	Date: October 2008	Version: 1.0	Id:
	ROSTUDEL Professional Services		
<h1>CONSTRAINT-BASED SCHEDULING AND MIXED APPROACHES</h1>			

*Abstract*

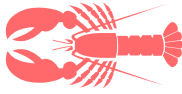
*Scheduling problems are hard combinatorial problems in practice, due to resource scarcity and other operational side-constraints. This complexity prevents unfortunately to tackle such problems with a brute Math Programming (MP) approach such as Mixed Integer Linear Programming (MILP).*

*We summarize here the advantage of Constraint Programming (CP) approaches to both model and solve highly combinatorial constrained problems, thanks to state-of-the art ILOG OPL6 platform. We also emphasize the benefits of integrating MP and CP in decomposition frameworks.*



**TABLE OF CONTENTS**

- 1. CONSTRAINT PROGRAMMING CONCEPTS.....3**
  - 1.1. CP CONCEPTS.....3
  - 1.2. CP AND SCHEDULING : CONSTRAINT-BASED SCHEDULING .....4
- 2. MODELING THE MAINTENANCE SCHEDULING FOR A POWER SYSTEM USING OPL6.....5**
  - 2.1. INTERVALS AS DECISION VARIABLES .....5
  - 2.2. OBJECTIVE FUNCTION .....6
  - 2.3. PRECEDENCE CONSTRAINTS.....6
  - 2.4. DISJUNCTIVE SCHEDULING .....7
  - 2.5. CUMULATIVE SCHEDULING.....7
  - 2.6. CUMULATIVE FUNCTIONS .....7
    - 2.6.1 *Sample 1* .....7
    - 2.6.2 *Sample 2* .....9
    - 2.6.3 *Sample 3* .....9
- 3. CP AND MP COLLABORATION .....10**
- 4. BIBLIOGRAPHY / BIBLIOGRAPHIE .....11**



## 1. CONSTRAINT PROGRAMMING CONCEPTS

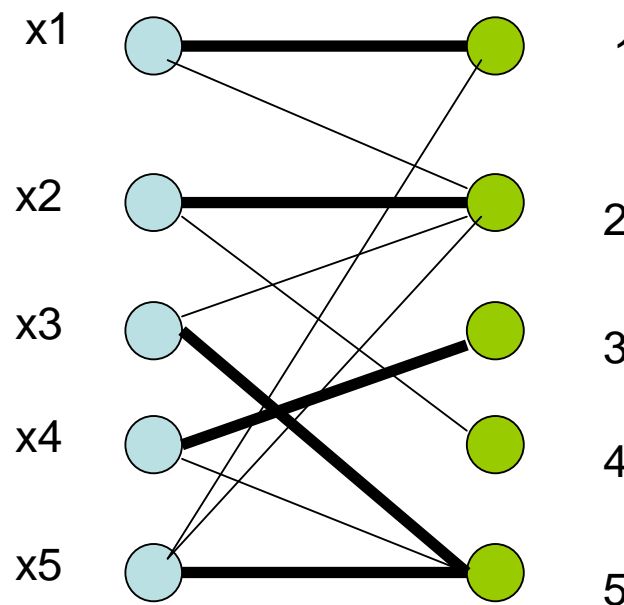
### 1.1. CP concepts

Constraint-based approaches take advantages of global constraints rather than linearizing them into several linear independent constraints. For instance, an “alldifferent” constraint between  $N$  variables is a global constraint that guarantees that no pair of variable can take the same value in any solution. One could derive a set of binary constraints of the form  $X[i] \neq X[j]$  and linearize them, but we will see that CP approach is more efficient to handle globally the set of variables.

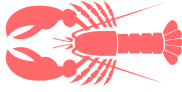
As in MP, searching for solutions includes domain an exhaustive search using branching and pruning, but while MP essentially searches for optimality, CP searches for feasibility. In MP, we prune branches for cost-related reasons, while in CP, we prune branches due to inconsistency (the domain of a variable becomes empty).

The concept of *inference* is at the core of constraint-based approaches: given a constraint involving a set of constraints, how can we “infer” or deduce some impacts on the set of variables domains? Are we sure that all these values are consistent or could we remove some of them? In constraint programming, one is interested to develop for each global constraint a scheme that guarantees incrementally that all the values potentially assignable to each variable are consistent with the constraint satisfiability. In other words, there is a filtering algorithm that removes all inconsistent values for variables and keeps only the values for each variable which are compatible with a feasible solution: this is the concept of “*arc-consistency*”.

Take again the “all different” sample and imagine that we have 3 variables  $A, B, C$  with respective domain  $\{1,2,3\}, \{2,3\}, \{2,3\}$ . If we treat independently this global constraint, we would post:  $A \neq B$ ,  $B \neq C$  and  $A \neq C$ . None of these 3 constraints is able to deduce itself that 2 and 3 are incompatible values to assign to  $A$ . Look now at the same constraint with another set of variables and values:



A dedicated filtering algorithm proposed by Regin[2] will rather treat the constraint as a global constraint by recognizing the “matching” structure between variables and values (in bold in the figure). In other words, the all different constraint is satisfied if and only iff there is a matching between variables and values in the above bipartite graph. And there are excellent and fast algorithms that are able to compute a maximal matching in polynomial time. A good filtering algorithm for the “all different” constraint will naturally embed such algorithm.



Applying high level Operations Research (OR) techniques to maintain arc or domain-consistency is at the core of constraint programming approaches, in order to prune unfeasible values as soon as possible, and keep an incremental behaviour of the global constraint when branching decisions are made during the search procedure.

For a review of integration of OR and CP, one will read with great interest Milano & Wallace[3] and John Hooker [4]. Another cornerstone of combinatorial optimization is local search, which is also a constraint-based reasoning [5].

## 1.2. CP and scheduling : constraint-based scheduling

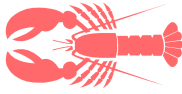
“*Constraint-Based Scheduling can be defined as the discipline that studies how to solve scheduling problems by using Constraint Programming*”[6]. It is a very successful application of CP due to its model-based computing aspect: concepts of activities, resources, precedence constraints... are natural and at a high level of encapsulation of variables and global constraints [7]. ILOG new release of OPL Studio platform – OPL6 [8] is even at a higher level of encapsulation with the concept of optional interval and powerful cumulative functions.

Once again, dedicated algorithms maintain arc-consistency such as the famous **edge-finder** method for **disjunctive scheduling**, which forbids two (or more) tasks to overlap. This is typically the case for a machine that can only perform one task at a time (**unary resource**).

Scheduling applications are numerous:

- one machine disjunctive scheduling with or without pre-emption
- cumulative scheduling : machines have a finite capacity
- precedence constraints between tasks
- open-shop : tasks to be computed by unary resources (machines) without any particular order
- Job-shop scheduling: jobs gather tasks that must be processed through unary resources with precedence constraints. One is then interested to compute an order between the machines.
- Scheduling with alternative resources
- Scheduling with state resources
- ...

Search strategies have been intensively studied and derived into algorithms such as “setTimes”, “setAlternatives” or Large Neighbourhood Search (LNS [9])



## 2. Scheduling models with OPL6

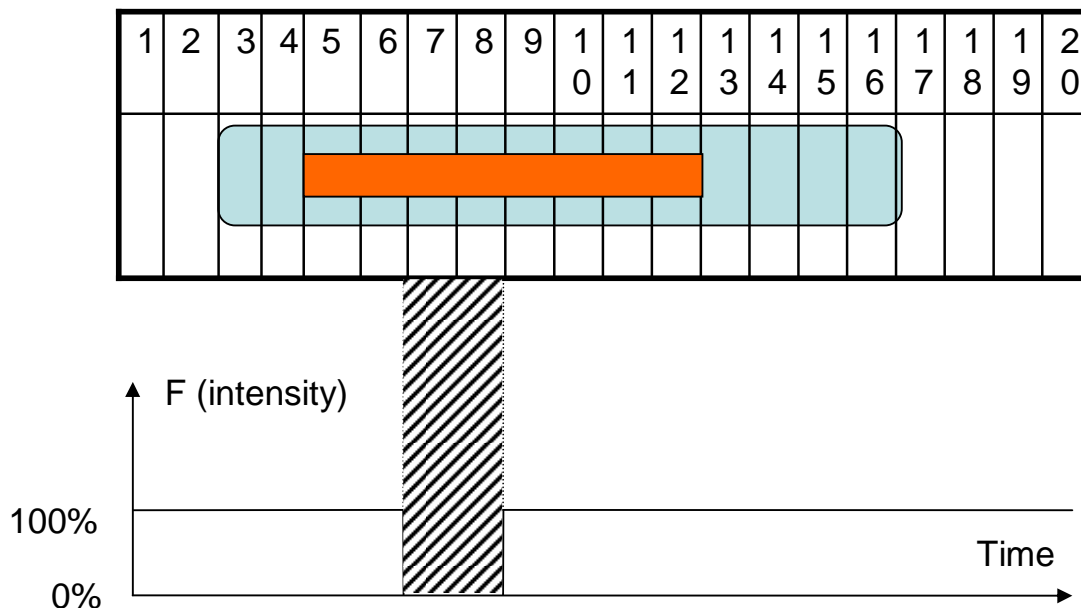
Several frameworks exist to implement Constraint Programming: ECLIPSE, CHOCO, KOALOG, ILOG SOLVER, ILOG SCHEDULER, ILOG OPL[8], COMET [7]... OPL and COMET are particularly interesting because they clearly separate model and search and provide a very powerful language to capture complex data structures, iterate on collections with filter, and of course provide a library of global constraints (alldifferent, cardinality constraint, inverse...), including scheduling constraints.

We will focus on the latest release of OPL6, because it integrates a new API dedicated for scheduling. This API is based on the **decision variable 'interval'** that enlarges the classic concept of activity, and **cumulative function**, that is a very original abstraction enabling to model the cumulated usage of a resource by the activities as a function of time.

We show hereafter some of the main concepts of scheduling in the OPL6 syntax, and finish with advanced concepts around cumulative functions.

### 2.1. Intervals as decision variables

Suppose we have a task (in orange in the figure below) with a fixed duration 6 days to be scheduled in a given time window (blue). We place ourselves in a non-preemptive mode, e.g. once the task begins, it is processed until its end. However, we want to be able to model some **calendar constraints** which restrict some days of the schedule (here days 7 and 8) to be off.

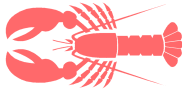


OPL6 comes with the very compact 'interval variable':

```
dvar interval task in 3..16 size 6 intensity F;
```

Note the difference between the size (duration) of the interval, which is actually 6 days and its length (its end minus its begin), which is here 8 days. This actually enforces the productivity, or intensity of the task.

In certain cases, it is convenient to use **optional intervals** simply adding the keyword 'optional' to the definition of the interval. We'll see further how optional intervals allow one to compute idle time between maintenance tasks for a given generator.



One can query through expression the natural characteristics of a given interval variable:

```
presenceOf(a) //is the interval present in the schedule
startOf(a, dval) //start of the interval
endOf(a, dval) //end of the interval
lengthOf(a, dval) //length of the interval , that is endOf(a) – startOf(a)
sizeOf(a, dval) //actual size of the interval, taken into account the intensity function if any (otherwise,
same as length)
```

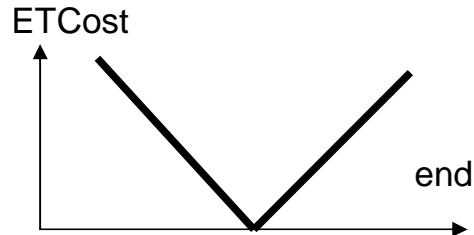
Note that all these expressions have a meaning when the interval is present in the schedule, or equivalently  $\text{presenceOf}(a)=1$ , which is automatically the case when the keyword 'optional' is not used. The default value  $\text{dval}$  (0 by default) is therefore used when the interval is not present or equivalently when  $\text{presenceOf}(a)=0$

## 2.2. Objective function

Given a set of tasks to schedule, one may be interested by different criteria to optimize. The classic **makespan** is given by:

```
minimize max(t in tasks) endOf(itvs[t]);
subject to {
...
}
```

Besides, OPL6 allows computing different expressions from intervals. Here is an illustration of the usage of a functional expression for modeling a V-shape **earliness/tardiness** cost:



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

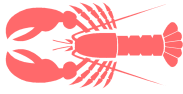
Note the simple definition of the V-shape piecewise function, and how it is used in the `endEval` operator that allows to compute  $\text{ETCost}(\text{endOf}(a[i]))$

## 2.3. Precedence constraints

A schedule is a partial order between tasks with respect to **precedence** between tasks. In OPL6, one simply uses ad-hoc global constraints.

```
forall(p in precedences)
  endBeforeStart(itvs[p.prec], itvs[p.suiv], p.delay);
```

Note the delay as the third parameter of the constraint, enabling for instance to model **set-up times** on transitions. `startBeforeEnd`, `startBeforeStart`, `endBeforeEnd` are also available to enrich the precedence constraints semantic.



## 2.4. Disjunctive scheduling

Activities that compete for a **unary resource** shall not overlap. This is enforced by another global constraint:

```
noOverlap(all(t in tasks : t.R1>0) itvs[t]);  
noOverlap(all(t in tasks : t.color==red) itvs[t]);
```

In the previous sample, we enforce that tasks that requires resource R1 cannot overlap. We enforce a similar constraint but on a set of tasks with no specific “resource” criterion, namely just the “color” of the task. Look at the documentation to see how to extend these constraints in order to capture a complete model of an **open-shop** scheduling.

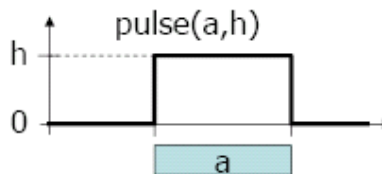
OPL6 allows manipulating a number of other constraints on intervals: alternative, span, synchronize, sequence, each of these defining a particular pattern.

## 2.5. Cumulative scheduling

Assume you have a set of tasks that consumes “manpower” resources for which 6 people are available. It is straightforward to model this limitation with the function “pulse”

```
sum(t in tasks) pulse(itvs[t],t.manpower) <= 6;
```

The pulse function is a function that has the following graph for a given interval  $a$ : it is 0 outside of the interval and the constant  $h$  between the start and the end of the activity. Summing these functions over the intervals naturally models the consumption of a given resource as a fixed level of resource during the whole execution of the interval.



In OPL6, several similar expressions are available, allowing one to model resource like reservoirs whose level increase at the beginning of some production activities and decrease from the beginning of some other consumption activities.

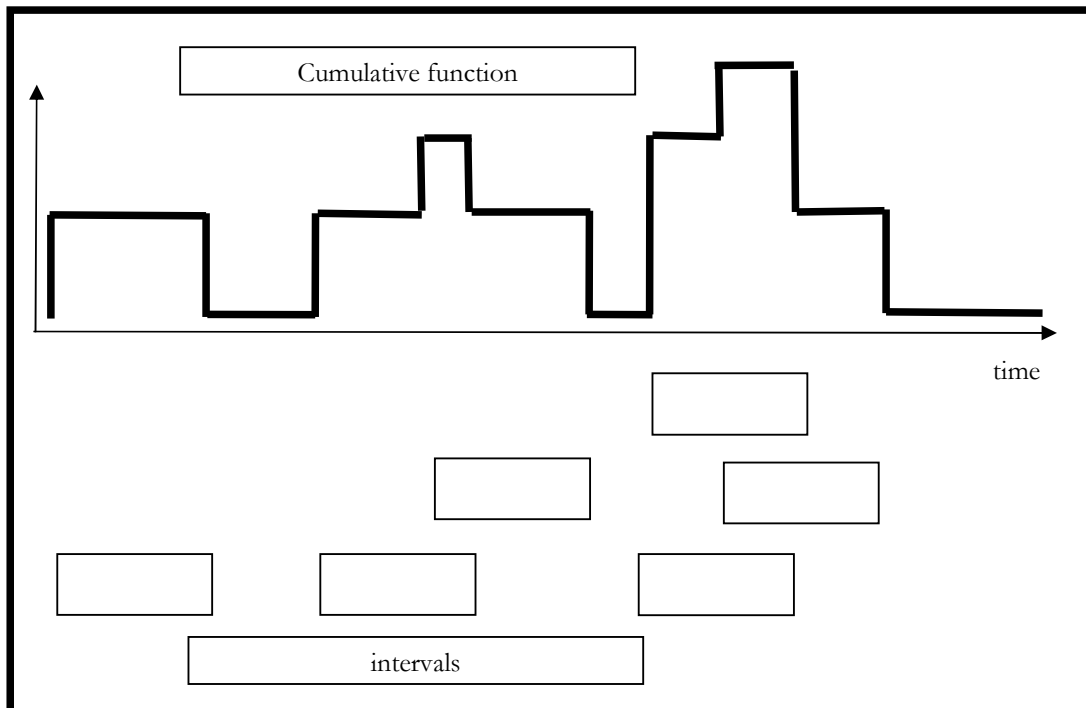
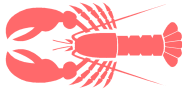
## 2.6. Cumulative functions

You can compose cumulative functions (add and subtract) such as in the above constraint with manpower resources. Cumulative functions can also be used to model high-level imbrications between intervals variables, such as in the examples below, given by ILOG R&D P. Laborie [10]

Optional interval variables can be used to capture the intervals of times during which at least one task is executed and intervals of time during which no task executes.

### 2.6.1 Sample 1

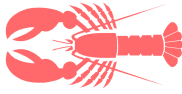
Given a set of tasks represented as interval variables  $A[i \text{ in } 1..n]$ , a chain of intervals  $W[j]$  that exactly cover the set of points where at least one task  $A$  executes can be defined as follows:



```
1. int n=...;
2. dvar interval A[i in 1..n] ...;
3. cumulFunction CA = sum(i in 1..n) pulse(A[i],1);
4. dvar interval W[i in 1..n] optional in 1..horizon; // Some work being done
   by some A[j]
5. cumulFunction CW = sum(i in 1..n) pulse(W[i],1);
6. constraints {
7. forall(i in 1..n-1) {
8.     endBeforeStart(W[i],W[i+1],1);
9.     presenceOf(W[i+1]) => presenceOf(W[i]);
10. }
11. forall(i in 1..n) {
12.     alwaysIn(CA, W[i], 1, n);
13.     alwaysIn(CW, A[i], 1, n);
14. }
15. };
```

#### Rationale:

- Line 3: the cumul function CA counts the number of tasks A executing at any time t
- Lines 4 and 7..10: W[i in 1..n] is a chain of optional intervals (to break symmetries, only the first k intervals will be present, indirectly, k is a decision variable of the problem).
- Line 4: the minimal size of 1 for intervals W (when they are present) also allows breaking some symmetries: if an interval W is present, it must represent a non null interval of time where some task execute
- Line 8: the minimal delay of 1 in the chain ensure that a block of tasks executing without interruption will be covered by exactly one interval W
- Line 5: the cumul function CW counts the number of intervals W executing at any time t (it can be either 0 or 1 as the intervals form a chain)
- Line 12: during the execution of a present interval W there must be at least one task A executed (that is:  $1 \leq CA \leq n$ )



- Line 13: during the execution of a task A there must be at least one interval W present (that is:  $1 \leq CW \leq n$ )

### 2.6.2 Sample 2

One can easily extend this model to also explicit the intervals of time during which no task executes by using an interval in between the intervals W

```
int n=...;
dvar interval A[i in 1..n] ...;
cumulFunction CA = sum(i in 1..n) pulse(A[i],1);
dvar interval W[i in 1..n] optional in 1..horizon; // Work
dvar interval M[i in 1..n-1] optional in 1..horizon; // Maintenance 6.
cumulFunction CW = sum(i in 1..n) pulse(W[i],1);
constraints {
  forall(i in 1..n-1) {
    endAtStart(W[i],M[i]);
    endAtStart(M[i],W[i+1]);
    presenceOf(M[i]) => presenceOf(W[i]);
    presenceOf(W[i+1]) => presenceOf(M[i]);
  }
  forall(i in 1..n) {
    alwaysIn(CA, W[i], 1, n);
    alwaysIn(CW, A[i], 1, n);
  }
};
```

### 2.6.3 Sample 3

Below is a small example that specifies that one cannot work more than 10 units without a maintenance period that will last 10 units. In this pedagogical example, tasks A[i] are supposed to have a processing time i and to require (n+1-i) units of a cumulative resource of capacity n.

```
int n=10;
dvar interval A[i in 1..n] size i;
cumulFunction Res = sum(i in 1..n) pulse(A[i], n+1-i); // Constraints on
activities
cumulFunction CA = sum(i in 1..n) pulse(A[i],1);
dvar interval W[i in 1..n] optional size 1..10; // Work
dvar interval M[i in 1..n-1] optional size 10; // Maintenance
cumulFunction CW = sum(i in 1..n) pulse(W[i],1);
minimize max(i in 1..n) endOf(A[i]); constraints {
  forall(i in 1..n-1) {
    endAtStart(W[i],M[i]);
    endAtStart(M[i],W[i+1]);
    presenceOf(M[i]) => presenceOf(W[i]);
    presenceOf(W[i+1]) => presenceOf(M[i]);
  }
  forall(i in 1..n) {
    alwaysIn(CA, W[i], 1, n);
    alwaysIn(CW, A[i], 1, n);
  }
  Res <= n;
};
```



---

### 3. CP and MP collaboration

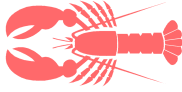
---

Remember that MIP formulations are unlikely to solve realistic instances of scheduling problems: assume we have 1000 tasks to schedule over one year (365 days). The MIP formulation creates  $365 \times 1000 = 365,000$  binary decision variables to decide whether each task is processed or not on a given day of the year.

Nevertheless, the relaxation of this problem as suggested in 1 may lead to interesting bounds for the objective function. However, a pure CP might be weak to minimize a difficult objective function.

Moving forward, one can keep the original objective function in a master problem and let CP generate good candidates according to their *reduced cost*. This decomposition is known as *column-generation* and has shown particular interest to solve large-scale combinatorial problems when the number of variables is huge. In this scheme, a CP model would generate schedule for a dynamic subset of the generators according to some *pricing* criteria (negative reduced cost) while the master problem will solve a *master linear problem* with very few constraints.

*Benders decomposition* has also been shown quite efficient to model interaction between a master and a slave model. In this decomposition scheme, the master problem assigns values to master variables that are used in the slave model, typically a given value for alpha. If the slave model is unfeasible, a “*no-good*” *Benders cut* is inferred from the unfeasibility. *Logic-based Benders decomposition* generalizes this to non-linear slave models, which is the case here. J.N Hooker[11] has shown some good results of this approach on planning and scheduling problems.



---

#### 4. BIBLIOGRAPHY / BIBLIOGRAPHIE

---

- [2] J-C. Régin: "A filtering algorithm for constraints of difference in CSPs", AAAI-94, Seattle, WA, USA, pp 362--367, 1994
- [3] Milano M. Wallace M. Integrating operations research in constraint programming 175. 4OR Volume 4. Number 3. 2006
- [4] J.N. Hooker: Integrated Methods for Optimization – Springer 2007
- [5] Constraint-Based Local Search. P. Van Hentenryck and L. Michel. The MIP Press 2005
- [6] Constraint-based Scheduling P. Baptiste, C. Le Pape, W. Nuijten . KLUWERS editions
- [7] COMET CP tutorial. P. Van Hentenryck UCL. October 2008
- [8] ILOG scheduling language guide: OPL6 documentation Help->All Topics panel : Language->Language Reference Manual->OPL, the modelling language->Scheduling
- [9] Randomized Large Neighborhood Search for Cumulative Scheduling. ICAPS05. D. Godard, P.Laborie and W.Nuijten
- [10] <http://forums.ilog.com/optimization/index.php?topic=622.0>
- [11] J. N. Hooker Planning and scheduling by logic-based Benders decomposition, Operations Research 55 (2007) 588-602