

Guiding Architectural Design Process of Hard Real-Time Systems with Constraint Programming

Pierre-Emmanuel Hladik¹, Hadrien Cambazard², Anne-Marie Déplanche¹, and
Narendra Jussien²
{hladik,deplanche}@irccyn.ec-nantes.fr {hcambaza,jussien}@emn.fr

¹ IRCCyN, UMR CNRS 6597, 1 rue de la Noë, 44321 Nantes Cedex 3, France

² École des Mines de Nantes, LINA, 4 rue Alfred Kastler, 44307 Nantes Cedex 3,
France

Abstract. In this paper, we present an original approach (CPRTA for "Constraint Programming for solving Real-Time Allocation") based on constraint programming to solve an allocation problem of hard real-time tasks. This problem consists in assigning periodic tasks to distributed processors in the context of fixed priority preemptive scheduling. CPRTA is built on dynamic constraint programming together with a learning method to find a feasible processor allocation under constraints. It is a novel approach for solving these kinds of problems which produces in its current version (still perfectible) as acceptable performances as classical algorithms do. Some experimental results are given to show it. Moreover, CPRTA shows very interesting properties. It is complete — *i.e.*, if a problem has no solution — the algorithm is able to prove it, it is non-parametric — *i.e.*, it does not require specific initializations — thus allowing a large diversity of models to be easily considered. Finally, thanks to its capacity to explain failures, it offers attractive perspectives for guiding the architectural design process. A first attempt in that way is proposed hereafter.

1 Introduction

Real-time systems have applications in many industrial areas: telecommunication systems, automotive, aircraft, robotics, etc. Today's applications are becoming more and more complex, as much in their software part (an increasing number of concurrent tasks with various interaction schemes), as in their execution platform (many distributed processing units interconnected through specialized network(s)), and in their numerous functional and non-functional requirements too (timing, resource, power, etc. constraints). One of the main issues in the architectural design of such complex distributed applications is to define an allocation of tasks onto processors so as to meet all the specified requirements. In general, it is a difficult constraint satisfaction problem. Even if it has to be solved off-line most of the time, it needs efficient and adaptable search techniques which are able to be integrated into a more global design process. Furthermore,

it is desirable that those techniques return relevant information intended to help the designer who is faced with architectural choices. The "binary" result in particular (has a feasible allocation been found?: yes and here it is, or no, and that's all) which is usually returned by the search algorithm is not satisfactory in failure situations. The designer would expect some explanations justifying the failure and enabling him to revisit his design. Therefore, more sophisticated search techniques that would be able to collect some knowledge about the problem they solve are required. Here are the general objectives of the work we are conducting.

More precisely, the problem we are concerned with consists in assigning a set of periodic, preemptive tasks to distributed processors in the context of fixed priority scheduling, to respect schedulability but also to account for requirements related to memory capacity, co-residence, redundancy, and so on. We assume that the characteristics of tasks (execution time, priority, etc.) and the ones of the physical architecture (processors and network) are all known a priori — Only static real-time systems are here considered —.

Assigning a set of hard preemptive real-time tasks in a distributed system under allocation and resource constraints is known to be an NP-Hard problem. It has been tackled with ad-hoc approaches, simulated annealing and genetic algorithms. Recently, Ekelin [2] have used constraint programming to produce an assignment and a pre-runtime scheduling of distributed systems under optimization criteria. Even if their context is different from ours, their results have shown the ability of such an innovative approach to solve an allocation problem for embedded systems and have encourage us to go further.

Like numerous hybridation schemes [3], the way we are investigating uses the complementary strengths of constraint programming and optimization methods from operational research. In this paper, we present its principle and study its performances. It is a decomposition-based method (related to logic Benders-based decomposition [4]) which separates the allocation problem from the scheduling one: the allocation problem is solved by means of *dynamic constraint programming* tools, whereas the scheduling problem is treated with specific real-time schedulability analysis. The main idea is to "learn" from the schedulability analysis to re-model the allocation problem so as to reduce the search space. In that sense, we can compare this approach to a form of *learning from mistakes*. Lastly we underline that a fundamental property of this method is the completeness : when a problem has no solution, it is able to prove it (contrary to heuristic methods that are unable to decide).

The remainder of this paper is organized as follows. In section 2, we describe the problem. Section 3 is dedicated to the description of the master- and sub-problems, and the relations between them. The logical Benders decomposition scheme is briefly introduced and the links with our approach are put forward. Some experimental results are presented in Section 4. Section 5 shows how it is possible to set up a failure analysis able to aid the designer to review his plans. It is a first attempt that proves its feasibility and will need to go deeper. The paper ends with concluding remarks in Section 6.

2 The problem description

2.1 The real-time system architecture

The hard real-time system we consider can be modeled by a software architecture: the set of tasks, and a hardware architecture: the execution platform for the tasks, as represented in Fig. 1 and Fig. 2.

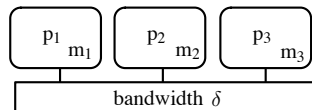


Fig. 1. Hardware architecture.

By *hardware architecture* we mean a set $\mathcal{P} = \{p_1, \dots, p_m\}$ of m processors with fixed memory capacity m_k and identical processing speed. Each processor schedules tasks assigned to it with a fixed priority strategy. It is a simple rule : a static priority is given to each task and at run-time, the ready task with the highest priority is put in the running state, preempting eventually a lower priority task. Those processors are fully connected through a communication medium with a bandwidth δ . In this paper, we look at a communication medium called a *CAN bus* which is currently used in a wide spectrum of real-time embedded systems. However any other communication network could be considered as far as its timing behaviour (including its protocol rules) is predictable. Thus the first experiments we have conducted addressed a token ring network.

CAN (Controller Area Network) [13] is both a protocol and physical network. CAN works as a broadcast bus meaning that all connected nodes will be able to read all messages sent on the bus. Each message has a unique identifier which is also used as the message priority. On each node waiting messages are queued. The bus makes sure that when a new message gets selected to transfer, the message with the highest priority, waiting on any connected node, will get transmitted first. When at least one bit of a message has started to be transferred it can't get preempted even though higher priority messages arrive. As a result, the CAN's behaviour will be seen subsequently as the one of a non preemptive fixed priority message scheduling.

The *software architecture* is modeled as a valued, oriented and acyclic graph $(\mathcal{T}, \mathcal{C})$. The set of nodes $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$ represents the tasks. A task in turn is a set of instructions which must be executed sequentially in the same processor. The set of edges $\mathcal{C} \subseteq \mathcal{T} \times \mathcal{T}$ refers to the data sent between tasks.

A task τ_i is defined through timing characteristics and resource needs: its period T_i (as a task is periodically activated ; the date of its first activation is free), its worst-case execution time without preemption C_i and its memory need μ_i . A priority $prio_i$ is given to each task. Task τ_j has priority over τ_i if and only if $prio_i < prio_j$. Edges $c_{ij} = (\tau_i, \tau_j) \in \mathcal{C}$ are weighted with its transmission time C_{ij} (the time it takes

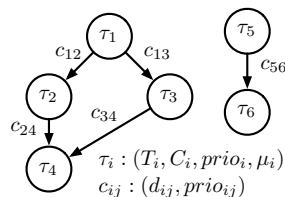


Fig. 2. Software architecture.

to transfer the message on the bus) together with a priority value $prio_{ij}$ (useful in the CAN context). Task priorities are assumed to be different. The same assumption is made on message priorities. In this model, we assume that communicating tasks have the same activation period. But we don't consider any precedence constraint between them : they are periodically activated in an independent way, and they read input data and write output data at the beginning and the end of their execution.

The underlying communication model is inspired from OSEK-COM specifications [14]. It is an uniform communication environment for automotive control unit application software. It defines common software communication interface and behaviour for internal communications (within an electronic control unit) and external ones (between networked vehicle nodes) which is independent of the communication

protocol used. It is the following. Tasks that are located on the same processor communicate through local memory sharing. Such a local communication cost is assumed to be zero. On the other hand, when two communicating tasks are assigned to two distinct processors, the data exchange needs the transmission of a message on the network. As a result, depending on the task allocation, an edge c_{ij} of the software architecture may give rise to two different equivalent schemes as illustrated in Fig. 3. In Fig. 3(b), M_{ij} inherits its period T_i from τ_i and its priority $prio_{ij}$ from c_{ij} .

Here, we are interested with the *periodic transmission mode* of OSEK-COM. In this mode data production and message transmission aren't synchronised : a producer task writes its output data into a local unqueued buffer from where a periodic protocol service reads it and sends it into a message. The building of protocol data units considered here is very simple : each data that has to be sent from a producer task τ_i to a consumer task τ_j in a distant way gives rise to its proper message M_{ij} . Moreover in this paper, for a sake of simplicity, the *asynchronous receiving mode* is preferred. It means that the release of a consumer task τ_j is strictly periodic and unrelated with the M_{ij} message arrival : when a node receives a message from the bus, its protocol records its data into a local unqueued buffer from where it can be read by the task τ_j . In [5] an extension of this work to a *synchronous receiving mode* is proposed in which a message reception notification activates the consumer task.

Therefore, from a scheduling point of view, messages on the bus are very similar to tasks on a processor. Like for tasks, each message M_{ij} is "activated" every T_i units of time and its (bus) priority is $prio_{ij}$.

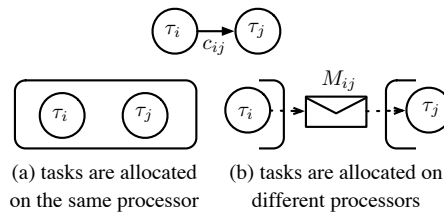


Fig. 3. Depending of the task allocation, a message exists or not.

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 the mapping A which respects the whole set of constraints described in the immediate below.

There are three classes of constraints the allocation problem must respect: timing, resource, and allocation constraints (refer to [6] for a complete description of these constraints).

- **Allocation constraints:** This set of constraints deal with the position (or relative position) of the tasks on the processors. Some tasks require specific processor characteristics to be executed (signal processor, compression processors, databases, etc.) and can only reside on a subset of the available processors. Others must not be put together on the same processor. Two sets of tasks may also have to be disjoint in any assignment.
- **Resource constraints:** The memory usage of a processor cannot exceed a fixed capacity.
- **Timing constraints:** A hard real-time system must respect all timing constraints to assure the security of the process. Temporal constraints define a schedulable allocation according to deadlines or due dates requirements.

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 to our problem is 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. They have proved quite effective in a wide range of applications (from planning and scheduling to finance — portfolio optimization — through biology) thanks to their main advantages: declarativity (the variables, domains, constraints description), genericity (it is not a problem dependent technique) and adaptability (to unexpected side constraints).

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

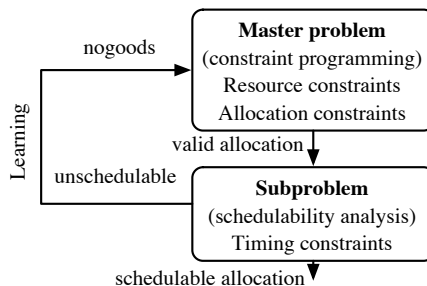


Fig. 4. Logic-based Benders decomposition to solve an allocation problem

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 [7].

We propose an approach inspired from methods used to integrate constraint programming into a logic-based Benders decomposition [3]. The allocation and resource constraints are considered on one side, and schedulability on the other (see Fig. 4). The master problem solved with constraint programming yields a valid allocation. The subproblem checks the schedulability of this allocation, eventually 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.

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 data 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.

One of the main objectives of the master problem is to solve efficiently the assignment part. It handles two kinds of constraints: allocation and resources. Due to the lack of space, we do not recall the close-form expressions for these constraints and refer to [6] for any further details.

3.2 Subproblem

The subproblem we consider here is to check whether a valid solution produced by the master problem is schedulable or not. A widely chosen approach for the schedulability analysis of a task set S is based on the following necessary and sufficient condition [8] : S is schedulable if and only if, for each task of S , its worst-case response time is less or equal to its relative deadline. Thus the subproblem solving leads us to compute worst-case response times for tasks on processors and for messages on the bus. According to the features of the considered task and message models, as well as the processor and bus scheduling algorithms, a "classical" computation can be used and its main results are given in the immediate following.

Task worst-case response time. For independent and periodic tasks with a pre-emptive fixed priority scheduling algorithm, it has been proven 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 $prio_i$. When D_i is (less or) equal to T_i , the worst-case response time for τ_i is given by [8]:

$$R_i = C_i + \sum_{\tau_j \in hp_i(A)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \quad (2)$$

where $hp_i(A)$ is the set of tasks with a priority higher than $prio_i$ and located on the processor $A(\tau_i)$ for a given allocation A and $\lceil x \rceil$ calculates the smallest integer $\geq x$. The summation gives us the number of times tasks with higher priority will execute before τ_i has completed. The worst-case response time R_i can be easily solved by looking for the fix-point of Eq. (2) in an iterative way.

Message worst-case response time. As mentioned earlier, message scheduling on the CAN bus can be viewed as a non-preemptive fixed priority scheduling strategy. Thus when doing a worst-case response time equation for a message, Eq. (2) has to be reused with some modifications. First it has to be changed so that a message only can be preempted during its first transmitted bit instead of its whole execution time. Second a blocking time, i.e. the largest time the message might be blocked by a lower priority message, must be added. The resulting worst-case response time equation for the CAN message M_{ij} is [9]:

$$R_{ij} = C_{ij} + L_{ij} \quad (3)$$

with

$$L_{ij} = \sum_{M' \in hp_{ij}(A)} \left\lceil \frac{L_{ij} + \tau_{bit}}{T'} \right\rceil C' + \max_{M' \in lp_{ij}(A)} \{C' - \tau_{bit}\} \quad (4)$$

where $hp_{ij}(A)$ (respectively $lp_{ij}(A)$) is the set of messages derived from the allocation A with a priority higher (respectively lower) than $prio_{ij}$; $\tau_{bit} = \frac{1}{\delta}$ is the transmission time for one bit (τ_{bit} is in relation with the bus bandwidth δ); C' is the worst-case transmission time for the message M' . The computation can be solved iteratively as Eq. (4).

3.3 Cooperation between master and subproblem(s)

We now consider a valid allocation (as the one the constraint programming solver may propose) in which some tasks are not schedulable. Our purpose is to explain why this allocation is unschedulable, and to translate this into a new constraint for the master problem.

Tasks. The explanation for the unschedulability of a task τ_i is the presence of tasks with higher priority on the same processor that interfere with τ_i . For any other allocation with τ_i and $hp_i(A)$ on the same processor, it is sure that τ_i will still be detected unschedulable. So the master problem must be constrained so that all solutions where τ_i and $hp_i(A)$ are together are not considered any further. This constraint corresponds to a *NotAllEqual* — A *NotAllEqual* on a set V of variables ensures that at least two variables among V take distinct values — on x :

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

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.

It is easy to see that this constraint is not totally relevant. For example, in Fig. 5, τ_4 that shares a processor with τ_1, τ_2 and τ_3 misses its deadline. Actually

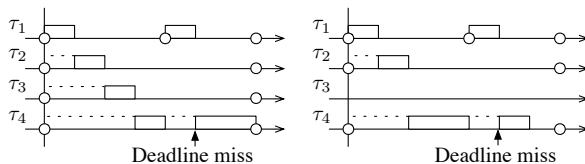


Fig. 5. Illustration of a schedulability analysis. The task τ_4 does not meet its deadline. The subset $\{\tau_1, \tau_2, \tau_4\}$ is identified to explain the unschedulability of the system.

the set $S_4(A) = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ explains the unschedulability but it is not minimal in the sense that if we remove one task from it, the set is still unschedulable. Here, the set $S_4(A)' = \{\tau_1, \tau_2, \tau_4\}$ 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* [10] (see algorithm [6]), has been used to determine a minimal (*w.r.t.* inclusion) set of involved tasks.

Messages. The reasoning is quite similar. If a message M_{ij} is found unschedulable, it is because of the messages in $hp_{ij}(A)$ and the longest message in $lp_{ij}(A)$. We denote $M_{ij}(A)$ their union together with $\{M_{ij}\}$. The translation of this information in term of constraint yields to:

$$\sum_{M_{ab} \in M_{ij}(A)} w_{ab} < \#M_{ij}(A)$$

where $\#X$ stands for the cardinality of X .

It is equivalent to a *NotAllEqual* constraint on a set of messages since to be met it requires that at least one message of $M_{ij}(A)$ "disappear" ($w_{ab} = 0$).

Like for tasks, so as to reduce the set of involved messages, QUICKXPLAIN has been implemented, using a similar adaptation of Eq. (3) and (4). It returns a minimal set of messages $M_{ij}(A)'$.

Appendix 1 presents an example allocation problem and shows how the algorithm is performed.

4 Experimental results

We have developed a dedicated tool named CEDIPE [16] that implements our solving approach (CPRTA). It is based on the CHOCO [15] constraint programming system and PALM [11], an explanation-based constraint programming system.

For the allocation problem, no specific benchmarks are available as a point of reference in the real-time community. Experiments are usually done on didactic examples [1] or randomly generated configurations [12]. We opted for this last solution. Our generator takes several parameters into account:

- n, m, mes : the number of tasks, processors (experiments have been done on fixed sizes: $n = 40$ and $m = 7$) and edges;

- $\%_{global}$: 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 included in residence constraints;
- $\%_{co-res}$: the percentage of tasks included in co-residence constraints;
- $\%_{exc}$: the percentage of tasks included in exclusion constraints;
- $\%_{msize}$: the size of a data is evaluated as a percentage of the period of the tasks exchanging it.

Task periods and priorities are randomly generated. Worst-case execution times are initially randomly chosen and evaluated again so as: $\sum_{i=1}^n C_i/T_i = m\%_{global}$. The memory need of a task is proportional to its worst-case execution time. Memory capacities are randomly generated while satisfying: $\sum_{k=1}^m m_k = (1 + \%_{mem}) \sum_{i=1}^n \mu_i$. For a sake of simplicity, only linear data communications between tasks are considered and the priority of an edge is inherited from the task producing it.

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 data exchanges impacts on both problems and the difficulty has been characterized by the ratios mes/n and $\%_{msize}$. $\%_{msize}$ expresses the impact of data exchanges on schedulability analysis by linking periods and message sizes.

Table 1 describes the parameters of each basic difficulty class. By combining them, categories of problems can be specified. For instance, a W-X-Y-Z category corresponds to problems with a memory difficulty in class W, an allocation difficulty in class X, a schedulability difficulty in class Y and a network difficulty in class Z.

Table 1. Details on difficulty classes

Mem.	$\%_{mem}$	Alloc.	$\%_{res}$	$\%_{co-res}$	$\%_{exc}$	Sched.	$\%_{global}$	Mes.	mes/n	$\%_{msize}$
1	60	1	0	0	0	1	40	1	0	0
2	30	2	15	15	15	2	60	2	0.5	70
3	10	3	33	33	33	3	90	3	0.875	150

4.1 Results

Table 2 summarizes some of the results of experiments with CPRTA. We do not give the results for all the intermediate classes of problems (like 1-1-1-1, 2-1-1-1, etc.) because they are easily solved and they do not exhibit a specific behaviour. $\%_{\text{RES}}$ gives the number of problem instances successfully solved (a schedulable solution has been found or it has been proven that none exists) within the time limit of 10 minutes per instance. $\%_{\text{VAL}}$ gives the percentage of schedulable solutions found (thus $\%_{\text{RES}} - \%_{\text{VAL}}$ gives the percentage of inconsistent problems). ITER is the number of iterations between the master problem and the subproblem. CPU is the mean computation time in seconds. NOG is the number of nogoods inferred from the subproblem. The data are obtained in average (on instances solved within the required time) on 100 instances (40 tasks, 7 processors) per class of difficulty with a Pentium 4 (3 GHz).

First, by examining the CPU column, we notice that CPRTA still remains very efficient in spite of its seeming complexity. Moreover, as measured by ITER and NOG, the cooperation between master and sub-problems is quite significant and the learning is of some importance.

The lines 1 to 5 in Table 2 show results for high difficulty classes without communications between tasks. The results in lines 1 to 3 are very good. They illustrate the basic ability of constraint programming to consider memory and allocation constraints. Lines 4 and 5 display some performances that are going down when the schedulability difficulty increases. Indeed, the schedulability constraints set is empty at the beginning of the search, and so, they have to be learnt from the subproblem. Furthermore, learning is only effective when a valid solution is produced by the master problem solver and as a consequence, it is not really integrated into the constraint programming algorithm. To improve CPRTA performances from this point of view, a new approach is now being developed that integrates schedulability analysis into the constraint programming algorithm so as not "to delay" its taking into account. A first basic implementation has been achieved without any optimization that makes us quite optimistic:

Table 2. Average results on 100 instances randomly generated into classes of problems

	cat.	$\%_{\text{RES}}$	$\%_{\text{VAL}}$	ITER	CPU	NOG
1	2-2-2-1	100.0	56.0	13.5	1.6	95.2
2	3-2-2-1	98.0	57.0	31.0	10.4	133.2
3	2-3-2-1	99.0	19.0	6.6	1.4	43.5
4	1-1-3-1	74.0	74.0	95.7	115.7	471.6
5	2-2-3-1	67.0	12.0	8.3	33.2	59.7
6	2-2-2-2	98.0	69.0	21.1	7.5	69.9
7	1-2-2-3	66.0	43.0	188.3	70.5	110.7
8	2-2-2-3	47.0	30.0	137.7	66.8	117.2

Table 3. Comparison between CPRTA and SA

cat.	SA		CPRTA	
	$\%_{\text{VAL}}$	CPU	$\%_{\text{VAL}}$	CPU
2-2-2-1	56.0	4.7	56.0	2.4
3-2-2-1	53.0	50.8	57.0	17.4
2-3-2-1	16.0	35.5	19.0	4.1
1-1-3-1	99.0	3.2	74.0	115.7
2-2-3-1	20.0	113.9	12.0	60.82
2-2-2-2	68.0	18.1	69.0	10.0
1-2-2-3	64.0	52.0	43.0	27.4
2-2-2-3	62.0	59.1	30.0	58.6

for the 1-1-3-1 category, $\%_{\text{RES}}$ grows to 82%, whereas it improves up to 79% for the 2-2-3-1 one.

The lines 6 to 8 deal with allocation problems where tasks may communicate. Once more, one can notice that when data exchanges increase (and thus message exchanges on the bus too), the CPRTA performances decrease. Reasons are the same as those of task schedulability: the more the messages are on the bus, the more their scheduling becomes difficult. Moreover, we have observed that nogoods inferred from message unschedulability are usually "weaker" (the search space cut is smaller) than the ones inferred from task unschedulability. Learning is then less efficient for this kind of problems. As for tasks, we hope to improve CPRTA by integrating the network schedulability as a global constraint into the master problem.

4.2 Comparison with simulated annealing

As to get comparative performances for CPRTA, a simulated annealing (SA) algorithm, inspired from [1], has been implemented. In [1] the energy function takes into account residence, exclusion and memory constraints as well as task deadline constraints. To be consistent with the CPRTA model, the schedulability of messages on the CAN bus and co-residence constraints have been integrated too. The implementation has been optimized so as to reduce computation times of this energy function.

SA is a heuristic method, it can only conclude on problems with a solution. Therefore, in Table 3 only CPRTA results for such problems are compared to SA. As seen on Table 3, except for problems for which CPRTA must be improved (see Section 4.1), CPRTA produces as satisfactory results as SA does, with better computation times. Introduction of schedulability as a constraint into the master problem should improve CPRTA and certainly increases its efficiency in a significant manner. Moreover, it should be pointed out that even if CPRTA is sometimes less efficient than SA, CPRTA solves on average more problems than SA does if we take into account problems without solution.

5 Explanations

In comparison with other search methods, using a constraint solver may help "intrinsically" to answer some classical queries when a problem is proven without solution such as: why does my problem have no solution ? Usually, when the domain of a variable of a CSP becomes empty (no value exists that will respect all the constraints on that variable), basic constraint programming systems notify the user that there is no solution. Nevertheless, thanks to the versatility of the explanation-based constraint approach we use, those relevant constraints, which explain the failure, are made available in addition [11].

Thus, in the case of an allocation problem for which no solution has been found, we analyse the set of constraints that is returned to explain the problem inconsistency. There can be many reasons to explain inconsistency. At the design level, we would like to be able to incriminate high level characteristics of the

system such as : allocation constraints, schedulability requirements of tasks, processors or network limitation. However, two points of view, based on the software or hardware architecture, can be adopted. We will first focus on the characteristics of the software architecture by analysing how each task is "responsible" for the failure. We will give there some insight on the way a critical task from the schedulability point of view can be identified. Each failure of the search process due to schedulability is analysed and transformed into a constraint criterion that encapsulates an accurate reason for this failure. The study of those criteria may lead to the guilty tasks. The rationale of this evaluation is based on the following remarks:

- The more a task appears within a nogood, the more this task has an impact on the schedulability inconsistency.
- The level of propagation performed by a nogood (either $NotAllEqual(x_i)$ or $\sum w_{ij} < B$), i.e its impact within the proof is strongly related to its size (the number of tasks it involves). "Small" $NotAllEqual$ have stronger impact.

In its general form, a constraint (learnt from a nogood) is defined by $NotAllEqual(x_i)$ or $\sum w_{ij} < B$ (see Section 3.3). We denote NAE the set of constraints in the $NotAllEqual$ form and SUM the set of constraints in the second form. For a task τ_i a constraint criterion \mathcal{C}_i is evaluated:

$$\mathcal{C}_i = \sum_{\substack{c \in \text{NAE} \\ x_i \in c}} \frac{1}{\#c} + \sum_{\substack{c \in \text{SUM} \\ \exists j, w_{ij} \in c \vee w_{ji} \in c}} \frac{1}{\#c}$$

This criterion considers the presence of a task in each constraint and its impact. Bigger \mathcal{C}_i is, bigger the impact of τ_i is on the inconsistency. By studying tasks with high \mathcal{C}_i and understanding why they have such an impact on the inconsistency (e.g. low priority allocation, too large processor utilization), it is possible to change some requirements (e.g. by adapting priorities, or choosing a different version for a task with an other period) and so to obtain a solution for the problem. Appendix 1 presents an example explanation on a problem without solution.

6 Conclusion and future work

In this paper, we present an original and complete approach (CPRTA) to solve a hard real-time allocation problem. We use a decomposition method which is built on a logic Benders decomposition scheme. The whole problem is split into a master problem handling allocation and resource constraints and a subproblem for timing constraints. A rich interaction between master and sub-problems is performed with the computation of minimal sets of unschedulable tasks and messages. It implements a kind of learning technique in an effort to combine the various issues into a solution that satisfies all constraints.

One important specificity of CPRTA is its completeness, *i.e.*, if a problem has no solution, the search algorithm is able to prove it. Moreover it offers good

potential means for building an analysis able to give an aid to the user in case of failure.

The results produced by our experiments encourage us to go a step further. Further works concern the inclusion of (task and message) schedulability analysis into the search process of the CP algorithms in the form of a global constraint. This should improve efficiency of CPRTA for hard-schedulability-constrained problems. Another interesting work deals with the explanation of failure. Our aim is to integrate into the design process an intelligent tool based on CPRTA ables to return pertinent explanations justifying the failure. We need to go deeper in that way and to try it out on some concrete cases.

References

1. Tindell, K., Burns, A., Wellings, A. : Allocating Hard Real-Time Tasks: An NP-Hard Problem Made Easy. *Real-Time Systems* (1992), 4(2), 145–165
2. Ekelin, C. : An Optimization Framework for Scheduling of Embedded Real-Time Systems. PhD, Chalmers University of Technology (2004)
3. Thorsteinsson, E. : Branch-and-check: A hybrid framework integrating mixed integer programming and constraint logic programming. *Lecture notes in Computer Science* (2001), Vol. 2239
4. Hooker, J., Ottoson, G. : Logic-based benders decomposition. *Mathematical Programming* (2003), Vol. 96, 33–60
5. Hladik, P.-E., Déplanche, A.-M. : Extension au réseau CAN des problèmes de placement. *IRCCyN* (2005), RI-2005-4
6. Hladik P.-E., Cambazard, H., Déplanche A.-M., Jussien, N. : Solving Allocation Problems of Hard Real-Time Systems with Dynamic Constraint Programming. *Proceeding of Real-Time and Network Systems (RTNS'06)* (2006)
7. Barták, R. : Constraint Programming: In Pursuit of the Holy Grail. *Proceeding of the Week of Doctoral Students (WDS99)* (1999)
8. Lehoczky, J. : Fixed priority scheduling of periodic task sets with arbitrary deadlines. *Proceedings of the 11th IEEE RTSS* (1990), 201–209
9. Tindell, K., Hansson, H., Wellings, A. : Analysis real-time communications: controller area network (CAN). *Proceedings of the 15th IEEE RTSS* (1994), 259–265
10. Junker, U. : QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. *Proceeding of the 8th IJCAI* (2001)
11. Jussien, N. : The versatility of using explanations within constraint programming. *École des Mines de Nantes* (2003), RR 03-04-INFO
12. Ramamritham, K. : Allocation and scheduling of complex periodic tasks. *Proceedings of the 10th ICDCS* (1990)
13. Bosch : CAN Specification version 2.0. (1991)
14. OSEK Group. : OSEK/VDX Communication version 3.0.2 (2003)
15. Choco. <http://choco.sourceforge.net/>
16. Cambazard, H., Hladik, P.-E. : EDIPE. <http://oedipe.rts-software.org/>

Appendix 1: Applying the method to an example

An example to illustrate the theory is developed hereafter. It will show how the cooperation between master- and sub-problems is performed and how explanations of failure are obtained.

Table 4 shows the characteristics of the software architecture (with 20 tasks). The entry " $x, y \rightarrow j$ " for the task τ_i indicates an edge c_{ij} with $C_{ij} = x$ and $prio_{ij} = y$. The hardware architecture is a set of 4 processors : $p_0 : m_0 = 102001$, $p_1 : m_1 = 280295$, $p_2 = m_2 = 36024$, and $p_3 : m_3 = 41617$.

The problem is constrained by :

- CC_1 : τ_0 must be allocated to p_0 or p_1 or p_2 .
- CC_2 : τ_{16} must be allocated to p_1 or p_2 .
- CC_3 : τ_{17} must be allocated to p_0 or p_3 .
- CC_4 : τ_7 , τ_{17} and τ_{19} must be on the same processor.
- CC_5 : τ_3 , τ_{11} and τ_{12} must be on different processors.

To start the resolution process, the solver for the master problem finds a valid solution in accordance with CC_1 , CC_2 , CC_3 , CC_4 and CC_5 . How the constraint programming solver finds such a solution is here out of our purpose. The valid solution it returns is:

- processor p_0 : $\tau_2, \tau_5, \tau_7, \tau_8, \tau_9, \tau_{17}, \tau_{19}$.
- processor p_1 : $\tau_4, \tau_6, \tau_{12}, \tau_{13}$.
- processor p_2 : $\tau_0, \tau_{11}, \tau_{14}, \tau_{15}, \tau_{16}$.
- processor p_3 : $\tau_1, \tau_3, \tau_{10}, \tau_{18}$.

One deduces that messages are $M_{0,13}$, $M_{1,8}$, $M_{4,9}$, $M_{8,18}$, $M_{10,15}$, and $M_{16,17}$.

It is easy to check it is a valid solution by considering allocation and resource constraints:

- $\mu_2 + \mu_5 + \mu_7 + \mu_8 + \mu_9 + \mu_{17} + \mu_{19} = 93383 \leq m_0$;
- $\mu_4 + \mu_6 + \mu_{12} + \mu_{13} = 278950 \leq m_1$;
- $\mu_0 + \mu_{11} + \mu_{14} + \mu_{15} + \mu_{16} = 151642 \leq m_2$;
- $\mu_1 + \mu_3 + \mu_{10} + \mu_{18} = 40761 \leq m_3$;

The subproblem checks now the schedulability of the valid solution. The schedulability analysis proceeds in three steps.

Table 4. Tasks and messages characteristics

τ_i	T_i	C_i	μ_i	$prio_i$	Message	τ_i	T_i	C_i	μ_i	$prio_i$	Message
τ_0	36000	2190	21243	1	600,1 \rightarrow 13	τ_{10}	12000	846	8206	4	200,2 \rightarrow 15
τ_1	2000	563	5855	6	500,3 \rightarrow 8	τ_{11}	36000	5836	60694	20	
τ_2	3000	207	2152	15	600,7 \rightarrow 7	τ_{12}	9000	2103	20399	10	
τ_3	8000	2187	21213	3		τ_{13}	36000	5535	54243	13	
τ_4	72000	17690	168055	7	300,4 \rightarrow 9	τ_{14}	18000	3905	41002	18	
τ_5	4000	667	6670	8	800,5 \rightarrow 19	τ_{15}	12000	1412	14402	5	
τ_6	12000	3662	36253	14		τ_{16}	6000	1416	14301	17	700,8 \rightarrow 17
τ_7	3000	269	2743	16		τ_{17}	6000	752	7369	19	
τ_8	2000	231	2263	12	100,6 \rightarrow 18	τ_{18}	2000	538	5487	11	
τ_9	72000	6161	59761	9		τ_{19}	4000	1281	12425	2	

First step: analysing the schedulability of tasks. The worst-case response time for each task is obtained by application of Eq. (2) and it is compared with its relative deadline. Here τ_5 , τ_{12} , τ_{16} and τ_{19} are found unschedulable.

Second step: analysing the schedulability of messages. The worst-case response time for each message is obtained by application of Eq. (3) and Eq. (4) and it is compared with its relative deadline. Here $M_{1,8}$ is found unschedulable.

Third step: explaining why this allocation is not schedulable. The unschedulability of τ_5 is due to the interference of higher priority tasks on the same processor: $hp_5 = \{\tau_2, \tau_7, \tau_8, \tau_9, \tau_{17}\}$. By applying QUICKXPLAINTASK with hp_5 , we find $\{\tau_5, \tau_9\}$ as minimal set. Consequently, the explanation of the unschedulability is translated into the new constraint:

$$CC_6 : \text{NotAllEqual}\{x_5, x_9\}$$

In the same way, by applying QUICKXPLAINTASK for τ_{12} , we find:

$$CC_7 : \text{NotAllEqual}\{x_6, x_{12}, x_{13}\}$$

for τ_{16} : $CC_8 : \text{NotAllEqual}\{x_{11}, x_{16}\}$ and for τ_{19} : $CC_9 : \text{NotAllEqual}\{x_9, x_{19}\}$.

For $M_{1,8}$, we have: $M_{1,8}(A) = \{M_{0,13}, M_{1,8}, M_{4,9}, M_{8,18}, M_{16,17}\}$. QUICKXPLAIN returns $\{M_{0,13}, M_{1,8}, M_{4,9}, M_{16,17}\}$ as $M_{1,8}(A)'$ the minimal set. So another constraint is created: $CC_{10} : w_{0,13} + w_{1,8} + w_{4,9} + w_{16,17} < 4$.

These new constraints CC_6 , CC_7 , CC_8 , CC_9 and CC_{10} are added to the master problem. They define a new problem for which it has to search for a valid solution and so on.

After 20 iterations between the master problem and the subproblem, this allocation problem is proven without solution. This results from 78 constraints learnt all along the solving process. This example has been solved using CEDIPE (see Section 4). On a computer with a G4 processor (800MHz), its computing time was 10.3 seconds.

We now focus on analysing how each task could be responsible of the failure. Table 5 gives C_i obtained on this example with CEDIPE [16]. Task τ_{19} has the biggest C_i . This task has a low priority together with a high processor utilization ($C_{19}/T_{19} = 0.32$). By just changing its priority to the highest one, and reusing CPRTA, we found a solution for this problem.

τ_i	C_i	τ_i	C_i	τ_i	C_i	τ_i	C_i
τ_{19}	6.33	τ_{13}	4.78	τ_2	3.22	τ_3	2.53
τ_{14}	5.98	τ_9	3.95	τ_1	2.85	τ_{16}	2.25
τ_{11}	5.98	τ_6	3.83	τ_{10}	2.77	τ_{18}	1.97
τ_5	5.42	τ_7	3.45	τ_4	2.65	τ_8	1.73
τ_{12}	5.42	τ_{15}	3.32	τ_{17}	2.55	τ_0	1.15

Table 5. Constraint criterions