceOf(w[j-1]);
    endBeforeStart(w[j-1],w[j],1);
  }
  forall(i in 1..n)
    alwaysIn(fw,a[i],1,1);
  forall(j in 1..m)
    alwaysIn(fa,w[j],1,n);
}
```

## 3.8 State function variables

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`

Author's personal copy

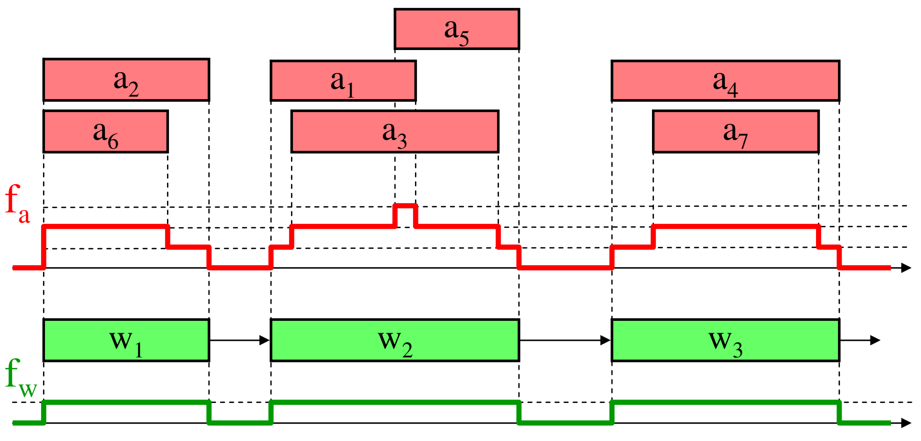

**Fig. 7** Modeling intervals $w$ when a cumulative resource is in use

constraints on cumul functions to constrain the values of the state function. A state function is a set of non-overlapping segments 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 Fig. 9):

$$\begin{array}{lll} [\text{start} = 0, & \text{end}=100): & \text{state}=0, \\ [\text{start} = 140, & \text{end}=300): & \text{state}=1, \\ [\text{start} = 320, & \text{end}=500): & \text{state}=2, \\ [\text{start} = 540, & \text{end}=650): & \text{state}=2, \cdots \end{array}$$

A *state function variable* $f$ is a variable whose value is a set of non-overlapping segments $[s_i, e_i)$ (with $s_i < e_i$). Each segment $[s_i, e_i)$ is associated with a non-negative integer value $v_i$ that represents the state of the function over the segment. Let $f$ be a fixed state function denoted $f = ( [s_i, e_i) : v_i )_{i\in[1,n]}$. We define $D(f) = \cup_{i\in[1,n]}[s_i, e_i)$ as the domain of $f$, that is, the set of points where the state function is associated with a state. For a fixed state function $f$ and a point $t \in D(f)$, we will denote $[s(f, t), e(f, t))$ the unique segment of the function that contains $t$ and $f(t)$ the value of this segment. For instance, in the oven example we would have $f(200) = 1$, $s(f, 200) = 140$ 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 $M\left[v, v'\right]$ is a transition distance matrix between state $v$ and state $v'$, we have: $\forall i \in [1, n-1],\ e_i + M[v_i, v_{i+1}] \leq s_{i+1}$.

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 $\text{algn}_s$, $\text{algn}_e$ two boolean values:

– The constraint `alwaysConstant`$(f, a, \text{algn}_s, \text{algn}_e)$ 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) $[s(a), e(a)) \subseteq [s(f, s(a)), e(f, s(a)))$
(b) $\text{algn}_s \Rightarrow s(a) = s(f, s(a))$

Author's personal copy

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

– The constraint $\texttt{alwaysEqual}(f, a, v, \text{algn}_s, \text{algn}_e)$ specifies that whenever $a$ is present the state function takes a constant value $v$ over interval $a$:

  (a) $\texttt{alwaysConstant}(f, a, \text{algn}_s, \text{algn}_e)$
  (b) $v = f(s(a))$

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

These constraints are also available on a fixed interval [Start, end) in place of interval variable $a$.

*Example* (State function value) The model in Fig. 8 defines a state function $f$ constrained by a set of fixed interval variables. A value for $f$ satisfying all the constraints of this model is shown in Fig. 9. As interval variables $b_0$ : [0, 100) and $b_1$ : [140, 300) are start and end aligned, they define two segments of the state function (with states 0 and 1 respectively). A transition distance 40 applies in between those states. Interval variables $b_{21}$ : [320, 400) and $b_{22}$ : [420, 500) require the same state of 2. Note that the transition distance between two segments of state 2 is 40, this is why intervals $b_{21}$ and $b_{22}$ necessarily belong to the same

```
dvar interval b0 in 0..100 size 100;
dvar interval b1 in 140..300 size 160;
dvar interval a1 in 140..240 size 100;
dvar interval a3 in 180..280 size 100;
dvar interval b21 in 320..400 size 80;
dvar interval b22 in 420..500 size 80;
dvar interval c in 510..530 size 20;
dvar interval a2 in 550..575 size 25;
dvar interval b23 in 540..600 size 60;

tuple triple { int i; int j; int t; }
{triple} M = {
  <0,0,0>,<0,1,40>,<0,2,40>,
  <1,0,40>,<1,1,0>,<1,2,20>,
  <2,0,40>,<2,1,20>,<2,2,40>
 };
stateFunction f with M;

constraints {
  alwaysEqual(f, b0, 0, true, true);
  alwaysConstant(f, b1, true, true);
  alwaysEqual(f, a1, 1, true, false);
  alwaysEqual(f, a3, 1, false, false);
  alwaysEqual(f, b21, 2, true, false);
  alwaysEqual(f, b22, 2, false, true);
  alwaysNoState(f,c);
  alwaysEqual(f, a2, 2, false, false);
  alwaysEqual(f, b23, 2, true, false);
}
```

**Fig. 8** A model with fixed interval variables on a state function

Author's personal copy

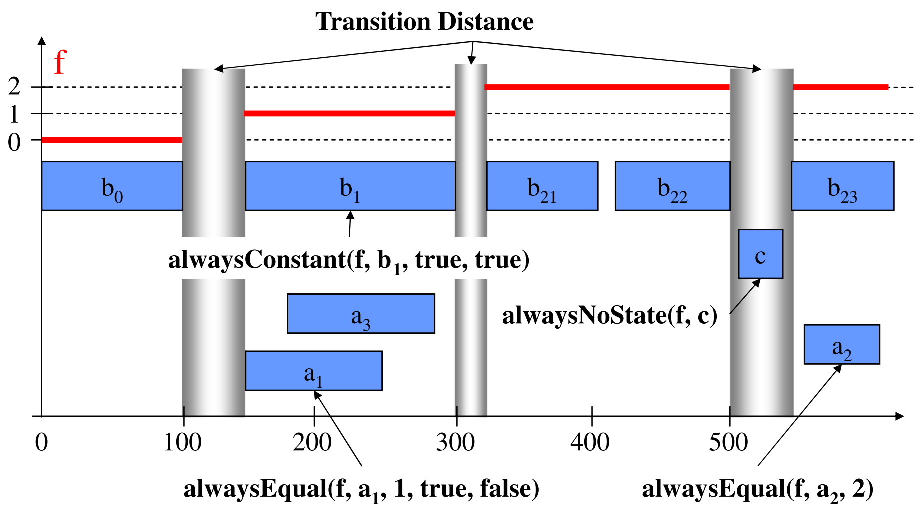

**Fig. 9** Example of a fixed State function

segment, as there is not enough time between the end of $b_{21}$ and the start of $b_{22}$ to have a transition between two segments of state 2.

*Example* (Simplified photo-lithography machine) The model in Fig. 10 illustrates the use of state functions to model a batching machine in the context of a photo-lithography scheduling problem. A set of $n$ operations is to be scheduled on the machine (line 11). Each operation `op[i]` consists in the treatment of a set of wafers on the machine. There are different families of operations. The machine can perform several operations at the same time (notion of batch) provided that (1) the processing time of the operations is the same as that of the batch, (2) the operations are from the same family and (3) the total capacity of the machine

```
 1  using CP;
 2  int n = ...;
 3  int capacity = ...;
 4  int ptmin[1..n] = ...;
 5  int ptmax[1..n] = ...;
 6  int nbwafers[1..n] = ...;
 7  int family[1..n] = ...;
 8  tuple triplet { int id1; int id2; int value; };
 9  {triplet} M = ...; // Transition time between pairs of families
10
11  dvar interval op[i in 1..n] size ptmin[i]..ptmax[i];
12  stateFunction machineFamily with M;
13  cumulFunction machineLoad = sum (i in 1..n) pulse(op[i], nbwafers[i]);
14
15  constraints {
16    machineLoad <= capacity;
17    forall(i in 1..n) {
18      alwaysEqual(machineFamily, op[i], family[i], true, true);
19    }
20  }
```

**Fig. 10** State functions illustrated on a simplified photo-lithography scheduling problem

Author's personal copy

in terms of number of wafers is not exceeded. Batches of operations are synchronized: that is, all operations in the same batch start (resp. end) at the same time. Furthermore, some family-dependent setup time is needed to configure the machine from a given batch family to the next batch family. The limited capacity is modeled as a cumul function (lines 13 and 16). The family-dependent setup time is captured by a matrix `M` (line 9) and batching constraints are defined using `alwaysEqual` constraints on a state function with start and end alignment (lines 12 and 18).

### 3.9 Integer variables and constraints

Traditional *integer variables* are of course also available in CP Optimizer and they can be used in combination with *interval variables* in the same model. But in general for scheduling problems, when possible, it is better to leverage the optionality status of interval variables rather than using explicit integer variables. When one would be tempted to use constraints on integer variables preconditioned by the presence status of an interval variable (something like `presenceOf`$(a) \Rightarrow Ct$ where $Ct$ is a constraint other than `presenceOf`), there is often a more direct and efficient model working with interval variables only and without preconditions.

Among the most useful numeric expressions and constraints in scheduling problems one can cite:

– Arithmetic expressions like `+`, `*`, `min`, `max` (makespan), `abs`, `/`, `exp` (for modeling net present value), `log`, `modulo` (for periodic scheduling)
– Array element expressions of the form `A[x]` where `A` is an array of fixed or variable numeric quantities and `x` an integer expression
– Comparative constraints like `==`, `<=`

## 4 Examples

We give in this section the complete CP Optimizer formulation of four scheduling problems. These examples illustrate how the concepts of the modeling language (in particular optional interval variables) can be leveraged to produce concise and efficient models. Each of the models given in this section, when solved with the automatic search, is on par with, or even outperforms, the pre-existing state of the art on this specific problem.

### 4.1 Flow-shop with earliness/tardiness costs

In this problem described in [40], $n$ jobs are to be executed on $m$ machines. Each job $i$ comprises 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 on which it will be executed. 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. 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 (measure of importance in cost terms) $w_i$. 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 [40], the

Author's personal copy

```
using CP;
int n = ...;
int m = ...;
int rd[1..n] = ...;
int dd[1..n] = ...;
float w[1..n] = ...;
int pt[1..n][1..m] = ...;
float W = sum(i in 1..n) (w[i] * sum(j in 1..m) pt[i][j]);

dvar interval op[i in 1..n][j in 1..m] size pt[i][j];

dexpr int C[i in 1..n] = endOf(op[i][m]);

minimize sum(i in 1..n) w[i]*abs(C[i]-dd[i])/W;
subject to {
  forall(i in 1..n) {
    rd[i] <= startOf(op[i][1]);
    forall(j in 1..m-1)
      endBeforeStart(op[i][j],op[i][j+1]);
  }
  forall(j in 1..m)
    noOverlap(all(i in 1..n) op[i][j]);
}
```

**Fig. 11** CP Optimizer model for Flow-shop with Earliness and Tardiness Costs

total earliness/tardiness cost is normalized by the sum of operation processing times so the global cost to minimize is:

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

A complete formulation for this problem from [29] is shown in Fig. 11.

Note that the earliness/tardiness cost could also have been represented as a piecewise-linear function (see Section 3.1.1).

This scheduling problem is non-regular (an optimal schedule does not necessarily execute all the activities as soon as possible). On this type of problem, a key ingredient for search efficiency is the linear relaxation that is automatically used to guide the search (see Section 6.4).

## 4.2 Multi-Mode RCPSP

The multi-mode resource-constrained project scheduling problem (MMRCPSP) is a classical extension of the resource-constrained project scheduling problem where each task can be executed in one of a number of alternative modes. The problem aims to select a single task mode from a set of available modes in order to construct a precedence- and resource-feasible project schedule with a minimal makespan [26].

A complete formulation of the MMRCPSP is shown in Fig. 12. The MMRCPSP problem illustrates well the use of optional interval variables. The allocation of a task to a given mode is directly captured by an optional interval variable (line 20) and there is no need for composite constraints in the model. This problem also relies on the use of cumul function expressions: we see that both renewable (`rUsage`) and non-renewable (`nUsage`) resources are modeled in a very similar way using expressions (line 22 and line 24) and constraints on

Author's personal copy

```
 1  using CP;
 2  int NbTasks = ...;
 3  int NbRs = ...;
 4  int NbNs = ...;
 5  int CapR[1..NbRs] = ...;
 6  int CapN[1..NbNs] = ...;
 7
 8  tuple Task { key int id; {int} succs; }
 9  {Task} Tasks = ...;
10
11  tuple Mode {
12    int tid;
13    int pt;
14    int dmdR [1..NbRs];
15    int dmdN [1..NbNs];
16  }
17  {Mode} Modes = ...;
18
19  dvar interval task[t in Tasks];
20  dvar interval mode[m in Modes] optional size m.pt;
21
22  cumulFunction rUsage[r in 1..NbRs] =
23     sum (m in Modes: m.dmdR[r]>0) pulse(mode[m], m.dmdR[r]);
24  dexpr int nUsage[r in 1..NbNs] =
25     sum (m in Modes: m.dmdN[r]>0) m.dmdN[r] * presenceOf(mode[m]);
26
27  minimize max(t in Tasks) endOf(task[t]);
28  subject to {
29    forall (t in Tasks)
30      alternative(task[t], all(m in Modes: m.tid==t.id) mode[m]);
31    forall (r in 1..NbRs)
32      rUsage[r] <= CapR[r];
33    forall (r in 1..NbNs)
34      nUsage[r] <= CapN[r];
35    forall (t1 in Tasks, t2id in t1.succs)
36      endBeforeStart(task[t1], task[<t2id>]);
37  }
```

**Fig. 12** CP Optimizer model for Multi-Mode RCPSP

the maximal value of these expressions (line 32 and line 34), the only difference being that the cumul function expression maintains a value *for all time-points*.

## 4.3 Satellite communication scheduling

This is an oversubscribed scheduling problem described in [27]. 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 is 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

Author's personal copy

in AMC or one antenna in AFSCN) of the resource $r_j \in Res_i$ that performs it. The duration $Dur_{i,j}$ of task $T_i$ depends on the allocated resource $r_j$.
- Tasks are categorized into one of five major priority classes, and task priorities must be respected: it is not possible to substitute a lower priority task for a higher priority task even if this choice enables additional lower priority tasks to be inserted into the schedule.
- 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 maximize the number of assigned tasks while respecting the priority rule.

A complete formulation of the problem (adapted from [29]) is shown in Fig. 13. It is worth noting that in this model optionality of interval variables is exploited to represent, for the same variable, (1) the temporal bounds (start, end) would the interval be present and (2) the boolean presence status of the interval. Optional tasks are created on line 11 and possible allocations of a task to a resource on line 12. Note how particular specifications of a possible resource allocation (like here the time-window and the resource-dependent activity duration) can be specified very easily without composite constraints. Alternative constraints at line 20 state that if a task is executed, then it must be allocated one resource within its time-window.

This example also illustrates the formulation of multi-objective problems in CP Optimizer, in particular lexicographical objectives. In the model, expression $nb[p]$ represents the number of executed tasks of priority $p$. The objective is to maximize the most priority criterion `nb[1]`, any improvement of this criterion is worth any loss on the other criteria. The second criterion `nb[2]` is the second most important one, and so on. The last criterion (here, `nb[5]`) is the least important one.

```
 1  using CP;
 2  tuple Resource { string name; key int id; int cap; }
 3  tuple Task { string name; key int id; int prio; }
 4  tuple Alt { int task; int res; int smin; int dur; int emax; }
 5
 6  {Resource} Res = ...;
 7  {Task} Tasks = ...;
 8  {Alt} Alts = ...;
 9  {int} Priorities = { 1,2,3,4,5 };
10
11  dvar interval task[t in Tasks] optional;
12  dvar interval alt[a in Alts] optional in a.smin..a.emax size a.dur;
13
14  dexpr int nb[p in Priorities] =
15    sum(t in Tasks: t.prio==p) presenceOf(task[t]);
16
17  maximize staticLex(nb);
18  subject to {
19    forall(t in Tasks)
20      alternative(task[t], all(a in Alts: a.task==t.id) alt[a]);
21    forall(r in Res)
22      sum(a in Alts: a.res==r.id) pulse(alt[a],1) <= r.cap;
23  }
```

**Fig. 13** CP Optimizer model for Satellite Communication Scheduling

Author's personal copy

## 4.4 Earth-observation scheduling

The GEO-CAPE observation scheduling problem was originally described in [10]. We are given a set of scenes $\Psi$ to observe and a set of time-slots $H$ representing the schedule horizon. Each instance of observation of a given scene takes one time-slot. Each scene $i \in \Psi$ is characterized by a baseline ($S^i_B$) and a threshold ($S^i_T$) separation value, together with the schedule quality values $V^i_B$ and $V^i_T$ when the separation between two consecutive observations of scene $i$ is equal to $S^i_B$ and $S^i_T$ respectively. This separation value must be always greater or equal to the minimum baseline separation. More formally, the "gain" $V(d)$ of two consecutive observations of the same scene $i$ separated by a duration $d$ such that $S^i_B \leq d \leq S^i_T$ is defined as: $V(d) = \alpha_i \cdot d + \beta_i$ where:

$$\alpha_i = \frac{V^i_T - V^i_B}{S^i_T - S^i_B} \qquad \beta_i = V^i_B - \left(S^i_B \cdot \frac{V^i_T - V^i_B}{S^i_T - S^i_B}\right)$$

The value $V(d)$ is zero when $d > S^i_T$. The objective is to maximize the total gain generated from consecutive observations. Scenes are not always observable. The available daylight and the cloud coverage of a scene $i \in \Psi$ are translated into a set of observable time-slots $C^i \subseteq H$. Each observation of scene $i$ must be scheduled at a time-slot belonging to $C^i$. Finally, the instrument can observe only one scene at a time so all observations must be scheduled at different time slots.

A complete CP Optimizer formulation of the GEO-CAPE observation scheduling problem ([34]) is shown in Fig. 14.

The gain function `V` is defined at Line 11 as a piecewise linear function. Lines 12-13 define, for each scene `i`, a step function `NoObs[i]` with value `0` when the scene is not observable and value `1` otherwise.

Decision variables of the problem are defined in Lines 15-18. For a scene `i`, the `j`th observation is modeled by an optional interval variable `a[i,j]` (Line 15). The separation time between the `j`th and `(j+1)`th observation is represented by an optional interval variable `s[i,j]` (Line 25) that is constrained to start at the start date of `a[i,j]` (Line 26) and end at the start date of `a[i,j+1]` (Line 26).

Two situations are possible for this separation time: it either lasts for less than `ST` and in this case it will produce some gain or, it lasts for more and will not produce any gain. These two situations are modeled by two interval variables `sv[i,j]` and `s0[i,j]`: the possible length of `sv[i,j]` is in `[SB,ST]` (Line 17) while the possible length of `s0[i,j]` is greater than `ST` (Line 18). The alternative choice between the two intervals is specified through an `alternative` constraint (Line 27).

The constraint at Line 32 states that `a[i,j]` is present if at least `j` observations are made on scene `i` while constraint at Line 24 enforces that the separation intervals `s[i,j]` before the last observation are all present. Constraints at lines 29, 30 and 33 enforce a dominance rule that forbids the presence of isolated observations: an observation is said to be isolated if it does not contribute to the gain. Line 33 says that a separation interval that does not produce any gain (`s0[i,j-1]`) must be followed by a separation interval that produces some gain ( `sv[i,j]`) as otherwise, observation `a[i,j]` would be isolated. Line 29 says that it is forbidden to have a scene with a unique (necessarily isolated) observation and Line 30 enforces the first observation to be non-isolated. Finally, the constraint at Line 36 says that if an observation is made, then it must be made at a time-slot where the scene is observable, while Line 38 ensures that none of the observations are performed at the same time. The objective function (Line 20) is to maximize the total gain, that is the sum of the values of the piecewise linear function `V` evaluated on the length of separation intervals `s[i,j]`.

Author's personal copy

```
using CP;
int   SB = ...; int   ST = ...;
float VB = ...; float VT = ...;
float A = (VT-VB) / (ST-SB);
int n = ...;
{int} T[1..n] = ...;
int TL = min(i in 1..n, t in T[i]) t;
int TU = max(i in 1..n, t in T[i]) t;
int m = (TU-TL) div SB;

pwlFunction V = piecewise{ A->ST; -VT->ST+1; 0 } (SB,VB);
stepFunction NoObs[i in 1..n] =
  stepwise(t in TL-1..TU) { (t in T[i]) -> t+1; 0 };

dvar interval a [1..n, 1..m+1] optional size 1;
dvar interval s [1..n, 1..m]   optional;
dvar interval sv[1..n, 1..m]   optional size SB..ST;
dvar interval s0[1..n, 1..m]   optional size ST+1..TU;

maximize sum(i in 1..n, j in 1..m) lengthEval(s[i,j], V);
subject to {
  forall(i in 1..n, j in 1..m+1) {
    if (j < m+1) {
      presenceOf(a[i,j+1]) == presenceOf(s[i,j]);
      startAtStart(a[i,j], s[i,j]);
      endAtStart(s[i,j], a[i,j+1]);
      alternative(s[i,j], append(sv[i,j], s0[i,j]));
      if (j == 1) {
        presenceOf(a[i,j+1]) == presenceOf(a[i,j]);
        !presenceOf(s0[i,j]);
      } else {
        presenceOf(a[i,j+1])  => presenceOf(a[i,j]);
        presenceOf(s0[i,j-1]) => presenceOf(sv[i,j]);
      }
    }
    forbidExtent(a[i,j], NoObs[i]);
  }
  noOverlap(a);
}
```

**Fig. 14** CP Optimizer model for the GEO-CAPE scheduling problem

This example illustrates the expressiveness of the modeling concepts: the complete problem can be represented without any need for reified constraints, except for the logical binary constraints between presence statuses that are efficiently handled in the logical network (see Section 6.2). Furthermore, the non-regularity of the objective function modeled as a piecewise linear function is directly exploited by the temporal linear relaxation to guide the search (see Section 6.4).

## 5 Solving

### 5.1 Automatic search

The usual way to solve a CP Optimizer model is to run the automatic search. This search algorithm has several important characteristics:

– It is *complete*. For a feasibility problem, it will either find a feasible solution or prove that the problem is infeasible. For an optimization problem, it will either find an optimal solution (and prove its optimality) or prove the problem is infeasible.

Author's personal copy

– It is *anytime*. Usually for an optimization problem a first feasible solution is found very quickly and the search algorithm iteratively improves the quality of the solution until it can prove optimality. Thus, once a first feasible solution has been found the search can be interrupted at any moment and it will provide the best available solution found so far.
– It is *parallel*. The search algorithm exploits the multiple cores of the machine to run parallel optimization (See Section 6.6).
– It is *randomized*. Internally, some ties are broken using random numbers. The seed of the random number generator is a parameter of the search.
– It is *deterministic*. Solving the same model twice on the same machine, even when using multiple parallel workers, will produce the same results. Determinism of the search is very important in the context of industrial applications because quite often the user would not accept that solving twice the same problem in the same conditions produces different results. Furthermore search determinism is essential for debugging.

The automatic search is continuously improved from version to version, for instance as illustrated in Section 7.3, version 12.7.1 is on average between 6 and 7 times faster than version 12.2 on the same machine. Some important ingredients of the automatic search are described in Section 6.

The behavior of the automatic search can be influenced by *search parameters*, *search phases* and *starting points*. These three concepts are not exclusive of each other and can be used in conjunction. They are described below.

## 5.2 Search parameters

Search parameters are global parameters of the automatic search. Here are some of the most useful parameters for scheduling problems. This first list concerns useful parameters for tuning the resolution:

Time limit: When the search is not able to prove optimality in reasonable time, a time limit may be provided so that the engine stops when the limit is met and returns the best feasible solution found so far. Note that other limits are provided (*e.g.* branch limit, solution limit)

Temporal relaxation: On some problems, the automatic search may decide to use a temporal linear relaxation of the problem to guide the search (See Section 6.4). This may not always improve the performance so it may occasionally be useful to switch off the temporal relaxation.

Inference levels: The propagation of several constraint types in the model can be controlled by dedicated inference levels. By default, quite a low inference level is used and the level can be increased using parameters. For instance setting parameter `NoOverlapInferenceLevel` to `Extended` would activate Edge-Finding-like propagation algorithms on no-overlap constraints.

Search type: Three types of search are provided. The default search is called *Restart*: in this type of search, the constructive search is restarted from time to time and guided toward an optimal solution. More details about the default search are provided in Section 6. The *DepthFirst* type of search is a single tree search algorithm, and can be quite useful while debugging and tuning the model, but generally will be less efficient than *Restart* search because it does not easily recover from poor branching decisions. Finally, the *Multi-point* search creates a set of solutions and combines the solutions in the set in order to produce better solutions. For some scheduling problems *Multi-point* may turn out to be more efficient than *Restart*.

Author's personal copy

Other useful parameters concern model debugging and model evaluation:

Workers: By default, the automatic search exploits parallelism and will use a number of parallel workers defined by the architecture of the machine. In some situations it may be advantageous to use less workers. In particular, while debugging and tuning the models, it is generally easier to work in sequential mode with a single worker.

Random seed: Although the search is deterministic, it uses some randomization in some strategies. This parameter sets the seed of the random generator used by these strategies. Solving the same model with different random seeds is useful to assess the robustness of the search on any particular model or model family.

## 5.3 Search phases

In some applications, there exists a group of key decision variables, such that once these variables are fixed, it is easy to extend the partial solution to the remaining variables. For instance, if we consider the problem in Fig. 7, the values of interval variables $w_j$ are a function of the values of variables $a_i$, so it may be useful from a search standpoint to fix all variables $a_i$ first. A *search phase* is a group of decision variables of the same type (integer, interval or sequence variables). Instantiation of groups of decision variables can be ordered by using several search phases. The decision variables in the first phase are instantiated before the variables in the second one and so on. When using search phases, the phases do not need to cover all variables of the model. The CP Optimizer search will instantiate all variables, and those that do not appear in any search phase will always be instantiated after those appearing in phases.

## 5.4 Starting points

The CP Optimizer search algorithm can be supplied with a *starting point* (a variable assignment) as input. There are use cases where better solutions can be produced more quickly by providing a starting point to the engine. For example:

- While the engine is solving a problem, the session has to be interrupted and the current best solution `sol` is stored. Later on, a new session for solving the problem is started and the stored solution `sol` is specified as the starting point of the new search so as to avoid restarting the search from scratch.
- For a particular problem, a heuristic is available to produce an initial solution `sol`. It would be helpful for this feasible solution to be injected into the engine to accelerate the search.
- A multi-objective optimization problem may involve a lexically ordered set of objective functions `(f1,f2,...,fn)`. In this case, the problem can be solved in $n$ successive steps: first, minimize objective `f1` to produce a solution `sol1`, then, add a constraint to avoid deteriorating `f1` and solve the problem with objective function `f2` using `sol1` as a starting point to produce a solution `sol2`, *etc.* Here, the solution to a given step is a feasible solution for the next step.
- Given an optimization problem for which finding a first solution is difficult, a possible first step is to relax the problem to make it easier to solve and minimize the constraint violations. For instance, in a detailed scheduling problem, a relaxation can be made by replacing activity deadlines by due dates and minimizing tardiness cost or by setting all activities as optional and minimizing the number of unscheduled activities. If this

Author's personal copy

first step is able to produce a solution with no violations, this feasible solution can be re-injected as starting point to the original optimization problem.
- Some applications require solving successive models that are very similar. For instance, in dynamic scheduling, a new request has to be integrated in an existing schedule that was computed in a previous step (notion of work-in-progress). In on-line scheduling, it is necessary to react to various uncertainties of the environment and reschedule when a perturbation occurs (resource breakdown, late activity); here, the new model is similar to the previous one except for the perturbations. In these applications, the previous solution could be used as a starting point to guide the search of the new solve process. Note that here, the previous solution is generally not a feasible solution for the new problem.

The starting point in general does not have to be either a complete nor feasible solution, however if it is the case, CP Optimizer will first visit this solution before going on to try to find improving solutions. If the starting point is infeasible or incomplete, the information contained in the starting point is used as a guideline for the search but of course there is no guarantee that the solutions traversed by the search will be "close" to the starting point solution.

### 5.5 User-defined constraints and search

Like most other CP solvers, it is possible to define new constraints and new search algorithms on top of CP Optimizer (in the native C++ API only). While this is particularly useful for CP researchers who want to experiment with some new constraints or some new propagation or search algorithms, this feature is seldom used in an industrial context because of the expressiveness of the modeling concepts and the efficiency, robustness and flexibility of the automatic search.[5] In an industrial context, the *model-and-run* paradigm offers a path that is more productive and much less risky than the development of hand-crafted constraints and search algorithms.

## 6 Under the hood

In this section we summarize some important ingredients of the automatic search: the presolve functionality, some constraint propagation algorithms, the temporal linear relaxation used to guide the search and the two search space exploration strategies that are used concurrently: the Large Neighborhood Search (for producing good quality solutions) and Failure-Directed Search (for proving infeasibility or optimality).

### 6.1 Presolve

As CP Optimizer is increasingly being adopted by users without a CP background we saw some models that do not take into account how constraint propagation works. Over time we accumulated a number of common model fixes that can improve the performance.

[5]The only industrial cases we have seen are applications that were ported from ILOG Scheduler to CP Optimizer, that did not have any explicit objective function (solution quality was ensured by specific heuristics) and required that exactly the same type of solution should be produced by CP Optimizer.

Author's personal copy

```
dvar interval pred;
dvar interval succ;

constraints {
  endOf(pred) <= startOf(succ);
  // Better: endBeforeStart(pred, succ);
}
```

**Fig. 15** Interval expressions used instead of precedence constraint.

Let us take a simple example. Many users are tempted to use interval variable expressions such as `endOf` instead of precedence constraints, see for example Fig. 15. Semantically, when both interval variables are present, constraint `endOf(pred) <= startOf(succ)` is equivalent to `endBeforeStart(pred, succ)`. However as we will see in Sections 6.2 and 7.1, propagation of `endBeforeStart` using the *temporal network* gives much better performance. Considering how common this modeling is and how easy it is to recognize the constraints involved, we decided to automatically *presolve* (rewrite) such constraints in the model.[6]

CP Optimizer certainly does not contain a presolve for every improvable modeling structure we encountered. However we believe that in a number of cases automatic presolve is the best option. It makes the modeling much simpler for the user. Additionally, sometimes the only other alternative is to make the modeling language more complicated and less intuitive.

Let us take another example: cost of different alternatives. Section 4.2 described a scheduling problem with multiple modes – MMRCPSP. It is common that different modes have not only different resource requirements and different durations but also different associated costs. Figure 16 shows a typical way how is the cost calculated.

The problem with this model is that the *sum* expression is propagated independently of the *alternative* constraint. In particular, the *sum* does not know that exactly one of the `presenceOf(modes[i])` will be true. Therefore the sum computes the range 0..14 instead of 3..6. Here, the CP Optimizer modeling language does not offer a better way to model the cost.[7] We considered extending the modeling language, however in the end we decided not to: we think that this way of modeling cost of different alternatives is the most intuitive one. Instead we added a presolve into CP Optimizer that improves propagation of this kind of expression.

## 6.2 Constraint propagation

**Logical network** All 2-SAT logical constraints between interval presence statuses of the form $[\neg]$`presenceOf`$(a) \vee [\neg]$`presenceOf`$(b)$ are aggregated in a logical network similar to the implication graph described in [4]. The objectives of the logical network are:

- The detection of inconsistencies in logical constraints.
- An $\mathcal{O}(1)$ access to the logical relation that can be inferred between any two intervals $(a, b)$. See for instance below how this is used for propagating temporal constraints.

[6]There is similar presolve for simple constraints between presence statuses of two interval variables: when possible those constraints are added into arcs in the *logical network*.

[7]A contrived better expression is `3+2*!presenceOf(modes[2])+presenceOf(modes[3])`. However it is hard to extend to more than 3 variables and we certainly do not advise users to model alternative costs this way.

Author's personal copy

```
dvar interval task;
dvar interval modes[1..3] optional;
int modecosts[1..3] = {5, 3, 6};
dexpr int cost = sum(i in 1..3) modecosts[i] * presenceOf(modes[i]);
...
constraints {
  alternative(task, modes);
  ...
```

**Fig. 16** Cost of different alternatives.

– 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.

**Temporal network** All temporal constraints (like `endBeforeStart`$(a, b, z)$) are aggregated in a temporal network whose nodes $p_i$ represent the set of interval start and end time-points and whose arcs $(p_i, p_j, z_{ij})$ represent precedences with minimal delay $z_{ij}$. A key element in the propagation of the temporal network is the exploitation of implication relations between interval presence statuses [35]. For a given arc $(p_i, p_j, z_{ij})$, whenever the logical network can infer the relation $x(p_i) \Rightarrow x(p_j)$ the propagation on the conditional bounds of $p_i$ (time-bounds *if* $p_i$ *were present*) can assume that $p_j$ will also be present; thus, the arc can propagate the conditional bounds from time-point $p_j$ on $p_i$: $t_{max}(p_i) \leftarrow \min(t_{max}(p_i), t_{max}(p_j) - z_{ij})$. Similarly, if the relation $x(p_j) \Rightarrow x(p_i)$ can be inferred by the logical network then the other half of the propagation that propagates on time-point $p_j$ can be performed: $t_{min}(p_j) \leftarrow \max(t_{min}(p_j), t_{min}(p_i) + z_{ij})$. This observation is crucial: it allows propagation on the conditional bounds of time-points even when their presence statuses are not fixed. Most of the classical algorithms for propagating on Simple Temporal Networks [8] 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. For example, if a positive cycle of optional interval variables exists, and the presence statuses of these intervals are all equivalent, then all of these intervals can be made *absent*.

**Resource constraints** Fast and powerful propagation of resource constraints is essential for constraint-based scheduling. In CP Optimizer, the default filtering algorithm for all kind of resources (disjunctive, cumulative or state) is the timetable (see also Section 7.1). Sometimes timetable propagation is not powerful enough (*e.g.* for a proof of optimality) and in this case CP Optimizer contains a number of more expensive edge-finding based algorithms that can propagate more. In particular there are multiple $\mathcal{O}(n \log n)$ algorithms for disjunctive resources [52], and an $\mathcal{O}(n^2)$ time-table edge-finding algorithm for cumulative resources [53]. These algorithms are all able to handle optional interval variables. Additional filtering is turned on automatically during failure-directed search (see Section 6.5) or by the user by setting appropriate *inference level* parameters (see Section 5.2).

Author's personal copy

## 6.3 Large neighborhood search

Large Neighborhood Search (LNS) [49] is a component of CP Optimizer automatic search for scheduling consisting of successive relaxation and re-optimization phases. It operates by first finding an initial feasible solution. Then a number of iterations are carried out, each iteration comprising a relaxation step followed by a re-optimization of the relaxed solution. This process continues until some condition is satisfied, typically, when the solution can be proved to be optimal (for instance as a result of FDS, see Section 6.5) or when a time limit is reached. In CP Optimizer, this approach is made more robust by the application of machine learning techniques to portfolios of large neighborhoods and completion strategies in order to converge on the best combined method for the problem being solved [33].

The large neighborhoods are all based on the initial generation of a Partial Order Schedule (POS) [42] constructed from a completely instantiated solution where interval variables have fixed start and end values. A POS is a directed graph G($\mathcal{A}$, $\mathcal{E}$) where the nodes in $\mathcal{A}$ are the interval variables of the problem and the edges in $\mathcal{E}$ are precedence constraints between interval variables. An important property of the POS is that any solution of the graph is also a resource-feasible solution of the original problem: that is, it satisfies the constraints imposed on *interval sequence variables*, *cumul expressions* and *state function variables*. Algorithms for transforming a fully instantiated solution into a POS $P$ are described in [16, 33]. Large neighborhoods in the portfolio differ in the way they select a subset of interval variables to be relaxed (this subset is called the LNS *fragment*). The relaxed POS $P'$ is obtained by removing from $P$ all the edges involving at least one selected interval variable and adding new edges to repair broken paths. Furthermore, the presence status of interval variables belonging to the LNS fragment is also relaxed, allowing the relaxed interval variables to be re-allocated on different resources. The relaxed POS $P'$ is then used to enforce precedence constraints between interval variables before applying a completion strategy to re-optimize the relaxed fragment.

Completion strategies implement some variants of the *SetTimes* strategy recapped in [16]. These strategies explore a search tree with a maximal number of failures equal to $\gamma \cdot n$ where $n$ is the number of interval variables of the problem and $\gamma$ is a self-adapting parameter. They consider interval variables by increasing indicative start times and try to schedule them as close as possible to their indicative times. When a failure occurs, then in the right branch, the interval variable is marked "unselectable" and will remain so until constraint propagation removes from the current domain of the variable the start or end dates that were tried on the left branch. In case of a non-regular objective function (like earliness costs), some of these strategies are guided by the temporal linear relaxation of the problem (see below).

The convergence of LNS is accelerated by the exploitation of *objective landscapes*, a light-weight structure computed before the search that provides some information on the relation between decision variables and objective values [31].

## 6.4 Temporal linear relaxation

Real-world scheduling applications often require optimizing irregular temporal costs (such as minimizing earliness/tardiness, minimizing or maximizing activities' durations or delays between activities, *etc.*) as well as non-temporal costs (resource allocation, non-execution, *etc.*). When this type of objective function is involved, providing a good time placement of activities can be a challenge even in the absence of resource constraints. In this context, CP Optimizer automatically uses a linear relaxation of the problem solved with CPLEX in order to compute indicative presence, start and end values for interval variables of the model.

Author's personal copy

This linear relaxation is computed and solved at the root node of each LNS move and its solution is used to guide the completion goal. The Temporal Linear Relaxation is described in detail in [36]: it contains a relaxation of the constraints of the original problem (logical and precedence constraints, alternatives, spans, energetic relaxation of resource constraints, automatic convexification of the objective) as well as the precedence constraints from the relaxed POS of the LNS.

### 6.5 Failure-directed search

The Large Neighborhood Search described in Section 6.3 makes no effort to explore the whole search space of the problem. Therefore it cannot prove optimality (except for some relatively simple cases) and may also get trapped in a local minimum. For this reason the automatic search of CP Optimizer combines LNS with Failure-Directed Search (FDS) [54] that focuses on complete exploration of the search space. FDS is turned on automatically only when the search space seems to be small enough and when LNS is not improving the current solution any more. The assumption is that at this point there is no better solution or it is very hard to find. Therefore FDS focuses on finding dead-ends (failures) as quickly as possible. For more information see [54].

### 6.6 Parallel optimization

Parallel optimization on a set of $n$ workers is achieved by running in parallel $n$ variants of the automatic search algorithm described in this section. Note that as the algorithm is randomized, a variant may just be to run the same configuration of the algorithm but starting from two different random seeds for the random generator. The $n$ workers communicate their solutions so that if a particular worker finds an improving solution, the other workers can benefit from it (their LNS will work on improving this solution, and their FDS will consider the improved upper-bound value). Determinism of the parallel search is ensured by synchronizing the workers only at some deterministic points in time.

## 7 Benchmarks and performance

### 7.1 Basic propagation algorithms

We illustrate in this section the performance of two of the most used propagation algorithms of CP Optimizer : temporal constraints and timetable propagation of cumul functions. Times are reported on a MacBook Pro, 2.5 GHz Intel Core i7, 16GB RAM.

**Temporal Constraints** As we have seen in Section 6.2, all temporal constraints of the model are aggregated into a *temporal network* that is propagated thanks to dedicated algorithms. As a result, propagation of temporal constraints is much faster than the individual propagation of constraints of type $x - y \leq c$. Suppose a problem that simply consists of propagating a chain of $n$ activities. The CP Optimizer model reads:

```
int n = ...;
dvar interval a[1..n] size 10;
constraints {
  forall(i in 1..n)
    endBeforeStart(a[i-1],a[i]);
}
```

Author's personal copy

This model can be compared with the similar problem modeled with integer variables for start and end of activities and arithmetic constraints (`==` for duration and `>=` for precedence constraints). Figure 17 compares the propagation time of CP Optimizer (`CP Optimizer Scheduling`) with the propagation time of the integer model (`CP Optimizer Integer` is solved by CP Optimizer, `Choco Integer` is a similar model solved by Choco 4.0.1 [43]). In fact, the propagation time in the CP Optimizer model with interval variables is negligible compared to the one of the integer models (it is less than $0.1s$ for 20000 activities).

**Time-table Propagation** The time-table propagation algorithm of CP Optimizer is the main algorithm used to propagate cumul functions. It relies heavily on an incremental structure that is maintained during the search. Figure 18 compares the performance of the time-table implementation in CP Optimizer compared with the recent implementations proposed in [11]. Even if it is difficult to compare algorithms implemented in different languages, one see that the time-table implementation of CP Optimizer tends to outperform the ones studied in [11].

## 7.2 Performance on classical benchmarks

As mentioned in the introduction, the CP Optimizer search algorithm was designed from the beginning with the main objective to efficiently solve complex industrial scheduling problems. These problems are quite different from academic benchmarks: they are in general more structured, less pure (involving more heterogeneous elements), larger, using a finer time granularity, *etc*. In spite of this gap between industrial and academic problems, the CP Optimizer search also proves to be very efficient on classical scheduling problems. The preliminary version of the automatic search described in [16] and [33] was shown to be on a par with or to even outperform dedicated resolution algorithms on 21 different classical scheduling benchmarks. In [29], three problems are studied (two of which are the ones described in Sections 4.1 and 4.3) for which the CP Optimizer automatic search outperforms specialized algorithms. The experimental study reported in [54] provides new results on classical scheduling benchmarks such as *job shop* (15 instances closed out of the previously 48 open instances), *job shop with operators* (208 instances closed out of the 222 open instances), *flexible job shop* (74 instances closed out of the 107 open instances), *RCPSP* (52 new lower bounds), *RCPSP with max delays* (51 new lower bounds out of 58 instances), *multi-mode RCPSP* (535 instances closed out of 552) or *multi-mode RCPSP*

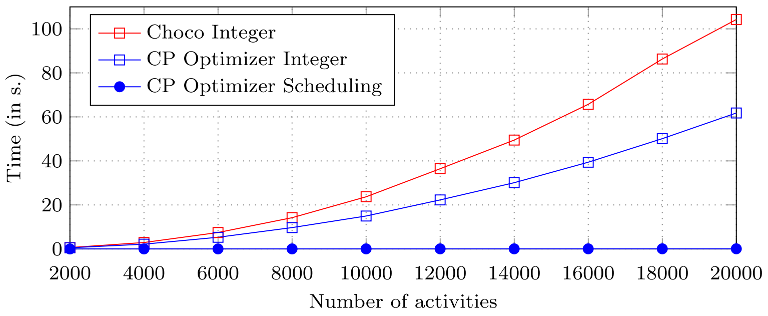

**Fig. 17** Propagation Time Comparison for a Chain of Activities

Author's personal copy

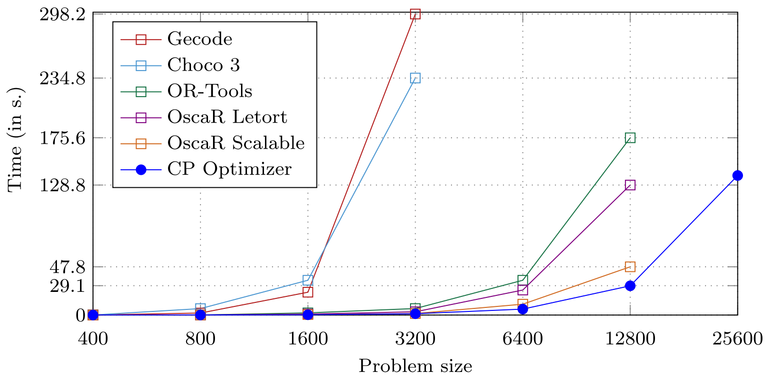

**Fig. 18** Comparison of Time-table implementations

*with max delays* (all instances of the benchmark closed). As far as *job shop* problems are concerned, a detailed comparison of the performance of CP Optimizer against a set of state-of-the-art MIP models was provided in [28] concluding that except for small problems, CP Optimizer dominates MIP both in terms of proving optimality and solution quality. Another performance example is the *Test Scheduling Problem* of the CSPLib [41] that was the topic of the Industrial Modeling Competition at CP 2015 won by a very concise CP Optimizer model solved with the automatic search. More recently, a very simple CP Optimizer model [32] was used to close the resource allocation and scheduling benchmark introduced in [20].

### 7.3 IBM scheduling benchmark

Since the creation of CP Optimizer, we have been continuously collecting scheduling problems from different sources (classical instances for famous scheduling problems like job-shop or RCPSP, benchmarks proposed in academic papers, industrial scheduling problems modeled by our consultants, partners or end-users, problems submitted on our Optimization forums, *etc.*). As of the end of 2017, our benchmark suite contains 145 different types of scheduling problem—each of the 4 problems described in Section 4 would be considered different types—with more than 3000 selected instances of these problems. Each problem instance consists of one `.cpo` file (see Section 8.2) that can be read and solved by the engine. The problems are quite diverse covering both disjunctive and cumulative scheduling, different problem sizes ranging from a few tens up to about one million activities and a wide range of objective functions involving makespan, weighted earliness-tardiness costs, resource allocation costs, activity non-execution penalties, resource transition costs, *etc.* This benchmark set is mostly used to monitor the performance improvements of the automatic search. The performance comparison between two versions of the automatic search is measured in terms of resolution time speed-up. As the engine is not able to solve all instances to optimality in reasonable time, the time speed-up estimates how many times faster a given version of the automatic search is able to reach similar quality solutions to the reference version. Figure 19 illustrates the performance evolution of the automatic search from version to version on scheduling problems since version 12.2 (released in Jan. 2011). For instance, we see that the latest release (version 12.7.1 released in Apr. 2017) is on average between 6 and 7 times faster than version 12.2 on the same machine. Figure 19

Author's personal copy

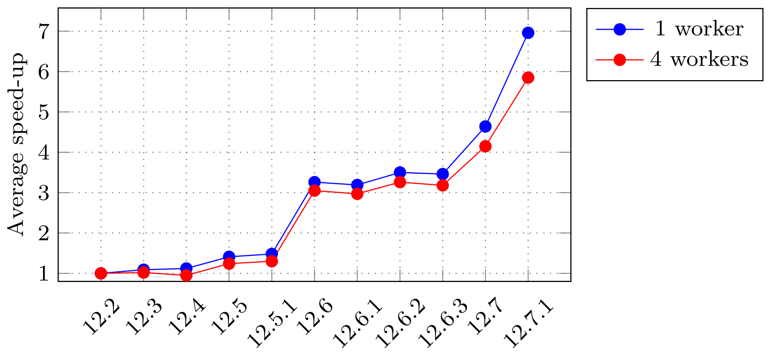

**Fig. 19** Performance evolution of CP Optimizer automatic search from version to version on scheduling problems. Reference is version 12.2

distinguishes the evolution of performance since version 12.2 for sequential search (1 worker) and parallel search with 4 workers. As of version 12.7.1, the search with 4 workers is about 2 times faster than the sequential search on scheduling problems.

## 8 APIs and tools

When working in an industrial context, the scheduling engine is usually integrated into a larger software component which constrains the choice of the programming language used for implementing the model. Furthermore, when developing the model, problem specifications are often changing and good quality data is not always available right at the beginning of the project. These considerations underline the importance of flexible APIs and efficient model development tools.

### 8.1 APIs

CP Optimizer is available in IBM ILOG CPLEX Optimization Studio together with the CPLEX Optimizers. Models can be designed and solved using OPL, C++, Python, Java or C#. All the modeling concepts (see Section 3) are available in all languages. As well as being distributed in CPLEX Optimization Studio, the Python API is also freely available for independent download or update from the Python Package Index as `docplex`.

### 8.2 Input/Output format

CP Optimizer is able to export the current model in a text file and import it back. The exported files are human readable and the `cpo` file format is fully documented. Unlike OPL, CPO files do not support *forall* statements or custom data types; on the other hand CPO files can be parsed and imported very quickly. The `cpo` files are a useful tool for debugging, especially for complicated applications since they are an exact representation of the generated model. When using a user-defined search, it is possible to export models from any search tree node. Models exported in this way contain the current variable domains, which is useful for inspecting constraint propagation.

Author's personal copy

## 8.3 Warnings

Like a compiler, CP Optimizer can analyze the model and print some warnings (see Fig. 20) when it thinks the model contains something potentially suspicious, independently of which language was used to create the model. The warnings include the suspicious elements of the model in the `cpo` file format, together with source code line numbers when available. CP Optimizer can currently report more than 70 types of warnings grouped in three importance levels.

## 8.4 Conflict refiner

The conflict refiner can be used to find an explanation for the infeasibility of a model. In the process of developing a model it often happens that, by nature of the model or because some errors slipped into the model or data, the problem is infeasible. In this case, the CP Optimizer conflict refiner functionality [30] can be used to identify a minimal infeasible subset of constraints.

Suppose for instance an initial version of the model of Fig. 13 for the Satellite Communication Scheduling problem for which all communications requests are compulsory (the `optional` keyword on line 11 is omitted). When trying to solve a particular instance of the problem, CP Optimizer would report the model as being infeasible. Running the conflict refiner on this problem would extract the following infeasible subset with 5 constraints:

```
Line    Iteration
20      t = <"134A",176,1>
20      t = <"144",191,1>
20      t = <"146",193,1>
20      t = <"146A",194,1>
22      r = <"LION",6,3>
```

On this instance looking at the data one can easily see that each of the 4 tasks in the conflict has a unique alternative on resource “LION” (capacity=3) with the following time windows and durations: [1232,1266]:19, [1238,1272]:31, [1228,1260]:22, [1230,1262]:22. Wherever we schedule these 4 tasks in their time-windows, all 4 overlap each other which

```
                              Warnings
cppfile.cpp:24: Warning: Unused interval variable 'x'.
    x = intervalVar(start=1..50, size=5..10)
javafile.java:20: Warning: Interval variable 'itv' has empty domain.
    itv = intervalVar(start=0..10, length=5, end=100..110)
pythonfile.py:7: Warning: Constraint is always true.
    x+y >= 5
pythonfile.py:8: Warning: Constraint is always false,
                 the model is infeasible.
    x+y < 5
satellite.cpo:2995:29: Warning: Constraint 'alternative':
                       there is only one alternative interval variable.
    alternative("task(134A,176,1)", ["alt(170,6,1232,19,1266)"], 1)
```

**Fig. 20** Example of Warnings

Author's personal copy

violates the resource capacity of 3. This conflict shows that the problem is over-constrained and the model can easily be adapted, typically by defining each task as optional and maximizing the number of executed tasks as done in Fig. 13.

The conflict refiner is an essential tool toward *explainable scheduling* for which it is necessary to produce an explanation why some particular decisions (resource allocation, time placement) were taken in a particular schedule.

## 9 Conclusion

In this article, we have presented a fairly complete overview of CP Optimizer for scheduling. CP Optimizer is a generic CP-based system to model and solve scheduling problems (among other combinatorial problems). It provides an algebraic language with simple mathematical concepts such as *intervals* and *functions* to capture the temporal dimension of scheduling problems in a combinatorial optimization framework. From the very beginning, CP Optimizer was designed with the goal of providing a similar experience as mathematical programming tools like CPLEX (also developed in our team) with a strong focus on usability [44]. In particular, CP Optimizer implements a model-and-run paradigm that vastly reduces the burden on the user to understand CP or scheduling algorithms: modeling is by far the most important. The automatic search provides good out-of-the-box performance and is continuously improving. The convergence with MP goes even further, with a convergence of the tools and functionalities around the engine, like an input/output format, modeling assistance with warnings and conflict refiner, *etc.* These tools accelerate the development and maintenance of models to be solved by the automatic search. They also limit the risks when developing a scheduling solution as it is very fast and easy to try new modeling ideas and face the changing requirements of the application.

CP Optimizer is continuously improving. As of today we are satisfied, and even sometimes pleasantly surprised, by the expressivity of the modeling language. Still, there are a few places where it could be parsimoniously extended in the future (for instance around cumul functions or sequence variables). Of course, we are planning to continue our effort to improve the performance and robustness of the automatic search both in general (integration of new search methods in the automatic search, improvement of the exploitation of parallelism) and also for some particular types of problems like routing problems or stochastic scheduling. Usability and consumability are also important topics: this may mean improving modeling assistance (for instance to make it easier to identify parts of the model that should be strengthened or reformulated) or making CP Optimizer even easier to deploy in applications.

## References

1. Aggoun, A., & Beldiceanu, N. (1993). Extending CHIP in order to solve complex scheduling problems. *Journal of Mathematical and Computer Modelling*, *17*, 57–73.
2. Booth, K., Nejat, G., Beck, C. (2016). A constraint programming approach to multi-robot task allocation and scheduling in retirement homes. In *Proceedings of the 22th international conference on principles and practice of constraint programming (CP 2016)* (pp. 539–555).
3. Booth, K., Tran, T., Nejat, G., Beck, C. (2016). Mixed-integer and constraint programming techniques for mobile robot task planning. *IEEE Robotics and Automation Letters*, *1*, 500–507.
4. Brafman, R.I. (2001). A simplifier for propositional formulas with many binary clauses. In *Proceedings of the 17th international joint conference on artificial intelligence (IJCAI 2001)* (pp. 515–522).

Author's personal copy

5. Cappart, Q., & Schaus, P. (2017). Rescheduling railway traffic on real time situations using time-interval variables. In *Proceedings of the 14th international conference on integration of AI and OR techniques in constraint programming (CPAIOR 2017)* (pp. 312–327).
6. Cesta, A., & Oddi, A. (1996). Gaining efficiency and flexibility in the simple temporal problem. In *Proceedings of the 3rd international workshop on temporal representation and reasoning (TIME 1996)* (pp. 45–50).
7. Cherkassky, B., Goldberg, A., Radzic, T. (1996). Shortest paths algorithms: theory and experimental evaluation. *Mathematical Programming*, *73*, 129–174.
8. Dechter, R., Meiri, I., Pearl, J. (1991). Temporal constraint networks. *Artificial Intelligence*, *49*(1-3), 61–96.
9. Dvořák, J., Heller, M., Hanzálek, Z. (2017). Makespan minimization of Time-Triggered traffic on a TarticleTarticleEthernet network. In *Proceedings of the IEEE 13th international workshop on factory communication systems (WFCS 2017)*.
10. Frank, J., Do, M., Tran, T.T. (2016). Scheduling ocean color observations for a GEO-Stationary satellite. In *Proceedings of the 26th international conference on automated planning and scheduling (ICAPS 2016)*.
11. Gay, S., Hartert, R., Schaus, P. (2015). Simple and scalable time-table filtering for the cumulative constraint. In *Proceedings of the 21st international conference on principles and practice of constraint programming (CP 2015)* (pp. 149–157).
12. GECODE: Gecode Toolkit (2016). Available at http://www.gecode.org/.
13. Gedik, R., Kirac, E., Milburn, A.B., Rainwater, C. (2017). A constraint programming approach for the team orienteering problem with time windows. *Computers & Industrial Engineering*, *107*, 178–195.
14. Gedik, R., Rainwater, C., Nachtmann, H., Pohlb, E. (2016). Analysis of a parallel machine scheduling problem with sequence dependent setup times and job availability intervals. *European Journal of Operational Research*, *251*, 640–650.
15. Giles, K., & van Hoeve, W.J. (2016). Solving a supply-delivery scheduling problem with constraint programming. In *Proceedings of the 22th international conference on principles and practice of constraint programming (CP 2016)* (pp. 602–617).
16. Godard, D., Laborie, P., Nuijten, W. (2005). Randomized large neighborhood search for cumulative scheduling. In *Proceedings of the 15th international conference on automated planning and scheduling (ICAPS 2005)* (pp. 81–89).
17. Gregory, A., & Majumdar, S. (2016). Energy aware resource management for MapReduce jobs with service level agreements in cloud data centers. In *Proceedings of the IEEE international conference on computer and information technology (CIT 2016)* (pp. 568–577).
18. Ham, A., & Cakici, E. (2016). Flexible job shop scheduling problem with parallel batch processing machines: MIP and CP approaches. *Computers & Industrial Engineering*, *102*, 160–165.
19. Han, J., Yuan, Z., Han, Y., Peng, C., Liu, J., Li, G. (2017). An adaptive scheduling algorithm for heterogeneous Hadoop systems. In *Proceedings of the IEEE/ACIS 16th international conference on computer and information science (ICIS 2017)* (pp. 845–850).
20. Hooker, J.N. (2007). Planning and scheduling by logic-based benders decomposition. *Operations Research*, *55*(3), 588–602.
21. IBM: ILOG CPLEX Optimization Studio 12.7.1: CP Optimizer Online Documentation (2017). Available at http://ibm.biz/COS1271Documentation.
22. Kinable, J. (2016). A reservoir balancing constraint with applications to bike-sharing. In *Proceedings of the 13th international conference on integration of AI and OR techniques in constraint programming (CPAIOR 2016)* (pp. 216–228).
23. Kinable, J., van Hoeve, W.J., Smith, S. (2016). Optimization models for a real-world snow plow routing problem. In *Proceedings of the 13th international conference on Integration of AI and OR techniques in constraint programming (CPAIOR 2016)* (pp. 229–245).
24. Kinnunen, T. (2016). Cost-efficient vacation planning with variable workforce demand and manpower. Technical report, Aalto University School of Science.
25. Kizilay, D., Eliiyi, D.T., Van Hentenryck, P. (2018). Constraint and mathematical programming models for integrated port container terminal operations. In *Proceedings of the 15th international conference on the integration of constraint programming, artificial intelligence, and operations research (CPAIOR 2018)*.
26. Kolisch, R., & Sprecher, A. (1996). PSPLIB - A project scheduling problem library. *European Journal of Operational Research*, *96*, 205–216.
27. Kramer, L.A., Barbulescu, L.V., Smith, S.F. (2007). Understanding performance tradeoffs in algorithms for solving oversubscribed scheduling. In *Proceedings of the 22nd AAAI conference on artificial intelligence (AAAI 2007)* (pp. 1019–1024).

Author's personal copy

28. Ku, W.Y., & Beck, J.C. (2016). *Mixed integer programming models for job shop scheduling: a computational analysis*. Computers & Operations Research.
29. Laborie, P. (2009). IBM ILOG CP Optimizer for detailed scheduling illustrated on three problems. In *Proceedings of the 6th international conference on integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CPAIOR 2009)* (pp. 148–162).
30. Laborie, P. (2014). An optimal iterative algorithm for extracting MUCs in a black-box constraint network. In *Proceedings of the 21st European conference on artificial intelligence (ECAI 2014)* (pp. 1051–1052).
31. Laborie, P. (2018). Objective landscapes for constraint programming. In *Proceedings of the 15th international conference on the integration of constraint programming, artificial intelligence, and operations research (CPAIOR 2018)*.
32. Laborie, P. (2018). An update on the comparison of MIP, CP and hybrid approaches for mixed resource allocation and scheduling. In *Proceedings of the 15th international conference on the integration of constraint programming, artificial intelligence, and operations research (CPAIOR 2018)*.
33. Laborie, P., & Godard, D. (2007). Self-adapting large neighborhood search: application to single-mode scheduling problems. In Baptiste, P., Kendall, G., Munier-Kordon, A., Sourd, F. (Eds.) *Proceedings of the 3rd multidisciplinary international conference on scheduling: Theory and applications (MISTA 2007)* (pp. 276–284). Paris.
34. Laborie, P., & Messaoudi, B. (2017). New results for the GEOCAPE observation scheduling problem. In *Proceedings of the 27th international conference on automated planning and scheduling (ICAPS 2017)* (pp. 382–390).
35. Laborie, P., & Rogerie, J. (2008). Reasoning with conditional time-intervals. In *Proceedings of the 21th international Florida artificial intelligence research society conference (FLAIRS 2008)* (pp. 555–560).
36. Laborie, P., & Rogerie, J. (2016). Temporal linear relaxation in IBM ILOG CP optimizer. *Journal of Scheduling*, *19*(4), 391–400.
37. Laborie, P., Rogerie, J., Shaw, P., Vilím, P. (2009). Reasoning with conditional time-intervals, part II: an algebraical model for resources. In *Proceedings of the 22th international Florida artificial intelligence research society conference (FLAIRS 2009)* (pp. 201–206).
38. Lazarev, A., Bronnikov, S., Gerasimov, A., Musatova, E., Petrov, A., Ponomarev, K., Kharlamov, M., Khusnullin, N., Yadrentsev, D. (2016). Mathematical modeling of the astronaut training scheduling. *Management of Large Systems*, *63*, 129–154. (in Russian).
39. Le Pape, C. (1994). Implementation of resource constraints in ILOG schedule: a library for the development of constraint-based scheduling systems. *Intelligent Systems Engineering*, *3*(2), 55–66.
40. Morton, T., & Pentico, D. (1993). *Heuristic scheduling systems*. NY: Wiley.
41. Mossige, M. CSPLib problem 073: Test scheduling problem. http://www.csplib.org/Problems/prob073.
42. Policella, N., Cesta, A., Oddi, A., Smith, S. (2004). Generating robust schedules through temporal flexibility. In *Proceedings of the 14th international conference on automated planning and scheduling (ICAPS 2004)* (pp. 209–218).
43. Prud'homme, C., Fages, J.G., Lorca, X. (2016). Choco documentation. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. http://www.choco-solver.org.
44. Puget, J.F. (2004). Constraint programming next challenge: simplicity of use. In *Proceedings of the 10th international conference on principles and practice of constraint programming (CP 2004)* (pp. 5–8).
45. Qin, T., Du, Y., Sha, M. (2016). Evaluating the solution performance of IP and CP for berth allocation with time-varying water depth. *Transportation Research*, *87*, 167–185.
46. Rainwater, C., Nachtmann, H., Adbesh, F. (2016). Optimal Dredge Fleet Scheduling within Environmental Work Windows. Technical report, Maritime Transportation Research and Education Center.
47. Roofigari-Esfahan, N., & Razavi, S. (2017). Uncertainty-aware linear schedule optimization: a space-time constraint-satisfaction approach. *Journal of Construction Engineering and Management, 143*(5).
48. Schmitt, M., & Stuetz, P. (2016). Perception-oriented cooperation for multiple UAVs in a perception management framework: system concept and first results. In *Proceedings of the IEEE/AIAA 35th digital avionics systems conference (DASC 2016)* (pp. 1–10).
49. Shaw, P. (1998). Using constraint programming and local search methods to solve vehicle routing problems. In *Proceedings of the 4th international conference on principles and practice of constraint programming (CP 1998)* (pp. 417–431).
50. Tran, T., Vaquero, T., Nejat, G., Beck, C. (2017). Robots in retirement homes: applying off-the-shelf planning and scheduling to a team of assistive robots. *Journal of Artificial Intelligence Research*, *58*, 523–590.
51. Van Hentenryck, P. (1999). *The OPL optimization programming language*. Cambridge: MIT Press.
52. Vilím, P. (2007). *Global constraints in scheduling*. Ph.D. thesis, Charles University in Prague, Faculty of Mathematics and Physics, Department of Theoretical Computer Science and Mathematical Logic,

Author's personal copy

KTIML MFF, Universita Karlova, Malostranské náměstí 2/25, 118 00 Praha 1, Czech Republic. http://vilim.eu/petr/disertace.pdf.

53. Vilím, P. (2011). Timetable edge finding filtering algorithm for discrete cumulative Resources. In Achterberg, T., & Beck, J. (Eds.) *Proceedings of the 8th international conference on integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CPAIOR-2011), Lecture notes in computer science* (Vol. 6697, pp. 230245). Berlin: Springer.
54. Vilím, P., Laborie, P., Shaw, P. (2015). Failure-directed search for constraint-based scheduling. In *Proceedings of the 12th international conference on integration of AI and OR techniques in constraint programming (CPAIOR 2015)* (pp. 437–453).