IBM ILOG CP Optimizer for Detailed Scheduling Illustrated on Three Problems

Philippe Laborie

ILOG an IBM company,
9, rue de Verdun, 94253 Gentilly Cedex, France
laborie@fr.ibm.com

Abstract. Since version 2.0, IBM ILOG CP Optimizer provides a new scheduling language supported by a robust and efficient automatic search. This paper illustrates both the expressivity of the modelling language and the robustness of the automatic search on three problems recently studied in the scheduling literature. We show that all three problems can easily be modelled with CP Optimizer in only a few dozen lines (the complete models are provided) and that on average the automatic search outperforms existing problem specific approaches.

Keywords: Constraint Programming, Scheduling.

1 Introduction

Since version 2.0, IBM ILOG CP Optimizer provides a new scheduling language supported by a robust and efficient automatic search. This new-generation scheduling model is based on ILOG's experience in applying Constraint-Based Scheduling to industrial applications. It was designed with the following requirements in mind [1,2]:

- It should be accessible to software engineers and to people used to mathematical programming;
- 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 fit naturally into a CP paradigm with clearly identified variables, expressions and constraints;
- 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 support a robust and efficient automatic search algorithm so that the user can focus on the declarative model without necessity to write any complex search algorithm (model-and-run development process).

The scheduling language is available in C++, Java, C# as well as in the OPL Optimization Programming Language[1]. The automatic search is based on a

[1] A trial version of OPL supporting this language can be downloaded on http://www.ilog.com/products/oplstudio/trial.cfm

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 148–162, 2009.

Self-Adapting Large Neighbourhood Search that iteratively *unfreezes* and *re-optimizes* a selected fragment of the current solution. The search algorithm is out of the scope of this paper, the principles of the approach have been described in [3] whereas more details about constraint propagation are available in [1].

The present paper illustrates the new modelling language and the efficiency and robustness of the automatic search on three problems recently studied in the scheduling literature. These problems were selected for several reasons:

- They are quite different in nature, covering cumulative and disjunctive scheduling, non-preemptive and preemptive scheduling, alternative modes, structured and unstructured temporal networks, *etc.*;
- All three problems are optimization problems with realistic non-regular objective functions (earliness/tardiness costs, number of executed tasks, complex temporal preference functions);
- They cover a range of different application domains (manufacturing, aerospace, project scheduling);
- Benchmarks and recent results are available to evaluate the efficiency of CP Optimizer's automatic search.

Section 2 recaps the modelling concepts of CP Optimizer used in the paper. Sections 3 to 5 are dedicated to the three scheduling problems: a flow-shop problem with earliness and tardiness costs [4], the oversubscribed scheduling problem studied in [5] and the personal task scheduling problem introduced in SelfPlanner [6]. Each of these sections starts with a description of the problem followed by a problem formulation in OPL. We show that all problems are easily modelled with CP Optimizer and that the resulting models are very concise (ranging from 15 to 42 lines of code). These models are then solved using the automatic search of CP Optimizer 2.1.1 with default parameter values on a 3GHz Linux desktop. We show that, in spite of its generality, the default search of CP Optimizer outperforms state-of-the-art approaches on all three problems.

2 CP Optimizer Model for Detailed Scheduling

This section recaps the conditional interval formalism introduced in [1,2]. It extends classical constraint programming by introducing with parsimony additional mathematical concepts (such as intervals, sequences or functions) as new variables or expressions to capture the temporal aspects of scheduling[2]. In this section we focus on the modelling concepts that are sufficient to understand the three models detailed in sections 3-5. A more formal and exhaustive description of the CP Optimizer concepts for detailed scheduling as well as several examples are provided in [7].

[2] In the present paper, a few concepts have been renamed so as to be consistent with their implementation in IBM ILOG CP Optimizer. In particular, we speak of *present*/*absent* rather than *executed*/*non-executed* interval variable and the notion of interval *duration* is replaced by the notion of interval *length*.

2.1 Interval Variables

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

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

Absent interval variables have special meaning. Informally speaking, an absent interval variable is not considered by any constraint or expression on interval variables it is involved in. For example, if an absent interval variable a is used in a precedence constraint between interval variables a and b then, this constraint does not influence interval variable b. Each constraint and expression specifies how it handles absent interval variables.

By default interval variables are supposed to be present but they can be specified as being **optional** meaning that $\perp$ is part of the domain of the variable and thus, it is a decision of the problem to have the interval present or absent in the solution. Optional interval variables provide a powerful concept for efficiently reasoning with optional or alternative activities. The following constraints on interval variables are introduced to model the basic structure of scheduling problems. Let a, a_i and b denote interval variables and z an integer variable:

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

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 optional activities, alternative modes/recipes/processes, etc.) in a well-defined CP paradigm.

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

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

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

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

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

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

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

2.2 Sequence Variables

Many problems involve scheduling a set of activities on a disjunctive resource that 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, additional constraints such as resource setup times or constraints on the relative position of activities in the sequence are common. To capture this idea we introduce the notion of *sequence variable*, a new type of decision variable whose value is a permutation of a set of interval variables. Constraints on 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).

A **sequence variable** p is defined on a set of interval variables A. A value of p is a permutation of all present intervals of A. For instance, if $A = \{a, b\}$ is a set of two interval variables with a being necessarily present and b optional, the domain of the sequence p defined on A consists of 3 permutations: $\{(a), (a, b), (b, a)\}$.

If p denotes a sequence variable and a, b two interval variables in the sequence, the **sequencing constraints** first(p, a) and last(p, a) respectively mean that if interval a is present, it is the first or last in sequence p. Sequencing constraints before(p, a, b) and prev(p, a, b) respectively mean that if both intervals a and b are present, then a is before or immediately before b in sequence p.

It is to be noted that the sequencing constraints above do not have any impact on the start and end values of intervals, they only constrain the possible values (permutations) of the sequence variable. The **no-overlap constraint** noOverlap(p) on a sequence variable p states that permutation p 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 permutation.

For modelling sequence dependent setup times, each interval variable a in a sequence p can be associated with a non-negative integer **type** $T(p, a)$ and the no-overlap constraint can be associated with a transition distance. A **transition distance** M is a function $M : [0, n) \times [0, n) \rightarrow \mathbb{Z}^+$. If a and b are two successive non-overlapping present intervals, the no-overlap constraint noOverlap(p, M) will express a minimal distance $M(T(p, a), T(p, b))$ between the end of a and the start of b.

2.3 Cumul Function Expressions

For cumulative resources, the cumulated usage of the resource by the activities is a function of time. An activity usually increases the cumulated resource usage function at its start time and decreases it when it releases the resource at its end time. For resources that can be produced and consumed by activities (for instance the content of an inventory or a tank), the resource level can also be described as a function of time: production activities will increase the resource level whereas consuming activities will decrease it. In these problem classes, constraints are imposed on the evolution of these functions of time, for instance a maximal capacity or a minimum safety level.

CP Optimizer introduces the notion of a *cumul function expression* which is a constrained expression that represents the sum of individual contributions of intervals. A set of elementary cumul functions is available to describe the individual contribution of an interval variable (or a fixed interval of time or a fixed date). These elementary functions cover the use-cases mentioned above: *pulse* for usage of a cumulative resource, and *step* for resource production/consumption (see Figure 1). It is important to note that the elementary cumul functions

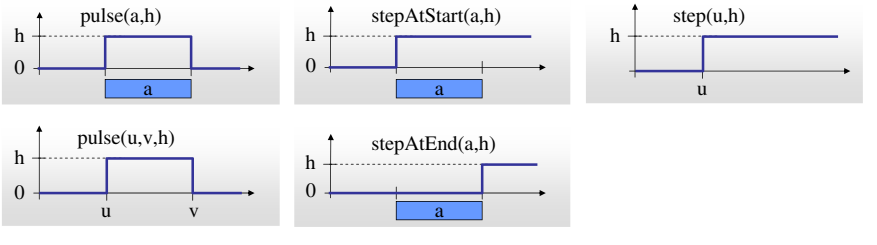

Fig. 1. Elementary cumul function expressions

defined on an interval variable are equal to the zero function when the interval variable is absent.

A **cumul function expression** f is defined as the sum of a set of elementary functions f_i or their negations: $f = \sum_i \epsilon_i \cdot f_i$ where $\epsilon_i \in \{-1, +1\}$. When the elementary cumul functions f_i that define a cumul function f are fixed (and thus, so are their related intervals), the cumul function itself is fixed and its value is a stepwise integer function. Several constraints are provided over cumul functions. These constraints allow restricting the possible values of the function over the complete horizon or over some fixed or variable interval. Let $u, v \in \mathbb{Z}$, $h, h_{min}, h_{max} \in \mathbb{Z}^+$ and a denote an interval variable. The following constraints are available on a cumul function f to restrict its possible values:

- alwaysIn$(f, u, v, h_{min}, h_{max})$ means that the values of function f must remain in the range $[h_{min}, h_{max}]$ everywhere on the fixed interval $[u, v)$.
- alwaysIn(f, a, h_{min}, h_{max}) means that if interval a is present, the values of function f must remain in the range $[h_{min}, h_{max}]$ between the start and the end of interval variable a.
- $f \leq h$: function f cannot take values greater than h.
- $f \geq h$: function f cannot take values lower than h.

3 Flow-Shop with Earliness and Tardiness Costs

3.1 Problem Description

The first problem studied in the paper is a flow-shop scheduling problem with earliness and tardiness costs on a set of instances provided by Morton and Pentico [8] that have been used in a number of studies including GAs [9] and Large Neighbourhood Search [4]. In this problem, a set of n jobs is to be executed on a set of m machines. Each job i is a chain of exactly m operations, one per machine. All jobs require the machines in the same order that is, the position of an operation in the job determines the machine it will be executed on. Each operation j of a job i is specified by an integer processing time $pt_{i,j}$. Operations cannot be interrupted and each machine can process only one operation at a time. The objective function is to minimize the total earliness/tardiness cost. Typically, this objective might arise in just-in-time inventory management: a late job has negative consequence on customer satisfaction and time to market, while an early job increases storage costs. Each job i is characterized by its release date rd_i, its due date dd_i and its weight w_i. The first operation of job i cannot start before the release date rd_i. Let C_i be the completion date of the last operation of job i. The earliness/tardiness cost incurred by job i is $w_i \cdot abs(C_i - dd_i)$. In the instances of Morton and Pentico, the total earliness/tardiness cost is normalized by the sum of operation processing times so the global cost to minimize is:

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

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

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

3.2 Model

A complete OPL model for this problem is shown in Model 1. The instruction at *line 1* tells the model is a CP model to be solved by CP Optimizer. The section between *line 2* and *line 8* is data reading and data manipulation. The number of jobs n is read from the data file at *line 2* and the number of machines m at *line 3*. A number of arrays are defined to store, for each on the n jobs, the release date (*line 4*), due date (*line 5*), earliness/tardiness cost weight (*line 6*) and, for each machine, the processing time of each operation on the machine (*line 7*). The normalization factor W is computed at line *line 8*. The model itself is declared between *line 9* and *line 20*. *Line 9* creates a 2-dimensional array of interval variables indexed by the job i and the machine j. Each interval variable represents an operation and is specified with a size corresponding to the operation's processing time. *Line 10* creates one integer expression $C[i]$ for each job i equal to the end of the m^{th} (last) operation of the job. These expressions are used in *line 11* to state the objective function. The constraints are defined between *line 13* and *line 19*. For each job, *line 14* specifies that the first operation of job i cannot start before the job release date whereas precedence constraints between operations of job i are defined at *lines 15-16*. *Lines 18-19* state that for each machine j, the set of operations requiring machine j do not overlap.

3.3 Experimental Results

Table 1 compares the results obtained by the default automatic search of CP Optimizer using the above model (col. CPO) with the best results obtained by various genetic algorithms as reported in [9] (col. GA-best) and the results of the

Table 1. Results for Flow-shop Scheduling with Earliness and Tardiness Costs

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

best Large Neighbourhood Search (S-LNS) studied in [4] (col. S-LNS-best). A time limit of one hour was used on a 3GHz processor for CP Optimizer similar to the two hours limit used in [4] on a 1.5GHz processor. The average improvement (using the geometric mean over the ratio $value_{CPO}/value_{Other}$) over the best GA is about 25% whereas the average improvement over the best LNS is more modest (1.7%).

4 Satellite Scheduling

4.1 Problem Description

The second illustrative model is an oversubscribed scheduling problem described in [5]. This model is a generalization of two real-world oversubscribed scheduling domains, the USAF Satellite Control Network (AFSCN) scheduling problem and the USAF Air Mobility Command (AMC) airlift scheduling problem. These two domains share a common core problem structure:

- A problem instance consists of n tasks. In AFSCN, the tasks are communication requests; in AMC they are mission requests.
- A set Res of resources are available for assignment to tasks. Each resource $r \in Res$ has a finite capacity $cap_r \geq 1$. The resources are air wings for AMC and ground stations for AFSCN. The capacity in AMC corresponds to the number of aircraft for a wing; in AFSCN it represents the number of antennas available at the ground station.
- Each task T_i has an associated set Res_i of feasible resources, any of which can be assigned to carry out T_i. Any given task T_i requires 1 unit of capacity (i.e., one aircraft in AMC or one antenna in AFSCN) of the resource $r_j \in Res_i$ that is assigned to perform it. The duration $Dur_{i,j}$ of task T_i depends on the allocated resource r_j.
- Each of the feasible alternative resources $r_j \in Res_i$ specified for a task T_i defines a time window within which the duration of the task needs to be allocated. This time window corresponds to satellite visibility in AFSCN and mission requirements for AMC.
- All tasks are optional; the objective is to minimize the number of unassigned tasks[3].

[3] A second type of model with task priorities is also described in [5]. In the present paper, we focus on the version without task priorities.

4.2 Model

A complete OPL model for this problem is shown in Model 2 using the AFSCN semantics. The section between *line 2* and *line 6* is data reading and data manipulation. A tuple defining ground stations data (with a name, a unique integer identifier and a capacity) is defined at *line 2* and read from the data file at *line 4*. A tuple defining a possible resource assignment for a task (specifying a task, a station, a task minimal start time, a task duration and a task maximal end time) is defined at *line 3* and read from the data file at *line 5*. The set of all tasks $Tasks$ is computed at *line 6* as the set of tasks used in at least one possible assignments.

Model 2 - OPL Model for Satellite Scheduling

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

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

The constraints *lines 11-12* state that each task, if present, is the alternative among the set of possible assignments for this task, this is modelled by an alternative constraint: if interval $task[t]$ is present, then one and only one of the intervals $alt[a]$ representing a ground station assignment for $task[t]$ will be present and $task[t]$ will start and end together with this selected interval. As specified by the semantics of the alternative constraint, if the task is absent, then all the possible assignments related with this task are absent too. The limited capacity (number of antennas) of ground stations is modelled by *lines 13-14*.

For each ground station s, a cumul function is created that represents the time evolution of the number of antennas used by the present assignments on this station s. This is a sum of unit pulse functions $\text{pulse}(alt[a], 1)$. Note that when the assignment $alt[a]$ is absent, the resulting pulse function is the zero function so it does not impact the sum. The resulting sum is constrained to be lower than the maximal capacity *cap* of the station. An interesting feature of the CP Optimizer model is that it handles optional tasks in a very transparent way: here, the fact that tasks are optional only impacts the declaration of task intervals at *line 7*. The notion of optional interval variable and the handling of absent intervals by the constraints and expressions of the model (here the alternative constraint and the cumul function expressions) allows an elegant modelling of scheduling problems involving optional activities and, more generally, optional and/or alternative tasks, recipes or modes.

4.3 Experimental Results

Table 2 compares the results obtained by the default automatic search of CP Optimizer using the above model (col. CPO) with the TaskSwap (TS) and Squeaky Wheel Optimization (SWO) approaches studied in [5] (col. TS and SWO). Figures represent the average number of unscheduled tasks for each problem set of the benchmark. The time limit for each instance was fixed to 120s for problem sets $x.1$, 180s for problem sets $x.2$ and 360s for problem sets $x.3$. In average, compared to the best approach described in [5] (SWO), the default automatic search of CP Optimizer assigns 5.3% more tasks.

Table 2. Results for Satellite Scheduling

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

5 Personal Task Scheduling

5.1 Problem Description

The third problem treated in this paper is the personal task scheduling problem introduced in [6] and available as an add-on to Google Calendar (`selfplanner.uom.gr/`). It consists of a set of n tasks $\{T_1, ..., T_n\}$. Each task T_i has a duration denoted dur_i. All tasks are considered preemptive, i.e. they

can be split into parts that can be scheduled separately. The decision variable p_i denotes the number of parts in which the i^{th} task has been split, where $p_i \geq 1$. T_{ij} denotes the j^{th} part of task T_i, $1 \leq j \leq p_i$. The sum of the durations of all parts of a task T_i must equal its total duration dur_i. For each task T_i, a minimum and maximum allowed duration for its parts, $smin_i$ and $smax_i$, as well as a minimum allowed temporal distance between every pair of its parts, $dmin_i$ are given. Depending on the values of $smax_i$ and $smin_i$ and the overall duration of the task dur_i, implicit constraints are imposed on p_i. For example, if $dur_i < 2 * smin_i$, then necessarily $p_i = 1$ and task T_i is non-preemptive. Each task T_i is associated with a domain $D_i = [s_{i1}, e_{i1}] \cup [s_{i2}, e_{i2}] \cup \cdots \cup [s_{iF_i}, e_{iF_i}]$, consisting of a set of F_i time windows within which all of its parts have to be scheduled. We denote respectively $L_i = s_{i1}$ and $R_i = e_{iF_i}$ the leftmost and rightmost values of domain D_i. A set of m locations is given, $Loc = \{l_1, l_2, \cdots, l_m\}$ as well as a 2-dimensional matrix $Dist$ with their temporal distances represented as non-negative integers. Each task T_i has its own spatial reference, $loc_i \in Loc$, denoting the location where the user should be in order to execute each part of the task. A set of ordering constraints, denoted $\prec (T_i, T_j)$ between some pairs of tasks is also defined, meaning that no part of task T_j can start its execution until all parts of task T_i have finished their execution. Time preferences are expressed for each task T_i. Five types of preference functions are available; they are depicted on Figure 2:

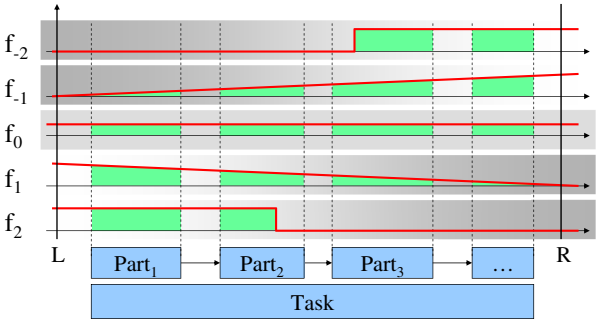

Fig. 2. Preference functions

f_{-2} Execute as much as possible of task T_i after a date d.
f_{-1} Execute as much as possible of task T_i as late as possible.
f_0 No preference.
f_1 Execute as much as possible of task T_i as early as possible.
f_2 Execute as much as possible of task T_i before a date d.

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

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

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

5.2 Model

A complete OPL model for the personal task scheduling problem is shown in Model 3. The section between *line 2* and *line 16* is data reading and data manipulation. A tuple representing a task description is declared at *line 2*, it specifies a unique integer task identifier, the location of the task, the task duration, the minimal and maximal duration of task parts, the minimal delay between two consecutive task parts, an identifier of the type of preference function for the task in $\{-2, -1, 0, 1, 2\}$, the threshold date in case preference function is of type f_{-2} or f_2 and two sets of integers *ds* and *de* respectively representing the start and end dates of the intervals $[s_i, e_i]$ of the task domain. The set of tasks is read from the data file at *line 3*. A triplet representing the temporal distance between two locations is declared at *line 4* and the transition distance matrix represented as a set of such triplets is read from the data file at *line 5*. A tuple storing an ordering constraint is defined on *line 6* and a set of such tuples is read from the data at *line 7*. *Lines 8-10* respectively compute, for each task t the leftmost value, rightmost value and diameter of the task domain. A tuple representing the i^{th} part of a task is defined at *line 11* and the total set of possible parts is computed at *line 12* considering that for each task of duration *dur* and minimal part duration *smin*, the maximal number of parts is $\lfloor dur/smin \rfloor$. *Lines 13-16* define a step function $holes[t]$ for each task t that is equal to 1 in the domain of t and to 0 everywhere else.

Variables and constraints are defined between *lines 17 and 42*. An array of interval variables, one interval $task[t]$ for each task t, is declared at *line 17*; each task is constrained to end before the schedule horizon (500 in the benchmark). *Line 18* defines an optional interval variable for each possible task part with a minimal and a maximal size given by *smin* and *smax*. A sequence variable is created at *line 19* on the set of all parts p, each part being associated with an integer type in the sequence corresponding to the location of the part. The satisfaction expression for each task t is modelled on *lines 20-25* depending on the preference function type; it uses the OPL conditional expression `c?e1:e2` where c is a boolean condition and $e1$ is the returned expression if c is true and $e2$ the returned expression if c is false. The normalization factors are the ones used in [6][4]. The objective function, as defined on *line 26* is to maximize the sum of all tasks satisfaction.

The constraints on *line 29* forbid any part of a task t to overlap a point where the step function $holes[t]$ is zero; this will constrain each task part to be executed in its domain. Constraints on *lines 31-32* state that the set of parts of a given task t forms a chain of optional intervals with minimum separation time *dmin* among which only the first ones will be executed, that is, each part $a[p]$ if present is constrained to be executed before its successor part $a[s]$ and the presence of

[4] The objective expression being quite complex, we used the solution checker provided with the instances to check that the constraints and objective function of our model are equivalent to the ones used in [6].

part $a[s]$ implies the presence of part $a[p]$. Constraints on *line 36* state that the total duration of the part of a task must equal the specified task duration *dur*. Note that when part $a[p]$ is absent, by default the value of sizeOf($a[p]$) is 0. *Line 37* constrains each task t to span its parts, that is to start at the start of first

Model 3 - OPL Model for Personal Task Scheduling

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

part and to end with the end of the last executed part. Ordering constraints are declared on *line 40* whereas *line 41* states that task parts cannot overlap and that they must satisfy the minimal transition distance between task locations defined by the set of triplets *Dist*.

5.3 Experimental Results

Table 3 compares the results obtained by the default automatic search of CP Optimizer using the above model (col. CPO) and a time limit of 60s for each problem with the Squeaky Wheel Optimization (SWO) approach implemented in SelfPlanner [6] (col. SWO). CP Optimizer finds a solution to more problems than the approach described in [6]: the SWO could not find any solution for the problems with 55 tasks whereas the automatic search of CP Optimizer solves 70% of them. Furthermore, SWO could not find any solution to 4 of the smaller problems with 50 tasks whereas CP Optimizer solves them all but for problem 50-2. On problems where SWO finds a solution, the average task satisfaction (average of the ratio between the total satisfaction and the number of tasks) is 78% whereas it is 87.8% with CP Optimizer. It represents an improvement of about 12.5% in solution quality.

Table 3. Results for Personal Task Scheduling

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

6 Conclusion

This paper illustrates the new scheduling support in IBM ILOG CP Optimizer. We selected three problems recently studied in the scheduling literature and provide a simple and concise CP Optimizer model for each of them. The size of the

OPL models range from 15 to 42 lines of code. These models are then solved using the automatic search of CP Optimizer with default parameter values. We show that on average, CP Optimizer outperforms state-of-the-art problem specific approaches on all the problems which is quite a remarkable result given the generality of the search and the large spectrum of problem characteristics. These results are consistent with our experience of using CP Optimizer on industrial detailed scheduling applications. In spite of the relative simplicity of the new scheduling language based on optional interval variables, it was shown to be expressive and versatile enough to model a large range of complex problems for which the automatic search proved to be efficient and robust. The major part of the future development of CP Optimizer will be the continued improvement of the automatic search process.

References

1. Laborie, P., Rogerie, J.: Reasoning with Conditional Time-intervals. In: Proc. 21th International FLAIRS Conference (FLAIRS 2008), pp. 555–560 (2008)
2. Laborie, P., Rogerie, J., Shaw, P., Vilím, P.: Reasoning with Conditional Time-intervals, Part II: an Algebraical Model for Resources. In: Proc. 22th International FLAIRS Conference (FLAIRS 2009) (2009)
3. Laborie, P., Godard, D.: Self-Adapting Large Neighborhood Search: Application to Single-mode Scheduling Problems. In: Proc. of the 3rd Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA), pp. 276–284 (2007)
4. Danna, E., Perron, L.: Structured vs. Unstructured Large Neighborhood Search: a Case Study on Job-shop Scheduling Problems with Earliness and Tardiness Costs. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 817–821. Springer, Heidelberg (2003)
5. Kramer, L.A., Barbulescu, L.V., Smith, S.F.: Understanding Performance Tradeoffs in Algorithms for Solving Oversubscribed Scheduling. In: Proc. 22nd AAAI Conference on Artificial Intelligence (AAAI 2007), pp. 1019–1024 (2007)
6. Refanidis, I.: Managing personal tasks with time constraints and preferences. In: Proc. 17th International Conference on Automated Planning and Scheduling Systems (ICAPS 2007), pp. 272–279 (2007)
7. Laborie, P., Rogerie, J., Shaw, P., Vilím, P., Wagner, F.: ILOG CP Optimizer: Detailed Scheduling Model and OPL Formulation. Technical Report 08-002, ILOG (2008), http://www2.ilog.com/techreports/
8. Morton, T., Pentico, D.: Heuristic Scheduling Systems. Wiley, Chichester (1993)
9. Vázquez, M., Whitley, L.D.: A Comparison of Genetic Algorithms for the Dynamic Job Shop Scheduling problem. In: Proc. GECCO 2000 (2000)