

Dynamic Constraint Programming for Solving Hard Real-Time Allocation Problems

Pierre-Emmanuel Hladik¹, Hadrien Cambazard²
Anne-Marie Déplanche¹, Narendra Jussien²

¹ IRCCyN, UMR CNRS 6597

1 rue de la Noë – BP 9210

44321 Nantes Cedex 3, France

{hladik,deplanche}@irccyn.ec-nantes.fr

²LINA CNRS

4 rue Alfred Kastler – BP 20722

44307 Nantes Cedex 3, France

{hcambaza,jussien}@emn.fr

Abstract

This paper is a synthesis of different works [12, 9, 8, 13]. Problems addressed in this paper consist in assigning periodic tasks to distributed processors in the context of fixed priority preemptive scheduling. A cooperative technique using an accurate management of constraint programming is presented to solve these problems. The approach breaks down a problem into two subproblems: allocation and schedulability. Allocation problem is tackled with dynamic constraint programming whereas schedulability is analyzed with classical real-time techniques. Decomposition is used as a way of learning when the allocation subproblem yields a valid solution while the schedulability analysis of the allocation does not. The rationale of this approach is to learn from the failures of the schedulability analysis to reduce the search space. Additionally, different search strategies are proposed to increase efficiency of the resolution.

1. Introduction

This paper addresses the issue of hard real-time periodic, preemptive task assignment to distributed processors in the context of fixed priority scheduling. An assignment must respect task schedulability but also account for requirements related to memory capacity, co-residence, redundancy, etc. We assume that the characteristics of tasks (such as execution time or priority) and the physical architecture (processors and networks) are known.

The problem of assigning a set of hard preemptive real-time tasks in a distributed system so as to meet constraints, is known to be NP-Hard [21]. It has been widely tackled with heuristic methods [27, 23], simulated annealing [30, 4], genetic algorithms [11, 26]. Recently, Szymanek et al. [28] and especially Ekelin [10] have used constraint programming to produce an assignment and a pre-runtime scheduling of distributed systems under optimization criteria.

This paper presents a decomposition-based method (related to logic Benders-based decomposition [15]) for solving allocation problem. This method separates the allocation problem from the schedulability one: allocation is solved by means of *dynamic constraint programming* tools, whereas

schedulability is checked with specific real-time scheduling analyses. The main idea is to "learn" from the schedulability analysis how to re-model the allocation problem and reduce the search space. In that sense, we can compare this approach to a form of *learning from mistakes*.

Extensions presented in this paper are related to heuristics for search strategies in constraint programming. They consist in defining an algorithm that specifies which variable with which value must be chosen to guide the method into the search tree. These heuristics preserve the completeness of the approach. Different search strategies are proposed and we provide experimental studies to evaluate their impact.

The remainder of this paper is organized as follows. In Section 2, the problem is described. Section 3 is dedicated to the description of the master/subproblems and the communication between them. The method is applied to a short toy example in Section 4. In Section 5, we describe different search strategies to solve allocation problem. Finally, experimental results are presented in Section 6.

2 The allocation problem

2.1 The real-time system architecture

Hard real-time systems we consider can be modeled by a software architecture: the set of tasks, together with a hardware architecture: the physical execution platform for the tasks, as depicted in Fig. 1.

The *hardware architecture* consists of a set $\mathcal{P} = \{p_1, \dots, p_k, \dots, p_m\}$ of m processors with fixed memory capacity m_k and identical processing speed. They are fully connected to a network (with a bandwidth δ) using a token ring protocol: a token travels around the ring and processors are allowed to send data only when they hold the token. It stays at the same place for a fixed maximum period of time which is large enough to ensure that all messages waiting on the processor are sent. This protocol was chosen by similarity with Tindell's approach in [30]. At this time, an extension to CAN protocol is being proceeded.

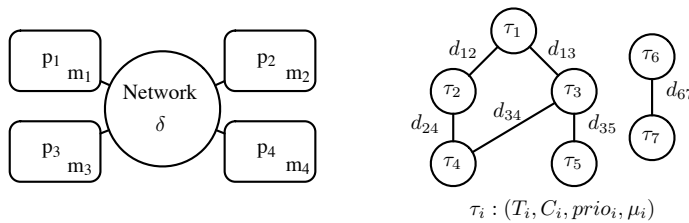


Figure 1. An example of hardware (left) and software (right) architecture.

The *software architecture* is modeled as a valued, oriented and acyclic graph $(\mathcal{T}, \mathcal{C})$. The set of nodes $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$ corresponds to the tasks and the set of edges $\mathcal{C} \subseteq \mathcal{T} \times \mathcal{T}$ refers to the messages sent between tasks.

A task τ_i is defined through timing attributes and resource needs: its period T_i (as a task is periodically activated), its worst-case execution time without preemption C_i and its memory need¹ μ_i . Edges $c_{ij} = (\tau_i, \tau_j) \in \mathcal{C}$ are valued with the amount of exchanged data: d_{ij} . Communicating tasks have the same activation period. Moreover, they are able to communicate in two ways: a local communication without any delay (via processor memory) that requires the tasks to be located on the same processor, and a distant communication which uses the network. In both situations, we do not consider

¹Within this model, only static real-time systems in which all memory resources are known *a priori* are considered.

any precedence constraints. Tasks are periodically activated in an independent way, and they read, respectively write, data at the beginning, respectively at the end, of their execution.

Finally, each processor is scheduled with a fixed priority strategy. Thus, a priority $prio_i$ is given to each task. Task τ_j has priority over τ_i if and only if $prio_i < prio_j$. The execution of a task may be preempted by tasks with higher priority.

2.2 The allocation problem

An allocation is a mapping $A : \mathcal{T} \rightarrow \mathcal{P}$ such that:

$$\tau_i \mapsto A(\tau_i) = p_k \quad (1)$$

The allocation problem consists in finding a mapping A which respects the whole set of constraints described in the immediate following.

There are three classes of constraints the allocation problem must respect: timing, resource, and allocation constraints.

Timing constraints. They are expressed by the means of deadlines for the tasks. A timing constraint enforces the duration between the activation date of any instance of the task τ_i and its completion time to be bounded by its deadline D_i (D_i is detailed in Section 3.2).

Resource constraints. Three kinds of constraints are considered:

- **Memory capacity:** The memory use of a processor p_k cannot not exceed its capacity (m_k):

$$\forall k = 1..m, \quad \sum_{A(\tau_i)=p_k} \mu_i \leq m_k \quad (2)$$

- **Utilization factor:** The utilization factor of a processor cannot exceed its processing capacity. The following inequality is a necessary schedulability condition :

$$\forall k = 1..m, \quad \sum_{A(\tau_i)=p_k} \frac{C_i}{T_i} \leq 1 \quad (3)$$

- **Network use:** To avoid overload, the amount of data carried along the network per unit of time cannot exceed the network capacity:

$$\sum_{\substack{c_{ij} = (\tau_i, \tau_j) \\ A(\tau_i) \neq A(\tau_j)}} \frac{d_{ij}}{T_i} \leq \delta \quad (4)$$

Allocation constraints. Allocation constraints are due to the system architecture. We distinguish three kinds of constraints.

- **Residence:** Sometimes, a task needs a specific hardware or software resource which is only available on specific processors (*i.e.* a task monitoring a sensor has to be run on a processor connected to the input peripheral). This constraint is expressed as a couple (τ_i, α) where $\tau_i \in \mathcal{T}$ is a task and $\alpha \subseteq \mathcal{P}$ is the set of available host processors for the task. A given allocation A must respect:

$$A(\tau_i) \in \alpha \quad (5)$$

- **Co-residence:** This constraint enforces several tasks to be assigned to the same processor (they share a common resource). Such a constraint is defined by a set of tasks $\beta \subseteq \mathcal{T}$ and any allocation A has to fulfil:

$$\forall(\tau_i, \tau_j) \in \beta^2, A(\tau_i) = A(\tau_j) \quad (6)$$

- **Exclusion:** Some tasks may be replicated for fault-tolerance and therefore cannot be assigned to the same processor. It corresponds to a set $\gamma \subseteq \mathcal{T}$ of tasks which cannot be placed together. An allocation A must satisfy:

$$\forall(\tau_i, \tau_j) \in \gamma^2, A(\tau_i) \neq A(\tau_j) \quad (7)$$

An allocation is said to be *valid* if it satisfies allocation and resource constraints. It is *schedulable* if it satisfies timing constraints. Finally, a solution for our problem is both a valid and schedulable allocation of the tasks.

3 Solving the problem

Constraint Programming (CP) techniques have been widely used to solve a large range of combinatorial problems. A *constraint satisfaction problem* (CSP) consists of a set V of variables defined by a corresponding set D of possible values (the so-called *domain*) and a set C of constraints. A solution to the problem is an assignment of a value in D to each variable in V such that all constraints are satisfied. This mechanism coupled with a backtracking scheme allows the search space to be explored in a *complete way*. For a deeper introduction to CP, we refer to [2].

Constraint Programming has proved to be quite effective in a wide range of applications (from planning and scheduling to finance – portfolio optimization – through biology) because of its main advantages: declarativity (the variables, domains, constraint description), genericity (it is not a problem dependent technique) and adaptability (to unexpected side constraints).

We propose an approach inspired from methods used to integrate constraint programming into a logic-based Benders decomposition [3, 15]. The allocation and resource constraints are considered on one side, and timing ones on the other (see Fig. 2). The master problem solved with constraint programming yields a valid allocation. The subproblem checks the schedulability of this allocation, finds out why it is unschedulable and designs a set of constraints, named *nogoods* which rules out all the assignments which are unschedulable for the same reason.

We will therefore introduce Benders decomposition [3] scheme to highlight the underlying concepts. Benders decomposition can be seen as a form of *learning from mistakes*. It is a solving strategy that uses a partition of the problem among its variables: x, y . The strategy can be applied to a problem of this general form:

$$\begin{aligned} \text{P : Min } & f(x) + cy \\ \text{s.t } & g(x) + Ay \geq a \text{ with } : x \in D, y \geq 0 \end{aligned}$$

A master problem considers only a subset of variables x (often integer variables, D is a discrete domain). A subproblem (SP) tries to complete the assignment on y and produces a Benders cut added to the master problem. This cut has the form $z \geq h(x)$ and constitutes the key point of the method, it is inferred by the dual of the subproblem. Let us consider an assignment x^* given by the master, the subproblem (SP) and its dual (DSP) can be written as follows:

$$\begin{array}{ll} \text{SP : Min } & cy \\ \text{s.t } & Ay \geq a - g(x^*) \text{ with } : y \geq 0 \end{array} \qquad \begin{array}{ll} \text{DSP : Max } & u(a - g(x^*)) \\ \text{s.t } & uA \leq c \text{ with } : u \geq 0 \end{array}$$

Duality theory ensures that $cy \geq u(a - g(x^*))$. As feasibility of the dual is independent of x^* , $cy \geq u(a - g(x))$ and the following inequality is valid: $f(x) + cy \geq f(x) + u(a - g(x))$. Moreover, according to duality, the optimal value of u^* maximizing $u(a - g(x^*))$ corresponds to the same optimal value of cy . Even if the cut is derived from a particular x^* , it is valid for all x and excludes a large class of assignments which share common characteristics that make them inconsistent. The number of solutions to explore is reduced and the master problem can be written at the I^{th} iteration:

$$\begin{aligned} \text{PM : Min } & z \\ \text{s.t : } & z \geq f(x) + u_i^*(a - g(x)) \quad \forall i < I \end{aligned}$$

From all of this, it can be noticed that dual variables need to be defined to apply the decomposition. However, [15] proposes to overcome this limit and to enlarge the classical notion of *dual* by introducing an *inference dual* available for all kinds of subproblems. He refers to a more general scheme and suggests a different way of thinking about duality: a Benders decomposition based on *logic*. Duality now means to be able to produce a proof, the logical proof of optimality of the subproblem and the correctness of inferred cuts. In the original Benders decomposition, this proof is established thanks to duality theorems.

For a discrete satisfaction problem, the resolution of the dual consists in computing the infeasibility proof of the subproblem and determining under what conditions the proof remains valid. It therefore infers valid cuts.

The success of the decomposition depends on both the degree to which decomposition can exploit structures and the quality of the inferred cuts. [15] suggests to identify classes of structured problems that exhibit useful characteristics for the Benders decomposition. Off-line scheduling problems fall into such classes and [16] demonstrates the efficiency of such an approach on a scheduling problem with dissimilar parallel machines.

Our approach is strongly connected to Benders decomposition and the related concepts. It is inspired from methods used to integrate constraint programming into a Benders scheme [29, 14]. The allocation and resource problem will be considered on one side and schedulability on the other side. The subproblem checks the schedulability of an allocation, finds out why it is unschedulable and designs a set of constraints (both symbolic and arithmetic) which rule out all assignments that are unschedulable for the same reason. Our approach concurs therefore the Benders decomposition on this central element: the Benders cut. The proof proposed here is based on off-line analysis techniques from real-time scheduling. One might think that a fast analytic proof could not provide enough relevant information on the inconsistency. As the speed of convergence and the success of the technique greatly depends on the quality of the cut, a conflict detection algorithm will be coupled with analytic techniques: QuickXplain [17]. Moreover, the master problem will be considered as a dynamic problem to avoid redundant computations as much as possible.

3.1 Master problem

As the master problem is solved using constraint programming techniques, we need first to translate our problem into CSP. The model is based on a redundant formulation using three kinds of variables: x, y, w .

Let us first consider n integer-valued variables x which are decision variables and correspond to each task, representing the processor selected to process the task: $\forall i \in \{1..n\}, x_i \in \{1, \dots, m\}$. Then, boolean variables y indicate the presence of a task on a processor: $\forall i \in \{1..n\}, \forall p \in \{1..m\}, y_{ip} \in \{0, 1\}$. Finally, boolean variables w are introduced to express whether a pair of tasks exchanging a message are located on the same processor or not: $\forall c_{ij} = (\tau_i, \tau_j) \in \mathcal{C}, w_{ij} \in \{0, 1\}$. Integrity constraints are used to enforce the consistency of the redundant model.

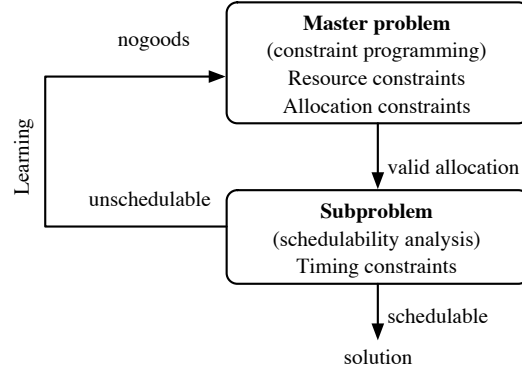


Figure 2. Logic-based Benders decomposition to solve an allocation problem

One of the main objectives of the master problem is to solve efficiently the assignment part. It handles two kinds of constraints: allocation and resource.

- **Memory capacity:** (cf. eq (2)) $\forall p \in \{1..m\}, \sum_{i \in \{1..n\}} y_{ip} \times m_i \leq \mu_p$
- **Utilization factor:** (cf. eq (3)) Let $lcm(T)$ be the least common multiple of periods of the tasks. The constraint can be written as follows:

$$\forall p \in \{1..m\}, \quad \sum_{i \in \{1..n\}} lcm(T) \times WCET_i \times y_{ip} / T_i \leq lcm(T)$$

- **Network use:** (cf. eq (4)) The network bandwidth is bounded by δ . Therefore, the size of the set of messages carried on the network cannot exceed this limit:

$$\sum_{i \in \{1..n\}} lcm(T) \times d_{ij} \times w_{ij} / T_i \leq lcm(T) \times \delta$$

- **Residence:** (cf. eq (5)) it consists of forbidden values for x . A constraint is added for each forbidden processor p of t_i : $x_i \neq p$
- **Co-residence:** (cf. eq (6)) $\forall (t_i, t_j) \in \beta^2, x_i = x_j$
- **Exclusion:** (cf. eq (7)) $alldifferent(x_i | t_i \in \gamma)$

Utilization factor and network use are reformulated with the lcm of tasks periods because our constraint solver cannot currently handle constraints with real coefficients and integer variables.

3.1.1 Incremental resolution.

Solving dynamic constraint problems has led to different approaches. Two main classes of methods can be distinguished: proactive and reactive methods. On the one hand, proactive methods propose to build robust solutions that remain solutions even if changes occur. On the other hand, reactive methods try to reuse as much as possible previous reasonings and solutions found in the past. They avoid restarting from scratch and can be seen as a form of learning. One of the main methods currently used to perform such learning is a justification technique that keeps trace of inferences made by the solver during the search. Such an extension of constraint programming has been recently introduced [18]: explanation-based constraint programming (*e-constraints*).

Definition 1 An explanation records information to justify a decision of the solver as a reduction of domain or a contradiction. It is made of a set of constraints C' (a subset of the original constraints of the problem) and a set of decisions dc_1, \dots, dc_n taken during search. An explanation of the removal of value a for the variable v will be written: $C' \wedge dc_1 \wedge dc_2 \wedge \dots \wedge dc_n \Rightarrow v \neq a$.

When a domain is emptied, a contradiction is identified. An explanation for this contradiction is computed by uniting each explanation of each value removal of the variable concerned. At this point, dynamic backtracking algorithms that only question a relevant decision appearing in the conflict are conceivable. By keeping in memory a relevant part of the explanations involved in conflicts, a learning mechanism can be implemented [19].

Here, explanations allow us to perform an incremental resolution of the master problem. At each iteration, the constraints added by the subproblem generate a contradiction. Instead of backtracking to the last choice point as usual, the current solution of the master problem is *repaired* by removing the decisions that occur in the contradiction as done by the MAC-DBT algorithm [18]. Tasks assigned at the beginning of the search can be moved without disturbing the whole allocation. In addition, the model reinforcement phase tries to transform a learnt set of elementary ones that have been added at previous iterations into higher level constraints. Explanations offer facilities to easily dynamically add or remove a constraint from the constraint network [18].

Notice that the master problem is never re-started. It is solved only once but is gradually *repaired* using the dynamic abilities of the explanation-based solver.

3.1.2 Model reinforcement.

Pattern recognition among a set of constraints that expresses specific subproblems is a critical aspect of the modelisation step. Constraint learning deals with the problem of automatically recognizing such patterns. We would like to perform a similar process in order to extract global constraints among a set of elementary constraints. For instance, a set of difference constraints can be formulated as an all-different constraint by looking for a maximal clique in the induced constraint graph. It is a well-known issue to this question in constraint programming and a version of the Bron/Kerbosh algorithm [5] has been implemented to this end (difference constraints occur when *NotAllEquals* involve only two tasks). In a similar way, a set of *NotAllEqual* constraints can be expressed by a *global cardinality constraint* (gcc) [25]. It corresponds now to a maximal clique in an hypergraph (where hyperarcs between tasks are *NotAllEquals*). However, it is still for us an open question that could significantly improve performances.

3.2 Subproblem(s)

The subproblem we consider here is to check whether a valid solution produced by the master problem is schedulable or not. Once assigned, tasks can be described as non-communicating or communicating. A task is said non-communicating if it do not send data, or if its receivers are on the same processor. A task is said communicating if one (at least) of its receivers is allocated to an other processor. For a non-communicating task, its deadline equals its period: $D_i = T_i$; for a communicating task, its deadline equal its period minus the maximal amount of time needed for transmitting the longest message: $D_i = T_i - TRT$ (it ensures a regular data refreshment).

A task is schedulable if its worst-case response time is lower than its deadline. A subproblem consists in checking this property for each task.

Classical calculation of worst-case response times and evaluation of the maximum transmission delay on a token ring network are given in the immediate following.

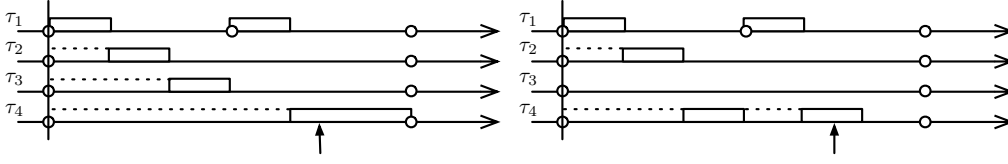


Figure 3. Illustration of a schedulability analysis. The task τ_4 does not meet its deadline. The subset $\{\tau_1, \tau_2, \tau_4\}$ is identified to explain the unschedulability of the system.

Worst-case response time. For independent tasks, it has been proved that the worst execution scenario for a task τ_i happens when it is released simultaneously with all the tasks which have a priority higher than τ_i . The worst-case response time for τ_i is given by (see, *e.g.*, [6]):

$$R_i = C_i + \sum_{\tau_j \in hp_i(A)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \quad (8)$$

with $hp_i(A)$ the set of tasks with priority higher than τ_i and located on the processor $A(\tau_i)$ for a given allocation A . Worst-case response time R_i is then easily computed by searching the fix-point in Eq. (8).

Transmission time on a token ring. The maximum transmission delay on a token ring network is bounded: let TRT (Token Rotation Time) denote the maximum duration for sending data on the network. An upper bound on this duration was proposed by Tindell in [30] and is computed by taking into account all the messages to be sent on the network:

$$TRT = \sum_{\substack{\{c_{ij} = (\tau_i, \tau_j) | \\ A(\tau_i) \neq A(\tau_j)\}}} \frac{d_{ij}}{\delta} \quad (9)$$

3.3 Cooperation between master and subproblem(s)

We now consider a valid allocation in which some tasks have been detected unschedulable. Our purpose is to explain why this allocation is unschedulable, and translate this into a new constraint for the master problem. For our problem, we separate two kinds of explanations.

Non-communicating tasks. The explanation for the unschedulability of a non-communicating task τ_i is the presence of tasks with higher priority on the same processor. For any other allocation with τ_i and $hp_i(A)$ on the same processor, it is sure that τ_i will still be unschedulable. So the master problem must be constrained so that all solutions where τ_i and $hp_i(A)$ are put together are not considered any further. This constraint corresponds to a *NotAllEqual*² on x :

$$NotAllEqual(x_j | \tau_j \in hp_i(A) \cup \{\tau_i\})$$

It is worth noticing that this constraint could be expressed as a linear combination of variables y . However, *NotAllEqual*(x_1, x_3, x_4) excludes the solutions that contain the tasks τ_1, τ_3, τ_4 gathered on any processor.

²A *NotAllEqual* on a set V of variables ensures that at least two variables among V take distinct values.

We can see that this constraint is not relevant. For example, in Fig. 3, the set $\{\tau_1, \tau_2, \tau_3, \tau_4\}$ is unschedulable. It explains the unschedulability but is not minimal in the sense that if we do not consider one task, the reduced set is still unschedulable. Here, the set $\{\tau_1, \tau_2, \tau_4\}$ is sufficient to justify the unschedulability.

In order to derive more precise explanations (to achieve a more relevant learning), a conflict detection algorithm, namely *QuickXplain* [17] (see Algorithm 1), has been used to determine a minimal (w.r.t. inclusion) set of involved tasks. A new function is defined, $R_i(X)$, the worst-case response time of τ_i with higher priority task in X :

$$R_i(X) = C_i + \sum_{\tau_j \in hp_i(A) \cap X} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \quad (10)$$

Algorithm 1 Minimal task set

QUICKXPLAINTASK(τ_i, A, D_i)

```

 $X := \emptyset$ 
 $\sigma_1, \dots, \sigma_{\#hp_i(A)}$  {an enumeration of  $hp_i(A)$ }
while  $R_i(X) \leq D_i$  do
   $k := 0$ 
   $Y := X$ 
  while  $R_i(Y) \leq D_i$   $k < \#hp_i(A)$  do
     $k := k + 1$ 
     $Y := Y \cup \{\sigma_k\}$ 
  end while
   $X := X \cup \{\sigma_k\}$ 
end while
return  $X$ 

```

To explain the unschedulability of a task, there could be more than one minimal task set, this is dependent of the order of enumeration of $hp_i(A)$ (the case study in Section 4 illustrates this).

Communicating tasks. The difficulty here is to avoid incriminating the whole system. If a communicating task τ_i is unschedulable, it is because of the tasks in $hp_i(A)$ and the message set $M(A)$ transmitted on the network. The translation of this information in terms of constraints yields:

$$NotAllEqual(x_j | \tau_j \in hp_i(A) \cup \{\tau_i\}) \vee \sum_{c_{kj} \in M(A)} w_{kj} < \#M(A)$$

$w_{ij} = 1$ if c_{ij} is transmitted on the network. So if a message of $M(A)$ is not transmitted, the sum becomes lower than $\#M(A)$ and the constraint is satisfied.

Again, we have used QUICKXPLAIN to refine the information. It has been appropriately changed to take into account messages (see Algorithm 2). $TRT(X)$ denotes the token rotation time for a set of messages X :

$$TRT(X) = \sum_{c_{ij} \in X} \frac{d_{ij}}{\delta} \quad (11)$$

Algorithm 2 Minimal message set

QUICKXPLAINCOM($\tau_i, A, R_i(T)$)

```
if  $R_i(T) < T_i$  then
   $X := \{c_{ij} | A(\tau_i) \neq A(\tau_j)\}$  {the minimal set must contain one message transmitted by  $\tau_i$ }
   $\sigma_1, \dots, \sigma_{\#M(A)-1}$  {an enumeration of  $M(A) - X$ }
else
   $X := \emptyset$ 
end if
while  $TRT(X) \leq T_i - R_i(T)$  do
   $k := 0$ 
   $Y := X$ 
  while  $TRT(Y) \leq T_i - R_i(T)k < \#M(A) - 1$  do
     $k := k + 1$ 
     $Y := Y \cup \{\sigma_k\}$ 
  end while
   $X := X \cup \{\sigma_k\}$ 
end while
return  $X$ 
```

Integration of nogoods in constraint programming solver. Dynamic integration of nogoods at any step of the search performed by the MAC (Maintaining Arc Consistency) algorithm of the constraint solver is based on the use of explanations. Explanations consist of a set of constraints and record enough information to justify any decision of the solver such as a reduction of domain or a contradiction. Dynamic addition/retraction of constraints are possible when explanations are maintained.

For example, the addition of a constraint at a leaf of the tree search will not lead to a standard backtracking from that leaf (which could be very inefficient as a wrong choice may exist at the beginning of the search because the constraint was not known at that time). Instead, the solver will jump (MAC-CBJ for conflict directed backjumping) to a node appearing in the explanation and therefore responsible for the contradiction raised by the new constraint. More complex and more efficient techniques such as MAC-DBT for dynamic backtracking exist to perform intelligent repair of the solution after the addition or retraction of a constraint.

4 Applying the method to an example

An example to illustrate the theory will be developed and will show how to perform the cooperation between master and subproblem. Figure 4 shows the characteristics of the considered hardware and software architectures.

The problem is constrained by :

- residence constraints:
 - C_1 : τ_3 must be allocated to p_1 or p_2 .
 - C_2 : τ_4 must be allocated to p_2 .
- co-residence constraint:
 - C_3 : τ_1 and τ_6 must be on the same processor.

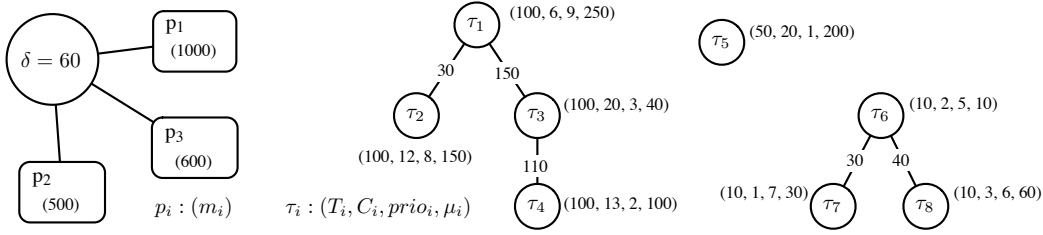


Figure 4. Hardware and software architecture characteristics

- exclusion constraints:
 - C_4 : τ_3 and τ_4 must be on different processors.
 - C_5 : τ_7 and τ_8 must be on different processors.

To start the resolution, the master problem begins to find a valid solution in respect of C_1, C_2, C_3, C_4 and C_5 . How the constraint programming solver finds a solution is not our purpose. A valid solution is found: $A = (p_1, p_1, p_1, p_2, p_1, p_1, p_2, p_3)$, where $A_i = A(\tau_i)$.

This solution is valid because all allocation and resource constraints are respected:

- $\mu_1 + \mu_2 + \mu_3 + \mu_5 + \mu_6 = 650 \leq p_1 = 1000$;
- $\mu_4 + \mu_7 = 130 \leq p_2 = 500$;
- $\mu_8 = 60 \leq p_3 = 600$;
- $\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} + \frac{C_5}{T_5} + \frac{C_6}{T_6} = 0.98 \leq 1$;
- $\frac{C_4}{T_4} + \frac{C_7}{T_7} = 0.23 \leq 1$;
- $\frac{C_8}{T_8} = 0.3 \leq 1$;
- $\frac{d_{34}}{T_3} + \frac{d_{67}}{T_6} + \frac{d_{68}}{T_6} = 8.1 \leq \delta = 60$.

The subproblem checks now the schedulability of the valid solution. The schedulability analysis is proceeded in four steps.

First step: determining TRT . Messages which are transmitted on the network are c_{34}, c_{67} and c_{68} , so (cf. eq (9)):

$$TRT = \frac{d_{34} + d_{67} + d_{68}}{\delta} = 3$$

Second step: computing task's deadline. If a task sends a message to a task on an other processor, the deadline equals its period less TRT , otherwise the deadline equals its period: $D_1 = D_2 = D_4 = T_1 = 100$, $D_5 = T_5 = 50$, $D_7 = D_8 = T_7 = 10$, $D_3 = T_3 - TRT = 97$, and $D_6 = T_6 - TRT = 7$.

Third step: analysing the schedulability. The worst-case response time for each task is obtained by application of equation (8): $R_1 = 6$, $R_2 = 18$, $R_3 = 48$, $R_4 = 15$, $R_5 = 72$, $R_6 = 20$, $R_7 = 1$ and $R_8 = 3$. τ_5 and τ_6 are found unschedulable.

Fourth step: explaining why an allocation is not schedulable. τ_5 do not communicate so the unschedulability is due only to tasks with higher priority on the same processor: $hp_5(A) = \{\tau_1, \tau_2, \tau_3, \tau_6\}$. This set is not minimal, since if one task is moved, τ_5 could stay unschedulable. By applying QUICKXPLAINTASK (see algorithm 1) with $hp_5(A)$ enumerated by increasing index, we find $\{\tau_2, \tau_3\}$ as minimal set. Consequently, the explanation of the unschedulability is translated on a global constraint:

$$C_6 : \text{NotAllEqual}\{x_5, x_2, x_3\}$$

If the enumeration of $hp_5(A)$ is τ_1, τ_3, τ_6 and τ_2 , the minimal set would be $\{\tau_1, \tau_3, \tau_6\}$.

For τ_6 , it is harder. By applying QUICKXPLAIN on tasks (algorithm 1) and next on messages (algorithm 2) we find:

$$C_7 : \text{NotAllEqual}\{x_6, x_1\} \vee \sum w_{67} + w_{34} < 2$$

And if QUICKXPLAIN is applied on messages and next on tasks, we find:

$$C_8 : \text{NotAllEqual}\{x_6, x_2\}$$

These new constraints are added to the master problem and a valid solution for this new problem is provided, and so on.

5 Search strategy

The time CP takes to find a solution depends on three parameters: the size of the search space, the ability to reduce that search space, and the *way it is searched*. In CP the search of solutions is managed by a *search strategy* that aims at quickly directing the search towards good solutions without loosing the completeness of CP. A search strategy is related to the order of choice of variable and value. It consists in defining an algorithm that specifies at each branching the variable to be chosen, together with its value.

To solve an allocation problem, different basic search strategies have been defined:

- **MINDOMAIN:** The first search strategy chooses the variable with the smallest domain size (number of possible values), *i.e.* the task (x_i) with the smallest number of processors where it could be assigned. There is no specific choice of the value. This search strategy is mainstream in CP and is a *first-fail strategy*. It is based on the principle of beginning where a failure is more likely to be detected. More parts of the search space are removed earlier, and important decisions are taken as soon as possible, rather than waiting for a extensive search to be completed.
- **MAXMEMORY:** The second search strategy is based on memory constraint. The variable (x_i) chosen for a branching is the task with the biggest memory need (μ_i) and its value is the processor where there is the biggest memory capacity while taking account of previous variable assignments.
- **MAXUTILIZATION:** The third strategy is similar to the previous one, where processor utilization is favored. The variable chosen (x_i) is the task with the biggest processor utilization (C_i/T_i) and its value is the processor where there is the smallest processor utilization at the current branching.
- **LEARNING:** The last search strategy uses the nogoods learned during the resolution. The rationale for this search strategy is based on the two following remarks:

- The more a task appears in nogoods, the more this task is constrained by schedulability;
- The smaller a nogood, the bigger its impact on the search space.

A nogood in its general form is defined by: $NotAllEqual(x_i) \vee \sum w_{ij} < C$. For a given nogood c , we denote $noe(c)$ the task set concerned by the $NotAllEqual$ and $cut(c)$ the set of messages concerned by the inequality. We denote NOG the set of nogoods learned. For a task τ_i a constraint criterion C_{c_i} is evaluated:

$$C_{c_i} = \sum_{\substack{c \in NOG \\ x_i \in noe(c)}} \frac{1}{\#noe(c)} + \sum_{\substack{c \in NOG \\ \exists j, w_{ij} \in cut(c) \vee w_{ji} \in cut(c)}} \frac{1}{\#cut(c)}$$

This criterion considers the presence of a task in each nogood and its impact. The LEARNING strategy chooses the variable with the biggest C_{c_i} . The value is chosen relatively to the tasks that have been already assigned at the current branching. A new criterion is defined for a couple of tasks to evaluate a constraint degree if these two tasks are allocated to the same processor:

$$C_{c_{i,j}} = \sum_{\substack{c \in NOG \\ (x_i, x_j) \in noe(c)^2}} \frac{1}{\#noe(c)} - \sum_{\substack{c \in NOG \\ w_{ij} \in cut(c) \vee w_{ji} \in cut(c)}} \frac{1}{\#cut(c)}$$

If x_i and x_j are in a $noe(c)$ then tasks τ_i and τ_j have not to be on the same processor, but if $w_{i,j}$ or $w_{j,i}$ is in a $cut(c)$ then τ_i and τ_j must be on the same processor to eliminate the message. For each processor p_k to which a task τ_i could be allocated, $\sum_{\forall j, A(\tau_j)=p_k} C_{c_{i,j}}$ is evaluated. Two strategies are defined. The first, named LEARNINGBEST, chooses the processor with the smallest value to conduct the search to a valid solution. And the second strategy, named LEARNINGFAIL, chooses the processor with the biggest value to yield an unschedulable solution and learn more nogoods on schedulability problem.

6 Experimental results

We chose to implement our approach with a tool named $\text{\textcircled{E}DIPE}$ [7], which is based on the CHOCO [20] constraint programming system and PALM [18], an explanation-based constraint programming system.

For the allocation problem, specific benchmarks are not available in the real-time research community. Experiments are usually done on didactic examples [30, 1] or randomly generated configurations [23, 22]. We opted for the latter. Our generator takes several parameters into account:

- n, m, mes : the numbers of tasks, of processors (experiments have been done on a fixed size: $n = 40$ and $m = 7$) and of messages;
- $\%_{global}$: the global utilization factor of processors;
- $\%_{mem}$: the memory over-capacity, *i.e.* the amount of additional memory available on processors with respect to the memory needs of all tasks;
- $\%_{res}$: the percentage of tasks involved in residence constraints;
- $\%_{co-res}$: the percentage of tasks involved in co-residence constraints;
- $\%_{exc}$: the percentage of tasks involved in exclusion constraints;

Alloc.	$\%_{mem}$	$\%_{res}$	$\%_{co-res}$	$\%_{exc}$	Sched.	$\%_{global}$	Mes.	mes/n	$\%_{msize}$
1	80	0	0	0	1	40	1	0.5	40
2	40	15	15	15	2	60	2	0.5	70
3	30	25	25	25	3	75	3	0.75	70
4	15	35	35	35	4	90	4	0.875	150

Table 1. Details on difficulty classes

- $\%_{msize}$: the size of messages is evaluated as a percentage of the period of the tasks exchanging it.

Task periods and priorities are randomly generated. Moreover, worst-case execution times are initially randomly chosen and evaluated again to respect: $\sum_{i=1}^n C_i/T_i = m\%_{global}$. The memory need of a task is proportional to its worst-case execution time. Memory capacities are randomly generated but must satisfy: $\sum_{k=1}^m m_k = (1 + \%_{mem}) \sum_{i=1}^n \mu_i$.

The number of tasks involved in allocation constraints is given by the parameters $\%_{res}$, $\%_{co-res}$, $\%_{exc}$. Tasks are randomly chosen and their number (involved in co-residence and exclusion constraints) can be set through specific levels. Several classes of problems have been defined depending on the difficulty of both allocation and schedulability problems. The difficulty of schedulability is evaluated using the global utilization factor $\%_{global}$ which varies from 40 to 90 %. Allocation difficulty is based on the number of tasks included in residence, co-residence and exclusion constraints ($\%_{res}$, $\%_{co-res}$, $\%_{exc}$). Moreover, the memory over-capacity, $\%_{mem}$ has a significant impact (a very low capacity can lead to solve a *packing* problem, sometimes very difficult). The presence of messages impacts on both problems and the difficulty has been characterized by the ratios mes/n and $\%_{msize}$. $\%_{msize}$ reflects the impact of messages on schedulability analysis by linking periods and message sizes.

Table 1 describes the parameters and difficulty classes of the considered problems. For instance, a class 2-1-4 indicates a problem with allocation difficulty in class 2, schedulability difficulty in class 1 and network difficulty in class 4. In the case of software architectures with no communication (the set of edges is empty: $C = \emptyset$), a class is specified by the only allocation and schedulability difficulties.

6.1 An experimental case study

The behaviour of our allocation algorithm for a problem of class 2-3 is outlined on Fig. 5. Computation time and learned constraints at each iteration between master and subproblems are plotted.

The master problem adapts the current solution to the nogoods thanks to its dynamic abilities and the learning process is very quick from the beginning. The number of learned constraints decreases until a hard satisfaction problem is formulated (*a-b* in Fig. 5). The master problem then is forced to evaluate an important amount of choices to provide a valid allocation (*b*). The process starts again with a quick learning of nogoods (*b-c*, *c-d*). This example shows the efficiency of using a learning method to guide the search for solution.

6.2 Study of search strategies on software architecture without messages

Experiments on search strategies have been conducted on software architecture with no communication. Figure 6 summarizes the results of our experiments. $\%_{RES}$ is the number of problem instances successfully solved (a schedulable solution has been found or the proof of inconsistency has been done) within the time limit of 10 minutes per instance. The data are obtained in average (on instances solved within the required time) on 100 instances per class of difficulty with a Pentium 4 (3 GHz).

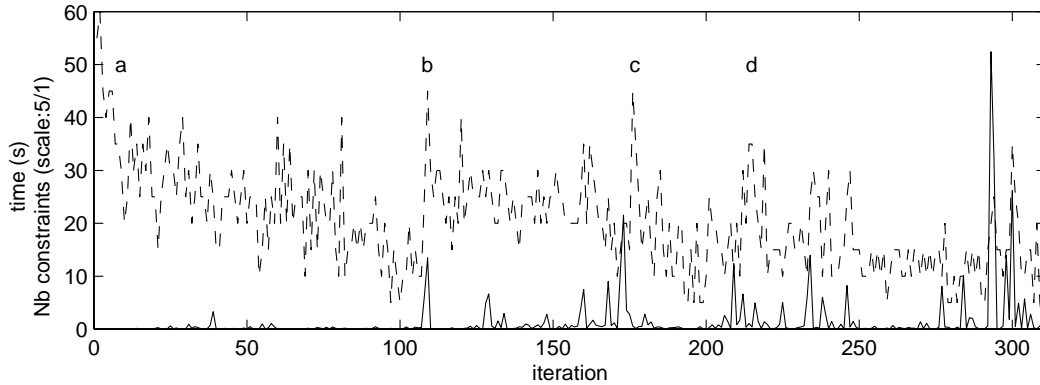


Figure 5. Behaviour of the allocation algorithm for a problem of class 2-3. Computation time (s) and number of nogoods (in dotlines with a scale of 5/1) inferred at each iteration are shown. (310 iterations, 1279 nogoods)

Class 1-4 is the hardest class of allocation problems. Without the allocation constraints, the initial search space is complete and all that concerns schedulability has to be learned. Moreover, these problems are close to inconsistency due to their scheduling hardness. Our approach seems to reach its limits in such a case. However, a more recent implementation increases the rate of solved problems to 52% with only few optimizations, which encourages us to go deeper into this way.

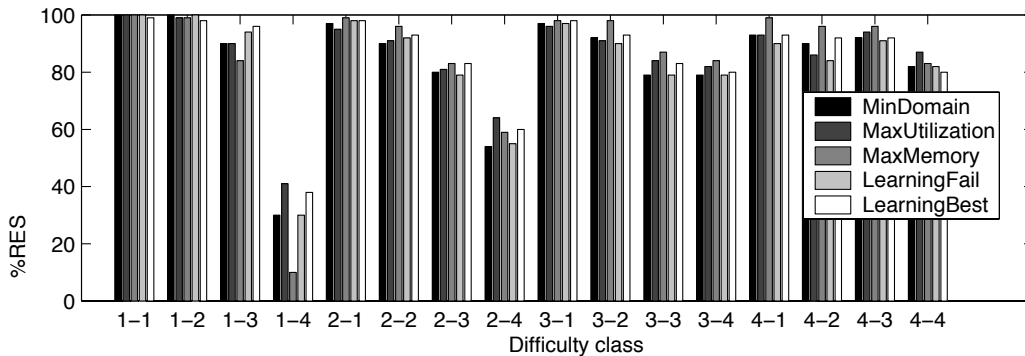


Figure 6. Average results on 100 randomly generated problem instances into various classes for the different search strategies

Experiments show that none of the mentioned strategies can solve all categories of problem:

- MINDOMAIN has a generic behavior for all classes of problem.
- MAXUTILIZATION produces good results for problems with a high schedulability difficulty and has mild performance for others.
- MAXMEMORY is excellent when the memory constraints become hard (allocation classes 3 and 4): it is then similar to a bin-packing algorithm. However, this strategy becomes inefficient when schedulability difficulty increases.

- LEARNING strategies have in average some good performances, in particular with the 1-4 class. LEARNINGBEST has a better behavior than LEARNINGFAIL. These good results are encouraging for our approach, because it shows that the learning from mistakes is effective.

These experiments show that all different constraints (allocation, resource and timing) have an importance in the hardness of a problem. Heuristics can be addressed to find a solution efficiently for a given problem, but it could be inefficient for an other class of problems. It seems interesting to focus on developing adaptive heuristics, that could adapt their behavior to difficulties of a stated problem instance.

6.3 Study of search strategies on software architectures with messages

Experiments on software architectures with messages have been conducted with all different search strategies. However, the observed behaviors are not different from the software architectures without messages. We just give some results with MINDOMAIN heuristics in Table 2. %VAL gives the percentage of schedulable solutions found (%RES - %VAL gives the percentage of problems found inconsistent). ITER is the number of iterations between master and subproblems, CPU is the computation time in seconds, NOG is the number of inferred nogoods. The data are obtained in average (on instances solved within the required time) on 100 instances.

One can see in Table 2 that when several hardness aspects compete on the problem, the difficulty increases (2-2-3 compared to 1-1-3). The presence of messages makes the problem much more complex for our approach because the independence between schedulability analysis on processors is lost and the nogoods learned on network are weak ones. Determining which tasks should be together, or not, is a tough issue, especially when a tight overall memory is combined to a difficult schedulability and many medium size messages. We hope to achieve better results with an efficient heuristic. An heuristic inspired from clustering (tasks that communicate together are regrouped) is developed to decrease network utilization. However, this heuristic is too specific and happens to be inefficient if resources or timing constraints are hard.

cat.	%RES	%VAL	ITER	CPU	NOG
2-1-1	98.0	94.0	34.8	24.7	103.8
2-1-2	93.0	90.0	40.1	18.6	126.9
2-1-3	56.0	49.0	92.0	106.6	249.2
2-1-4	36.0	19.0	58.9	113.5	192.8
2-2-1	82.0	65.0	59.0	72.7	191.2
2-2-2	74.0	61.0	55.3	60.5	217.8
2-2-3	38.0	24.0	77.6	142.1	275.7
2-3-4	10.0	0.0	3.6	7.9	49.5

Table 2. Average results on 100 randomly generated problem instances into various difficulty classes for the MINDOMAIN strategy

We experiment the technique on Tindell’s problem instance [30], solved thanks to simulated annealing. The Tindell instance falls into the 2-2-4 category except that it involves a lower over-capacity memory which increases the difficulty and much more residence constraints which simplify it. This instance exhibits a particular structure: the network plays a critical part and feasible solutions have an almost minimal network utilization. We were forced to specialize our generic approach on this particular point through the use of an allocation heuristic that tries to gather tasks that exchange messages.

One can obtain the Tindell's solution very quickly (5 iterations) if the network workload is minimized at each iteration.

6.4 Discussion on the approach

Guiding the search is the key element of a constraint approach for solving combinatorial problems. This is even more the case when the problem exhibits some specific structures that makes it tractable once they have been identified. A subset of variables can have a great influence over the rest of the whole problem and should be considered initially so as to, *e.g.*, simplify the problem as early as possible. The previous heuristics designed for task allocation attempt to take into account various aspects at the heart of the complexity of the problem.

Focusing on memory consumption can pay off if the packing problem of the current instance is hard; oppositely, using C_i/T_i is profitable to detect hard tasks when the schedulability problem is critical. Our next step is to design a complete generic search heuristic that can handle simultaneously all aspects of the problem. One current research track in Constraint Programming is based on the information provided by the propagation steps. The key issue is to establish the impact of each component or variable of the problem, from the solver point-of-view and not from the problem semantic (as we did before). The results obtained with the LEARNING heuristic encourage us to go further and incorporate network, allocation, memory as well as schedulability considerations into the whole search scheme, in a generic way. Propagation and explanations can both be used to analyze the combinatorial relationships between variables and their impact [24] on the whole problem (forgetting that this impact is due to either the memory or the worst-case execution time of the tasks).

7 Conclusion and future work

In this paper, an original approach to solve the allocation problem with constraint programming is presented. We use a decomposition method which is, up to a certain extent, built on a logic Benders decomposition for learning. The overall problem is split into a master problem for allocation and resource constraints and a subproblem for timing constraints, using the learning technique in an effort to combine the various issues into a solution that satisfies all constraints. Our approach offers a new answer to the problem of real-time task allocation. It opens new perspectives on integrating techniques coming from a broader horizon, other than optimization, within CP in a Benders scheme.

The first experiments have encouraged us to go a step further into the implementation of different heuristics to assist and speed up the search. These heuristics are efficient for specific problems but cannot adapt their behavior to different hardness. For the moment, we try to combine them into some adaptive approaches so as to deal with problems of various hardness. Tests are being conducted.

Future work includes (but is not limited to) the extensive comparison of our approach with other methods such as tabu search, simulated annealing, etc. We also believe that the extension of this work to other kinds of network protocols such as CAN, TTP (largely used in real-time embedded systems), as well as to software architectures with precedence constraints conveys serious interest.

References

- [1] P. Altenbernd and H. Hansson. The slack method: A new method for static allocation of hard real-time tasks. *Real-Time Systems*, 15(2):103–130, 1998.
- [2] R. Barták. Constraint programming: In pursuit of the holy grail. In *Proc. of the Week of Doctoral Students (WDS99)*, 1999.
- [3] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962.

- [4] G. Borriello and D. Miles. Task scheduling for real-time multiprocessor simulations. In *Proc. of 11th Workshop on RTOSS*, pages 70–73, 1994.
- [5] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *ACM*, 16(9):575–577, 1973.
- [6] G. C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications*, volume 23 of *Real-Time Systems*. Springer, 2nd edition, 2005.
- [7] H. Cambazard and P. Hladik. ŒDIPE.
- [8] H. Cambazard, P.-E. Hladik, A.-M. Déplanche, N. Jussien, and Y. Trinquet. Decomposition and learning for a hard real-time task allocating problem. In *proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP 2004)*, 2004.
- [9] H. Cambazard, P.-E. Hladik, A.-M. Dplanche, N. Jussien, and Y. Trinquet. Dcomposition et apprentissage pour un problème d’allocation de tches temps rel. *Journées Nationales sur la rsolution Pratique de Problmes NP-Complets*, 2004.
- [10] C. Ekelin. *An Optimization Framework for Scheduling of Embedded Real-Time Systems*. PhD thesis, Chalmers University of Technology, 2004.
- [11] E. Ferro, R. Cayssials, and J. Orozco. Tuning the cost function in a genetic/heuristic approach to the hard real-time multitask-multiprocessor assignment problem. In *Proc. of the 3th World Multiconference on Systemics Cybernetics and Informatics*, pages 575–577, 1999.
- [12] P.-E. Hladik. *Ordonnançabilité et placement des systèmes temps réel distribués, préemptifs et à priorités fixes*. PhD thesis, Université de Nantes, 2004.
- [13] P.-E. Hladik, H. Cambazard, A.-M. Déplanche, and N. Jussien. How to solve allocation problems with constraint programming. In *Proc. of the Work In Progress of the 17th Euromicro*, 2005.
- [14] J. Hooker, G. Ottosson, E. Thorsteinsson, and H. Kim. A scheme for unifying optimization and constraint satisfaction methods. *Knowledge Engineering Review*, 15(1):11–30, 2000.
- [15] J. N. Hooker and G. Ottosson. Logic-based benders decomposition. *Mathematical Programming*, 96:33–60, 2003.
- [16] V. Jain and I. E. Grossmann. Algorithms for hybrid milp/cp models for a class of optimization problems. *INFORMS Journal on Computing*, 13:258–276, 2001.
- [17] U. Junker. Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In *Proc. of the 8th International Joint Conference on Artificial Intelligence (IJCAI 01)*, 2001.
- [18] N. Jussien. The versatility of using explanations within constraint programming. Technical Report RR 03-04-INFO, École des Mines de Nantes, 2003.
- [19] N. Jussien and O. Lhomme. Local search with constraint propagation and conflict-based. *Artificial Intelligence*, 139(1):21–45, 2002.
- [20] F. Laburthe. Choco: implementing a cp kernel. In *proceedings of CP 00 Post Conference Workshop on Techniques for Implementing Constraint programming Systems*, 2000.
- [21] E. Lawler. Recent results in the theory of machine scheduling. *Mathematical Programming: The State of the Art*, 1983.
- [22] Y. Monnier, J.-P. Beauvais, and A.-M. Déplanche. A genetic algorithm for scheduling tasks in a real-time distributed system. In *Proc. of the 24th Euromicro Conference*, 1998.
- [23] K. Ramamritham. Allocation and scheduling of complex periodic tasks. In *proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS 1990)*, 1990.
- [24] P. Refalo. Impact-based search strategies for constraint programming. In *proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP 2004)*, 2004.
- [25] J. Régin. Generalized arc consistency for global cardinality constraint. In *Proc. of AAAI / IAAI*, pages 209–215, 1996.
- [26] F. E. Sandnes. A hybrid genetic algorithm applied to automatic parallel controller code generation. In *proceedings of the 8th Euromicro Workshop on Real-Time Systems*, pages 70–75, 1996.
- [27] J. Santos, E. Ferro, J. Orozco, and R. Cassials. A heuristic approach to multitask-multiprocessing assignment problem using the empty-slots method and rate monotonic scheduling. *Real-Time Systems*, 13(2):167–199, 1997.

- [28] R. Szymanek, F. Gruian, and K. Kuchcinski. Digital systems design using constraint logic programming. In *proceedings of The Practical Application of Constraint technologies and Logic Programming (PACLP 2000)*, 2000.
- [29] E. S. Thorsteinsson. Branch-and-check: A hybrid framework integrating mixed integer programming and constraint logic programming. *Lecture notes in Computer Science*, 2239, 2001.
- [30] K. W. Tindell, A. Burns, and A. Wellings. Allocating hard real-time tasks: An np-hard problem made easy. *Real-Time Systems*, 4(2):145–165, 1992.