Journal of Discrete Event Dynamic systems, Kluwer Academic Publishers, to be published

Off-line computation of real-time schedules using Petri nets

Emmanuel Grolleau, Annie Choquet-Geniet
LISI-ENSMA, Teleport 2, 1 avenue Clement Ader, BP 40109, 86961 Futuroscope Chasseneuil Cedex, France

Abstract. We present a methodology of off-line analysis of real-time systems, composed of periodic, synchronous or asynchronous precedence and resource constrained real-time tasks. As there is no polynomial optimal scheduling technique for such tasks sets, we present an enumerative method based on the construction of the state graph of a Petri net. The time is modeled by the Petri net through the earliest firing rule.

Keywords: real-time, scheduling, off-line, schedule length, Petri nets

1. Introduction

1.1. Real-time systems

Real time applications, most of the time dedicated to the process control, are characterised by the existence of temporal constraints, induced by the dynamics of the controlled process (Stankovic, 1988). The correctness of such applications does not only rely on the functional correctness, which we assume to be proven, but also on the temporal correctness, because an exact result provided out of time can be as bad as a false result.

For the sake of proving its correctness, the application is modeled by a set of periodic, aperiodic and sporadic tasks (Stankovic et al., 1998) which can communicate and share critical resources. This set is supplied by a more or less formal study of the specifications of the application. Each task is implemented using classical instructions of a high-level algorithmic language, and specific real-time primitives. These primitives insure the interactions between tasks and are provided by the real time kernel. The tasks are furthermore characterized by temporal parameters (Liu and Layland, 1973).

In this paper, we only consider periodic tasks, and we assume that non periodic tasks are carried out by a periodic server, or processed in the background (Buttazzo, 1997).

Each task is characterized by a period, a first release time, a processor load, which is its worst case computation time, and a relative deadline, which is the maximum delay from the release of an instance


© 2002 *Kluwer Academic Publishers. Printed in the Netherlands.*

of the task, to its completion. The time at which the instance has to be completed is its (absolute) deadline. If the first release times are all equal, the application is said synchronous, otherwise it is said asynchronous.

1.2. Real-time scheduling

We are concerned with the problem of the temporal validation as well as with the effective computation of feasible schedules, satisfying all the structural (critical resources, precedence constraints) and temporal constraints of the tasks. We assume that the applications run in a preemptive uniprocessor environment, and we deal only with deterministic real time applications: the temporal parameters are a priori known, since such applications are the only ones for which the respect of the temporal constraints can be guaranteed.

The temporal validation of the application relies on the opportunity of choosing an appropriate scheduling policy. Two approaches are usually considered in order to solve the schedulability problem:

1.2.1. *On-line method*

The scheduling policy is implemented within the scheduler. On-line scheduling algorithms are usually priority based: the processor is assigned to the pending task with the highest priority. The priority assignment is either empirical, or deduced from the temporal parameters of the tasks (Rate Monotonic (RM), Earliest Deadline (ED), and Least Laxity (LL) (Liu and Layland, 1973; Leung and Merrill, 1980; Mok and Dertouzos, 1978)). In restricted contexts these algorithms are optimal in the sense that if a feasible schedule exists, the schedule computed by the algorithm is feasible. Furthermore, if the tasks are synchronous, there exists polynomial time analytical criteria for testing feasibility. But if critical resources are used, the scheduling problem is NP-hard (Mok, 1983; Baruah et al., 1990; Leung and Merrill, 1980), and only sufficient feasibility conditions are still ahead (Kaiser, 1982; Sha et al., 1990; Chen and Lin, 1990; Baker, 1991). In this context, there is no optimal on-line algorithm (Mok, 1983).

1.2.2. *Off-line method*

A schedule, already computed, is stored in a table used by the dispatcher, avoiding the cost caused by the execution of the scheduling algorithm. The main benefit comes from the raise of the scheduling power compared to the on-line method: on-line scheduling algorithms make decision according to the instantaneous state of the system, meanwhile off-line methods are clairvoyant (they are based on a complete

knowledge of the task system). It follows that off-line methods are not reduced to work-conserving schedules (in a work-conserving schedule, a task never intentionally waits). This is mandatory in order for an algorithm to be optimal, since in some cases where critical resources are involved, all the pending tasks have to wait in order to avoid a future blockage of an urgent task on a resource. Another advantage of off-line scheduling is that it gets rid of the problem of the instability: the behaviour of the application becomes deterministic, since every action is planned within the schedule. Therefore, off-line methods have been developed for the sake of highly constrained task systems. The methods that can be found in the literature are usually exponential in time. Most of them rely on exhaustive branch-and-bound enumeration techniques, or on non deterministic algorithms. They all consider restricted context: either they consider a finite set of aperiodic tasks, or, what is equivalent, a set of synchronous periodic tasks (only the set of instances occurring within the time interval $[0..lcm(P_i)_{i=1..n}]$ has to be considered (Leung and Merrill, 1980)); or they consider independent tasks in a non preemtive environment.

(Zamorano et al., 1997) deals with systems of independent tasks, and proceeds by induction on the set of tasks: once all the valid schedules for a set of k tasks are computed, a (k + 1)th task is added to the system, and its possible locations within the previous schedules are considered.

(Xu and Parnas, 1990) presents the more general method, and deals with non periodic task systems, with only one instance of each shared resource, and precedence constraints. Initially, the ED schedule is computed. Then it is corrected if a task does not meet its deadline. For that purpose, tasks are sliced in several sub-tasks, and new precedence and preemption constraints are added. This process is iterated until either failure is established, or a valid schedule is computed.

Tree structured methods have been proposed in (Baker and Su, 1974; Bratley et al., 1975), they consider uni- and multiprocessor non pre-emptive environment, and independent tasks. In both uni and multiprocessor cases, their methods consist in computing all the permutations of the tasks, and in eliminating those which induce lateness. In the multiprocessor case, the placement is jointly considered. Analysis methods based on time or timed Petri nets (Ramchandani, 1974; Merlin and Farber, 1976) have also been developed. Their main goal is to establish diagnoses (Tsai et al., 1995; Menasche and Berthomieu, 1983; Berthomieu and Diaz, 1991).

Approximated methods of least cost have also been considered: genetic algorithms, simulated annealing...(Monnier et al., 1998) deal jointly with the scheduling and the placement problems, according to the Least Laxity policy, in a multiprocessor environment.

1.3. The proposed method

Our approach deals with highly coupled applications, dealing with critical resources as well as with reader/writer problems, with interleaved critical sections (generalising the assumption of overlapping critical sections, required for the use of the resource management protocols), and including communications, in a uniprocessor environment. Moreover, our method deals with synchronous as well as with asynchronous tasks. It is, as far as we know, the most general framework for studying uniprocessor schedulability for periodic task sets that can be found in the literature. Our method is optimal, since it consists in an exhaustive enumeration of the feasible schedules.

The method is implemented in a tool called PeNSMARTS and deals with realistic task systems. Our first goal is to provide a diagnostic of schedulability for the application, especially when analytical criteria cannot be used, and when the classical on-line algorithms fail. Our second goal is to help the designer for the choice of a valid schedule according to some quality criteria. For that sake, we propose the construction of schedules that are optimal with respect to some further criteria, that cannot be expressed with the only help of the temporal parameters. E.g., we can extract the feasible schedules that minimize the average response time for a subset of tasks, or that minimize the reaction rates of some tasks... An other point of interest concerns the location of the idle slots. This location does particularly matter when non periodic tasks have to be scheduled in the background, and when the designer uses an off-line schedule for periodic tasks, jointly with an on-line scheduling algorithm which handles non periodic tasks: since the schedules are known in advance, idle slots can be accurately managed by an on-line scheduler, particularly if those idle slots are thoroughly distributed over the pre-run-time schedule. Our method enables the user to locate them a priori, and to choose a specific arrangement, e.g. a distribution which is as uniform as possible, or where idle slots occur as late as possible (priority driven or work conserving algorithms). Let us finally notice that, even if a classical on-line algorithm produces a valid schedule, it is not necessarily the best one according to the considered criteria.

The used method relies on modeling the application by a Petri net. The set of tasks is modeled by a classical autonomous Petri net (Petri, 1962; Peterson, 1981; Murata, 1989). The power of this net is increased, on the one hand by a behaviour which obeys the earliest firing rule (Starke, 1990), which allows the implementation of a logical clock; and on the other hand by the adjunction of terminal markings, which take relative deadlines into account. From this net, we can build the marking

graph, which represents exactly the set of feasible schedules. Then it is possible to extract optimal schedules according to some performance criteria of the application. This extraction is performed by the use of shortest paths algorithms in the marking graph, labeled according to the desired criteria. Up to now, we only know an upper bound of the size of the marking graph, which is exponential in the number of tasks. But this bound is very pessimistic, since it takes into account neither the interactions between tasks, nor the deadlines, which both greatly reduce the size of the graph. Furthermore, in order to decrease the size of the graph, we use some heuristics, as e.g. the limitation of the number of context switches, by forbidding multiple pre-emptions between independent parts of tasks. Furthermore, we reduce the time used for the construction of the state graph with an efficient cut of states leading to non feasible schedule as soon as it can be detected. These heuristics are described in Sec. 4.3 and allow our tool implementing this method to cope with realistic task systems (our prototype has successfully scheduled task systems with up to 20 complex tasks).

The sequel of this paper is organized as follows: in section 2, we introduce the task model, including the idle task, which is of great importance for the completeness of the method, in section 3, we present the modeling step of our methodology, in section 4, we present the analysis step, heuristics used in order to reduce the state graph, and we show how to get optimal feasible schedules. Finally, in section 5 we present a case study through the use of our tool PeNSMARTS, which implements the presented method.

2. The task model

2.1. The temporal model

The application is decomposed into a set of tasks $\{\tau_i : i = 1..n\}$, supposed to be periodic (Stankovic et al., 1998), coming from a preliminary design based on the study of the specifications. The temporal model mostly used in real-time scheduling theory is an extension of the model of (Liu and Layland, 1973) (see fig. 1) where each task τ_i is characterized by four parameters:

- r_i: first release date of τ_i
- C_i: run-time of τ_i (or worst case execution time), which is always supposed to be predictable
- D_i: relative deadline of τ_i, the maximal time elapsed between the release of an instance of τ_i and its completion

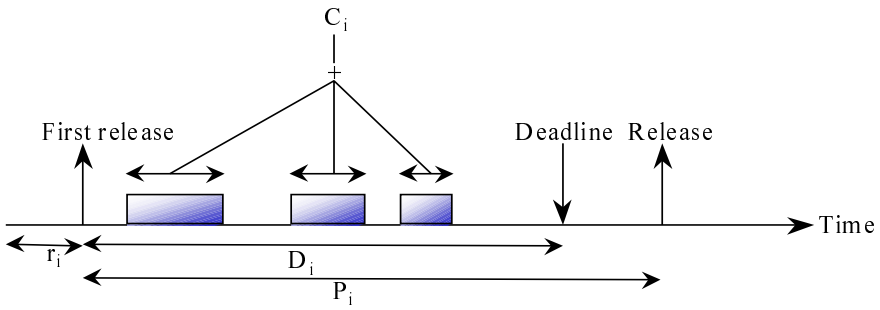

Figure 1. Temporal parameters of a periodic real-time task $\tau_i\langle r_i, C_i, D_i, P_i\rangle$.

– P_i: release period of τ_i. τ_i is released at the dates $r_i + kP_i$ for $k \in IN$

In the sequel, a task is denoted $\tau_i\langle r_i, C_i, D_i, P_i\rangle$ and a task system $S = \{\tau_i\langle r_i, C_i, D_i, P_i\rangle\}_{i=1..n}$. Without loss of generality, we suppose $min_{i=1..n}\{r_i\} = 0$, and the latest release date is denoted $r = max_{i=1..n}\{r_i\}$. A task τ_i is said synchronous if $r_i = 0$, and asynchronous if $r_i > 0$ for some i. Finally, we call the hyperperiod of a task system S the least common multiple of the periods of the tasks: $P = LCM_{i=1..n}\{P_i\}$.

Moreover, we assume that tasks can communicate: the communications are modeled by one to one mailboxes, under the assumption that the emission rate to a mailbox is equal to the reception rate. Tasks also use critical resources, that can be accessed either in write mode, or in read-only mode (i.e. several read-only accesses can occur simultaneously). Moreover, some parts of the tasks can be non-preemptible. In the sequel, we consider tasks as sequences composed of sequential blocks (written in a high level language), the worst case duration of each block is known, and of real-time primitives (lock/unlock resources, emission/reception of messages), of which the durations are included within the duration of the preceding block.

Example 1. `A task sending a message and using a resource`

```
Task τ1 is
  ri=3;
  Di=8;
  Pi=12;
Begin
  Bloc1 duration 2;
  Send(mailboxj,a_message);
  Bloc2 duration 1;
  Lock(a_ressource);
  Bloc3 duration 1;
```

```
  Unlock(a_ressource);
End;
```

2.2. The idle task

A characteristic measure of the task set is its processor load, $U = \sum_{i=1}^{n} \frac{C_i}{P_i}$. It represents the utilization rate of the processor by the whole application. A trivial necessary feasibility condition on a uniprocessor system is that $U \leq 1$.

If $U < 1$, it can be shown that the processor remains idle $P(1-U)$ units of time each hyperperiod. These idle times, called cyclic idle times, since they arise regularly, are collected within a new task, called idle task, $\tau_0 = \langle r_0, P(1-U), P, P \rangle$. We will come back later on its release time r_0. Notice that this task is purely fictitious, since it does not appear effectively within the design of the application. Note that this task can be used to allow execution of aperiodic or sporadic. It can be of great interest since if the idle slots are periodically distributed in the schedule, it can be guaranteed that an urgent aperiodic or sporadic task never has to wait more than a delay related to the period of the idle slots.

The benefits provided by the consideration of this idle task are multiple: first, one needs to consider only a task system with a processor load equal to 1, which in many ways is much easier to deal with. For the purpose of on-line scheduling, such a task is of little interest, since it will always have the lowest priority. But, if the study can be enlarged to non work conserving scheduling policies (which can be useful in order to avoid deadlock situations or some priority inversions), this task becomes necessary, and it can be processed even if other tasks are pending. Furthermore, the explicit modeling of the cyclic idle slots enables the control of their number within schedules, and guarantees that they will occur in proper amount.

If the tasks are synchronous, no other idle slots can occur. On the other hand, if some tasks are asynchronous, a finite number of idle slots, called acyclic idle slots (since they do not appear regularly), may appear (Grolleau, 1999). They are induced by the system loading of the task system. We assume that these acyclic idle slots are processed only when no task (including the idle task) is pending. As evidence of the possible appearance of acyclic idle slots, let us consider the following system, constituted of three independent tasks: $S = \{\tau_1\langle r_1 = 0, C_1 = 1, D_1 = P_1 = 4\rangle, \tau_2\langle r_2 = 1, C_2 = 3, D_3 = P_3 = 6\rangle, \tau_3\langle r_3 = 3, C_3 = 1, D_3 = P_3 = 4\rangle\}$. The processor load of S is equal to $\frac{1}{4} + \frac{3}{6} + \frac{1}{4} = 1$, thus no cyclic idle slots will occur. The fig. 2 shows the schedule of S according to the ED policy.

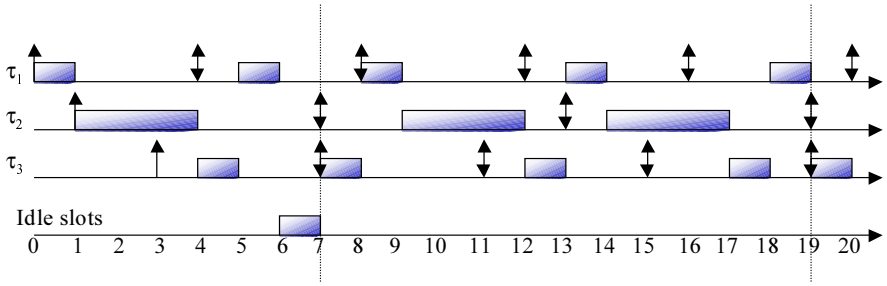

Figure 2. EDF schedule for the task system $S = \{\ \tau_1\langle r_1 = 0, C_1 = 1, D_1 = P_1 = 4\rangle, \tau_2\langle r_2 = 1, C_2 = 3, D_3 = P_3 = 6\rangle,\ \tau_3\langle r_3 = 3, C_3 = 1, D_3 = P_3 = 4\rangle\ \}$.

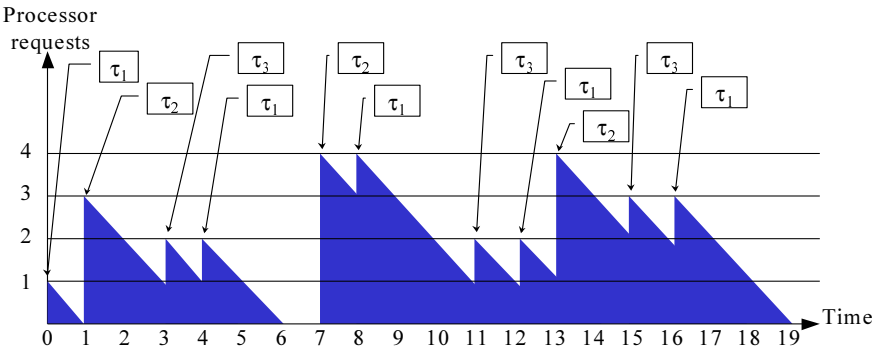

Figure 3. The process request diagram of the application depicted on fig. 2.

We can notice the occurrence of a unique idle slot, although the processor load is $U = 1$. No other idle slot appears within the schedule, this one is the only one. Furthermore, we can notice that the system is in the same state at times 7 and 19. The part between those two dates will be infinitely iterated: it is the cyclic part of the schedule. The fig. 3 shows the processor request diagram of the application. It gives, at each time, the total remaining computing time units. This request diagram is equivalent whatever the work-conserving scheduling policy is since there is no distinction between the tasks: if there is at least one pending task, then one unit of time of a task is computed, else there is an idle slot.

The results concerning the acyclic idle slots (Grolleau, 1999) are summarized hereafter:

- the number n_c of acyclic idle slots is bounded, and can be deduced from the processor request diagram.

- the last acyclic idle slot occurs before the time $r + P$.

- let t_c be the occurrence date of the last acyclic idle slot, (with by convention t_c = -1 if there is no acyclic idle slot at all), the system is in the same state at the dates $t_c + 1$ and $t_c + P + 1$. This means that after the last acyclic idle slot, the system behaves cyclically over the hyperperiod.
- the date t_c does not depend on the scheduling policy (provided acyclic idle slots are processed as late as possible)
- without loss of generality, the first release time of the (cyclic) idle task τ_0 can be chosen such that the application enters its cyclic part as soon as possible : $r_0 = t_c + 1$.

The proof is based on the fact that the positions of the acyclic idle slots are independent from the work-conserving scheduling algorithm. Then we show that after the last acyclic idle slot, the system behaves cyclically thanks to a consideration of the processor requests. Finally, we bound the date of the last acyclic idle slot.

The acyclic idle slots are collected within a further non periodic task, the acyclic task $\tau_c = \langle r_c = 0, C_c = n_c, D_c = t_c \rangle$.

3. Petri net model of allowable schedules

This section introduces the model used to schedule real-time task systems. The model is a constrained marking colored Petri net (Valk and Vidal-Naquet, 1981), under the earliest firing rule (EFR). Sec. 3.1.2 shows that this rule can model time progress in the behavior of the Petri net. The expressiveness of Petri nets under the maximal firing rule is equivalent to the expressiveness of T-timed Petri nets under the earliest firing rule (Starke, 1990).

The modeled applications are composed of a set of tasks as depicted in example 1, and of both idle and acyclic tasks. In this way, the processor is fully loaded. As the interactions of the tasks are complex, the expressiveness of a Petri net (PN) model is appropriate. Furthermore, the operational semantic involving the time must be associated to the PN modeling the task system. We assume that a quantum of preemption is defined: a running task can be preempted by any pending task at each preemption point. Therefore, the scale used to express the time is given in terms of preemption points, and a preemptible task can be preempted each time unit.

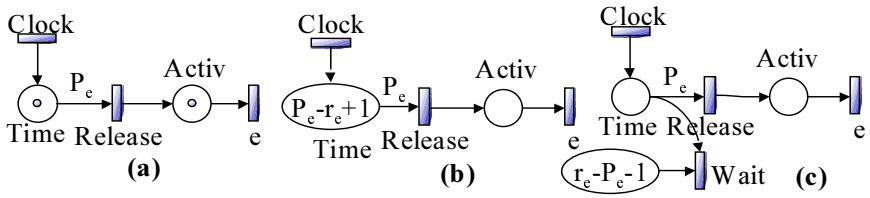

Figure 4. (a) Petri net model for a periodic action e (b) e is delayed by $r_e \leq P_e + 1$ (c) e is delayed by $r_e > P_e + 1$.

3.1. Modeling the time

3.1.1. *The earliest firing rule (EFR)*

A PN obeys the EFR if each time a firing of transitions occurs, a maximal set of simultaneously enabled transitions fires: let I be the set of enabled transitions for a marking M (note that I is not a multiset, since a transition t cannot fire more than once in the same firing step). A maximal set of simultaneously enabled transitions $I' \subseteq I$ is defined by: $\forall t \in I \backslash I'$, t is in conflict with some transitions of I' (i.e. the transitions of I' are not in conflict with each other, and any other enabled transition is in conflict with some transition of I').

A PN with EFR behaves exactly like a timed PN at maximal speed where all the durations of the transitions are one, nevertheless, their implementation is easier since the residual firing vector does not have to be taken into account. Furthermore, the expressiveness of PN with the EFR is equivalent to the expressiveness of a Turing machine, therefore it can model the whole set of interactions between tasks.

3.1.2. *Modeling the periodic actions*

Fig. 4.a presents the basic PN component which models the periodicity of an elementary action e, occurring with period P_e (with $P_e > 1$).

The transition *Clock* is a source transition which is always enabled without any conflict. It follows that each time a maximal set of transitions fires, the transition *Clock* fires, producing a token in the place *Time* which acts like a local time marker. Thus we assimilate the firing of *Clock* to a time unit in our scale of time. As soon as Time holds P_e tokens, the transition *Release* fires (simultaneously with *Clock* which is always fired), so that a token is produced in the place *Activ* each P_e units of time, i.e. at the dates $kP_e - 1$ for $k \in IN$. Finally, as soon as *Activ* contains a token, the transition e fires, the action e occurs at the dates kP_e. The same principle is used to model a delayed periodic action (first release date greater than 0). If the first release date of the action e is $r_e \leq P_e + 1$, the model differs from the previous model only by the initial marking (see fig. 4.b), and if $r_e > P_e + 1$, a place and a

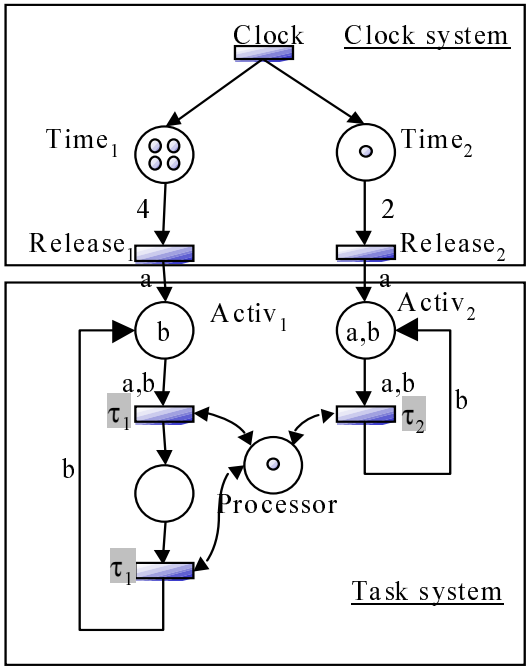

Figure 5. Modeling of an application $S = \{\tau_1\langle r_1 = 1, C_1 = 2, D_1 = 3, P_1 = 4\rangle, \tau_2\langle r_2 = 0, C_2 = 1, D_2 = 2, P_2 = 2\rangle, \}$ by means of a Place/transition net composed of two parts: a net which models the task system and a temporal structure which includes a global real time clock RTC and clocks local to the tasks ($Time_i$ and $Release_i$)). *In the following the arcs from/to the processor are not drawn in order to increase the readability of the Petri nets.*

transition are added to the model (see fig. 4.c). On fig. 4.c, transition $Wait$ fires during the $r_e - P_e - 1$ first units of time. Then the place $Time$ needs P_e units of time to enable $Release$, which fires at the date $r_e - 1$, so e fires for the first time at the date r_e.

3.2. The complete model

The model which we propose includes two parts, both modeled by a place/transition net (see fig. 5): the task system, which is obtained through a classical modeling of the functional description of the tasks, and the clock system, which models time. The periods of the tasks are modeled within the clock system as explained in sec. 3.1.2, the release times are taken into account by the initial marking, the relative deadlines are dealt using terminal markings and finally, if the EFR is used, the computations times can be deduced from the task system.

3.2.1. *The clock system*
The clock system models the behavior of the time like the clock system of section 3.1.2: there is a unique $Clock$ transition counting the time in each place $time_i$ (one for each task τ_i) which acts as a local clock used to release periodically the related task (see Fig. 5). The body of the tasks can be complex and is described in the next section, but remark that the transition $Release_i$ produces a token of color a (for activation) in the place $Activ_i$, and that when the task completes its execution, a b-token is produced in $Activ_i$. This token expresses the fact that we consider non reentrant tasks: an instance of a task cannot start if the previous one is not completed.

In addition, some constraints on the allowed markings are introduced in order to model the relative deadlines of the tasks. We thus introduce a terminal set defined by:

- $\forall i \in 1..n, M(Time_i) > D_i \Rightarrow M(Activ_i) = b$
- $\forall i \in 1..n, M(Time_i) = 1 \Rightarrow M(Activ_i) = a + b$ or $M(Activ_i) = b$

The first equation, used when relative deadlines are less than the periods ($D_i < P_i$), expresses the fact that an instance must be completed when its deadline occurs: no token a remains in the place $Activ_i$, thus the instance has started its execution, and there is a token b, which says that the execution has run until completion. The second equation concerns tasks with deadline equal to the periods ($D_i = P_i$). Just after the task has been released, the place $Activ_i$ must contain one single token a, indicating that current instance has started execution, and one token b indicating that the previous instance has been completed. But it is possible that $Time_i$ contains one token when the task has never been released (it is possible when $r_i > 0$), in this case, $M(Activ_i) = b$.

3.2.2. *The task system*
Each task body of duration C_i of the task system is modeled by a series of at most C_i transitions (it can be less than C_i since a series of $m > 3$ transitions can be compressed into a series of 3 transitions, see Fig. 6). Each transition of the task system uses a common resource: the processor. Accordingly, only one transition of the task system can be fired at a time. Interactions between the tasks are modeled in a classical way by mailbox places, and resource places (in exclusion mode or in read/write mode). The beginning of a non-preemptible part is modeled by the fact that the first transition of the part locks the processor but does not unlock it; the processor is liberated by the last transition of the non-preemptible part.

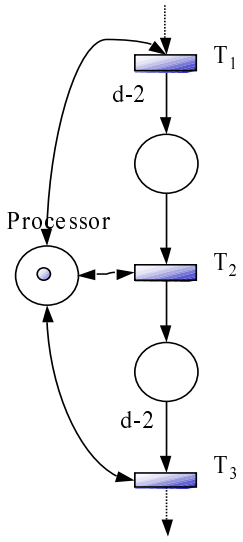

Figure 6. Modeling $d \geq 3$ units of time with a series of 3 transitions.

3.2.3. *Initial state and language*

We complete the definition of the model by the description of its initial marking.

If a task τ_i is synchronous ($r_i = 0$), then the place $Activ_i$ holds an a-token because τ_i is active at the beginning plus a b-token in order to fit the marking constraints. The place $Time_i$ contains one token, therefore τ_i is reactivated by the production of an a-token in $Activ_i$ at the date P_i.

If a task τ_j is asynchronous ($r_j > 0$), the place $Activ_j$ contains only a b-token because the task is not released initially. The marking of the local clock $Time_j$ is $P_j - r_j + 1$ in order to release τ_j at the date r_j. Note that if $P_j - r_j + 1 < 0$, then it is possible to add a place used to delay τ_j after $P_j + 1$ (see Fig. 4.c).

Since we focus on the generation of valid schedules, each transition of τ_i is labeled with τ_i, and the other transitions are labeled with the empty word.

The language of the Petri net model where all the reachable markings meet the terminal constraints, and where each word is infinite, is given by the enumeration of the state graph of the Petri net. This language is called the center of the terminal language (Valk and Vidal-Naquet, 1981). Since a marking meeting the terminal constraints corresponds to a state of the task system where no temporal constraint is violated, the center of the terminal language of the Petri net corresponds to feasible schedules of the modeled task system.

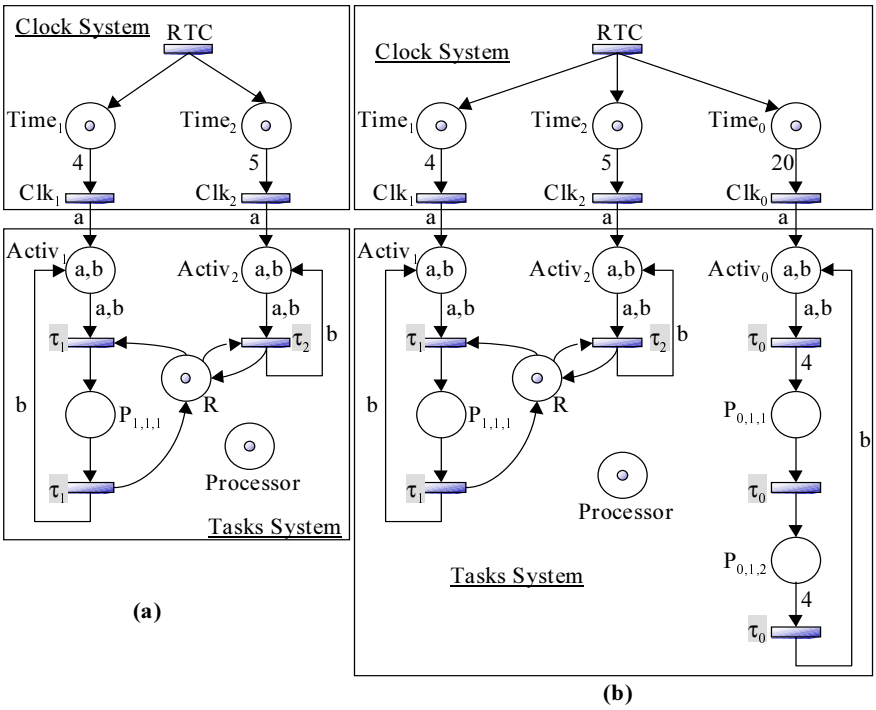

Figure 7. (a) A Petri net whose language contains the work-conserving schedules of $S = \{\tau_1\langle 0, 2, 4, 4\rangle, \tau_2\langle 0, 1, 1, 5\rangle\}$.(b) The idle task has been added to the model. Then the language contains the whole set of feasible schedules thanks to the adjunction of an idle task $\tau_0\langle 0, P(1-U) = 6, P = 20, P = 20\rangle$. Note that there is no work-conserving feasible schedule for S, therefore, the Petri net on (a) does not produce any schedule. Finally, note on the modelization of τ_0 on (b) how a series of 3 transitions can model a long duration.

Recall that the firing rule is the EFR, therefore the language corresponds only to the whole set of feasible work-conserving sequences. But since an idle task including the cyclic idle slots is always added to the task system, the language of the Petri net computes the whole set of non work-conserving sequences too since the idle task may be processed even when there are other pending tasks. This is a crucial point since if some tasks share resources, the work-conserving sequences are not optimal. Fig. 7a presents a Petri net whose language consists only of work-conserving schedules, and Fig. 7b presents the Petri net with the idle task, whose language is the whole set of feasible schedules (see Fig. 8).

Finally, in order for the model to generate schedules where the acyclic idle slots are not necessarily placed as late as possible, a place and a transition modeling the acyclic idle task (see Sec. 2.2) is intro-

duced in the model. The place holds as many token as the possible number of acyclic idle slots, and the transitions linked to this place use the processor like the transitions modeling the body of the tasks.

Therefore, the whole set of feasible schedules is given by the center of the terminal language of the Petri net model. The Petri net model can be viewed as a very flexible enumeration method of feasible schedules because it can easily model mailboxes, multi-instance resources, read/write resources, preemptive and non-preemptive parts...

4. Analysis tool: PeNSMARTS

The model is implemented and analyzed by a tool, called PeNSMARTS for *Petri Net Scheduling, Modeling and Analysis of Real-Time Systems.* This section presents how the tool extracts optimal schedules from the PN model, and how we cope with the exponentially large state graph for realistic task systems.

4.1. The state graph

The feasible schedules are obtained through the construction of the state graph. Sec. 4.3 discusses the heuristics used to cope with the state explosion problem. As in practice, we cannot deal with infinite state graph, we focus now on the cyclicity of the schedules in order to bound the depth of the state graph to be computed. As an example, let $S = \{\tau_0\langle 0, 6, 20, 20\rangle,\ \tau_1\langle 0, 2, 4, 4\rangle,\ \tau_2\langle 0, 1, 1, 5\rangle\}$ be a task system where τ_1 and τ_2 share a resource R during their execution (see Fig. 7b).

The Fig. 8 presents the set of feasible schedules (i.e. the state graph) obtained from the simulation of the Petri net given on Fig. 7b. The initial marking of this graph is a home marking (all the paths of the graph reach this state infinitely often), and a valid schedule (obtained by a path from the initial marking to itself) can be repeated infinitely often. As an example, $(\tau_2\tau_1\tau_1\tau_0\tau_0\tau_2\tau_1\tau_1\tau_1\tau_1\tau_2\tau_0\tau_0\tau_1\tau_1\tau_2\tau_1\tau_1\tau_0\tau_0)^*$, where $*$ is the Kleene star, is a feasible schedule.

Since the length of each word is infinite, we have to show that each word w of the PN language can be written $w_a w_c^*$.

In the case of synchronous task systems, at the date 0, all the tasks are simultaneously released. One hyperperiod $P = lcm_{i=1..n} Pi$ later, all the tasks are in the same state as at the date 0. It follows that in this case, the marking of the PN at the date P is the same as the initial marking M_0. Thus, in the state graph of the PN, M_0 is an home marking, and each word can be written as w_c^* where $|w_c| = P$ ($|w|$ denotes the length of w).

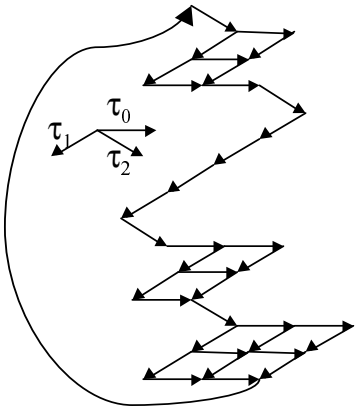

Figure 8. The state graph of the Petri net given on Fig. 7b and equivalently the set of all feasible schedules for the modeled task system.

In the case of asynchronous task systems, we have presented a cyclicity result in sec. 2.2 that claims that for all feasible schedules, a date t_c can be computed, with $-1 \leq t_c < r + P$. This date is unique for a task system, and the state of a system at the date $t_c + 1$ is the same as at the date $t_c + P + 1$. It follows that each word can be written in the form $w_a w_c^*$ where $|w_a| = t_c + 1 \leq r + P$ and $|w_c| = P$. Thus the state graph contains a marking at the depth t_c which is a unique home marking. Fig. 9 presents a state graph of an asynchronous task system.

As an example, let $S = \{\tau_1\langle 0, 3, 6, 8\rangle, \tau_2\langle 6, 2, 8, 8\rangle, \tau_3\langle 7, 4, 16, 16\rangle\}$ be a task system where τ_2 and τ_3 share a resource, and a part of τ_1 precedes τ_2 (see Fig. 10). Since the tasks are not synchronous, we have to compute the date of the last acyclic idle slot : this date is obtained through a simulation of the processor request diagram, that is $t_c = 3$ (see Fig. 11). This date is obtained according to this rule (Grolleau, 1999): the processor load of the task system is $U = \frac{7}{8}$, and its hyperperiod is $P = 16$. It follows that for each time window of size $P = 16$ there are $\frac{7}{8} * 16 = 2$ cyclic idle slots. Fig. 11 shows that there are 3 idle slots in the window $[0..P]$. It follows that one of them is an acyclic idle slot. It follows that the number of acyclic idle slots is $n_c = 1$. We could choose arbitrarily one of the three idle slots to be the acyclic idle slot, but the one minimizing the schedule length is always the first one. Therefore we choose $t_c = 3$ and finally, the parameters of the (cyclic) idle task are $\tau_0\langle t_c + 1 = 4, P(1 - U) = 2, P = 16, P = 16\rangle$ and the parameters of the acyclic task are $\tau_c\langle 0, n_c = 1, t_c = 3\rangle$. The Petri net model for this task system is given on Fig. 10.

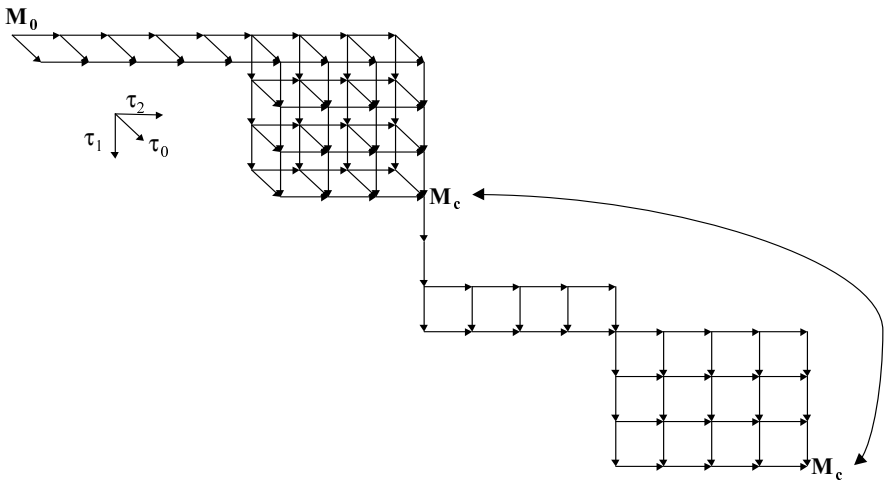

Figure 9. A state graph of an asynchronous task system $S = \{\tau_1\langle 5,3,7,7\rangle, \tau_2\langle 0,8,14,14\rangle\}$. The marking M_c follows the last acyclic idle slot (labeled t_c on the graph). The first diamond corresponds to the initial load of the system, from M_0 to M_c at the depth t_c+1. The second diamond corresponds to the cyclic part of the schedules from M_c to the same marking M_c at the depth t_c+P+1.

4.2. Extraction of optimal schedules

The depth of the state graph is bounded for synchronous and asynchronous task systems, and contains a unique home marking. The state graph contains the entire set of feasible schedules, thus each path is a permutation of another, and the graph has a diamond shape.

The diamond shape of this graph is very interesting in order to find schedules optimal in the sense of minimizing the average response time of a set of tasks. In order to get the set of sequences (i.e. the sub-graph of the state graph) minimizing the average response time of τ_1 in the example given on Fig. 12, the arcs corresponding to the termination of τ_1 are weighted by the response time of τ_1 corresponding to the arc. In other words, if the depth of an arc corresponding to the termination of τ_1 is h, its weight is $h - P_1\lfloor\frac{h-r1}{P_1}\rfloor - r_i$. The other arcs are weighted by 0.

Then the schedules minimizing the average response time of τ_1 are the paths in the graph minimizing the cost (i.e. the dates of termination of τ_1). On Fig. 12, there is only one path minimizing the average response time of τ_1, that is $\tau_1\tau_1\tau_1\tau_2\tau_2\tau_2\tau_2\tau_1\tau_1\tau_1\tau_2\tau_2\tau_2\tau_2$. The average response time of τ_1 for this path is $\frac{6}{2} = 3$.

Thus, in order to find optimal schedules, we just have to search in the graph the shortest path regarding these weights. As the graph is

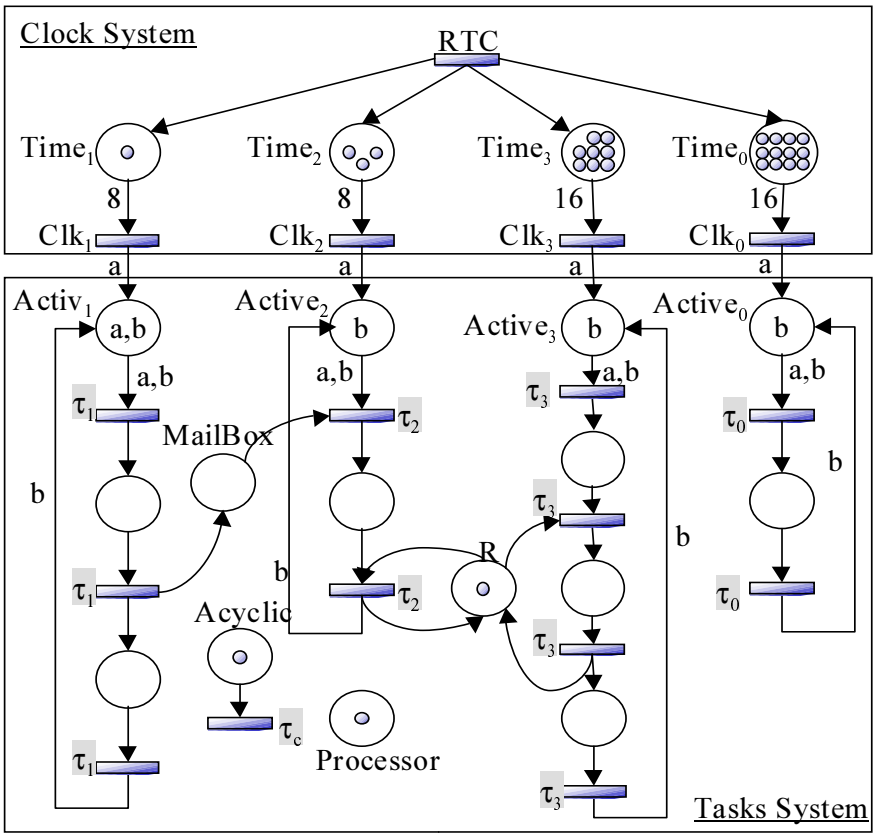

Figure 10. A Petri net model including an idle task, and an acyclic task.

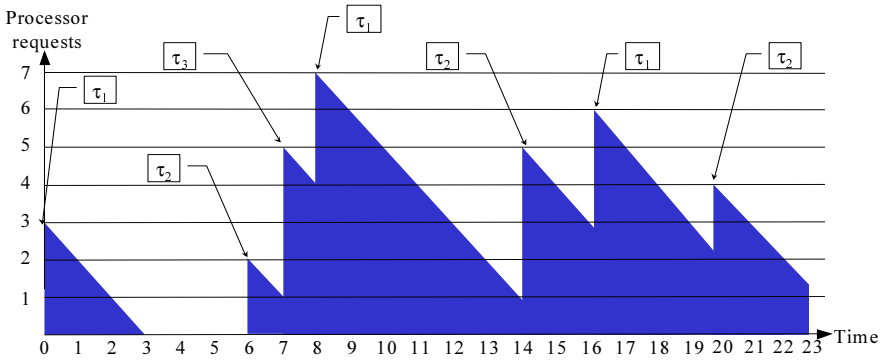

Figure 11. Processor request diagram of $S = \{\tau_1\langle 0, 3, 6, 8\rangle, \tau_2\langle 6, 2, 8, 8\rangle, \tau_3\langle 7, 4, 16, 16\rangle\}$.

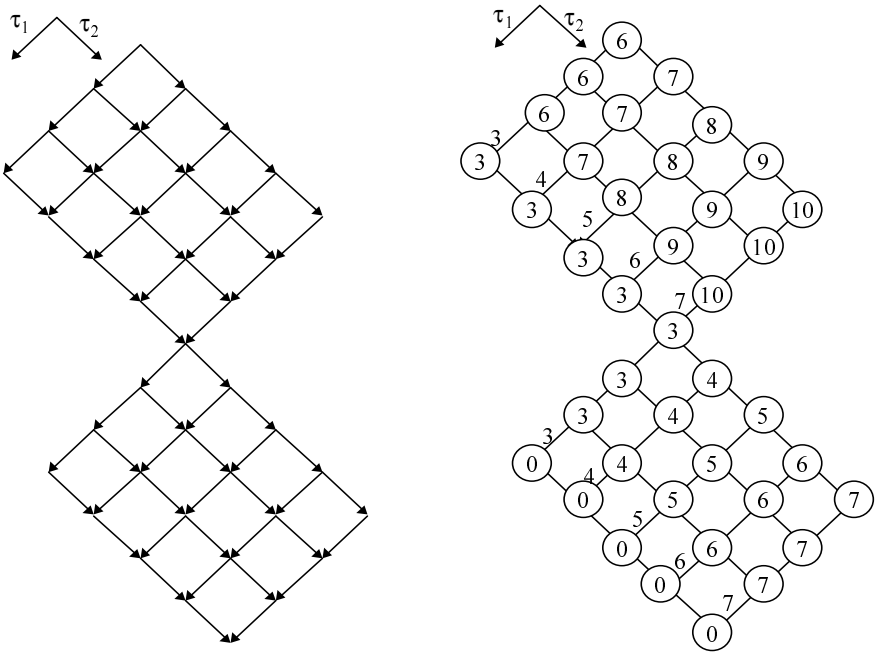

Figure 12. Refinement of a state graph in order to minimize the average response time of τ_1 in the graph of feasible schedules for $S = \{\tau_1\langle 0, 3, 7, 7\rangle, \tau_2\langle 0, 8, 14, 14\rangle\}$. Each node N is labelled by the minimal cost up to the last marking, which is given by the formula $Cost(N) = min_{N' \in Children(N)}\{Weight(Arc(N, N')) + Cost(N')\}$.

a diamond, this search is performed in a time linear in the number of nodes plus the number of arcs by a reverse topological pass through the state graph. We call this extraction of a sub-graph from the state graph a refinement. Several refinements can be applied recursively on sub-graphs.

The extraction technique is based on a weight function depending on criteria. E.g., if we are interested in minimizing the response time of the tasks of a subset S', each arc of the graph is weighted as follows:

- if the arc corresponds to the termination of a task in S', then its weight is the associated response time (e.g. if an instance of τ_i terminates at height h, the weight of the corresponding arc is $h - r_i - P_i\lfloor\frac{h-r_i}{P_i}\rfloor$)

- if the arc does not correspond to any termination of a task in S', then its weight is null.

Similar weighting techniques of the arcs can be used to consider the reaction rates, the importance, the lateness,... Note that each feasible

schedule contains the same number of non-zero weights since each contains the same number of instances of the tasks. Therefore, the paths minimizing the cost are optimal with respect to the response time. In order to minimize the average response time, we consider the function $Cost(N) = min_{N' \in Children(N)}\{Weight(Arc(N, N')) + Cost(N')\}$ used to weight the nodes (see Fig. 12). We use the function $Cost(N) = min_{N' \in Children(N)}\{max\{Weight(Arc(N, N')), Cost(N')\}\}$ in order to minimize the worst case response time.

The same refinement techniques are used in order to extract, from a diamond graph, optimal schedules minimizing the average (or the worst case) maximal tardiness, or the maximal reaction rates of a set of tasks. In these cases, the only difference is the way the arcs are weighted.

Therefore, once the state graph is obtained, the designer can get optimal schedules by several refinements of the state graph. Another way to get interesting schedules, is to apply constraints on the schedules like: "response time of τ_i must lie between C_i and $C_i + 2$", in order to, e.g., bind the jitter of τ_i. This kind of constraints can be applied prior to the construction of the state graph, reducing its space requirements.

Similar methods can be used in order to accurately distribute the idle slots. This can be useful in order to handle sporadic requests accepted on-line.

4.3. Optimizations of the state graph generation

The fact that a finite word w reaches only valid markings does not imply that there exists an infinite word in the center of the terminal language whose prefix is w. Thus, the construction of the state graph of the center of the terminal language implies backtracking (i.e. some reached markings have to be given up). This phenomenon can be reduced thanks to optimization: a necessary condition for schedulability can be tested for each marking reached (see Sec. 4.3.1). Several heuristics are used in order to reduce the (non polynomial) size of the state graph. One of them significantly reduces the graph by forbidding unnecessary preemptions: two concurrent parts of tasks, independent from each others, cannot interleave (see Sec. 4.3.2). The main advantage of this heuristic is that it does not reduce the scheduling power of the model (i.e. if there exists a feasible schedule, then this heuristic preserves at least one schedule). Finally, a appropriate data representation is useful in order to efficiently represent the state graph (see Sec. 4.3.3).

4.3.1. *Reduce the backtracking*
Recall we focus on the center of the terminal language (see Sec. 3.2.2) of the Petri net which is the whole set of feasible schedules for the considered task system. The property of the center of a language is that each word obtained is infinite. It follows that, in the construction of the state graph, some reached marking cannot lead to an infinite word (i.e. the reached state cannot be part of a feasible schedule). Therefore some time and space are lost in the exploration of states which do not lead to a feasible schedule. In this case, as soon as the sterility of the node is established, it has to be cut, and in this case, its parent state could be cut, and so on. This necessary backtracking of the state graph generation has to be reduced by a good prune technique.

Our technique is, given a state, to consider the best possible response time of the current instance of each task (without consideration of any resource or precedence constraint). Let $S = \{\tau_i\langle r_i, C_i, D_i, P_i\rangle\}_{i=1..n}$ a task system and M a marking corresponding to a state. Our goal is to give a lower bound on the response time of the current instance of the tasks after this state. Let $C_{rest,i,M}$ the remaining processing time for τ_i and $d_{i,M}$ the remaining time to its deadline. We verify that $\forall i, \sum_{d_{j,M}<d_{i,M}} C_{rest,j,M} + C_{rest,i,M} \leq d_{i,M}$. The idea behind this formula is that when the deadline of a task τ_i occurs, the deadlines of the more urgent tasks will have occured. It follows that the remaining time to the deadline of τ_i has to be sufficient to process τ_i and the more urgent tasks. This condition can be tested for all the tasks in linear time in the number of tasks. In order to illustrate the gain, consider a task system $S = \{\tau_1\langle 0, 10, 10, 20\rangle\, \tau_2\langle 0, 10, 20, 20\rangle\}$. Fig. 13 shows in dotted line the markings studied if the cut are made only on non terminal markings, in normal line the studied markings when pruning, and in bold line the final state graph. The arrow shows a state M where the graph is pruned: $d_{1,M} = 9$, $C_{rest,1,M} = 10$, therefore, τ_1 cannot meet its next deadline, and M can be pruned.

4.3.2. *Discarding unnecessary schedules*
If two parts of tasks are independent, if a feasible schedule exists, then it can be shown that a feasible schedule exists where a part precedes the other (without preemtions of the parts of the tasks). It follows that it is possible to significantly reduce the size of the state graph in forbidding unnecessary preemptions. These conditions are handled in our model by successor constraints reflecting the following constraints:

- A released task can preempt
- A task which was waiting for an event (message or resource) which has just arised can preempt

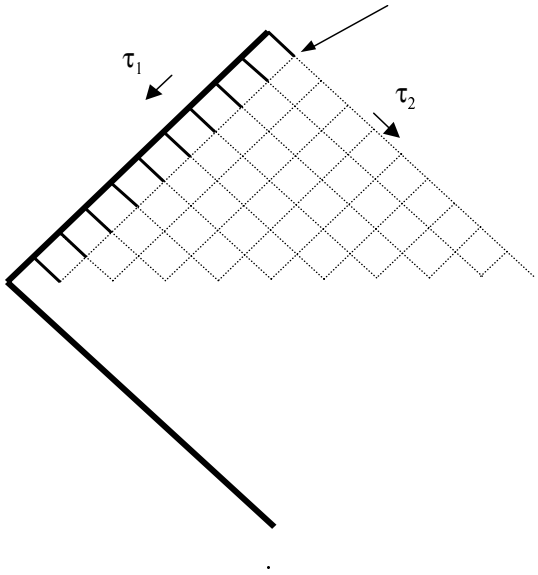

Figure 13. Gain obtained by pruning

– If the running task needs to lock a resource, it can be preempted by the other tasks

This optimization does not reduce the scheduling power of the model since it lets at least one feasible schedule if the system is feasible. But it significantly reduce the space and time used for the state graph. Fig. 14 shows the gain for the trivial example $S = \{\tau_1 \langle 0, 10, 20, 20\rangle \, \tau_2 \langle 0, 10, 20, 20\rangle\}$. With the successor constraints, the part of the graph in dotted line is not built. To illustrate the gain on a larger example, the state graph corresponding to the example of the mine pump presented in Sec. 5 does not fit in a memory of 128 MO.

4.3.3. *Construction of the state graph*
The state graph can contain thousands of markings. It follows that PeNSMARTS needs an efficient internal representation of the markings. First, each place of our model can be bounded in the maximal number of each token it can contain. Then for a place P_k of the Petri net, bounded by b_k, PeNSMARTS uses $\lceil log_2(b_k) \rceil$ bits. Moreover, the state graph can be obtained in a depth first construction (to get only a feasible schedule) and stopped as soon as a feasible schedule is obtained; or in a breadth first construction (to get an optimal schedule).

Finally, in order to minimize the insertion and deletion of markings, we use AVL-trees to represent them. We use one AVL-tree per unit of time (i.e. there are H AVL-trees for a synchronous task system). This representation facilitates hierarchical pass in the state graph.

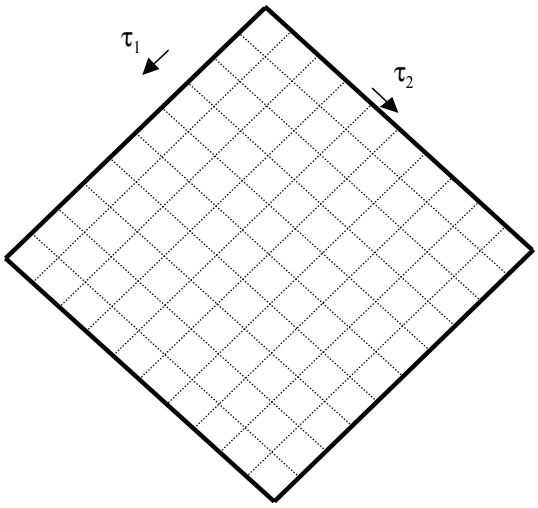

Figure 14. Gain obtained by the successor constraints

5. Case study

Consider a task system dedicated to the control of a mine pump : a mine has to be irrigated, the level of water must remain between a lower and an upper level, infiltration irrigates the mine in a natural way, and the task system has to ensure that the upper level is never exceeded (this example is a free adaptation of an example of (Joseph, 1996)). When the water level becomes too high, a pump is triggered until a lower level is reached. Simultaneously, the methane level has to be controlled in order to trigger an alarm when a high level of methane is reached, and to disable the pump if a dangerous level of methane is reached. The entire process is displayed on a control terminal. Moreover, an additional task traces the levels of methane and water and stores the values on a hard disk drive.

The task system is implemented with seven interacting tasks and three heavily loaded used shared resources (the control terminal, a buffer used to share the methane level, and a buffer used to share the water level). No on-line scheduling algorithm can be validated for this task system.

Studying the tasks, we get the task system given in Fig. 15, where durations are given in milliseconds. Acq_Water (resp. $Acq\_Methane$) uses a shared buffer in order to store the level of water (resp. methane).

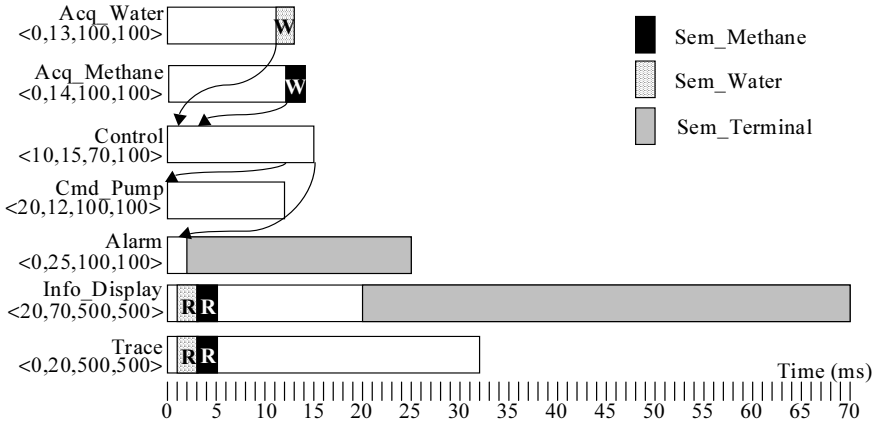

Figure 15. Task system of the mine pump. The temporal parameters of the tasks are given under their name. Note that $Info\_Display$ and $Trace$ use Sem_Water and $Sem\_Methane$ for reading.

The buffer is used for reading by both $Info\_Display$ and $Trace$. It follows that these two tasks can simultaneously access the shared buffers.

This kind of task system has the following characteristics: asynchronous tasks (to the best of our knowledge, no other off-line method deals with asynchronous task systems), processor load equal to 99.4%, high shared resources utilization (which implies that it cannot be scheduled by an on-line algorithm) and read/write access to some resources.

Using our tool PeNSMARTS on this system, we get a graph of all feasible schedules containing 218346 nodes and more than 18.10^{28} feasible schedules (note that in order to choose the optimal schedules, we never deal with the paths, but only with the nodes). The depth of the state graph is 500 time units. Using the method described in section 4, we obtain 5 different schedules optimizing the average response time of all the tasks but the idle task.

6. Summary

The enumeration method based on a Petri net implementation is very flexible and allows the modeling of complex real-time tasks: communication by means of mailboxes, resources in exclusive or read/write access, preemptive and non preemptive sections. To our knowledge, this off-line method handles a wider class of real-time systems than other methods currently available.

This method is exponential in time and space, and several heuristics are used in order to reduce the state graph of the Petri net. The

usefulness of the Petri net model has been studied through a tool, called PeNSMARTS for Petri Net Scheduling, Modeling and Analysis of Real-Time Systems, which has been developed. It is able to find optimal schedules for synchronous and asynchronous task systems. The algorithms used in order to extract optimal schedules are linear in the size of the state graph.

Furthermore, the study of the cyclicity of schedules in the case of asynchronous task systems allows the use of off-line scheduling methodologies to take asynchronous task systems into account.

7. Acknowledgements

The authors are very grateful to the anonymous reviewers for their important and constructive comments.

References

Baker, K. and Z. Su: 1974, 'Sequencing with due-dates and early start times to minimize maximum tardiness'. *Naval Research Logistic Quarterly* **21**, 171–176.

Baker, T.: 1991, 'Stack-based scheduling of real-time processes'. *The Journal of Real-Time Systems* **3**, 67–99.

Baruah, S., L. Rosier, and R. Howell: 1990, 'Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor'. *Real-Time Systems* **2**, 301–324.

Berthomieu, B. and M. Diaz: 1991, 'Modeling and verification of time dependent systems using time Petri nets'. *IEEE Transactions on Software Engineering* **17**(3), 259–273.

Bratley, P., M. Florian, and P. Robillard: 1975, 'Scheduling with earliest start and due date constraints on multiple machines'. *Naval Research Logistic Quarterly* **22**(1), 165–173.

Buttazzo, G.: 1997, *Hard real-time computing systems.* Kluwer Academic Publishers.

Chen, M. and K. Lin: 1990, 'Dynamic priority ceilings : a concurrency control protocol for real-time systems'. *Real-Time Systems* **2**(4), 325–346.

Grolleau, E.: 1999, 'Ordonnancement temps reel hors-ligne optimal a laide de reseaux de Petri en environnement monoprocesseur et multiprocesseur'. Ph.D. thesis, ENSMA-Universite de Poitiers.

Joseph, M. e.: 1996, *Real-time systems, specification, verification and analysis.* Prentice Hall.

Kaiser, C.: 1982, 'Exclusion mutuelle et ordonnancement par priorite'. *Technique et Science Informatiques* **1**(1), 59–68.

Leung, J. and M. Merrill: 1980, 'A note on preemptive scheduling of periodic real-time tasks'. *Information Processing Letters* **11**(3), 115–118.

Liu, C. and J. Layland: 1973, 'Scheduling algorithms for multiprogramming in real-time environment'. *Journal of the ACM* **20**(1), 46–61.

Menasche, M. and B. Berthomieu: 1983, 'Time Petri nets for analyzing and verifying time dependent communication protocols'. In: H. R. West and C.H. (eds.): *Protocol Specification, Testing, and Verification*, Vol. 3. IFIP, pp. 161–172.

Merlin, P. and D. Farber: 1976, 'Recoverability of communication protocols - Implications of a theoretical study'. *IEEE Transactions on Communications* pp. 1036–1043.

Mok, A.: 1983, 'Fundamental design problems of distributed systems for the hard real-time environment'. Ph.D. thesis, Massachussets Institute of Technologie.

Mok, A. and M. Dertouzos: 1978, 'Multiprocessor scheduling in a hard real-time environment'. In: *7th Texas Conference on Computer Systems*. pp. 5.1–5.12.

Monnier, Y., J. Beauvais, and A.-M. Deplanche: 1998, 'A Genetic Algorithm for Scheduling Tasks in a Real-Time Distributed System'. In: *24th Euromicro Conference*. Vasteras, Sweden.

Murata, T.: 1989, 'Petri nets: properties, analysis and applications'. *Proc. of the IEEE* **17**(3), 541–580.

Peterson, J.: 1981, *Petri nets theory and the modeling of systems*. Prentice-Hall.

Petri, C.: 1962, 'Kommunikation mit automaten'. Ph.D. thesis, Bonn Institut fur Instrumentelle Mathematik.

Ramchandani, C.: 1974, 'Analysis of asynchronous concurrent systems by timed Petri nets'. Ph.d., MIT.

Sha, L., R. Rajkumar, and J. Lehoczky: 1990, 'Priority inheritance protocols : an approach to real-time synchronization'. *IEEE Transactions on Computers* **39**(9), 1175–1185.

Stankovic, J.: 1988, 'Misconceptions about real-time computing'. *Computer* **21**(10), 10–19.

Stankovic, J., M. Spuri, K. Ramamritham, and G. Buttazzo: 1998, *Deadline scheduling for real-time systems*. Kluwer Academic Publishers.

Starke, P.: 1990, 'Some properties of timed nets under the earliest firing rule'. *Advances in Petri nets 1989, Venice in Lecture Notes in Computer Science* **424**, 418–432.

Tsai, J., S. Yang, and Y.-H. Chang: 1995, 'Timing constraint Petri nets and their application to schedulability analysis of real-time system specifications'. *IEEE Transactions on Software Engineering* **21**(1), 32–49.

Valk, R. and G. Vidal-Naquet: 1981, 'Petri nets and regular languages'. *Journal of Computer and System Sciences* **23**(3), 399–325.

Xu, J. and D. Parnas: 1990, 'Scheduling processes with release times, deadlines, precedence, and exclusion relations'. *IEEE Transactions on Software Engineering* **16**(3), 360–369.

Zamorano, J., A. Alonso, and J. d. l. Puente: 1997, 'Automatic generation of cyclic schedules'. In: *WRTP'97, 22nd IFACIFIP Workshop on Real-Time Programming*. Elsevier Science, pp. 145–151.