# Solving dynamic resource constraint project scheduling problems using new constraint programming tools

Abdallah Elkhyari, Christelle Guéret, Narendra Jussien

# Solving dynamic timetabling problems as dynamic resource constrained project scheduling problems using new constraint programming tools

Abdallah Elkhyari[1], Christelle Guéret[1,2], and Narendra Jussien[1]

[1] École des Mines de Nantes, BP 20722
F-44307 Nantex Cedex 3, France
{aelkhyar,gueret,jussien}@emn.fr
[2] IRCCyN
Institut de Recherche en Communications et Cybernétique de Nantes
France

**Abstract.** Timetabling problems have been studied a lot over the last decade. Due to the complexity and the variety of such problems, most work concern static problems in which activities to schedule and resources are known in advance, and constraints are fixed. However, every timetabling problem is subject to unexpected events (consider for example, for university timetabling problems, a missing teacher, or a slide projector breakdown). In such a situation, one has to quickly build a new solution which takes these events into account and which is preferably not too different from the current one. We introduce in this paper constraint-programming based tools for solving dynamic timetabling problems modelled as RCPSP (Resource-Constrained Project Scheduling Problems). This approach uses explanation-based constraint programming and operational research techniques.

## 1 Introduction

Timetabling problems have been studied a lot over the last decade. Due to the complexity and the variety of such problems, most work concern static problems in which activities to schedule and resources are known in advance, and constraints are fixed. However, every timetabling problem is subject to unexpected events (consider for example, for university timetabling problems, a missing teacher, or a slide projector breakdown). In such a situation, one has to quickly build a new solution which takes these events into account and which is preferably not too different from the current one.

In this paper, we present an exact approach for solving dynamic timetabling problems which uses explanation-based constraint programming and operational research techniques. In this approach, timetabling problems are modelled as RCPSP (Resource-Constrained Project Scheduling Problem).

This paper is organized as follows: Section 2 introduces *timetabling problems* and RCPSP, and explains how *timetabling problems* can be modelled as RCPSP. Section 3 presents the basics of explanation-based constraint programming. Our approach is presented in Section 4. Dynamic events that can be taken into account in our system are listed in Section 5 and computational results are reported in Section 6.

## 2 Timetabling problems and RCPSP

### 2.1 Timetabling problems

Timetabling problems can be defined as the scheduling of a certain number of activities (lectures, tasks, etc.) which involve specific groups of people (students, teacher, employees, etc.) over a finite period of time, requiring certain resources (rooms, materials, etc.) in conformity with the availability of resources and fulfilling certain other requirements. A huge variety of timetabling problems exist: school timetabling, examination timetabling, employee timetabling, university timetabling, etc.

Due to the complexity and the variety of such problems, most work concern static problems in which both activities to schedule and resources are known in advance, and constraints are fixed. The different techniques used are: graph coloring [8], integer programming [33], genetic algorithms [29], tabu search [18], etc. Several different authors have presented *constraint programming* techniques for solving timetabling problems, [4, 15, 16, 24, 27]. To our knowledge, no work concern the resolution of dynamic timetabling problems.

### 2.2 RCPSP

The RCPSP can be defined as follows: let $A = \{1, 2, \ldots, n\}$ be a set of activities, and $R = \{1, \ldots, r\}$ a set of renewable resources. Each resource $k$ is available in a constant amount $R_k$. A resource $k$ is called disjunctive if $R_k = 1$. Otherwise it is called cumulative. Each activity $i$ has a duration $p_i$ and requires a constant amount $a_{ik}$ of the resource $k$ during its execution. Preemption is not allowed. Activities are related by precedence constraints, and resource constraints require that for each period of time and for each resource, the total demand of resource does not exceed the resource capacity. The objectives considered here are to find a feasible solution or a solution for which the end of the schedule is minimized (see for example Fig. 1). The problem is *NP-hard* [3].

Let $S_i$ be the starting time of activity $i$. Several extensions of the RCPSP can be considered :

- generalized precedence constraints: such a constraint imposes that an activity $j$ must be executed after another activity $i$ and that there are exactly $d$ units of time between the end of $i$ and the starting of $j$, or that there are at least $d_{min}$ and at most $d_{max}$ units of time between them. These constraints will be noted $i \rightarrow^{d} j$ and $i \rightarrow_{d_{min}}^{d_{max}} j$ respectively.

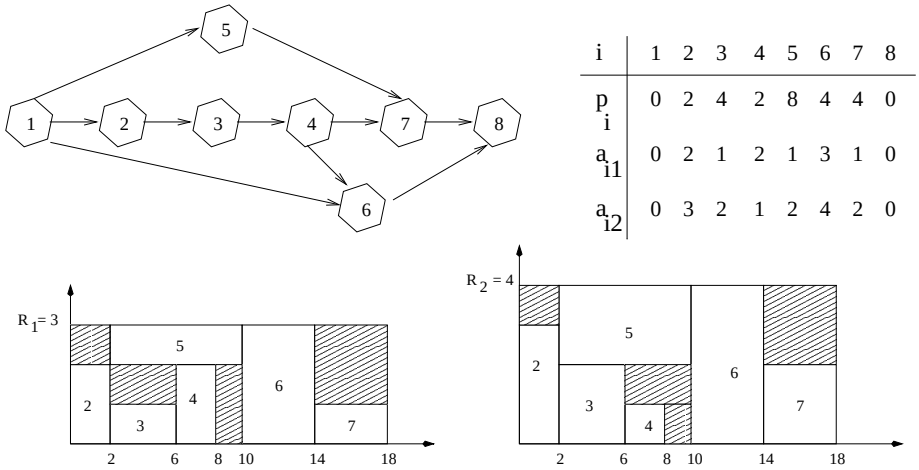


**Fig. 1.** *One project with 8 activities and 2 resources, and a feasible solution*

- generalized disjunctive constraints: such a constraint imposes that $i$ and $j$ are in disjunction and that there are exactly $d$, or at least $d_{min}$ and at most $d_{max}$ units of time between them. These constraints will be noted $i \leftrightarrow^{d} j$ and $i \leftrightarrow_{d_{min}}^{d_{max}} j$ respectively.
- generalized overlapping constraints: two activities may have to overlap (be executed during at least a common unit of time) during exactly $d$, or at least $d_{min}$ and at most $d_{max}$ units of time. These constraints will be noted $i \|^{d} j$ and $i \|_{d_{min}}^{d_{max}} j$ respectively.
- generalized resource constraints: the capacity of resources may change during certain periods of time.

The static RCPSP has been extensively studied ([5, 26]). Currently, the most competitive exact algorithms for the RCPSP are the ones of Brucker *et al.* [6], Demeulemeester and Herroelen [12], Mingozzi *et al.* [28] and Sprecher [32]. One of the main difficulty in the RCPSP is to maintain the resource limitation during the planning period. The classical deduction rules used for that purpose are *core-times* [25] and *task-interval* [9, 10].

The dynamic RCPSP is seldom studied. Two classical methods are used to solve it:

- recomputing a new schedule 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 online scheduling problems) [31]. This is unacceptable in the context of timetabling problems since the complete timetable must be known in advance.

Recently, Artigues *et al.* [1] introduced a formulation of the RCPSP based on a flow network model. The authors developed a polynomial algorithm based on this model to insert an unexpected activity.

### 2.3 Timetabling problems as RCPSP

Timetabling problems are special cases of RCPSP. In [7], *Brucker* shows that the *High school timetabling* can easily be formulated as a RCPSP. And *Dignum et al.* [13] translated the particular timetabling problem of an educational establishment of Maastricht into a Time and Resource Constrained Scheduling Problem (TRCSP), *i.e.* a RCPSP in which the objective is to determine a schedule which is completed on time and such that the total additional costs are minimized.

Let us consider the following example of a French university timetable [16]: $m$ classes have to follow a set of lectures given by $n$ teachers over an horizon of $T$ time periods. All lectures are of the same length. Each lecture may require specific material (slide-projector, specific room, etc.)

The constraints are the following:

- C1: a class cannot follow more than one lecture at a time
- C2: a teacher cannot give more than one lecture at a time
- C3: some teachers or classes are not available in some periods
- C4: material availability must be respected
- C5: some lectures have to be scheduled at the same time (shared gymnastic rooms, class divided in several groups for foreign language lectures,...)
- C6: two lectures concerning the same subject should not be scheduled on too close periods
- C7: some lectures are linked by precedence constraints (prerequisite, ...)

Clearly this example can be modelled as a RCPSP in which lectures are the activities to schedule and resources are teachers, classes and materials. Some of these resources are disjunctive (teachers for example), others are cumulative (if several slide-projectors are available for example). These lectures are subject to (generalized) precedence constraints (C7), (generalized) overlapping constraints (C5) and (generalized) disjunctive constraints (C1, C2 and C6). Resource constraints must be respected (C4), and some resource capacities are time dependent (C3).

## 3 Explanation-based constraint programming

Constraint programming techniques have been widely used to solve scheduling problems. A *constraint satisfaction problem* (CSP) consists in a set $V$ of variables defined by a corresponding set of possible values (the domains $D$) and a set $C$ of constraints. A solution for the network is an assignment of a value to each variable such that all the constraints are satisfied. An extension of classical constraint programming has been recently introduced. It is called explanation-based constraint programming (*e-constraints*) and it has already proved its interest in

many applications [19] including dynamic constraint solving. We recall in this section what it is and how it can be used.

In the following, we consider a constraint satisfaction problem $(V, D, C)$. Decision making during the enumeration phase (variable assignments) amounts to add (*eg.*, upon decision making) or remove (*eg.*, upon backtracking) constraints from the current constraint system. Enumeration is therefore considered as a dynamic process.

### 3.1 Explanations

A **contradiction explanation** (*a.k.a.* **nogood** [30]) is a subset of the current constraints system of the problem that, left alone, leads to a contradiction. Thus, no feasible solution contains a nogood. A contradiction explanation is composed of two parts: a subset of the original set of constraints ($C' \subset C$) and a subset of decision constraints introduced so far in the search:

$$C \vdash \neg (C' \wedge v_1 = a_1 \wedge \cdots \wedge v_k = a_k) \tag{1}$$

In a contradiction explanation composed of at least one decision constraint, a variable $v_j$ is selected and the previous formula is rewritten as[3]:

$$C \vdash C' \wedge \bigwedge_{i \in [1..k] \setminus j} (v_i = a_i) \rightarrow v_j \neq a_j \tag{2}$$

The left hand side of the implication constitutes an **eliminating explanation** for the removal of value $a_j$ from the domain of variable $v_j$ and is noted $\texttt{expl}(v_j \neq a_j)$.

Classical CSP solvers use domain-reduction techniques (removal of values). Recording eliminating explanations is sufficient to compute contradiction explanations. Indeed, a contradiction is identified when the domain of a variable $v_j$ is emptied. A contradiction explanation can easily be computed with the eliminating explanations associated with each removed value:

$$C \vdash \neg \left( \bigwedge_{a \in d(v_j)} \texttt{expl}(v_j \neq a) \right) \tag{3}$$

There exist generally several eliminating explanations for the removal of a given value. Recording all of them leads to an exponential space complexity. Another technique relies on *forgetting* (erasing) eliminating explanations that are no longer relevant[4] to the current variable assignment. By doing so, the space complexity remains polynomial. We keep only **one** explanation at a time for a value removal.

[3] A contradiction explanation that does not contain such a constraint denotes an over-constrained problem.

[4] A nogood is said to be relevant if all the decision constraints in it are still valid in the current search state.

### 3.2 Computing explanations

During propagation, constraints are awoken (like agents or daemons) each time a variable domain is reduced (this is an event) possibly generating new events (value removals). A constraint is fully characterized by its behavior regarding the basic events such as value removal from the domain of the variables and domain bound updates. Explanations for events are computed when the events are generated.

**Explanations for basic constraints** It is easy to provide explanations for basic constraints. The following example shows how to compute them.

*Example 1.* Let us consider a two-variables toy problem: $x$ and $y$ with the same set of possible values *[1, 2, 3]*. Let us state the constraint $x > y$. The resulting sets of possible values are *[2, 3]* for $x$ and *[1, 2]* for $y$. An explanation for this situation is the constraint $x > y$. Now, let us suppose that we choose to add the constraint $x = 2$. The only resulting possible value for $x$ is *2*. The explanation of the modification is the constraint $x = 2$. The other consequence is that the remaining value for $y$ is *1*. The explanation for this situation is twofold: a direct consequence of the constraint $x > y$ and also an indirect consequence of constraint $x = 2$.

**Explanations for global constraints** Computing a precise explanation for global constraints may not be easy because it is necessary to study the algorithms used for propagation. However, there always exists a generic explanation: the current state of the domains of each variable of the constraints. In section 4, we will describe how to provide more precise explanations for timetabling-related constraints.

### 3.3 Using explanations

Explanations are useful in many situations [19]. The following sections detail some of them: providing user information, improving search and handling dynamic problems.

**Providing user information**

- explanations can be scanned to determine past effects of selected constraints: these are the events for which the associated explanation contains one of the selected constraint;
- considering the union of the explanations of the currently removed values in the current solution is a justification of that situation;
- when encountering a contradiction, a contradiction explanation will provide a subset of the constraints system that justifies the contradiction and that can be provided to the user;
- etc.

**Improving search strategies** Explanations 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. Explanations can be used to improve standard backtracking and to exploit information gathered to improve the search: to provide intelligent backtracking [17], to replace standard backtracking with a jump-based approach *à la Dynamic Backtracking* [21], or even to develop new local searches on partial instantiations [22].

The common idea of these techniques is, upon encountering a contradiction, to determine an explanation from which a constraint will be selected either to determine a relevant backtracking point (for intelligent backtracking) or to only be dynamically removed and replaced with its negation (as in *dynamic backtracking*).

**Dynamically adding/removing constraints** We can use the explanations for adding or removing constraints because they help pointing out past effects of constraints that can be effortlessly undone without a complete re-computation from scratch. Notice that adding constraint to a problem is a well known issue but removing it is not so easy. For dynamically removing constraints [2, 11], one needs to: disconnect the constraint from the constraint network, set back values by undoing the past events (which are easily accessible thanks to the recorded explanations, see above), and re-propagate to get back to consistent state.

## 4 Solving dynamic timetabling problems

Using explanations provides efficient solving techniques for dynamic problems [20]. In this section, we describe a branch-and-bound algorithm used to solve RCPSP and the addition of explanation capabilities into it in order to provide a dynamic timetabling problem solver.

### 4.1 Principle

We developed an environment for solving dynamic RCPSP and timetabling problems [14] which is based upon:

- a branch-and-bound algorithm for RCPSP (inspired from [6]) within a constraint programming solver: in each node, deduction rules are applied in order to determine redundant information (constraint[5] propagation).
- an extensive use of *explanations*. Explanations are recorded during search and propagation (*i.e.* using the propagation rules – namely *core-times* and *task-interval*) which has been upgraded in order to provide a precise explanation for every deduction made (see section 4.3). They are used to handle

[5] We call *constraint* each initial constraint of the problem (precedence and resource constraints), but also each decision taken by the branching scheme, and each deduction made (thanks to propagation rules) during the search as mentioned in section 3.

dynamic events. Indeed, an unexpected event leads to add, modify or remove a constraint in the system. In the first two cases, if the current solution is no more valid, then the explanations tell us which are the constraints responsible for the contradiction. Repairing is done by removing at least one constraint (preferably a search decision) from the explanation of the contradiction, and by adding its negation. The resulting solution is generally quite similar to the previous one, and is found faster than if we have had solved the problem from scratch.

### 4.2 Branch-and-bound algorithm

The branch-and-bound algorithm used in our approach is inspired from [6].

**The branching scheme** Each node of the tree search is defined by three disjoint sets: a *conjunction* set, a *parallel* set and a *flexible* set. The *conjunction* set $C$, contains all pairs of activities $(i, j)$ in conjunction (*i.e.* that satisfy one of the relations $i \rightarrow j$ or $j \rightarrow i$). The *parallel* set $P$, consists in all pairs of activities $(i, j)$ that overlap (*i.e.* that satisfy the relations $i \parallel j$) and finally the *flexible* set $F$, contains all the remaining pairs of activities $(i, j)$.

Branching is done by transferring one pair of activities $(i, j)$ from $F$ either to set $C$ by imposing the relation $i \rightarrow j$ or $j \rightarrow i$, or to set $P$ (see Fig. 4.2). This branching is repeated until set $F$ becomes empty.

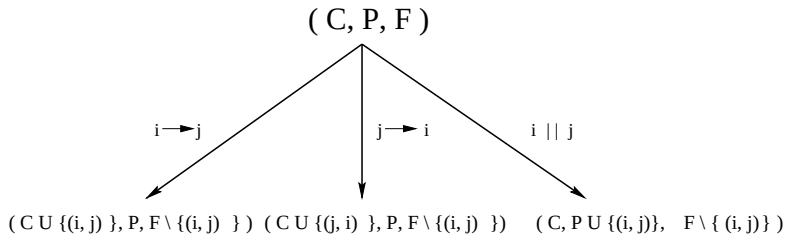


**Fig. 2.** *Branching scheme*

In each node, constraint propagation updates sets $C$ and $P$ by removing pairs from set $F$ when constraints are added. Repairs (or backtracks) put back pairs from $C$ and $P$ to $F$.

**Simple expression of relations: notion of distance** In order to implement our approach, it is necessary to be able to easily express both a decision taken in the search tree and its negation. For example, if we consider a possible decision $i \rightarrow j$, its negation is the disjunction $j \rightarrow i \vee i \parallel j$. Disjunctions are not

easily posted nor handled when performing search in constraint programming. To overcome that problem, we introduced the notion of *distance*[6].

*Definition*

*Let $i$ and $j$ be two distinct activities.*
*The distance $d_{ij}$ between $i$ and $j$ is defined as the time between the ending date of $i$ and the starting date of $j$.*
*We have: $d_{ij} = t_j$ - $t_i$ - $p_i$ (see Fig. 3)*

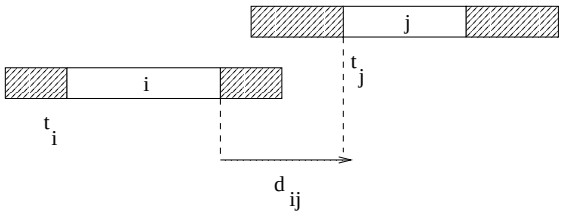


**Fig. 3.** *Distance between two activities*

The relative positions of activities $i$ and $j$ can easily be deduced according to the value of the distance $d_{ij}$ (see Fig. 4).

- $d_{ij} \geq 0$ iff $i \rightarrow j$.
- $d_{ij} \leq -p_i - p_j$ iff $j \rightarrow i$.
- $-p_i - p_j < d_{ij} < 0$ iff $i$ and $j$ overlap.

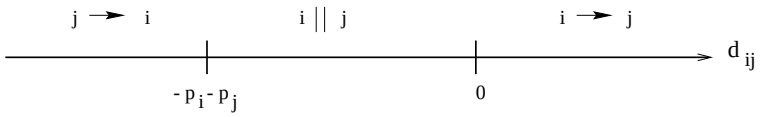


**Fig. 4.** *Positions of two activities $i$ and $j$ according to the value of $d_{ij}$*

We can deduce that:

- $d_{ij} \times d_{ji} \leq 0$ iff $i$ and $j$ are in disjunction.
- $d_{ij} \times d_{ji} > 0$ iff $i$ and $j$ overlap.
- $d_{ij}$ and $d_{ji}$ cannot both take a strictly positive value at the same time.

[6] This notion is a generalization of the one introduced by Brucker in [6].

Using this notion of distance, the decisions taken in the search tree and their negations can easily be translated in term of mathematical constraints (see Table 1).

**Table 1.** *Decisions as constraints on distances*

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

Furthermore, as we will see in Section 5.1, generalized temporal constraints can also easily be translated using distances.

The next two sections describe how explanations are added in the constraints that are used to enforce decisions made during search and also in the initial constraints of the problem (both temporal and resource-related).

### 4.3 Adding explanations to temporal constraints

Providing explanations for temporal binary constraints is straightforward. Therefore, here are the explanations for basic temporal constraints (notice that explanations for generalized temporal constraints can easily be deduced):

- $d_{ij} \geq 0$ (resp. $d_{ij} > -p_i - p_j$)
  The lower bound of the variable $d_{ij}$ is updated to $0$ (resp. $-p_i - p_j + 1$). This modification is only due to the constraint itself, and hence the explanation of the modification is the constraint itself.
- $d_{ij} < 0$ (resp. $d_{ij} \leq -p_i - p_j$)
  The upper bound of the variable $d_{ij}$ is updated to $-1$ (or $-p_i - p_j - 1$). This modification is only due to the constraint itself, and hence the explanation of this modification is the constraint itself.
- $d_{ij} \times d_{ji} > 0$
  If the upper bound of the variable $d_{ij}$ (resp. $d_{ji}$) becomes strictly negative then the upper bound of the variable $d_{ji}$ (resp. $d_{ij}$) is updated to $-1$[7]. This modification is due first to the use of the constraint itself and second to the previous modification of the upper bound of variable $d_{ij}$ (resp. $d_{ji}$), and hence the explanation of this modification is twofold: the explanation of the previous modification for the upper bound of the variable $d_{ij}$ (resp. $d_{ji}$) and the constraint itself.
- $d_{ij} \times d_{ji} \leq 0$
  If the lower bound of the variable $d_{ij}$ (resp. $d_{ji}$) becomes positive then the

[7] We manipulate here integer variables.

upper bound of the variable $d_{ji}$ (resp. $d_{ij}$) is updated to *0*. This modification is due to the use of the constraint itself and the previous modification of the lower bound of the variable $d_{ij}$ (resp. $d_{ji}$). The explanation of this modification is twofold: the explanation of the previous modification for the lower bound of the variable $d_{ij}$ (resp. $d_{ji}$) and the constraint itself.

### 4.4 Adding explanations to resource constraints

Explanations for resource management constraints are not that easy. It is necessary to study the algorithms used for propagation. Classical techniques for maintaining resource limitations for scheduling problems are: *core-times* [25], *task-interval* and *resource-histogram* [9, 10].

**Resource-histogram constraints** The principle of the *resource-histogram* technique is to associate to each resource $k$ an array $\texttt{level}(k)$ 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 [25] is used for detecting contradictions. A *core-time* $\texttt{CT}(i)$ is associated to each activity $i$. It is defined as the interval of time during which a portion of an activity is always executed wether it starts at its earliest or latest starting time. A lower bound of the schedule is obtained when considering only the *core-time* of each activity.

Combining these two techniques provides an efficient resource-conflict detecting constraint. A timetable for each resource is computed as follows: for each activity $i$, its *core-time* $\texttt{CT}(i) = [f_i, r_i + p_i)$ ($f_i$ is the latest starting time of $i$, $r_i$ its earliest starting time) is computed and the amount $a_{ik}$ of resource $k$ for each time interval $[f_i, f_i + 1), \ldots, [r_i + p_i - 1, r_i + p_i)$ is reserved. We associate to each *timetable constraint* two histograms (see Fig. 5):

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

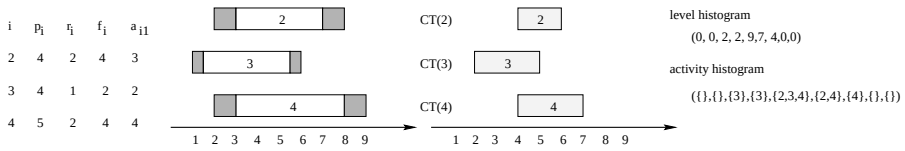


**Fig. 5.** Example of timetable.

These histograms are used in the following ways:

– detecting resource conflicts when the required level of a resource $k$ at one time period $t$ exceeds the resource capacity at that time[8]. The conflict set associated to this contradictory situation is constituted from the set of activities ($S_t$) stored in slot $t$ of the activity histogram. Let $t_v$ be the variable representing the starting time of activity $v$. The explanation of this conflict is given by the following equation ($c$ being the histogram constraint itself):

$$\left( \bigwedge_{v \in S_t} \left( \bigwedge_{a \in d(t_v)} \texttt{expl}(t_v \neq a) \right) \right) \bigwedge c \tag{4}$$

– tightening the time window of an activity. It may occur that the current bounds of the time window of an activity are not compatible with the other activities. In that situation, the tightening of the time-window will be explained by the set $S_t$ of activities requiring the resources during the incompatible time-period $[t-1, t)$. The following equation is used to provide an explanation for the modification of a time window ($c$ being the histogram constraint itself):

$$\left( \bigwedge_{v \in S_t} \left( \bigwedge_{a \in d(t_v)} \texttt{expl}(t_v \neq a) \right) \right) \bigwedge c \tag{5}$$

**Task-interval constraints** The technique of *task-intervals* [9, 10] used for managing cumulative resources can detect conflicts, deduce precedences and tighten time-windows.

A *task-interval* $T = [i, j]$ is associated to each pair of activities $(i, j)$ which require the same resource $k$. It is defined as the set of activities $\ell$ which share the same resource and such that $r_i \leq r_\ell$ and $d_\ell \leq d_j$ ($r_i$ being the earliest starting time of the activity $i$ and $d_i$ its due date) *i.e.* that need to be scheduled between tasks $i$ and $j$. The set $\texttt{inside}(TI)$ represents the set of activities constituting the *task-interval*, while $\texttt{outside}(TI)$ contains the remaining activities. Let $\texttt{energy}(TI) = \sum_{\ell \in \texttt{inside}(TI)} (d_\ell - r_\ell) \times a_{\ell k}$ be the energy required by a *task-interval*.

Several propagations rules can be defined upon *task-interval*. We consider here the following two:

**Integrity rule** If *energy(TI)* is greater than the total *energy* $(d_j - r_i) \times R_k$ available in the interval then a conflict is detected. The conflict explanation set is built from the set of activities *inside* the *task-interval i.e.* ($c$ being the constraint associated to this rule):

$$\left( \bigwedge_{v \in inside(TI)} \left( \bigwedge_{a \in d(t_v)} \texttt{expl}(t_v \neq a) \right) \right) \bigwedge c \tag{6}$$

[8] It may not be a constant value when considering resources whose capacity is variable during time.

**Throw rule** This rule consists in tightening the time window of an activity intersecting a given *task-interval TI*. This is done by comparing the necessary energy shared by the activities in *inside(TI)* and another activity $u$ intersecting *TI*, and the energy available during the *task-interval*. The explanation of this update is built from the set of activities *inside* the *task-interval* ($c$ being the constraint associated to this rule):

$$\left(\bigwedge_{v \in inside(TI)} \left(\bigwedge_{a \in d(t_v)} \texttt{expl}(t_v \neq a)\right)\right) \bigwedge c \bigwedge \left(\bigwedge_{a \in d(t_u)} \texttt{expl}(t_u \neq a)\right) \quad (7)$$

## 5 Dynamic events taken into account

This section lists the various dynamic events taken into account in our system[9].

### 5.1 Temporal events

**Adding constraints** Adding a new (generalized) precedence, disjunctive or overlapping constraint between two lectures can be done thanks to the different following procedures adding temporal events in our system:

- **add(i, j, Relation)**: imposes the relationship "Relation" between activities $i$ and $j$. Table 2 gives the constraint added for each possible relationship.

**Table 2.** Constraints added by the procedure **add(i, j, Relation)** for each possible relation between $i$ and $j$

| Relation | Constraint |
|---|---|
| i $\rightarrow$ j | $d_{ij} \geq 0$ |
| j $\rightarrow$ i | $d_{ij} \leq -p_i - p_j$ |
| i $\leftrightarrow$ j | $d_{ij} \times d_{ji} \leq 0$ |
| i $\parallel$ j | $d_{ij} \times d_{ji} > 0$ |

- **add(i, j, Relation, d)**: imposes the relationship "Relation" between activities $i$ and $j$ and that there are exactly $d$ units of time between $i$ and $j$. Table 3 gives the constraint added for each possible relationship.
- **add(i, j, Relation, Dmin, Dmax)**: imposes the relationship "Relation" between activities $i$ and $j$ and that there are at least $Dmin$ and at most $Dmax$ units of time between $i$ and $j$. Table 4 gives the constraint added for each relationship.

[9] Notice that events that can be modelled as simple arithmetic constraints (such as Teacher $A$ cannot give its lecture on March $24^{th}$) are automatically handled by the underlying constraint solver we use. Only specific complex constraints introduced for solving RCPSP and timetabling problems are presented here

**Table 3.** Constraints added by the procedure **add(i, j, Relation, d)** for each possible relation between $i$ and $j$

| Relation | Constraint |
|---|---|
| i $\rightarrow$ j | $d_{ij} = d$ |
| j $\rightarrow$ i | $d_{ij} = -d - p_i - p_j$ |
| i $\leftrightarrow$ j | $d_{ij} = d \vee d_{ji} = d$ |
| i $\parallel$ j | $d_{ij} = -d$ |

**Table 4.** Constraints added by the procedure **add(i, j, Relation, Dmin, Dmax)** for each possible relation between $i$ and $j$

| Relation | Constraint |
|---|---|
| i $\rightarrow$ j | $D_{min} \leq d_{ij} \leq D_{max}$ |
| j $\rightarrow$ i | $-D_{max} - p_i - p_j \leq d_{ij} \leq -D_{min} - p_i - p_j$ |
| i $\leftrightarrow$ j | $D_{min} \leq d_{ij} \leq D_{max} \vee -D_{max} - p_i - p_j \leq d_{ij} \leq -D_{min} - p_i - p_j$ |
| i $\parallel$ j | $-D_{max} \leq d_{ij} \leq -D_{min}$ |

- **add(i, Dmin, Dmax)**: imposes a time window $[Dmin, Dmax]$ for activity $i$. The constraint added is $Dmin \leq i \leq Dmax$.

**Removing constraints** Our system is able to dynamically remove all these temporal events thanks to the following procedures: **remove(i, j, Relation)**, **remove(i, j, Relation, d)**, **remove(i, j, Relation, Dmin, Dmax)**, **remove(i,Dmin, Dmax)**

**Modifying constraints** Each of the temporal events that can be added or removed can also be replaced by another one:

- **modify(i, j, Old_Relation, New_Relation)** replaces relation "Old_Relation" by "New_Relation" between $i$ and $j$. Actually, this procedure removes the constraint associated to the relationship "Old_Relation" and adds the constraint associated to the relationship "New_Relation".

### 5.2 Activity related events

The procedures concerning the activity related events are:

- **add(i, Duration, Res_requirements, Predecessors, Successors)**: this procedure adds an activity $i$ with duration "Duration", resources requirements "Res_requirements". Its predecessors are "Predecessors" and its successors "Successors".
  This procedure creates a new variable $i$, connects it to the constraints network, adds the temporal constraints associated and the disjunction constraints resulting from the capacity limitations, and inserts variable $i$ in the *timetable* and in the *task-interval* constraints.

- **remove(i)**: this procedure removes activity $i$ from the problem.
  It disconnects variable $i$ from the constraints network, removes all the constraints related to either $i$, any $d_{ij}$ or any $d_{ji}$. It also removes variable $i$ from the *timetable* and the *task-interval* constraints, and removes all *task-interval* constraints having $i$ as starting or ending activity.

Activity modification are handled as an activity removal followed by an activity addition (the modified one).

### 5.3 Resource related events

It is possible to add or remove a resource:

- **add(Resource, Capacity, Capacities requirements)** adds a new resource to the problem and the disjunctive constraints which result from its capacity limitation. It also adds the *timetable* and *task-interval* constraints associated to it.
- **remove(Resource)**: removes a resource from the problem, as well as the disjunctive constraints which results from the capacity limitation of this resource, the *timetable* and all the *task-interval* constraints associated.

As for activities, modifying a resource is handled as a resource removal followed by a resource addition.

### 5.4 Translating timetabling-related events into our event system

Obviously, the events described so far are not meant to be directly used by a final user of our system. A translation between the real-life events and our events needs to be done. For example, when considering timetabling problems, the following events could be handled:

- The addition of *new lectures for an existing course*, for example because a teacher needs more hours than expected, can be handled as the addition of new activities which may be related to other activities by (generalized) precedence, disjunctive or overlapping constraints.
- The suppression of one or several lectures of a teacher who finishes his/her course earlier than expected implies the removal of one or several activities.
- The installation of a new classroom equipped with computers, or the purchase of video-conference materials corresponds to the addition of a new resource or to the modification of the capacity of a resource.
- The fact that *a teacher is no longer available on the afternoons* can be handled as a modification (namely its capacity level) of a resource.
- etc.

### 5.5 The special case of over-constrained problems

Uncontrolled addition, removal or modification of constraints can lead to an over-constrained problem. Our explanation-based system is then able to provide a contradiction explanation which will help the user either to select a constraint/relation to be removed or modified, or to allow an increase of the resulting makespan of the project.

## 6 Computational experiments

We present here our first experimental results. Our experiment consists in comparing our dynamic scheduler with a static scheduler: a scheduling problem $P$ 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.

All experiments[10] were conducted on RCPSP as our system was primarily design for such problems. Experiments on real-life timetabling problems are planned in a near future.

### 6.1 First benchmark: Patterson RCPSP instances

We use here some RCPSP test problems introduced by Patterson[11].

For this set of problems, we tried to evaluate the impact of a single modification when comparing static and dynamic scheduling approaches. The following events were evaluated:

- Adding a precedence constraint (**add(i, j, $\rightarrow$)**)
  Table 5 presents the results obtained on original problems from Patterson from which a single precedence constraint has been removed (which gives the starting problem) then added (which gives the final problem – the original Patterson one[12]).
- Adding a generalized precedence constraint (**add(i, j, $\rightarrow$, d)**)
  Table 6 presents the results obtained on original problems from Patterson to which a generalized precedence constraint is added (this constraint is an existing precedence in the optimal solution for which the existing time lag is increased).
- Adding a generalized precedence constraint (**add(i, j, $\rightarrow$, Dmin, Dmax)**)
  Table 7 presents the results obtained on original problems from Patterson to which a generalized precedence constraint is added (this constraint is an existing precedence in the optimal solution for which the existing time lag is increased).

[10] All experimental data and results are available online at `http://www.emn.fr/jussien/RCPSP`.

[11] `www.bwl.uni-kiel.de/Prod/psplib/dataob.html`

[12] This explains that some results obtained for the static solver are the same: the final problem is the same but not the original one as a different constraint has been removed.

– Adding an overlapping constraint (**add(i, j, ||)**) Table 8 presents the results obtained on original problems from Patterson to which a randomly chosen overlapping constraint is added.

.

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 in seconds needed for solving optimally problem $P$;
– $t_e$: the `cpu` time in seconds spent to solve this problem after dynamically adding the event $e$
– $t_{pe}$: the `cpu` time in seconds needed to obtain an optimal solution of the problem $P \cup \{e\}$ from scratch.
– $\frac{t_{pe}-t_e}{t_{pe}}$: the overall interest of 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 %!

**Table 5.** Adding a precedence constraint

| Name | #Act | #Res | $t_p$ | $t_e$ | $t_{pe}$ | $\frac{t_{pe}-t_e}{t_{pe}}$% |
|---|---|---|---|---|---|---|
| T1P1a | 14 | 3 | 7.26 | 1.16 | 6.96 | 83.3 |
| T1P1b | 22 | 3 | 6.75 | 0.23 | 6.96 | 96.7 |
| T1P2 | 7 | 3 | 0.23 | 0.02 | 0.19 | 89.5 |
| T1P3a | 13 | 3 | 25.82 | 4.16 | 12.58 | 67.0 |
| T1P3b | 13 | 3 | 15.64 | 1.21 | 12.58 | 90.4 |
| T1P4a | 22 | 3 | 8.01 | 1.46 | 3.63 | 59.8 |
| T1P4b | 22 | 3 | 4.50 | 0.50 | 3.63 | 86.2 |
| T1P5a | 22 | 3 | 20.37 | 8.70 | 13.80 | **37.0** |
| T1P5b | 22 | 3 | 11.34 | 1.76 | 13.80 | 87.3 |
| T1P6a | 22 | 3 | 135.11 | 19.60 | 78.61 | 75.1 |
| T1P6b | 22 | 3 | 36.70 | 7.81 | 78.61 | 90.1 |
| T1P6c | 22 | 3 | 87.80 | 0.96 | 78.61 | **98.8** |
| T1P8 | 9 | 1 | 2.11 | 0.32 | 2.30 | 86.1 |
| T1P10 | 8 | 2 | 0.34 | 0.06 | 0.25 | 76.0 |

**Table 6.** Adding a generalized precedence constraint

| Name | #Act | #Res | $t_p$ | $t_e$ | $t_{pe}$ | $\frac{t_{pe}-t_e}{t_{pe}}\%$ |
|---|---|---|---|---|---|---|
| T2P1a | 14 | 3 | 7.31 | 0.87 | 8.84 | 90.2 |
| T2P1b | 14 | 3 | 6.58 | 0.82 | 9.55 | 91.4 |
| T2P5a | 22 | 3 | 14.90 | 3.45 | 8.33 | 58.6 |
| T2P5b | 22 | 3 | 16.90 | 3.44 | 8.19 | 58.0 |
| T2P5c | 22 | 3 | 13.82 | 3.50 | 8.22 | 57.4 |
| T2P6a | 22 | 3 | 62.45 | 5.14 | 24.43 | 79.0 |
| T2P6b | 22 | 3 | 64.42 | 1.87 | 19.56 | 90.5 |
| T2P6c | 22 | 3 | 62.50 | 1.24 | 23.40 | **94.7** |
| T2P8 | 9 | 1 | 2.19 | 0.22 | 2.58 | 91.5 |
| T2P10 | 8 | 2 | 0.18 | 0.02 | 0.26 | 92.3 |
| T2P11 | 8 | 2 | 0.39 | 0.20 | 0.42 | **52.4** |

**Table 7.** Adding a generalized precedence constraint

| Name | #Act | #Res | $t_p$ | $t_e$ | $t_{pe}$ | $\frac{t_{pe}-t_e}{t_{pe}}\%$ |
|---|---|---|---|---|---|---|
| T3P1a | 14 | 3 | 7.12 | 0.81 | 8.50 | 90.5 |
| T3P1b | 14 | 3 | 9.69 | 0.61 | 9.81 | 93.8 |
| T3P3 | 13 | 3 | 17.49 | 0.45 | 14.34 | **96.9** |
| T3P5a | 22 | 3 | 14.46 | 4.82 | 7.59 | 36.5 |
| T3P5b | 22 | 3 | 15.25 | 3.10 | 7.81 | 60.3 |
| T3P6a | 22 | 3 | 68.38 | 9.63 | 19.30 | 50.1 |
| T3P6b | 22 | 3 | 75.88 | 2.52 | 21.12 | 88.1 |
| T3P6c | 22 | 3 | 66.00 | 17.74 | 27.34 | **35.1** |
| T3P6d | 22 | 3 | 65.45 | 16.25 | 30.65 | 47.0 |
| T3P8 | 9 | 1 | 2.90 | 0.12 | 2.84 | 95.8 |

**Table 8.** Adding an overlapping constraint

| Name | #Act | #Res | $t_p$ | $t_e$ | $t_{pe}$ | $\frac{t_{pe}-t_e}{t_{pe}}\%$ |
|---|---|---|---|---|---|---|
| T4P1 | 14 | 3 | 7.43 | 0.38 | 10.62 | **96.4** |
| T4P5 | 22 | 3 | 15.16 | 16.39 | 24.84 | 34.0 |
| T4P6a | 22 | 3 | 77.73 | 8.44 | 14.60 | 42.2 |
| T4P6b | 22 | 3 | 70.45 | 6.63 | 18.59 | 64.3 |
| T4P6c | 22 | 3 | 64.87 | 9.85 | 12.86 | **23.4** |
| T4P8a | 9 | 1 | 2.37 | 0.38 | 2.35 | 83.9 |
| T4P8b | 9 | 1 | 2.27 | 0.12 | 2.64 | 95.5 |

### 6.2 Second benchmark: Kolish, Sprecher and Drexl RCPSP instances

Our second set of experiments performed on Kolish, Sprecher and Drexl RCPSP instances[13] considers 4 consecutive modifications (any kind of events[14]) to an original problem. As we can see on table 9, our results are quite promising. Even bad results (instance T1KSD10) get better in the long run. However, notice that some results (see table 10 – results for 5 consecutive modifications) show that dynamic handling is not always the panacea and rescheduling from scratch can be very quick.

**Table 9.** Some Kolish, Sprecher and Drexl instances (4 consecutive dynamic events). Relative speed-up (in %)

| 12 act./4 res. | Modif. 1 | Modif. 2 | Modif. 3 | Modif. 4 |
|---|---|---|---|---|
| T1KSD1 | 12.76 | 26.09 | 35.84 | 35.58 |
| T1KSD2 | 46.24 | 54.92 | 62.68 | 63.13 |
| T1KSD3 | 35.88 | 44.12 | 45.82 | 45.87 |
| T1KSD5 | 32.01 | 67.90 | 75.15 | 75.65 |
| T1KSD6 | 3.81 | 26.38 | 46.62 | 55.92 |
| T1KSD8 | 22.13 | 10.64 | 21.08 | 22.21 |
| T1KSD9 | 46.72 | 52.58 | 47.75 | 49.27 |
| T1KSD10 | -29.21 | -0.35 | 15.74 | 30.27 |

**Table 10.** Other Kolish, Sprecher and Drexl instances (5 consecutive dynamic events). Relative speed-up (in %)

| 22 act./4 res. | Modif. 1 | Modif. 2 | Modif. 3 | Modif. 4 | Modif. 5 |
|---|---|---|---|---|---|
| T2KSD11 | 47.15 | 12.49 | 5.87 | 6.89 | -1.98 |
| T2KSD13a | 34.81 | 43.55 | 43.04 | 49.89 | 50.98 |
| T2KSD13b | 39.19 | -126.79 | -174.22 | -216.67 | -146.42 |
| T2KSD15 | -47.62 | -42.50 | -41.00 | -65.06 | -88.20 |
| T2KSD17a | -2.40 | 1.65 | -5.90 | -16.23 | -43.06 |
| T2KSD17b | 3.58 | 13.11 | -12.76 | -10.51 | -74.91 |
| T2KSD18a | 30.61 | 26.76 | -52.76 | -19.07 | 0.85 |
| T2KSD18b | 40.39 | 56.09 | 40.54 | 42.56 | 45.15 |

[13] `www.wior.uni-karlsruhe.de/RCPSP/ProGen.html`
[14] See `http://www.emn.fr/jussien/RCPSP` for the details of each instance.

## 7 Conclusion

In this paper, we presented the integration of explanations within scheduling-related global constraints and their interest for solving dynamic timetabling problems. We presented first experimental results of a system that we developed using those techniques. These results demonstrate that incremental constraint solving for scheduling problem is useful.

We are currently improving our system with user-interaction capabilities still using explanations following [23]. Moreover, our main goal now is to give our system to the timetabling service in our institution in order to evaluate it in a real-world situation.

## References


1. 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):179–198, 2000.
2. Christian Bessière. Arc consistency in dynamic constraint satisfaction problems. In *Proceedings AAAI'91*, 1991.
3. 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.
4. Patrice Boizumault, Yann Delon, and Laurent Péridy. Constraint logic programming for examination timetabling. *Special Issue of the Journal of Logic Programming: Applications of Logic Programming*, 26(2):217–233, 1996.
5. P. Brucker, A. Drexl, R. Möring, K. Neumann, and E. Pesch. Resource-constrained project scheduling: notation, classification, models and methods. *European Journal of Operational Research*, 112:3–41, 1999.
6. 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.
7. Peter Brucker and Sigrid Knust. Resource-constraints project scheduling and timetabling. In *Selected papers of the 2000 Practice and Theory of Automated Timetabling international conference*, volume 2079 of *Lecture Notes in Computer Science*, pages 277–293. Springer-Verlag, 2001.
8. E. K. Burke, D. G. Elliman, and R. F. Weare. A university timetabling system based on graph colouring and constraint manipulation. *Journal of Research on Computing in Education*, 27(1):1–18, 1994.
9. Y. Caseau and F. Laburthe. Cummulative scheduling with task-intervals. In *JICSLP'96: Joint International Conference and Symposium on Logic Programming*, 1996.
10. 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.
11. Romuald Debruyne, Gérard Ferrand, Narendra Jussien, Willy Lesaint, Samir Ouis, and Alexandre Tessier. Correctness of constraint retraction algorithms. In *FLAIRS'03: Sixteenth international Florida Artificial Intelligence Research Society conference*, pages ???–???, St. Augustine, Florida, USA, may 2003. AAAI press.

12. E. Demeulemeester and W. Herroelen. A branch and bound procedure for the multiple resource-constrained project scheduling problem. *Management Science*, 38(12):1803–1818, 1992.
13. F. P. M. Dignum, W. P. M. Nuijten, and L. M. A. Janssen. Solving a time tabling problem by constraint satisfaction. Technical report, Eindhoven University of Technology, 1995.
14. Abdallah Elkhyari, Christelle Guéret, and Narendra Jussien. Conflict-based repair techniques for solving dynamic scheduling problems. In *Principles and Practice of Constraint Programming (CP 2002)*, number 2470 in Lecture Notes in Computer Science, pages 702–707, Ithaca, NY, USA, September 2002. Springer-Verlag. Short paper.
15. H.J. Goltz. Combined automatic and interactive timetabling using constraint logic programming. In E. Burke and W. Erben, editors, *Int. Conf. on the Practice and Theory of Automated Timetabling (PATAT'00)*, 2000.
16. Christelle Guéret, Narendra Jussien, Patrice Boizumault, and Christian Prins. Building university timetables using Constraint Logic Programming. In Edmund Burke and Peter Ross, editors, *The Practice and Theory of Automated Timetabling*, volume 1153 of *Lecture Notes in Computer Science*, pages 130–145. Springer-Verlag, 1996.
17. 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.
18. A. Hertz. Tabu search for large scale timetabling problems. *European Journal of Operations Research*, 54:39–47, 1991.
19. Narendra Jussien. e-constraints: explanation-based constraint programming. In *CP01 Workshop on User-Interaction in Constraint Satisfaction*, Paphos, Cyprus, 1 December 2001.
20. Narendra Jussien and Patrice Boizumault. Dynamic backtracking with constraint propagation – application to static and dynamic csps. In *CP97 Workshop on The Theory and Practice of Dynamic Constraint Satisfaction*, Schloss Hagenberg, Austria, 1 November 1997.
21. 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.
22. Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1):21–45, July 2002.
23. Narendra Jussien and Samir Ouis. User-friendly explanations for constraint programming. In *ICLP'01 11th Workshop on Logic Programming Environments (WLPE'01)*, Paphos, Cyprus, 1 December 2001.
24. L. Kang and G. M. White. A logic approach to the resolution of constraint in timetabling. *European Journal of Operational Research*, 61:306–317, 1992.
25. 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.
26. R. Kolisch and S. Hartmann. *Heuristic Algorithms for Solving the Resource-Constrained Project Scheduling Problem: Classification and Computational Analysis*, pages 147–178. Weglarz, J. (Ed.): Handbook on Recent Advances in Project Scheduling, Kluwer, Dordrecht, 1998.

27. G. Lajos. Complete university modular timetabling using constraint logic programming. In P. Ross and E. Burke, editors, *The practice and Theory of Automated Timetabling: Selected Papers from the 1st International Conference*, number 1153 in Lecture Notes in Computer Science, pages 148–161, 2000.
28. A. Mingozzi, V. Maniezzo, S. Ricciardelli, and L. Bianco. An exact algorithm for project scheduling with resource constraints based on a new mathematical formulation. *Management Science*, 44:714–729, 1998.
29. P. Ross, D. Corne, and H-L. Fang. Improving evolutionary timetabling with delta evaluation and directed mutation. *Parallel Problem Solving in Nature*, III:565–566, 1994.
30. Thomas Schiex and Gérard Verfaillie. Nogood recording for static and dynamic CSP. In $5^{th}$ *IEEE International Conference on Tools with Artificial Intelligence*, pages 48–55, Boston, MA., 1993.
31. J. Sgall. On-line scheduling - a survey. *On-Line Algorithms*, 1997.
32. J. P. Stinson, E. W. David, and B. M. Khamawala. Multiple resource-constrained scheduling using branch and bound. *IIE Transactions*, 1:252–259, 1978.
33. A. Tripathy. School timetabling - a case in large binary integer linear programming. *Management Science*, 30:1473–1489, 1984.