Journées des Thèses 2011
Branche TIS (DCSD/DCPS/DTIM)
26 Janvier à Palaiseau et 7-8 Février à Toulouse

Multiprocessor Schedulability Analysis for Dependent Periodic Tasks

Mikel Cordovilla (Mikel.Cordovilla@onera.fr)
ONERA–DCSD, Toulouse
ISAE, Toulouse

Thèse encadrée par : Claire Pagetti (Onera-DTIM), Frédéric Boniol (Onera-DTIM) et Eric Noulard (Onera-DTIM)

Résumé

Within the context of hard real-time systems, the schedulability analysis *of a task set is a major issue. The problem consists in proving that the tasks always satisfy their temporal and precedence constraints for a given scheduling policy and a given platform. Extensive work has been done in the last decades for defining sufficient criteria and exact algorithms. Sufficient criteria usually have an excellent complexity but often lead to an over-dimension of the system. On the opposite, exact algorithms, especially in the case of multiprocessor platform, suffer from an exponential complexity.*

*In this paper, we study an exact technique : we apply a brute force search combined with a model checker (*Uppaal*) that determines whether the exploration is complete. We consider periodic tasks with precedences constraints which execute on parallel platforms composed of homogeneous processors. Under these hypotheses, we have encoded for policies : fixed task priority, gEDF, gLLF and LLREF. The analyser is user friendly and provides promising performances.*

A. INTRODUCTION

The evolution of real-time embedded systems leads to the increase of functionalities and the use of more powerful architectures. We consider homogeneous multicore architectures which are composed of at least two identical cores integrated on the same chip. The functional specification is given as a set of nodes (or tasks) that share some data and the communication pattern of which is precisely defined.

The objective is to implement the specification as a multi-threaded code on the multicore. For this, we provide a tool set which verifies the schedulability and generates an implementation which corresponds to the semantics of the initial specification.

1. Real-time scheduling

A real-time system can be represented as a taskset $\mathbf{S} = \{\tau_1, \tau_2, \ldots, \tau_n\}$. A periodic task is a functional code that must satisfy some timing constraints. Its attributes are (T, C, O, D) where T is the appearance period, C is the worst case execution time estimation, O is the first arrival instant and D is the relative deadline. The deadline is assumed to be less than the period, that is $D \leq T$. These notions are illustrated in the figure 1. Since a periodic task executes periodically, there are several executions which are called *instance* or *job*. $\tau[p]$ denotes the p^{th} job of τ. HP or hyperperiod is the least common multiple (aka lcm) of the task periods. This model of task is a standard adaptation of the Liu and Layland model [LL73].

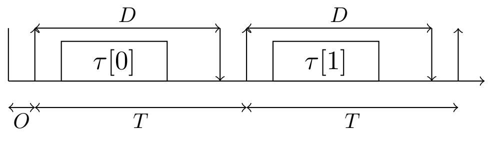


Fig. 1 – *Task model*

The multicore is composed of m identical cores. Several priority-based scheduling policies have been proposed to give access to the tasks on the CPU resources.

A policy is *preemptive* if it can suspend a task during its execution and resume it later. A policy can allow the *migration* : a task can start on a core and after a suspension can pursue on another core. Several priority affectation algorithms can be found in the literature. *Fixed priority* assigns a static priority to each task. gEDF (Global Earliest Deadline

First) assigns the highest priority to the task with the shortest relative deadline. gLLF (Global Least Laxity first)[Mok83] assigns the highest priority to the task with the shortest laxity, i.e. the relation between the remaining execution time and the relative deadline. LLREF(Largest Local Remaining Execution First) [CRJ06] divides the time in slots where the bound is the closest deadline, wake up time or period. The priority is assigned regarding the local laxity first and the largest remaining execution time.

An off-line scheduling is computed before runtime and stored in a table executed by a dispatcher. Conversely, an on-line scheduler takes decisions at run-time whenever a task instance terminates or a new task instance asks for the computing resources.

We consider in the following on-line preemptive scheduling with migrations since these strategies allow powerful and flexible implementations. We assume that a task cannot execute on two different cores at the same time (there is no intra-task parallelism).

2. Precedences constraints

Real-time tasks often share some data. In practice, a task produces a data which is consumed by another one. The exchange can be asynchronous and is realised by a message passing ; or the exchange can be synchronous and is realised by the use of a global variable. In both cases, we distinguish two kinds of behaviours :

– direct : tasks exchange data directly, the producer must end before the consumer starts. In real-time it is encoding as a precedence between the producer and the consumer. We then speak of "dependent task set" ;
– not defined : there is no information about the type of exchange. There is no assumption on the ordering between the producer and the consumer. Consequently, these communications do not generate additional constraints on the task set.

There are several formalisations for representing the precedences in real-time systems. We use the notion of extended precedences of [FBG+10].

Simple precedences describe communications between tasks with an identical period. Precedence $\tau_i \rightarrow \tau_j$ expresses that τ_i must execute before τ_j. A task can be constrained by several precedences but the complete graph of precedences must not contain any cycle, i.e. we cannot have $\tau_i, \rightarrow \tau_j \rightarrow \cdots \rightarrow \tau_i$.

Let us introduce some notations. *precs(τ_i)* is composed of all the predecessors of τ_i : *preds*$(\tau_j) = \{\tau_i | (\tau_i, \tau_j) \in \rightarrow\}$. In the same way, *succs(τ_j)* is composed of all the successors of τ_j : *succs*$(\tau_i) = \{\tau_j | (\tau_i, \tau_j) \in \rightarrow\}$.

The idea of the *extended precedences* is to formalise the precedences between multiperiodic tasks. Note that if τ_i produces some data for τ_j and $T_i < T_j$, τ_i can executes several instances before τ_j executes its first instance. A precedence between the n^{th} instance of τ_i and the n'^{th} instance of τ_j is denoted as $\tau_i[n] \rightarrow \tau_j[n']$.

Definition 1. *An extended precedence between τ_i and τ_j is a set of task instance precedences. It is coded as a set $M_{i,j} \subseteq \mathbb{N}^2$ thus $\forall (n, n') \in M_{i,j}, \tau_i[n] \rightarrow \tau_j[n']$.*

The set $M_{i,j}$ may be infinite. J. Forget proposes periodic extended precedences to turn $M_{i,j}$ into a finite set. Let $p_{i,j} = lcm(T_i, T_j)$ and $M_{i,j} \subseteq p_{i,j}/T_i \times p_{i,j}/T_j$, $\tau_i \xrightarrow{M'_{i,j}} \tau_j$ means that $\tau[n] \rightarrow \tau[n']$ where $(n, n') \in M_{i,j}$ such that :

$$\exists k \in \mathbb{N}, (n, n') = (m, m') + \left(k\frac{p_{i,j}}{T_i}, k\frac{p_{i,j}}{T_j}\right) \quad (1)$$

We can verify that a simple precedence $\tau_i \rightarrow \tau_j$ is a particular case of periodic extended precedences and equivalent to $\tau_i \xrightarrow{M_{i,j}} \tau_j$ with $M_{i,j} = \{(n, n) | n \in \mathbb{N}\}$.

Example 1. *Let us consider the communication $\tau_i \xrightarrow{M} \tau_j$.*

– *for $T_i = 2T_j$ and $M = \{(0, 0)\}$, we have : $\tau_i[0] \rightarrow \tau_j[0], \tau_i[1] \rightarrow \tau_j[2], \tau_i[2] \rightarrow \tau_j[4], \ldots$*
– *for $T_i = T_j/2$ and $M = \{(0, 1)\}$, we have : $\tau_i[0] \rightarrow \tau_j[0], \tau_i[2] \rightarrow \tau_j[1], \tau_i[4] \rightarrow \tau_j[2], \ldots$*
– *for $T_i = T_j/3$ and $M = \{(0, 2)\}$, we have $\tau_i[0] \rightarrow \tau_j[2], \tau_i[1] \rightarrow \tau_j[5], \tau_i[2] \rightarrow \tau_j[8], \ldots$*
– *for $T_i = T_j \times 3/5$ and $M = \{(0, 0), (2, 1), (3, 2)\}$, we have : $\tau_i[0] \rightarrow \tau_j[0], \tau_i[2] \rightarrow \tau_j[1], \tau_i[3] \rightarrow \tau_j[2], \tau_i[5] \rightarrow \tau_j[3], \tau_i[7] \rightarrow \tau_j[4] \ldots$*

3. Contribution

The purpose of this work is to propose a methodology for the schedulability analysis of a dependent task set on a multicore architecture for a given scheduling policy. Our approach is based on a brute-force technique. The idea is to describe all the behaviours and to use the model checker UPPAAL which is designed to efficiently encode state space exploration and to detect when the system returns in previously visited states. We have applied in [CBNP11] the approach for independent task sets with efficient explorations. We extend the results to take into account the precedences.

In the section B., we present the task encoding and the schedulability analysis for simple and extended precedences. In the last section C. we describe a methodology to generate several performance analysis for the proposed encoding.

B. SCHEDULABILITY ANALYSIS OF DEPENDENT TASKS

A schedulability analysis takes as an input a multicore architecture, a task set and a scheduling policy. The objective is to determine whether any execution always respects the temporal constraints, i.e. there is no deadline miss and the precedences are respected.

To represent the evolution of the time we define a time step. At each firing, we compute the new configuration according to the used scheduling policy.

1. Schedulability analysis

The technique works as follow : we encode the attributes of the tasks as tabular data structures. We describe the exploration with an automaton and we encode the precedences in arrays that intervene in the choice of the m highest priority tasks.

Task encoding First, for encoding a task $\tau = (T_\tau, C_\tau, O_\tau, D_\tau)$ in UPPAAL we use the variables :

- $T^c_\tau : \mathbb{N} \to [0, T_\tau]$ where $T^c_\tau(t)$ represents the remaining time till the beginning of the next activation,
- $C^c_\tau : \mathbb{N} \to [0, C_\tau]$ where $C^c_\tau(t)$ represents the remaining execution time of the current job,
- $O^c_\tau : \mathbb{N} \to [0, O_\tau]$ where $O^c_\tau(t)$ represents the remaining time till the starting time of the first job,
- $D^c_\tau : \mathbb{N} \to [0, D_\tau]$ where $D^c_\tau(t)$ represents the remaining time till the deadline of the current job,
- $n^c_\tau : \mathbb{N} \to \mathbb{B}$ where $n^c_\tau(t)$ represents the access to a processor. The value of this variable depends on the policy.

The parameters of a task are computed each time t. The configuration of task τ at t is the tuple $(T^c_\tau(t), C^c_\tau(t), O^c_\tau(t), D^c_\tau(t), n^c_\tau(t))$.

According to several values of the tuple we can divide tasks in two groups : *active* and *inactive*. A task τ_i is *active* at t if $O_i = 0 \wedge C_i > 0$ otherwise it is *inactive*.

The variable *list_prio(t)[n]* gives the ordered list of tasks at t. A task τ_i, accesses a processor at t, i.e. when $n_i(t) = true$ which occurs if it is active, it is free of precedences and the position of τ_i in *list_prio(t)[n]* is less than m :

$$n_i(t) = true \Leftrightarrow \begin{cases} \tau_i \in active(t) \wedge \\ \tau_i = list\_prio(t)[k] \wedge k \leq m \wedge \\ \#precs_i = 0 \end{cases} \tag{2}$$

The definition of the set *precs* will be given in the next sections, it depends on the time and on the precedences.

Exploration To encode the temporal exploration, we use the automaton shown in figure 2.

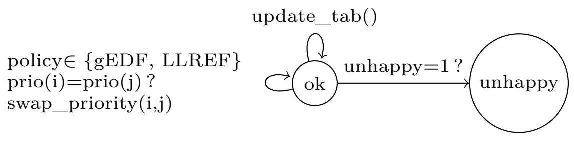


Fig. 2 – *Execution in* UPPAAL

Initially, the automaton is in state *OK* and $T_\tau(0) = T_\tau; D_\tau(0) = D_\tau; C_\tau(0) = C_\tau; O_\tau(0) = O_\tau$.

The first loop labelled *update_tab* represents the temporal evolution of the system, i.e. $t \to (t+1)$. The function *update_tab* modifies the tuple depending on the task state. A task evolutes in this manner :

```
1  if $O_\tau(t) > 0$ then /* $\tau$ is not active at $t$ because the release date has not still been exceed */
2
3      $O^c_\tau(t+1) = O^c_\tau(t) - 1$;
4  else
5      if $T^c_\tau(t) = 1$ then /* a new job is awaken */
6
7          $T^c_\tau(t+1) = T_\tau, D^c_\tau(t+1) = D_\tau$;
8          $C^c_\tau(t+1) = C_\tau$
9      else
10         $T^c_\tau(t+1) = T^c_\tau(t) - 1, D^c_\tau(t+1) = D^c_\tau(t) - 1$;
11         if $C^c_\tau(t) > 0$ then
12             if $\tau$ is not one of the $m$ highest priority then
13                 $C^c_\tau(t+1) = C^c_\tau(t)$
14             else
15                 $C^c_\tau(t+1) = C^c_\tau(t) - 1$
```

On gEDF and LLREF, two tasks with the same priority can execute indistinctly. However, considering that the priority allocation is realised by a C-like function, the non determinism cannot be represented. The second loop is used to reorder the same priority tasks and to force a non deterministic scheduling.

Finally, the transition labelled $unhappy = 1?$ can be fired only if the variable *unhappy* is equal to 1 which only occurs when the task set is not schedulable.

2. Simple precedences

For representing the simple precedences we create two bi-dimensional arrays : *tabInitial* for the initial values and *tabCurrent* for stocking the current state of the precedences.

Example 2. *Let us for instance consider the task set :*

	T	C	O	D	*Prio*	
τ_0	*6*	*1*	*0*	*6*	*1*	$\tau_0 \to \tau_1$
τ_1	*6*	*2*	*0*	*6*	*2*	$\tau_0 \to \tau_2$
τ_2	*6*	*2*	*1*	*6*	*3*	$\tau_0 \to \tau_3$
τ_3	*6*	*1*	*1*	*6*	*4*	$\tau_1 \to \tau_3$
						$\tau_2 \to \tau_3$

For these precedence constraints tabInitial = $[(-1,-1,-1),(0,-1,-1),(0,-1,-1),(0,1,2)]$. *The* -1 *value means that there is no precedence constraint.*

At time 0, *tabCurrent* is identical to *tabInitial* but with the time evolution the validity of the precedences changes :

– when task τ_i ends its execution at $t+1$ and if $tabCurrent[\tau_j](t) = (\cdots, i, \cdots)$, then we remove the constraint $tabCurrent[\tau_j](t+1) = (\cdots, -1, \cdots)$;
– when $t = HP$ we restart the initial precedence values, *tabCurrent* =*tabInitial*.

A task which is submitted to, at least, a simple precedence cannot execute. $precs(\tau_i, t)$ represents the valid precedences at t, so, $tabCurrent(t) = [precs(\tau_i, t)]$. The size of *tabInitial* and *tabCurrent* is $nbTask \times max(\#precs(\tau_i)\forall i)$.

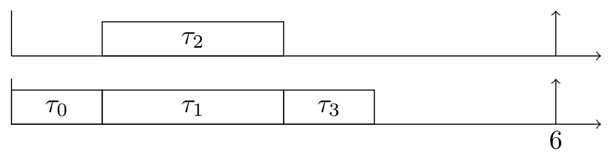
Definition 2. *$precs(\tau_i, t)$ is composed of all predecessors of τ_j which do not finish its execution at t.*

$$precs(\tau_i, t) = \{\tau_j | (\tau_j, \tau_i) \in \to \wedge C_{\tau_i}(t) \neq 0\}$$

Example 3. *We want to analyse if the task set of example 2 is schedulable on 2 cores with a fixed priority.*

time	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
τ_0								
τ_1	*0*							*0*
τ_2	*0*							*0*
τ_3	*0,1,2*	*1,2*	*1,2*					*0,1,2*

At the time 0 τ_0 *and* τ_1 *are active but only* τ_0 *executes because* τ_1 *depends on* τ_0. *At time* 1, *all tasks depending on* τ_0 *can execute. The scheduler assigns* τ_1 *and* τ_2 *to each core. At* 3 *they finish their execution and they disappear of the* $precs(3, \tau_3)$. τ_3 *is available to execute from* 3. *We emphasizes at* 6 *tabCurrent restarts with the initial configuration.*



3. Extended precedences

We apply the same idea for the extended precedences. We define two bi-dimensional arrays, *tabInitial* for stocking the initial value and *tabCurrent* for the current state of every precedences. Let us start with an example to illustrate the encoding in *tabInitial* :

Example 4. *For a task set :*

	T	C	O	D	*Prio*	
τ_0	*5*	*1*	*0*	*5*	*1*	$\tau_1 \xrightarrow{\{(0,0)\}} \tau_0$
τ_1	*5*	*1*	*0*	*5*	*2*	$\tau_1 \xrightarrow{\{(1,0)\}} \tau_3$
τ_2	*5*	*1*	*1*	*5*	*3*	$\tau_2 \xrightarrow{\{(0,0)\}} \tau_0$
τ_3	*10*	*1*	*1*	*10*	*4*	$\tau_2 \xrightarrow{\{(1,0)\}} \tau_5$
τ_4	*10*	*1*	*1*	*10*	*5*	$\tau_5 \xrightarrow{\{(0,0)\}} \tau_3$
τ_5	*20*	*1*	*1*	*20*	*6*	$\tau_5 \xrightarrow{\{(0,0)\}} \tau_4$

We present in the figure 3 the encoding of the precedences of τ_0 .

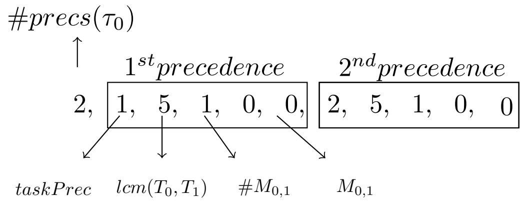


Fig. 3 – *Extended precedence encoding*

The first number, "2" represents the number of precedence constraints of τ_0, *with* τ_1 *and* τ_2. *Rectangles demarcate the data belonging to each precedence. For the first precedence, the first value, "1" means that the predecessor task is* τ_1. *5 is the* $lcm(T_0, T_1)$. *1 is the number of sets in* $M_{0,1}$ *and the two lasts values,* $(0,0)$ *is the unique element of* $M_{0,1}$.

More generally, the precedences contraining a task τ are expressed like $initialTab(\tau) = nbPrec + \#precs(nbPrec) \times (taskPrec, lcm, nbWords, \#M(taskPrec) \times (n, n')$ with :

– *nbPrec* : the number of precedences constraints for τ_i ;
– *taskPrec* : the name of one of the predecessors task ;
– *lcm(taskPrec,taskDep)* : the *lcm* calculated between the precedent task and the dependant task ;
– *nbWords* : each extended precedence between two tasks τ_i and τ_j corresponds to a set of precedences between the instances of the tasks. This variable stocks the cardinal of this set.
– (n, n') : Every precedence between two instances is defined by a couple (n, n') that denote a precedence from the instance n of the precedent task to the instance n'.

So, the size of both tables *initialTab* and *currentTab* is $nbTasks \times max(\#preds(nbPrec) \times (taskPrec, lcm, nbWords, \#M(taskPrec) \times (n, n')))$.

time	0	1	2	5	7	8	10	20
τ_0	2,1,5,1,0,0,2,5,1,0,0	1,2,5,1,0,0	0	2,1,5,1,0,0,2,5,1,0,0	0	0	2,1,5,1,0,0,2,5,1,0,0	2,1,5,1,0,0,2,5,1,0,0
τ_1	0	0	0	0	0	0	0	0
τ_2	0	0	0	0	0	0	0	0
τ_3	2,1,10,1,1,0,5,20,1,0,0	2,1,10,1,1,0,5,20,1,0,0	2,1,10,1,1,0,5,20,1,0,0	2,1,10,1,1,0,5,20,1,0,0	1,5,20,1,0,0	0	1,1,10,1,1,0	2,1,10,1,1,0,5,20,1,0,0
τ_4	1,5,20,1,0,0	1,5,20,1,0,0	1,5,20,1,0,0	1,5,20,1,0,0	0	0	0	1,5,20,1,0,0
τ_5	1,2,20,1,1,0	1,2,20,1,1,0	1,2,20,1,1,0	0	0	0	0	1,2,20,1,1,0

Fig. 4 – *Evolution of current table*

Extended precedences main purpose is to generate precedences constraints between two specific jobs. Therefore, to decide if a job is constrained by another one, we have to know which is the current job of each tasks. We declare a table *list_job[nbTasks]* to stock this value. We actualise this table each time we start a new job of a task. When $t = HP$ we restart the states of all the tasks, $tabCurrent = tabInitial$ and we restart *list_job[nbTasks]* to 0.

Definition 3. *At time t, the current job of τ_i is $q = list\_job[i]$. From the definition of $M_{i,j}$, we know all the jobs required before $\tau_i[q]$. $\tau_j[m] \rightarrow \tau_i[q]$ with $(q,m) = (l,l') + (k\frac{p_{i,j}}{T_i}, k\frac{p_{i,j}}{T_j})$ where $k = min(list\_job[i], list\_job[j]$ thus*

$$precs(t,\tau_i) = \{\tau_j \left| \begin{array}{l} (l,l') \in M_{i,j} \\ C_j(t) > 0 \wedge \\ m = l' + k \times \frac{p_{i,j}}{T_i} \wedge \\ q = l + k \times \frac{p_{i,j}}{T_j}\} \end{array} \right.$$

To update *tabCurrent* we use two functions, *updateDep(taskId)* and *initialiseDep(taskId)*. *updateDep(taskId)* is called at the end of a task execution to cancel every precedence constraints generated by this task. When a task restart a job *initialiseDep(taskId)* is called to initialise $tabCurrent[taskId]$.

Example 5. *We want to analyse if the task set of example 4 is schedulable on 2 cores with a fixed priority.*

The table 4 describes the temporal evolution of the tabCurrent *and the figure 5 illustrates in a Gantt chart the tasks execution. Initially, only τ_0 and τ_1 tasks are in active state, but only τ_0 has no precedences and starts on core 1. At 1 the rest of tasks are awake and therefore active. τ_1 finishes its execution, consequently, the analyser actualises the precedences in the table* tabCurrent *and changes the value active of every tasks which depend on $\tau_1[0]$. At the second time, τ_2 ends the execution of the first job. The task τ_0 dependes on $\tau_1[0]$ and $\tau_2[0]$, can start in 3. This motif is repeated every 5 instants because between these three task have the same period (5) and consequently they are constrained by simple precedences.*

τ_5 depends on $\tau_2[1]$ and start the execution at 7. τ_4 must execute before $\tau_5[0]$ so from time 8 is free of precedences constraints to execute its two jobs. τ_3 must execute before $\tau_5[0]$ and $\tau_1[1]$. From time 8 can be allocated in a core but at time 10 τ_1 restart and τ_3 must to wait to $\tau_1[1]$ to execute de second job.

C. PERFORMANCE EVALUATION

We have developed a prototype named `converter` which converts a simple text description of a dependent task set. Example 4 can be described by :

```
# Example1 task set
# Task "Name" T C D O
Task "Tau0" 5 1 5 0
Task "Tau1" 5 1 5 0
Task "Tau2" 5 1 5 1
Task "Tau3" 10 1 10 1
Task "Tau4" 10 1 10 1
Task "Tau5" 20 1 20 1
# ExtDependency "pred" "succs" words
ExtDependency "Tau1" "Tau0" 0 0
ExtDependency "Tau1" "Tau3" 1 0
ExtDependency "Tau2" "Tau0" 0 0
ExtDependency "Tau2" "Tau5" 1 0
ExtDependency "Tau5" "Tau3" 0 0
ExtDependency "Tau5" "Tau4" 0 0
```

Task description starts with the keyword "Task", following with the task name, the period, the WCET, the relative deadline and the offset. We can use two notations to describe precedences. For simple precedences we start with the keyword "Dependency" following with the predecessor task name and the successor task name. However, to define a extended precedences, we use the keyword "ExtDependency" following with the predecessor task name, the successor task name and every instance couples contained in $M_{prec,succ}$.

`converter` takes several command line arguments : the file name containing the task set description, the number of cores, the scheduling policy (FP, gEDF, gLLF, LLREF) and the type of dependency in the system. The source code is around 5000 lines of portable C code. The `converter` is part of the wider software kit which contains building blocks for embedded real-time middleware consisting of off-line and on-line tools.

In the future, we plan to create an automatic task set generator with varying temporal and precedence constraints. The idea is to evaluate impact, mainly the memory consummation and the time performances.

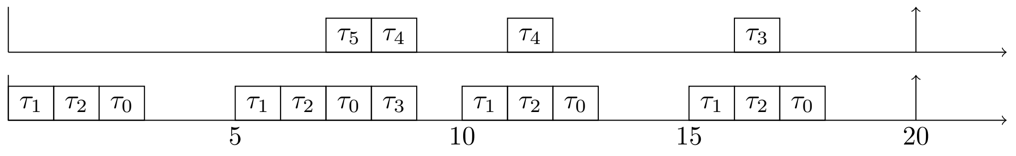


Fig. 5 – *Extended precedences execution*

D. CONCLUSIONS

The aim of this research is to present a brute force approach for analysing the schedulability of a task set with precedence constraints. We consider several scheduling policies such as fixed priority, gEDF, LLF or LLREF. We have then proposed an encoding to take into account the *simple precedences* and the *extended precedences*. We include this implementation in a prototype `converter` which takes a textual task specification and converts it in a UPPAAL specification.

The next work is to propose a methodology for the performance analysis as well as to make extensve performance benchmarks. The second work to explore consists of adding some metrics to the model or the analyser, such as the number of task migrations and of context changes.

RÉFÉRENCES

[CBNP11] Cordovilla, Mikel, Frédéric Boniol, Eric Noulard, and Claire Pagetti. Multiprocessor schedulability analyser. *26th Symposium On Applied Computing (SAC 11)*, march 2011.

[CRJ06] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. *Real-Time Systems Symposium, IEEE International*, 0 :101–110, 2006.

[FBG$^+$10] Julien Forget, Frédéric Boniol, E. Grolleau, D. Lesens, and Claire Pagetti. Scheduling dependent periodic tasks without synchronization mechanisms. *In 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10)*, April 2010.

[LL73] Cheng L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1) :46–61, 1973.

[Mok83] A. K. Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Cambridge, MA, USA, 1983.