

---

# Conflict-based repair techniques for solving dynamic scheduling problems

---

Abdallah Elkhyari, Christelle Guéret, and Narendra Jussien

École des Mines de Nantes

4, rue Alfred Kastler – BP 20722

F-44307 Nantes Cedex 3

email: {aelkhyar, gueret, jussien}@emn.fr

## Abstract

Scheduling problems considered in the literature are often static (activities are known in advance and constraints are fixed). However, every schedule is subject to unexpected events. In these cases, a new solution is needed in a preferably short time and as close as possible to the current solution.

In this paper, we present an exact approach for solving dynamic Resource-Constrained Project Scheduling Problems or RCPSP. This approach combines explanation-based constraint programming and operational research techniques.

## 1 Introduction

Scheduling problems have been studied a lot over the last decade. Due to the complexity and the variety of such problems, most works consider static problems in which activities are known in advance and constraints are fixed. However, every schedule is subject to unexpected events (consider for example a new activity to schedule, or a machine breakdown). In these cases, a new solution taking these events into account is needed in a preferably short time and as close as possible to the current solution.

Using the formalism and techniques of dynamic *Constraint Satisfaction Problems* (CSP) naturally comes to the mind. Indeed, CSP and dynamic CSP (an extension of the CSP framework where the set of variables or/and constraints evolves throughout computation [10]) are increasingly used for representing and handling scheduling problems.

In this paper, we present an exact approach for solving dynamic Resource-Constrained Project Scheduling Problems or RCPSP using a dynamic CSP. In these problems, a set of activities have to be scheduled under precedence and resource constraints in order to minimize the total project duration. Solving dynamic RCPSP consists in reacting to some unexpected events. A simple (but time consuming) way to deal with dynamic scheduling is to perform global rescheduling each time an event occurs. The objective

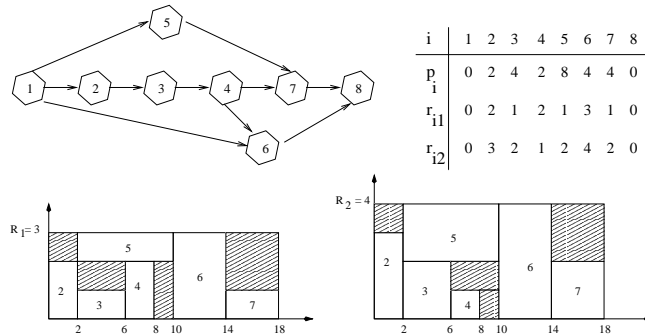


Figure 1: A feasible solution for a RCPSP with 8 activities and 2 resources.

of our approach is to get quickly a new solution by updating an initial static solution, hoping to obtain a schedule not too different from the current one. This approach combines conflict-based constraint programming and operational research techniques.

This paper is organized as follows: Section 2 introduces the RCPSP. Section 3 presents explanation-based constraint programming systems. Our approach for solving dynamic RCPSP is described in Section 4. The different events that can be taken into account are listed in Section 5. Computational results are given in Section 6.

## 2 Static and dynamic Resource Constrained Project Scheduling Problems

The Resource Constrained Project Scheduling Problem (RCPSP) consists of a set of activities  $A = \{1, 2, \dots, n\}$  and a set of renewable resources  $R = \{1, \dots, r\}$ . Each resource  $k$  is available in a constant amount  $R_k$ . Each activity  $i$  has a duration  $p_i$  and requires a constant amount  $a_{ik}$  of resource  $k$  to be processed. Preemption is not allowed. Activities are related by two sets of constraints: temporal constraints induced by a set of precedence constraints, and resource constraints that state that for each time period and for each resource, the total demand does not exceed the resource capacity. The objective considered here is the minimization of the makespan (total duration) of the project. An example is given Fig. 1. This problem, denoted by  $PS/prec/C_{max}$  [6], is *NP-hard* [5].

*Branch and bound* is the most common way to deal with the static case of this problem in order to get optimal solutions. Currently, one of the most competitive exact algorithms for the RCPSP is the one of *Brucker et al.* [7]. In this Branch and bound, four disjoint sets for pairs of activities are defined: activities in conjunction, activities in disjunction, activities which overlap (processed in parallel for at least one time period) and undecided activities. The idea of the branching scheme is to transfer pairs of activities from the undecided set to the other ones.

In the dynamic RCPSP one has to quickly react to unexpected events, as the arrival of a new activity, the modification of an activity time window, etc. The dynamic RCPSP is seldom studied. Two classical methods used to solve such problem are:

- recomputing a new schedule from scratch each time an event occurs. This is quite time consuming and may lead to a solution very different from the previous one.

- constructing a partial schedule and completing it progressively as time goes by (like in on-line scheduling problems). In this case the schedule cannot be constructed in advance which may not be acceptable for planning purposes.

Recently, *Artigues et al.* [3, 2] introduced a formulation of the RCPSP based on a flow network model. The idea is that any resource unit used during the realization of an activity must be transferred to another activity. They developed a polynomial algorithm based on this model which updates an initial static schedule in order to insert an unexpected activity.

### 3 Conflict-based constraint programming for solving dynamic problems

A CSP is defined by a set of variables  $V = \{v_1, v_2, \dots, v_n\}$ , their respective value domains  $D_1, D_2, \dots, D_n$  and a set of constraints  $C = \{c_1, c_2, \dots, c_m\}$ . A solution of the CSP is an assignment of a single value to each of the variables such that all constraints in  $C$  are satisfied. We denote by  $\mathbf{sol}(V, C)$  the set of solutions of the CSP  $(V, C)$ .

In the following, we consider variable domains as unary constraints. Moreover, the classical enumeration mechanism that is used to explore the search space is modelled as a series of constraints additions (value assignments) and retractions (backtracks and negating constraints). Those particular constraints are called *decision constraints*. This rather unusual consideration allow the easy generalization of the concepts that are presented in this paper to any kind of decision constraints (not only assignments but also ordering constraints between tasks for scheduling problems or splitting constraints in numeric CSP, etc.).

#### 3.1 Conflict sets and explanations

Let us consider a constraint system whose current state (*i.e.* the original constraints and the set of decisions made so far) is contradictory. A **conflict set** (*a.k.a.* **nogood** [21]) is a subset of the current constraint system that, left alone, leads to a contradiction (no feasible solution contains a nogood). A conflict divides into two parts: a subset of the original set of constraints ( $C' \subset C$  in equation 1) and a subset of decision constraints introduced so far in the search (here  $dc_1, \dots, dc_k$ ).

$$\mathbf{sol}(V, (C' \wedge dc_1 \wedge \dots \wedge dc_k)) = \emptyset \quad (1)$$

An operational viewpoint of conflict sets can be made explicit by rewriting equation 1 the following way:

$$C' \wedge \left( \bigwedge_{i \in [1..k] \setminus j} dc_i \right) \rightarrow \neg dc_j \quad (2)$$

Let us assume  $dc_j : v = a$  in the previous formula. Equation 2 leads to the following result ( $s(v)$  is the value of variable  $v$  in the solution  $s$ ):

$$\forall s \in \mathbf{sol} \left( V, C' \wedge \left( \bigwedge_{i \in [1..k] \setminus j} dc_i \right) \right), s(v) \neq a \quad (3)$$

The left hand side of the implication in equation 2 is called an **eliminating explanation** (explanation for short) because it justifies the removal of value  $a$  from the domain  $d(v)$  of variable  $v$ . It is noted:  $\text{expl}(v \neq a)$ .

Explanations can be combined to provide new ones. Let us suppose that  $dc_1 \vee \dots \vee dc_j$  is the set of all possible choices for a given decision (set of possible values, set of possible sequences, etc.). If a set of explanations  $C'_1 \rightarrow \neg dc_1, \dots, C'_j \rightarrow \neg dc_j$  exists, a new explanation can be derived:  $\neg(C'_1 \wedge \dots \wedge C'_j)$ . This new conflict provides more information than each of the old ones.

For example, a conflict can be computed from the empty domain of a variable  $v$  using the explanation of each of the value removals:

$$\bigwedge_{a \in d(v)} \text{expl}(v \neq a) \quad (4)$$

### 3.2 Using conflicts and explanations

Explanations can be used in several ways [1, 16, 15, 18]. Debugging purposes pop to the mind: to explain **clearly** failures, to explain differences between intended and observed behavior for a given problem (eg. why is value  $i$  not assigned to variable  $x$ ?).

Explanations can be very useful when dealing with dynamic problems. Incremental constraint addition to a problem is a well known issue but incremental constraint *retraction* is not so easy. Bessière already used a simplified explanation system to perform such a retraction [4].

When an explanation-based system is used, the incremental retraction of a given constraint  $c$  can be achieved in 4 steps: *disconnecting* the constraint from the constraint network, *undo past effects* (direct and indirect ones) of the constraint (just consider all the explanations containing  $c$ ), *checking* that those value restorations are still valid and *re-propagating* constraints in order to reach a consistent state.

It is exactly the state that would have been obtained if the retracted constraint ( $c$ ) would not have been introduced into the system.

Explanations and conflicts can also be used to efficiently guide search. Indeed, classical backtracking-based searches only proceed by backtracking to the last choice point when encountering failures. Conflicts can be used to improve standard backtracking and to exploit information gathered to improve the search: to provide intelligent backtracking [12], to replace standard backtracking with a jump-based approach *à la Dynamic Backtracking* [11, 16], or even to develop new local searches on partial instantiations [17].

For exactly solving dynamic RCSP, we will use explanations to both efficiently guiding the search and dealing with dynamic problems.

### 3.3 Computing explanations

The most interesting explanations (and conflicts) are those which are minimal regarding inclusion. Those explanations allow highly focused information about dependency relations between constraints and variables. Unfortunately, computing such an explanation can be exponentially costly [13].

A good compromise between precision and ease of computation is to use the solver-embedded knowledge to provide interesting explanations. Indeed, constraint solvers al-

ways know, although it is scarcely explicit, *why* they remove values from the domain of the variables<sup>1</sup>. By making that knowledge explicit and therefore sort of *tracing* the behavior of the solver, quite precise and interesting explanations can be computed.

To achieve this behavior, the obvious action is to alter the code of the solver itself [15] in order to provide explicit explanations. In section 4.2, we present the integration of explanations into the constraints used to model and solve RCPSPs.

## 4 Our approach for solving dynamic RCPSP

We developed a branch and bound algorithm (inspired from [7]) within a constraint programming solver: at each node, deduction rules are applied in order to determine redundant information. We introduced conflict-based repair techniques in it. The global algorithm must have the following characteristics:

- decision constraints must have a straightforward negation in order to easily do or undo a decision. We therefore introduced the notion of distance between two activities to easily define them.
- explanation capabilities must be added to all constraints (temporal and resource maintenance constraints) that are used. This may not be straightforward [14].

In the remainder of the paper, the following notations are used:

- $(i \rightarrow j)$ , called *precedence* relation, means that activity  $j$  must start after the end of activity  $i$ .
- $(i \leftrightarrow j)$ , called *disjunction* relation, means that either  $(i \rightarrow j)$  or  $(j \rightarrow i)$  holds.
- $(i \parallel j)$ , called *overlap* relation, means that activities  $i$  and  $j$  overlap at least one time unit.

### 4.1 Branching scheme

#### 4.1.1 Principle

Our branching scheme is inspired from [7]. Given a solution, each pair of activities  $(i, j)$  satisfies exactly one of the three relations:  $(i \rightarrow j)$ ,  $(j \rightarrow i)$  or  $(i \parallel j)$ . A solution for the problem is then an allocation of one of these relations to all pairs of activities such that temporal and resource constraints are satisfied. In the following, we consider only pairs of activities  $(i, j)$  such that  $i < j$ .

Each node of the tree search is defined by three disjoint sets:  $C$ ,  $P$  and  $F$ .  $C$ , the *conjunction* set, contains all pairs of activities  $(i, j)$  that satisfy one of the relations  $(i \rightarrow j)$  or  $(j \rightarrow i)$ .  $P$ , the *overlap* set, contains all pairs of activities  $(i, j)$  such that  $(i \parallel j)$  is satisfied. And  $F$ , the *flexible* set, contains all other pairs of activities (denoted by  $(i \sim j)$ ).

Branching is done by choosing one pair of activities  $(i, j) \in F$  and transforming the relation  $(i \sim j)$  either into a *precedence* relation  $(i \rightarrow j)$  or  $(j \rightarrow i)$ , or into an *overlap* relation  $(i \parallel j)$  (see Fig. 2). This branching is repeated until set  $F$  is emptied.

---

<sup>1</sup>For example, in binary CSP, a value  $a$  is removed by propagation when all its supports on a given constraint  $c$  have been removed *i.e.*  $\text{expl}(v_i \neq a) = \bigcup_{b \in \text{supp}(c, v_i, a)} \text{expl}(v_j \neq b)$ .

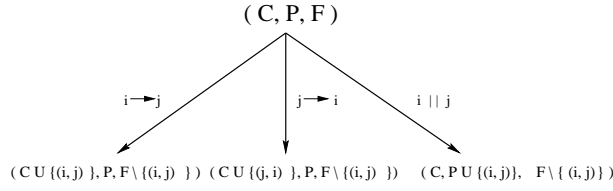


Figure 2: Branching scheme.

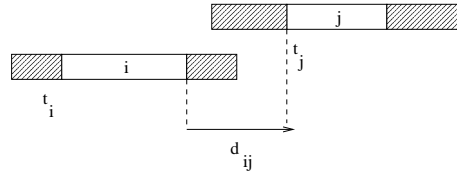


Figure 3: Distance between two activities.

Initially,  $C$  contains all the original precedence relations and  $P$  and  $F$  are empty sets.

#### 4.1.2 Our notion of distance

**Definition 1** Let  $i$  and  $j$  be two distinct activities.

The distance  $d_{ij}$  between  $i$  and  $j$  is defined as the time between the completion time of  $i$  and the starting time  $t_j$  of  $j$  (see Fig. 3).

We have  $d_{ij} = t_j - t_i - p_i$ .

Notice that a distance  $d_{ij}$  can have negative values.

The value of  $d_{ij}$  alone gives information about the relative position of  $i$  and  $j$  (see Fig. 4).

It is easy to prove that:

- $d_{ji} = -d_{ij} - p_i - p_j$ .
- $d_{ij} \times d_{ji} \leq 0$  iff  $i$  and  $j$  are in disjunction.
- $d_{ij} \times d_{ji} > 0$  iff  $i$  and  $j$  overlap.
- $d_{ij}$  and  $d_{ji}$  cannot be both strictly positive.

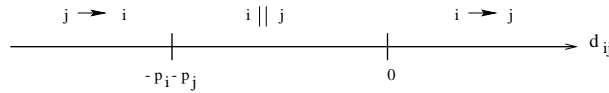


Figure 4: Positions of two activities according to the value of their distance.

Decision	Constraint	Opposite decision	Associated constraint
$(i \rightarrow j)$	$d_{ij} \geq 0$	$(i \nrightarrow j)$	$d_{ij} < 0$
$(j \rightarrow i)$	$d_{ij} \leq -p_i - p_j$	$(j \nrightarrow i)$	$d_{ij} > -p_i - p_j$
$(i \parallel j)$	$d_{ij} \times d_{ji} > 0$	$(i \leftrightarrow j)$	$d_{ij} \times d_{ji} \leq 0$

Table 1: Correspondance between branching decisions and constraints on distances

### 4.1.3 Distance and branching

The distance between two activities is used to set up the decision constraints used in the branching scheme and their negation. Table 1 states the correspondance.

Notice that the set of possible relations between two activities  $i$  and  $j$  is restricted by the current domain values of the distance between them.

### 4.1.4 Branch selection.

Several heuristics can be used to select the flexible pair of activities on which branching is done in each node of the search tree:

- *Min\_Dom*: choose the variable  $d_{ij}$  which has less values in its domain.
- *Min\_Deg\_Dom*: select the variable  $d_{ij}$  which has the smallest ratio between the number of values in its domain and its degree in the constraint network.
- *Max\_Sum\_Duration*: choose the variable  $d_{ij}$  for which the sum  $p_i + p_j$  is maximum.
- *Max\_Sum\_Energy*: select the variable  $d_{ij}$  for which the sum of energies  $w_{ik} + w_{jk}$  is maximum, for any resource  $k$  (the energy consumption of a resource  $k$  for an activity  $i$  is  $w_{ik} = p_i \times a_{ik}$ ).

It appears that the *Max\_Sum\_Energy* heuristic detects more early resource conflicts. It is the one that is used in our system.

## 4.2 Adding explanations into temporal and resource-management constraints

Whereas adding explanations to temporal constraints is quite easy (they are indeed modelled using simple mathematical relations), resource-management constraints are more global constraints and do not provide a straightforward explanation mechanism.

The classical techniques for maintaining the resource limitation over the time horizon for the RCPSP are: *core-times* [20], *task-interval* [9, 8] and *resource-histogram* [9, 8]. We upgraded these techniques to provide explanations.

Indeed, we developed two sets of constraints in order to fulfill resource management requirements: *timetable* constraints (a combination of the *core-times* and *resource-histogram* techniques) and *task-interval* constraints.

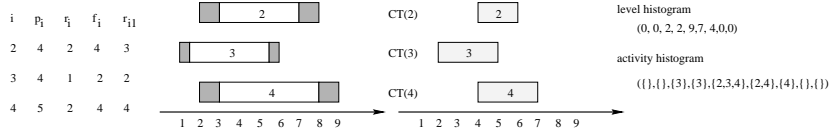


Figure 5: Example of timetable.

#### 4.2.1 Timetable constraints

The principle of the *resource-histogram* technique [9, 8] is to associate to each resource  $k$  an array  $level(k)$  in order to keep a timetable of the resource requirements. This histogram is used for detecting a contradiction and reducing the time windows of activities.

The *core-times* technique [20] is used for computing lower bounds using a destructive method. A *core-times*  $CT(i)$  is associated to each activity  $i$ . It is defined as the interval of time during which a portion of an activity is executed whether it starts at its earliest or latest start time. The procedure consists in scheduling all the  $CT(i)$ . If a resource conflict is detected then the value of the lower bound is upgraded.

We combined these two techniques in order to detect some resource conflicts. We propose to construct a timetable for each resource as follows: for each activity  $i$ , we compute its *core-times*  $CT(i) = [f_i, r_i + p_i]$  ( $f_i$  is the latest start time of  $i$ ,  $r_i$  its earliest start time) and we reserve an amount  $a_{ik}$  of resource  $k$  for each time periods  $[f_i, f_i + 1), \dots, [r_i + p_i - 1, r_i + p_i)$ . We associate to each *timetable constraint* two histograms (see Fig. 5):

- a *level histogram*: which contains the amount of resource required at each time period  $[t - 1, t)$ .
- an *activity histogram* which contains the sets  $S_t$  of activities which require an amount of resource for each time period  $[t - 1, t)$ . It is essentially used for providing explanations.

We use these histograms for two purposes:

- detecting resource conflicts when the required level of a resource  $k$  at one time period  $t$  exceeds the resource capacity. The conflict set associated to this contradiction is the union of the explanations of all the value removals from the domain of the activities in  $S_t$  (given by the *activity histogram*). A contradiction explanation can be computed as:  $\neg \left( \bigwedge_{v \in S_t} \left( \bigwedge_{a \in d(v)} \text{expl}(v \neq a) \right) \right)$ .
- updating the time window of an activity: in Fig. 6, we see that the resource considered can not support activity 2 at the time period  $[2, 3)$  and hence the earliest starting time of the activity 2 can be updated to value 3.

The explanation of this modification is the union of all the explanations of the value removals from the domain of the activities using the resource at time period  $[2, 3)$  and the constraint  $c$  itself. An explanation for this modification can be computed as:  $\left( \bigwedge_{v \in S_3} \left( \bigwedge_{a \in d(v)} \text{expl}(v \neq a) \right) \right) \wedge c$ .

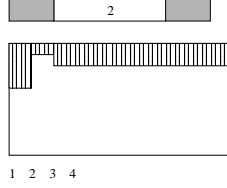


Figure 6: Timetable rule.

#### 4.2.2 Task-interval constraints.

The notion *task-interval* [9, 8] provides several rules: *integrity rule*, *precedence rule*, *throw rule*, *reject rule* and *exclusion rule*. These rules can detect conflicts, deduce precedence relations and upgrade the activity domains (see Section 4.2.2).

The *task-interval*  $T = [i, j]$  associated to a pair of activities  $(i, j)$  which require the same resource  $k$ , is the set of activities  $l$  which share the same resource and such that  $r_i \leq r_l$  and  $d_l \leq d_j$  ( $r_i$  is the earliest start time of the activity  $i$  and  $d_i$  its due date). Each *task-interval*  $TI = [i, j]$  is defined by the set of activities inside it ( $inside(TI)$ ), its energy ( $energy(TI) = \sum_{l \in inside(TI)} w_{lk}$ ) and its starting task  $i$  and ending task  $j$ . The set of activities outside  $TI$  will be denoted by ( $outside(TI)$ ).

Three situations are possible for an activity  $l$  in the set  $outside(TI)$  relatively to the *task-interval*  $TI$  where the activity must be partially processed:

- the activity completely overlaps interval  $TI$ ;
- the activity partially overlaps interval  $TI$ . In this case, the minimum time period during which the activity overlaps  $TI$  is equal to the minimum between  $(r_l + p_l - r_i)$  (the activity starts at its earliest start date) and  $(d_j - f_l)$  (the activity starts at its latest start date).

The minimum amount of energy, required by an activity  $l$  in the set  $outside(TI)$  in the interval  $TI$  can then be computed by the formula:  $w_l(TI) = \max\{0, \min\{(d_j - r_i) \times a_{lk}, (r_l + p_l - r_i) \times a_{lk}, (d_j - f_l) \times a_{lk}\}\}$ .

We use the *integrity rule* and *throw rule* [9, 8] for detecting a resource conflicts and updating the time windows of activities.

**Integrity rule** If  $energy(TI)$  is greater than the total energy  $(d_j - r_i) \times R_k$  available in the interval then we have a contradiction. The conflict set is the union of all the explanations of value removals in the activities in  $inside(TI)$ :  $\neg \left( \bigwedge_{v \in inside(TI)} \text{expl}(v \neq a) \right)$ .

If the *integrity rule* does not detect any conflict then we take into account the set  $inter(TI)$  of activities intersecting  $TI$ . The conflict set is the union of all the explanations of value removals from the domain of the activities in  $inside(TI)$  and in  $inter(TI)$ :  $\neg \left( \bigwedge_{v \in inside(TI) \cup inter(TI)} \text{expl}(v \neq a) \right)$ .

**Throw rule** This rule consists in pushing to the right (or left) an activity intersecting  $TI$ , to determine the quantity of energy intersecting  $TI$  and compare it to the quantity of remaining energy. The explanation of this update is the union of all the explanations of the value removals from the domain of the activities in  $inside(TI)$  and the constraint  $c$  itself:  $\left( \bigwedge_{v \in inside(TI)} \text{expl}(v \neq a) \wedge c \right)$ .

When there is no modification then we take into account the activities intersecting  $TI$ . The explanation of the modification in this case is the union of all the explanations of value removals from the domain of the activities in  $inside(TI) \cup inter(TI)$  and the constraint  $c$  itself:  $(\bigwedge_{v \in inside(TI) \cup inter(TI)} \text{expl}(v \neq a) \wedge c)$ .

### 4.3 The overall search algorithm

We use a conflict-based search algorithm for solving the RCPSP. The main idea is to consider search as continuously making decisions (and propagating them) until a contradiction occurs. In that case, instead of using backtracking in order to browse the search space, we use repair techniques. Those repair techniques use the conflict set explaining the contradiction in order to determine which decision is to be undone (not necessarily the latest made). This search algorithm is a generalization of *dynamic backtracking* [11] and *mac-dbt* [16] to scheduling.

As we are solving optimization problems, we use the classical approach used in constraint programming: as soon as a solution is found, a dynamic constraint stating that the objective value should be improved is added and search continues.

### 4.4 Handling over-constrained problem

When a problem is over-constrained (no solution exists), one should try to relax the problem by removing some constraints. The process of finding a relaxation is to use explanations for identifying the constraints to be removed [19].

In our system, after adding constraints, the problem can become over-constrained. Then, we compute the constraints to be removed. The user can choose between removing some of these constraints or increasing the total duration of the project.

## 5 Handling dynamic events

In this section, we introduce the set of events that can currently be dynamically taken into account in our solver and how we model those events.

### 5.1 Temporal events

Temporal events introduce, modify or remove temporal constraints. The set of temporal constraints taken into account and their translation using distance variables are the following:

- **Precedence relations:**

- $i \rightarrow j$ :  $d_{ij} \geq 0$ .
- $i \rightarrow j$  and there is exactly  $l$  time units between the end of  $i$  and the beginning of  $j$ :  $d_{ij} = l$
- $i \rightarrow j$  and there is at least  $l_{min}$  and at most  $l_{max}$  time units between the end of  $i$  and the beginning of  $j$ :  $l_{min} \leq d_{ij} \leq l_{max}$

- **Overlapping relations:**

- $i \parallel j : d_{ij} \times d_{ji} > 0$
- $i \parallel j$  and there is exactly  $l$  time units between the end of  $i$  and the beginning of  $j$ :  $d_{ij} = -l$
- $i \parallel j$  and there is at least  $l_{min}$  and at most  $l_{max}$  time units between the end of  $i$  and the beginning of  $j$ :  $-l_{max} \leq d_{ij} \leq -l_{min}$

- **Disjunctive relations:**

- $i \leftrightarrow j$ :  $d_{ij} \times d_{ji} \leq 0$
- $i \leftrightarrow j$  and there is exactly  $l$  time units between the end of  $i$  and the beginning of  $j$ :  $d_{ij} = l$  or  $d_{ji} = l$
- $i \leftrightarrow j$  and there is at least  $l_{min}$  and at most  $l_{max}$  time units between the end of  $i$  and the beginning of  $j$ :  $l_{min} \leq d_{ij} \leq l_{max}$  or  $l_{min} \leq d_{ji} \leq l_{max}$

- **Time windows:**

- Activity  $i$  must start between  $l_{min}$  and  $l_{max}$ :  $l_{min} \leq t_i \leq l_{max}$

There are three possible events considering those relations: **adding a relation**, **removing a relation**, and **modifying a relation** which is handled as removing the old relation and adding the new one.

## 5.2 Activity related events

The set of activity related events taken into account is:

- **Adding an activity:** we can add an activity  $i$  with a given duration and some resources requirements, and two sets  $P$  and  $S$  of activities which precede it and follow it respectively.

We create a new variable  $i$ , connect it to the constraint network, add the temporal constraints (from sets  $P$  and  $S$ ), add the disjunctive constraints which results from the capacities limitation and add the variable  $i$  to the associated resource constraints.

- **Removing an activity:** An activity  $i$  can be removed from the problem.

We remove all the constraints that use variable  $i$ ,  $d_{ij}$  or  $d_{ji}$ , and remove variable  $i$  from the associated resource constraints.

## 5.3 Resource related events.

The set of resource related events taken into account is:

- **Adding a resource:** To add a new resource in the problem, we the resource constraints: (*timetable constraint* and *task-interval constraints*).

- **Removing a resource:** A resource can be removed from the problem.

In order to remove a resource from the problem, we remove the disjunctive constraints which results from the capacity limitation of this resource, remove the *timetable constraint* and all the *task-interval constraints* which are related to this resource.

## 6 First computational experiments

We present here our first experimental results on dynamic RCPSP.

Our experiment consists in comparing our dynamic scheduler with a static scheduler. Our experimental protocol consists in considering a scheduling problem  $P$  which is first optimally solved. Then, an event  $e$  is added to this problem. We then compare a re-execution from scratch (as a static scheduler would do) and the dynamic addition of the related constraints in terms of cpu time.

We use here some test problems introduced by *Patterson*<sup>2</sup>. We have added in some of these problem, one of the following events:

- addition of precedence relation between two activities  $i$  and  $j$  (*cf.* Table 2-a),
- addition of precedence relations between two activities  $i$  and  $j$  with the additional constraints stating that there is exactly  $l$  time between the end of  $i$  and the beginning of  $j$  (*cf.* Table 2-b),
- addition of precedence relations between two activities  $i$  and  $j$  with the additional constraints that there is at least  $l_{min}$  and at most  $l_{max}$  time units between the end of  $i$  and the beginning of  $j$  (*cf.* Table 2-c),
- addition of overlapping relation between two activities  $i$  and  $j$  (*cf.* Table 2-d).

In all the tests, activities  $i$  and  $j$  are randomly selected.

In all the tables, we designate by:

- # Act: the number of activities of the problem;
- # Res: the number of resources of the problem;
- $t_p$ : the cpu time needed for solving optimally problem  $P$ ;
- $t_e$ : the cpu time spent to solve this problem after dynamically adding the event  $e$
- $t_{pe}$ : the cpu time needed for obtaining an optimal solution of the problem  $P \cup \{e\}$  from scratch.
- $\frac{t_{pe}-t_e}{t_{pe}}$ : the overall interest using a dynamic scheduler compared to a static scheduler expressed as the percentage of improvement.

These results clearly show that using a dynamic scheduling solver is of great use compared to solving a series of static problems. In our first experiments, the improvement is never less than 23% and can even get to 98.8 %!

---

<sup>2</sup>[www.bwl.uni-kiel.de/Prod/psplib/dataob.html](http://www.bwl.uni-kiel.de/Prod/psplib/dataob.html)

# Act	# Res	$t_p$	$t_e$	$t_{pe}$	$\frac{t_{pe}-t_e}{t_{pe}}\%$
7	3	0.23	0.02	0.19	89.5
8	2	0.34	0.06	0.25	76.0
9	1	2.11	0.32	2.30	86.1
13	3	25.82	4.16	12.58	67.0
13	3	15.64	1.21	12.58	90.4
14	3	7.26	1.16	6.96	83.3
14	3	6.75	0.23	6.96	96.7
22	3	8.01	1.46	3.63	59.8
22	3	4.50	0.50	3.63	86.2
22	3	20.37	8.70	13.80	<b>37.0</b>
22	3	11.34	1.76	13.80	87.3
22	3	135.11	19.60	78.61	75.1
22	3	36.70	7.81	78.61	90.1
22	3	87.80	0.96	78.61	<b>98.8</b>

(a)

# Act	# Res	$t_p$	$t_e$	$t_{pe}$	$\frac{t_{pe}-t_e}{t_{pe}}\%$
8	2	0.39	0.20	0.42	<b>52.4</b>
8	2	0.18	0.02	0.26	92.3
9	1	2.19	0.22	2.58	91.5
14	3	7.31	0.87	8.84	90.2
14	3	6.58	0.82	9.55	91.4
22	3	14.90	3.45	8.33	58.6
22	3	16.90	3.44	8.19	58.0
22	3	62.45	5.14	24.43	79.0
22	3	64.42	1.87	19.56	90.5
22	3	62.50	1.24	23.40	<b>94.7</b>
22	3	13.82	3.50	8.22	57.4

(b)

# Act	# Res	$t_p$	$t_e$	$t_{pe}$	$\frac{t_{pe}-t_e}{t_{pe}}\%$
9	1	2.90	0.12	2.84	95.8
13	3	17.49	0.45	14.34	<b>96.9</b>
14	3	7.12	0.81	8.50	90.5
14	3	9.69	0.61	9.81	93.8
22	3	68.38	9.63	19.30	50.1
22	3	75.88	2.52	21.12	88.1
22	3	66.00	17.74	27.34	<b>35.1</b>
22	3	14.46	4.82	7.59	36.5
22	3	65.45	16.25	30.65	47.0
22	3	15.25	3.10	7.81	60.3

(c)

# Act	# Res	$t_p$	$t_e$	$t_{pe}$	$\frac{t_{pe}-t_e}{t_{pe}}\%$
9	1	2.37	0.38	2.35	83.9
9	1	2.27	0.12	2.64	95.5
14	3	7.43	0.38	10.62	<b>96.4</b>
22	3	77.73	8.44	14.60	42.2
22	3	70.45	6.63	18.59	64.3
22	3	15.16	16.39	24.84	34.0
22	3	64.87	9.85	12.86	<b>23.4</b>

(d)

Table 2: Adding temporal constraints

## 7 Conclusion

In this paper, we presented a system for solving optimally dynamic scheduling problems using conflict-based repair techniques. Instead of recomputing a new solution from scratch each time an event occurs, this system gets quickly a new solution by updating the current schedule. The first results obtained show that this approach is very interesting: indeed, our system gets new solutions up to 98,8% faster than a global rescheduling. Furthermore, contrarily to the schedule found by a recomputing from scratch, the new schedule obtained by our system is generally close to the initial one.

Another interest of our system is that it can help the user in case of over-constrained problem by providing the set of constraints responsible of the conflict. The user can then choose the constraints to remove for obtaining a solution.

For the moment, our system can take into account several dynamic events like the addition of a new activity, the modification of an activity time window or the addition of a new precedence between two activities. We are currently working on the introduction of more complicated events like the variations of activity durations, variations of resource requirements or variations of resource capacity limitations.

## References

- [1] Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, and Narendra Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In *16th IEEE conference on Automated Software Engineering (ASE'01)*, San Diego, USA, November 2001.
- [2] C. Artigues and F. Roubellat. A polynomial activity insertion algorithm in a multi-resource schedule with cumulative constraints and multiple modes. *European Journal of Operational Research*, 127(2):297–316, 2000.
- [3] Christian Artigues, Philippe Michelon, and S. Reusser. Insertion techniques for static and dynamic resource constraint project scheduling. Research Report 152, LIA, 2000.
- [4] Christian Bessière. Arc consistency in dynamic constraint satisfaction problems. In *Proceedings AAAI'91*, 1991.
- [5] J. Blazewicz, J.K. Lenstra, and A.H.G. Rinnoy Kan. Scheduling projects subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5:11–24, 1983.
- [6] P. Brucker, A. Drexler, R. Möring, Neumann, and E. Pesch. Resource-constrained project scheduling: notation, classification, models and methods. *European Journal of Operational Research*, 112:3–41, 1999.
- [7] P. Brucker, S. Knust, A. Schoo, and O. Thiele. A branch and bound algorithm for the Resource-Constrained Project Scheduling Problem. *European Journal of Operational Research*, 107:272–288, 1998.
- [8] Y. Caseau and F. Laburthe. Cumulative scheduling with task-intervals. In *Joint International Conference and Symposium on Logic Programming (JICSLP)*, 1996.

- [9] Yves Caseau and François Laburthe. Improving clp scheduling with task intervals. In P. Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming, ICLP'94*, pages 369–383. MIT Press, 1994.
- [10] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *AAAI'88*, pages 37–42, St Paul, MN, 1988.
- [11] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
- [12] Christelle Guéret, Narendra Jussien, and Christian Prins. Using intelligent backtracking to improve branch and bound methods: an application to open-shop problems. *European Journal of Operational Research*, 127(2):344–354, 2000.
- [13] Ulrich Junker. QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, Seattle, WA, USA, August 2001.
- [14] Narendra Jussien. e-constraints: explanation-based constraint programming. In *CP01 Workshop on User-Interaction in Constraint Satisfaction*, Paphos, Cyprus, 1 December 2001.
- [15] Narendra Jussien and Vincent Barichard. The palm system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques for Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, Singapore, September 2000.
- [16] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming (CP 2000)*, number 1894 in Lecture Notes in Computer Science, pages 249–261, Singapore, September 2000. Springer-Verlag.
- [17] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. In *Seventh National Conference on Artificial Intelligence – AAAI'2000*, pages 169–174, Austin, TX, USA, August 2000.
- [18] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 2002. to appear.
- [19] Narendra Jussien and Samir Ouis. User-friendly explanations for constraint programming. In *ICLP'01 11th Workshop on Logic Programming Environments*, Paphos, Cyprus, 1 December 2001.
- [20] R. Klein and A. Scholl. Computing lower bounds by destructive improvement: an application to Resource-Constrained Project Scheduling Problem. *European Journal of Operational Research*, 112:322–345, 1999.
- [21] Thomas Schiex and Gérard Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.