The Scheduling Problem of Self-Suspending Periodic Real-Time Tasks

Yasmina Abdeddaïm, Damien Masson

The Scheduling Problem of Self-Suspending Periodic Real-Time Tasks*

Yasmina Abdeddaïm and Damien Masson
Université Paris-Est
LIGM UMR CNRS 8049, ESIEE Paris
2 bld Blaise Pascal, BP 99, 93162 Noisy-le-Grand CEDEX, France
y.abdeddaim/d.masson@esiee.fr

September 19, 2012

Abstract

In this paper, we address the problem of scheduling periodic, possibly self-suspending, real-time tasks. We show how to use model checking to obtain both a necessary and sufficient feasibility test, and a schedulability test for classical scheduling policies (RM, DM, EDF). When these algorithms fail to schedule a feasible system, we show how to generate an appropriate scheduler. We provide also a method to test the sustainability of a schedule w.r.t execution and suspension durations. Finally, using a model checking tool we validate our approach.

1 Introduction

A real-time task can suspend itself when it has to communicate, to synchronize or to perform external input/output operations. Classical models neglect self-suspensions, considering them as a part of the task computation time [24]. Models that explicitly consider suspension durations exist but their analysis is proved difficult. In [29], the authors present three negative results on systems composed by hard real-time self-suspending periodic tasks scheduled on-line. We propose in this paper the use of model checking on timed automata to address these three negative results: 1) the scheduling problem for self-suspending tasks is NP-hard in the strong sense, 2) classical algorithms do not maximize tasks completed by their deadlines and 3) scheduling anomalies can occur at run-time. Result 1) means that there cannot exist a non-clairvoyant on-line algorithm that takes its decisions in a polynomial time and always successfully schedules a feasible self-suspending task set. We so propose to use model checking to generate off-line a feasible scheduler for each specific instances of the problem, i.e. for each task sets. Result 2) implies that traditional on-line schedulers are not optimal, whereas using our method to produce a schedule is an optimal approach. Result 3) points out that changing the properties of a feasible task set in a positive way (e.g. reducing an execution or a suspension duration, extending a period) can affect its feasibility. To overcome this problem, we consider the cases where both the execution and the suspension times of each task are constrained within an interval of possible values. The generated schedulers then have an important property: the feasibility of a task set is sustainable w.r.t the execution and suspension durations.

We review some related work in Section 2. Section 3 presents the task model, Section 4 introduces the timed automaton modeling a self-suspending task, Section 5 exposes how to check the feasibility and the schedulability with PFP (Preemptive Fixed Priority) and EDF (Earliest Deadline First), Section 6 shows how to prove the sustainability of a schedule and how to generate a sustainable scheduler, Section 7 presents experiments and finally we conclude in Section 8.

2 Related work

Recent works have shown the relevance of considering possible self-suspensions for the scheduling problem of real-time tasks. For example, when an application is executed on a multi-core or on a multi threaded architecture, significant suspension intervals can occur due to tasks migrations [20] or to resource sharing amongst several threads [18].

The problem can be partly addressed by the use of specific configurable synchronization protocols [7], or by model transformations. For example, a real-time task that suspends itself just once can be considered as two different subtasks, the first one having a shorter deadline and the second one having a release jitter. Analysis methods of such tasks with release jitter are proposed in [31]. Additional mechanisms to enforce period and release have also been proposed in [27, 30].

However, considering the problem this way introduces a high degree of pessimism. Pessimistic schedulability analysis of periodic tasks are detailed in [16, 24, 26]. Their pessimism level has been assessed in [28]. Unfortunately, the exact-case feasibility problem for self-suspending periodic tasks was shown to be NP-hard in [29].

In [19], the authors prove that the critical scheduling instant characterization is easier in the context of sporadic real-time tasks. They provide, for systems scheduled under a rate-monotonic priority assignment rule (RM), a pseudo-polynomial response-time test and propose slack enforcement policies to improve the schedulability of tasks with self-suspensions. Our approach addresses the problem for periodic tasks, and is not restricted to RM. Some other recent works on self-suspending tasks focus on the multiprocessor context [22, 23].

In this paper, we propose a timed-automata-based model to solve this scheduling problem. The timed automata approach has been already used to solve job shop scheduling problems [1, 10]. In [11], the authors present a model based on timed automata to solve real-time scheduling problems, our model can be seen as an extension of this model to take into account the possible suspensions of a task. The principal benefits of the timed automata approach is first that it proposes a model for both the scheduling and the formal verification of the system, and second that it manages to handle open problems, where no results

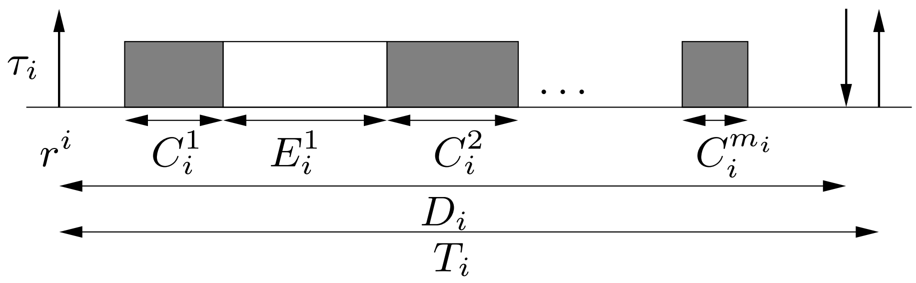


Figure 1: A self-suspending task τ_i

are currently known.

3 Self-Suspending Task Model

We consider the problem of scheduling a set $\Sigma = \{\tau_1 \dots \tau_n\}$ of n synchronous independent possibly self-suspending periodic tasks on one processor.

A self-suspending task is characterized by the tuple $\tau_i = (\mathcal{P}_i, T_i, D_i)$ where, T_i the period of task τ_i, $D_i \leq T_i$ its relative deadline and $\mathcal{P}_i$ its execution pattern. An execution pattern is a tuple $\mathcal{P}_i = (C_i^1, E_i^1, C_i^2, E_i^2, ..., C_i^{m_i})$ defining the durations of the computation and suspension steps where:

- m_i is the number of computation steps separated by $m_i - 1$ suspension steps for task τ_i,
- $C_i^k \in I\!N$ is the worst-case computation time of the k^{th} computation step of task τ_i,
- $E_i^k \in I\!N$ is the worst-case duration time of the k^{th} self-suspension step of task τ_i.

If a task τ_i has no suspension at step k, then the computation steps k and $k+1$ are merged as a single step with computation time $C_i^k = C_i^k + C_i^{k+1}$. This model and notations are inspired by existing literature on self suspending tasks [19, 22, 24, 29]. Figure 1 represents the self-suspending task model.

We call a task an *uncertain task* if the duration of its computation and suspension steps takes values within an interval. The execution pattern of an uncertain task becomes $\mathcal{P}_i = ([C_{i,l}^1, C_{i,u}^1], [E_{i,l}^1, E_{i,u}^1], [C_{i,l}^2, C_{i,u}^2], \dots [C_{i,l}^{m_i}, C_{i,u}^{m_i}])$ s.t. for a task τ_i:

- $C_{i,l}^k \in I\!N$ and $C_{i,u}^k \in I\!N$ are respectively the lower and upper bounds of the computation time of the k^{th} computation step and
- $E_{i,l}^k \in I\!N$ and $E_{i,u \in I\!N}^k$ are respectively the lower and upper bounds of the suspension duration of the k^{th} suspension step.

Note that if $C_{i,l}^1 = C_{i,u}^1$ and $E_{i,l}^1 = E_{i,u}^1$ the task τ_i is a regular task.

A scheduling problem $\Sigma = \{\tau_1, ..., \tau_n\}$ is feasible, if there exits a schedule for Σ where no task missed its deadline.

4 The Modeling step

In this section we present a timed automata based model for self-suspending tasks. This model is an improvement of the one we proposed in [3]. We first introduce the definition and the semantic for the basic timed automaton model.

4.1 Timed Automata

A Timed Automaton [4] is a model extending the classical automaton model with a set of variables, called clocks. Clocks are real variables evolving synchronously and continuously with time. Thanks to these variables, it is possible to express constraints over delays between transitions. Indeed, each transition of a timed automaton can be labeled by a clock constraint called guard which controls the firing of a transition. Clocks can be reset to zero in a transition and each location is constrained by a staying condition called invariant.

Formally, let $\mathcal{X}$ be a set of real variables called *clocks* and $\mathcal{C}(\mathcal{X})$ the set of clock constraints ϕ over $\mathcal{X}$ generated by $\phi ::= x\sharp c \mid x - y\sharp c \mid \phi \wedge \phi$ where $c \in I\!N$, $x, y \in \mathcal{X}$, and $\sharp \in \{<, \leq, \geq, >\}$. A *clock valuation* is a function $v : \mathcal{X} \rightarrow I\!R_+ \cup \{0\}$ which associates to every clock x its value $v(x)$. Given a value $d \in I\!R$ we write $v + d$ for the clock valuation associating with clock x the value $v(x) + d$. If r is a subset of $\mathcal{X}$, $[r = 0]x$ is the valuation v' such that $v'(x) = 0$ if $x \in r$, and $v'(x) = v(x)$ otherwise.

Definition 1 (Timed Automaton)**.** *A* timed automaton *(TA) is a tuple* $\mathcal{A} = (\mathcal{Q}, q_0, \mathcal{X}, \mathcal{I}, \Delta, \Sigma)$ *where* $\mathcal{Q}$ *is a finite set of states,* q_0 *is the initial state,* $\mathcal{X}$ *is a finite set of clocks,* $\mathcal{I} : Q \rightarrow \mathcal{C}(\mathcal{X})$ *is the invariant function,* $\Delta \subseteq \mathcal{Q} \times \mathcal{C}(\mathcal{X}) \times \Sigma \times 2^{\mathcal{X}} \times \mathcal{Q}$ *is a finite set of transitions and* Σ *is an alphabet of actions.*

A configuration of a timed automaton is a pair $(q, \mathbf{v})$ where q is a state and $\mathbf{v}$ a vector of clock valuations. The semantic of a timed automaton is given as a timed transition system with two kinds of transitions between configurations defined by the following rules:

- a discrete transition $(q, \mathbf{v}) \xrightarrow{a} (q', \mathbf{v}')$ where there exists $\delta = (q, \phi, a, r, q') \in \Delta$ such that $\mathbf{v}$ satisfies ϕ and $\mathbf{v}' = [r = 0]x$,

- a timed transition $(q, \mathbf{v}) \xrightarrow{d} (q, \mathbf{v} + d\mathbf{1})$ with $d \in I\!R_+$, where $\mathbf{v}$ and $\mathbf{v} + d\mathbf{1}$ satisfying $\mathcal{I}(q)$ the invariant of state q. We note $\mathbf{1}$ the unit $dim(\mathcal{X})$ vector $(1 \ldots 1)$.

Timed transitions represent the elapse of time in a state, and discrete transitions represent the ones between states. A timed transition is enabled if clocks valuations satisfy the invariant of the state and a discrete one is enabled if clocks valuations respect the guard on the transition. Then, we define a run in a timed automaton as a sequence of timed and discrete transitions.

A network of timed automata is the parallel composition of a set of timed automata. The parallel composition use an interleaving semantic where synchronous communication can be done using input actions denoted $a?$ and output actions denoted $a!$.

Note that this basic model can be extended to allow integer bounded variables whose values can be tested and assigned. However, a timed automaton augmented with a set $V = \{V_i \ldots V_n\}$ of bounded variables can be trans-

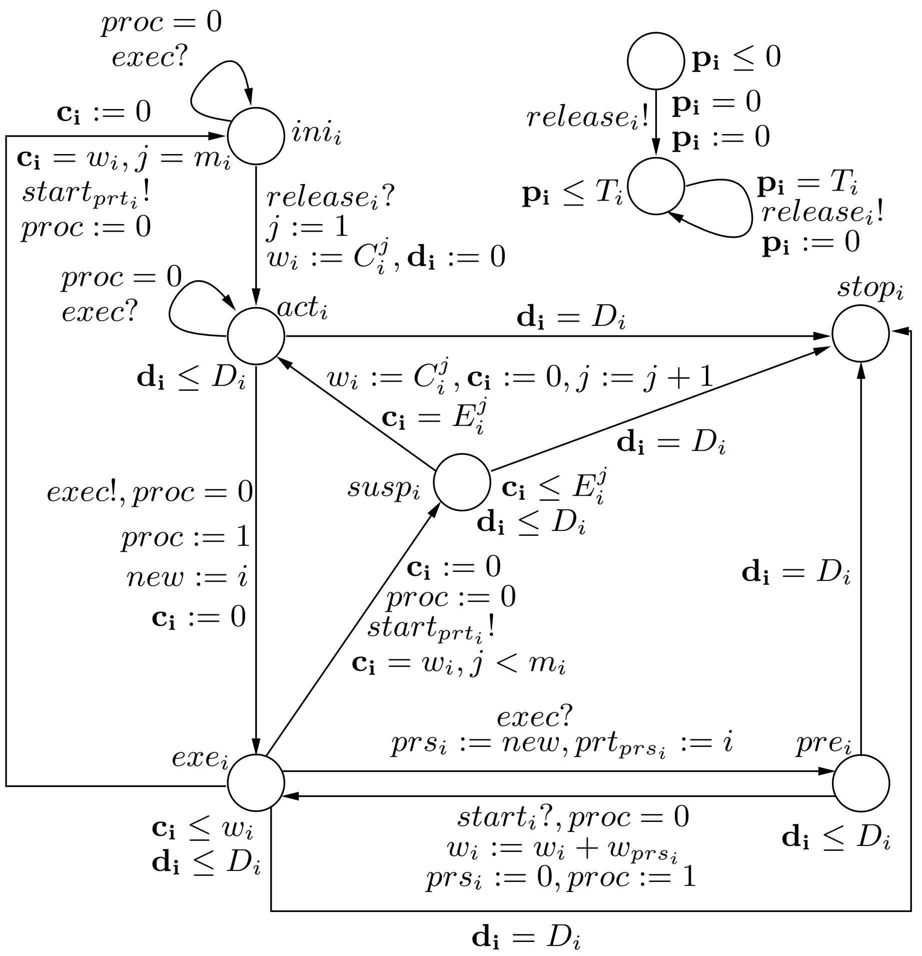


Figure 2: Periodic Self-Suspending Task Automaton

formed into a time automaton as defined in Definition 1 where the set of states $\mathcal{Q} = \mathcal{Q} \times [l_1, u_2] \times \ldots \times [l_n, u_n]$ where $[l_i, u_i]$ is the domain of variable V_i.

4.2 Self-suspending Timed Automaton

We model a self-suspending task $\tau_i = (\mathcal{P}_i, T_i, D_i)$ using a timed automaton $\mathcal{A}_i$ as shown in Figure 2. The automaton $\mathcal{A}_i$ has two clocks $\{c_i, d_i\}$ and a set $\mathcal{Q} = \{ini_i, act_i, exe_i, pre_i, susp_i, stop_i\}$ of states. A second timed automaton $\mathcal{T}_i$ is used to model the periodicity of the task. This automaton sends an action $release_i$ every period T_i. When a task is released, the automaton $\mathcal{A}_i$ captures the action $release_i$ and moves from state ini_i to state act_i. State act_i is the waiting state where the task is active but not yet executed. When a task starts its execution, the automaton moves to state exe_i and the clock c_i is reset to zero. Note that using a guard $proc = 1$, a task can be executed only if the processor is idle. The clock c_i is used to measure the response time of the task noted w_i. Using an invariant $c_i \leq w_i$ on state exe_i, a task cannot stay more than its response time in an execution state. When the task terminates ($c_i = w_i$), the automaton moves either to state ini_i if the task was executing its last step ($j = m_i$), or otherwise ($j < m_i$) to state $susp_i$. State $susp_i$ is the state modeling the suspension of a task. At step j, the task τ_i is suspended exactly E_i^j time units, this is modeled using an invariant $c_i \leq E_i^j$ and a guard $c_i = E_i^j$ on the transition from state $susp_i$ to state act_i.

In our model, a task can be preempted only if an other task is assigned to the processor. This preempting task noted *new* can be a new instance of a task or a task terminating its suspension step. Thus, the preemption of a task synchronizes with the execution of an other task using the action *exec*!. When the task τ_i is preempted, the automaton moves to state pre_i and the variable prs_i records the identifier of the preempting task. A task cannot resume if its preempting task has not terminate yet, indeed, we are restricting ourselves to fixed-job priority schedules. In a fixed-job priority schedule, when the relative priority assignment between two jobs has been decided, it cannot change. EDF is an example of fixed-job priority scheduling algorithm. Least Laxity First (LLF) is a well known counter example.

Every termination of a task is synchronized using an action $start_i$ with the resuming of the task it preempts (if this task exits). Then, when the preempted task τ_i resumes, the response time of the preempting task pre_i is added to the response time of τ_i.

When a task τ_i is activated, the clock d_i is reset to zero. If this clock reaches the deadline of the task before its completion, the automaton moves to state $stop_i$.

Note that the proposed model is a basic timed automaton extended with a finite set of integer bounded variables $\{new, prs_i, prt_i, proc, j, w_i\}$. These variables can be read, written, and are subject to common arithmetic operations. As shown in the presentation of the model, the variables $\{new, prs_i, prt_i\}$ are never incremented or decremented and represents instances of a task, thus they are lower bounded by 1 and upper bounded by n. The variable *proc* is a boolean variable indicating if the processor is idle or not. The variable j indicates the step number of a task, this variable is settled to 1 in the transition from ini_i to act_i and never decreases, thus it is lower bounded by 1 and when the variable reaches m_i, the automaton moves to state ini_i, thus the variable is upper bounded by m_i. Finaly, w_i measures the response time of a task τ_i. The variable w_i is initialized to C_i^j and never decreases, so it is lower bounded by C_i^j, this case is reached if the task is not preempted. Variable w_i is upper bounded by $C_i^j + D_i$. Indeed, in our model, if a task is preempted when the tasks resumes, its responses time is augmented by the response time of its preempting task, nowever, the duration of the responses times of all the preempting tasks cannot exceed the deadline of the task τ_i, otherwise the clock d_i will reach the deadline and the automaton moves and stays in state $stop_i$.

5 Feasibility and Schedulability using Model Checking

Model checking is a method for automatic verification where the system is modeled using a formal model M and the correctness property is stated with a formal specification language ϕ. Given a model M and a property ϕ, model checkers are used to automatically decide whether M satisfied ϕ or not.

In this section, we present how we use CTL [17] model checking to test the feasibility of a task set and its schedulability with PFP and EDF.

CTL properties are generated using the following grammar:

$$\phi ::= p|(\neg\phi)|(\phi \wedge \phi)|(\phi \vee \phi)|AX\phi|EX\phi|AG\phi|EG\phi|A[\phi U \phi]|E[\phi U \phi]$$

where p is a set of atomic formulas. CTL formulas are interpreted on a transition system s.t. the initial state s_0 satisfies: $AG\phi$ iff in all the paths starting at s_0 all the states satisfy ϕ, $EG\phi$ iff there exists a path starting at s_0 where all the states satisfy ϕ, $AX\phi$ iff in all the paths starting at s_0 in the next state ϕ is satisfied, $EX\phi$ iff there exits a path starting at s_0 where in the next state ϕ is satisfied, $E[\phi_1 U \phi_2]$ iff there exists a path starting at s_0 where ϕ_1 is satisfied until ϕ_2 is satisfied and $A[\phi_1 U \phi_2]$ iff for all the paths starting at s_0 ϕ_1 is satisfied until ϕ_2 is satisfied.

5.1 Feasibility

Let $\Sigma = \{\tau_1 \ldots \tau_n\}$ be a finite set of self-suspending tasks. We associate to every task τ_i a self-suspending automaton $\mathcal{A}_i$. We note $\mathcal{A}_\mathcal{P}$ the parallel composition of the automata $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$. We add to our model a global clock t which is never reset.

We consider a configuration of $\mathcal{A}_\mathcal{P}$ as a tuple $(q, \mathbf{v}, v(t))$ where $q = (s_1, \ldots s_n)$ and $\mathbf{v} = (v(c_1), v(d_1), \ldots v(c_n), v(d_n))$ s.t. $v(t)$ is the valuation of clock t and $\forall i \in [1, n]$:

1. s_i is a state of the automaton $\mathcal{A}_i$,
2. $v(c_i), v(d_i)$ are the clocks valuations of c_i and d_i respectively.

The configurations of the timed transition system of $A_\mathcal{P}$ represent the possible configurations of a task: active, executing, preempted, suspended, stopped. Note that for the sake of clarity we omit to mention in a configuration the values of the integer variables and the valuation of the period automaton clock.

The following proposition provides a feasibility test for the scheduling problem $\Sigma = \{\tau_1 \ldots \tau_n\}$.

Proposition 1 (feasibility)**.** *Let $\Sigma = \{\tau_1 \ldots \tau_n\}$ be a set of self-suspending tasks. Σ is feasible iff the network $A_\mathcal{P}$ modeling Σ satisfies the CTL Formula 1*

$$\phi_1 : EG\neg(\bigvee_{i\in[1,n]} stop_i) \tag{1}$$

Proposition 1 states that the self-suspending problem is feasible iff there exists an infinite run ξ in $A_\mathcal{P}$ where all the configurations satisfy the property

$EG\neg(\bigvee_{i\in[1,n]} stop_i)$. We call ξ a feasible run. In other words, ξ is a run where no configuration contains a state $stop_i$.

The proposition is justified by the fact that in each timed automaton $\mathcal{A}_i$, the state $stop_i$ is reached iff the clock d_i reached the deadline D_i. Suppose that the scheduling problem Σ is feasible and Formula 1 is not satisfied. If the problem is feasible, then there exists a schedule where all the instances of all tasks never miss their deadline. This schedule corresponds in the self-suspending automaton to a feasible run. This contradicts the hypothesis that Proposition 1 is not satisfied. Suppose now that Formula 1 is satisfied and the scheduling problem is not feasible. If the formula is not satisfied, then all the infinite runs are not feasible i.e all the infinite runs lead to a $stop_i$ state. This contradicts the fact that the problem is feasible, the contradiction comes from the fact that the automaton capture all possible behaviors of task instances.

An on-line scheduling algorithm can be obtained using a feasible run satisfying Formula 1 if one exists. To compute this algorithm, we first check Formula 1 to generate a feasible infinite run if one exists. Model checking for timed automata is decidable but PSPACE-complete [5], however, in our approach, the feasible run is computed off line. Then, given a feasible run of the network $A_{\mathcal{P}}$, a schedule can be derived. This schedule defines the rules controlling when and how transitions between different configurations of a task occur. This on-line scheduling algorithm can be computed as a scheduling function $F_{Sched} : \{0 \dots t^*\} \rightarrow \{1 \dots n\} \cup \{\epsilon\}$ s.t. if:

1. $F_{Sched}(t) = i \in \{0 \dots n\}$ then task τ_i is executing at time t,
2. $F_{Sched}(t) = \epsilon$ then the processor is idle at time t.

The time point t^* is the valuation $v(t)$ of the first configuration of ξ where the task set is again in its initial configuration. This configuration is reached at least at the hyper-period, the least common multiple of the periods of all tasks. We then just have to repeat this algorithm to obtain an infinite schedule.

Note that using our model, a computed schedule can be a non work-conserving one. Work-conserving schedules are ones where the processor can be idle only if there is no ready task. Indeed, in the network $A_{\mathcal{P}}$, the processor can be idle ($F_{Sched}(t) = \epsilon$) if no task is active, but also if there exits an active but not suspended task and no other task is in its preemption state. Scheduling theory often implicitly addresses problems for work-conserving schedulers because leaving the processor idle when tasks are ready seems to result in a resource wasting. To produce work-conserving schedules, we use Formula 2 rather than Formula 1 to compute the scheduling function F_{Sched}.

$$\begin{aligned}\phi_2 : EG\neg\Big(\bigvee_{i\in[1,n]} stop_i \bigvee_{i\in[1,n]} &\big((act_i \wedge d_i > 0 \wedge c_i > 0) \\ \bigwedge_{j\neq i\in[1,n]} ((act_j \wedge d_j > 0 \wedge c_j > 0) &\vee (susp_j \wedge c_j > 0) \\ &\vee (ini_j \wedge c_j > 0))\big)\Big)\end{aligned} \tag{2}$$

Formula 2 forbids executions where an active task is not scheduled and the processor is idle. Indeed, this formula is not satisfied if there exists a run where a task τ_i is active since a time $t > 0$ ($act_i \wedge d_i > 0 \wedge c_i > 0$) and all the other different tasks τ_j with $j \neq i$ are not executed since a time $t > 0$ ($(act_j \wedge d_j > 0 \wedge c_j > 0) \vee (susp_j \wedge c_j > 0) \vee (ini_j \wedge c_j > 0)$), in other words Formula 2 is not satisfied if there is no work-conserving schedule.

5.2 Schedulability

To test schedulability according to a given a fixed-job priority scheduling policy, one can model the scheduling policy in the CTL checked formula.

To test fixed priority (PFP) schedulability, we have to test if there exists a feasible infinite run where some configurations are forbidden: the ones where a task is executing while a greater priority task is not.

Proposition 2 (PFP Schedulability)**.** *Let $\Sigma = \{\tau_1 \dots \tau_n\}$ be a set of self-suspending tasks sorted according to the priorities of the tasks. Σ is schedulable according to PFP iff the network $A_{\mathcal{P}}$ modeling Σ satisfies the CTL Formula 3.*

$$\begin{aligned}\phi_3 : EG\neg\Big(&\bigvee_{i\in[1,n-1]} \bigvee_{j\in[i+1,n]} (act_i \wedge c_i > 0 \wedge d_i > 0 \wedge exe_j) \\ &\bigvee_{i\in[1,n-1]} \bigvee_{j\in[i+1,n]} (pre_i \wedge exe_j)\Big) \wedge \phi_2\end{aligned} \tag{3}$$

Formula 3 states that the problem is schedulable according to PFP iff there exists a feasible run where, in all the configurations, a task τ_j cannot be in its execution state exe_j if a highest priority task τ_i ($i < j$) is active since a time $t > 0$ ($act_i \wedge c_i > 0 \wedge d_i > 0$) or preempted ($pre_i$).

Using this approach, we can also test the EDF schedulability.

Proposition 3 (EDF Schedulability)**.** *Let be $\Sigma = \{\tau_1 \dots \tau_n\}$ a set of self-suspending tasks. Σ is schedulable according to EDF iff the network $A_{\mathcal{P}}$ modeling Σ satisfies the CTL Formula 4.*

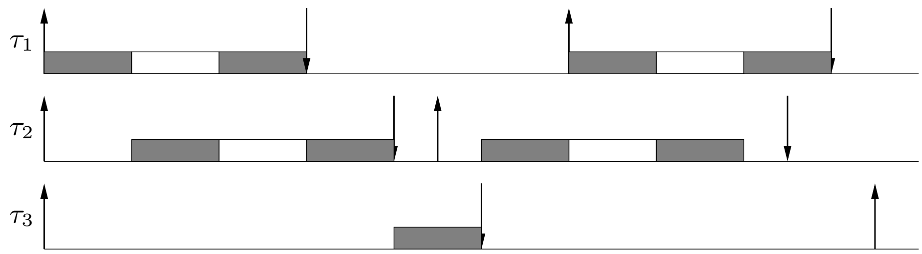
$$\begin{aligned}\phi_4 : EG\neg\Big(&\bigvee_{i\in[1,n]} \bigvee_{j\neq i\in[1,n]} (act_i \wedge c_i > 0 \wedge d_i > 0 \wedge exe_j \wedge p_{ij}) \\ &\bigvee_{i\in[1,n]} \bigvee_{j\neq i\in[1,n]} (pre_i \wedge exe_j \wedge p_{ij})\Big) \wedge \phi_2\end{aligned} \tag{4}$$

p_{ij} is a state of an observer automaton reachable when $d_i - d_j > D_i - D_j$ with d_i and d_j the deadline clocks of tasks τ_i and τ_j respectively.

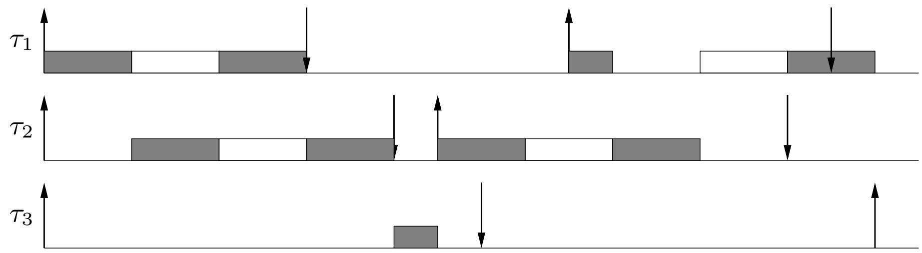
Under the EDF scheduling policy, the processor is assigned to a task if it is the closest to its deadline. Formula 4 states that the problem is schedulable according to EDF iff there exists a feasible run where, in all the configurations, a task cannot be in its execution state (exe_i) if a task τ_j with a closer deadline ($d_i - d_j > D_j - D_j$) is active since a time $t > 0$ ($act_i \wedge c_i \wedge d_i$) or preempted ($pre_i$).

6 Sustainabability

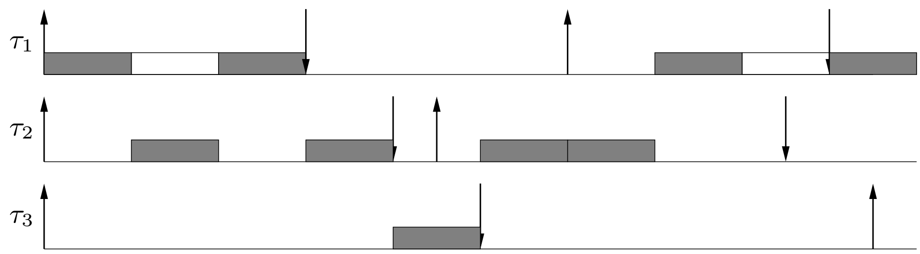
The schedulability of a task set with a given algorithm is said sustainable w.r.t. a parameter when a schedulable task set remains schedulable when this parameter



(a) EDF Feasible Schedule



(b) EDF Unfeasible if $C_1^3 = C_1^3 - 1$



(c) EDF Unfeasible if $E_1^2 = E_1^2 - 2$

Figure 3: Unsustainable EDF Schedule

is changed in a positive way. The sustainability is an important property since it permits to study the worst case scenario. As mentioned in the introduction, schedulability is not sustainable w.r.t execution and suspensions durations for the self-suspending scheduling problem. As an example, let $\Sigma = \{\tau_1, \tau_2, \tau_3\}$ be a set of self-suspending tasks where $\tau_1 = ((2, 2, 2), 6, 12)$, $\tau_2 = ((2, 2, 2), 8, 9)$ and $\tau_3 = ((0, 2, 0), 10, 19)$. Figure 3(a) represents the work conserving EDF schedule of interval $[0, 20]$ for this problem. In Figures 3(b) and 3(c), one can see that the diminution of either execution or suspension times leads to new deadline misses for τ_1.

In this section, we show how to prove that a task set is sustainable using timed game automata.

A Timed game automaton (TGA) [25] is an extension of the time automaton model where the set of transitions is split into controllable (Δ_c) and uncontrollable (Δ_u) transitions. This model defines the rules of a game between a controller (mastering the controllable transitions) and the environment (mastering the uncontrollable transitions). Given a timed game automaton and a logic formula, solving a timed game consists in finding a strategy f s.t. a TGA supervised by f always satisfies the given formula whatever are the actions chosen by the environment. A strategy is formally a partial mapping from the set of runs of the TGA to the set $\Delta_c \cup \{\lambda\}$ s.t. for a finite run ξ:

- if $f(\xi) = e \in \Delta_c$ then the controller has to execute the transition e from the last configuration of ξ,
- if $f(\xi) = \lambda$ then the controller has to wait in the last configuration of ξ.

It has been shown that solving a timed game is a decidable problem [25].

In our task model, if a task is an uncertain task its execution and suspension durations can be bounded within intervals. In the remaining of the paper, we say that the schedulability is sustainable when the task set is feasible with all the possible values in the intervals. By extension we say that an algorithm is sustainable when the schedulability of a task set with this algorithm is sustainable.

To check sustainability, we introduce first a timed game automaton modeling a game between the environment and a scheduler. The environment fixes the execution and the suspension times of a task, and the scheduler decides to execute or preempt a task.

6.1 Sustainable Schedulability w.r.t Duration of Suspension

Let us first consider the uncertain scheduling problem where the execution times of a task are given as constants C_i^j representing the worst case execution times, but the suspensions of a task are restricted to be bounded within an interval $[E_{i,l}^j, E_{i,u}^j]$.

A self-suspending task τ_i is modeled using a timed game automaton as represented in Figure 4, this model is almost similar to the timed automaton model presented in Section 3.

Considering that the suspension durations are controlled by the environment, the transition from state $susp_i$ to state act_i is an uncontrollable transition, while the start and preemption transitions are controllable ones fixed by the scheduler.

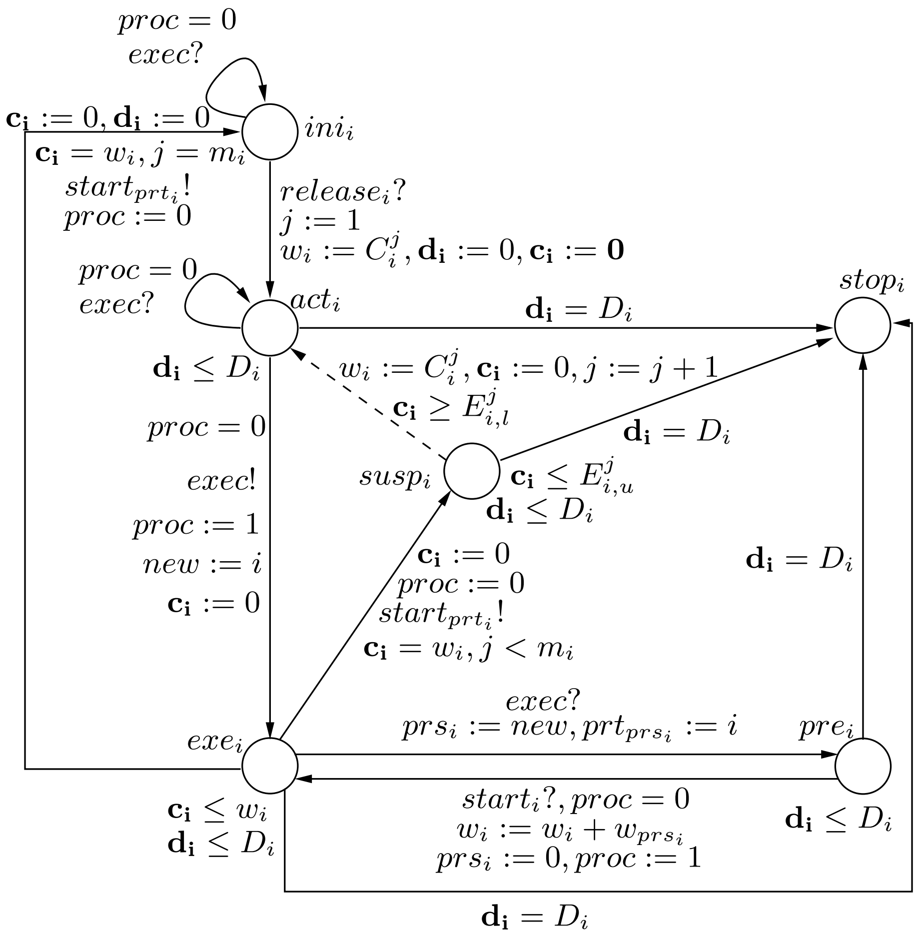


Figure 4: Self-Suspending Task TGA with Uncertain Suspensions. Uncontrollable transitions are represented using dashed lines.

Using a guard $c_i \geq E_{i,l}^j$ from state $susp_i$ to state act_i and an invariant $c_i \leq E_{i,u}^j$ on state $susp_i$, the duration of each suspension step j is no more fixed but can have any possible value in the interval $[E_{i,l}^j, E_{i,u}^j]$.

Proposition 4 (Sustainability)**.** *Let $\Sigma = \{\tau_1 \dots \tau_n\}$ be a finite set of uncertain self-suspending tasks. A sustainable scheduling algorithm exists for Σ iff there exists a strategy f s.t the timed game automata network $A_{\mathcal{P}}$ modeling Σ supervised by f satisfies the CTL Formula 5.*

$$\phi_5 : AG\neg(\bigvee_{i\in[1,n]} stop_i) \tag{5}$$

The strategy f is called a scheduling strategy of Σ.

Proof. Let Σ be a finite set of uncertain self-suspending tasks and $A_{\mathcal{P}}$ the timed game automata network modeling Σ. Formula 5 states that there exists a strategy function f s.t. for every configuration of $A_{\mathcal{P}}$ and every possible suspension duration, there is a way to avoid $stop_i$ states.
($\Rightarrow$) Suppose that: (1) A sustainable scheduling algorithm exists for Σ and (2) for every possible strategy f the network $A_{\mathcal{P}}$ controlled by f does not satisfy Formula 5. If there exists a sustainable scheduling algorithm for Σ, then there exits a feasible schedule S s.t. whatever are the suspension durations of each step j of each task τ_i within the interval $[E_{i,l}^j, E_{i,u}^j]$, the schedule remains feasible. This feasible schedule defines a policy deciding at each possible configuration of the task set to: execute a task, preempt a task, let the processor idle or stay in the same configuration. Note that this schedule is a set of possible feasible schedules corresponding to each possible duration of each suspension. According to the fact that all the durations (execution and suspension) of tasks are integer and considering synchronous activation of tasks, the set of possible configurations of the task set is finite and corresponds to all the possible configurations in the hyper period of the scheduling problem.

To formalize this policy, let us define the tuple $(r, time)$ as the state of the schedule S where:

1. $r = (r_1 \dots r_n)$ is a vector with $\forall i \in [1,n]$ r_i is a possible configuration of task τ_i: $r_i \in \{\{inactive \cup_{j\in[1,m_i]} \{active_j, execution_j, preempted_j, suspended_j\}\}$ and,

2. $time = (time_1 \dots time_n)$ is a vector with $\forall i \in [1,n]$ $time_i = (time1_i, time2_i)$ is a tuple where

   (a) (i) $time1_i$ is the time delay since the task τ_i has terminate its last execution step if $r_i = inactive$ (ii) $time1_i$ is the time delay since the step j of task τ_i has been activated if $r_i = active_j$ (iii) $time1_i$ is the time delay since the task τ_i has been executing its step j if $r_i = execution_j$ or $preempted_j$ (iiii) $time1_i$ is the time delay since the step j of task τ_i has been suspended if $r_i = suspended_j$.

   (b) (i) $time2_i$ is the time delay since the step j of task τ_i has been activated if $r_i = execution_j$ or $suspended_j$ or $preempted_j$ and (ii) if $time2_i$ is not defined $time2_i = time1_i$.

We formalized the scheduling policy as a partial mapping F_S from the set of state tuples $\{(r^1, time^1)\dots(r^m, time^m)\}$ to $\{(r^1, time^1)\dots(r^m, time^m)\}$ where m is the number of tuple states and n is the number of tasks s.t if: $F_S((r, time))$ $= (r', time')$ then move from the configuration $(r, time)$ of the schedule to the configuration $(r', time')$. This mapping is a partial mapping, because in some configurations the schedule S do not have a policy but the configuration has to move to a new one where a suspension has terminated.

Let us construct now a set of configurations $Q_S = \{(q^i, \mathbf{v}^i)\}$ of $A_{\mathcal{P}}$ s.t. for every possible state tuple $(r^i, time^i)$ of the schedule S we associate a configuration $(q^i, \mathbf{v}^i)$ of $A_{\mathcal{P}}$ with $q^i = (q^i_1 \dots q^i_n, nb^i_1 \dots nb^i_n)$ where q^i_j is the state of the automaton of the task τ_j in the configuration q^i, nb^i_j is the step of the task τ_j in the configuration q^i and $\mathbf{v}^i$ is the vector of clock valuations in the configuration q^i of the network $A_{\mathcal{P}}$ s.t. $\forall i, k$ if :

- $r^i_k = inactive$ then $q^i_k = ini_k$, $nb^i_k = m_k$ and $v^i(c_k) = v^i(d_k) = time1^i_k$,
- $r^i_k = active_j$ then $q^i_k = act_k$ and $nb^i_k = j$ and $v^i(c_k) = v^i(d_k) = time1^i_k$,
- $r^i_k = execution_j$ then $q^i_k = exe_k$ and $nb^i_k = j$, $v^i(c_k) = time1^i_k$ and $v^i(d_k) = time2^i_k$,
- $r^i_k = preempted_j$ then $q^i_k = pre_k$ and $nb^i_k = j$, $v^i(c_k) = time1^i_k$ and $v^i(d_k) = time2^i_k$,
- $r^i_k = suspended_j$ then $q^i_k = susp_k$ and $nb^i_k = j$, $v^i(c_k) = time1^i_k$ and $v^i(d_k) = time2^i_k$.

The scheduling policy F_S can be used to compute a scheduling strategy f_S that mimics the decisions of the sustainable schedule S.

The scheduling strategy f_S is a partial mapping form the set of runs of $A_{\mathcal{P}}$ to the set $\{\Delta_c, \lambda\}$ where Δ_c is the set of controllable transitions of $A_{\mathcal{P}}$. Let us note q_e the last configuration of a run ξ and $(r_e, time_e)$ the state tuple corresponding to the configuration q_e. The strategy f_S is defined as follows:

- if $F_S((r_e, time_e)) = (r'_e, time'_e)$ and $r'_e = r_e$ then $f_S(\xi) = \lambda$ and
- if $F_S((r_e, time_e)) = (r'_e, time'_e)$ and $r'_e \neq r_e$ then $f_S(\xi) = tr \in \Delta_c$ where tr is a controllable transition leading to the configuration corresponding to the tuple state $(r'_e, time'_e)$.

The strategy f_S starts at the initial configuration of $A_{\mathcal{P}}$ and then mimics the decisions of the scheduler S, thus it cannot reach a configuration with no equivalence in the set of tuples state of S knowing that the set of tuples represents all the possible configurations of the task set.

According to the fact that S remains feasible whatever are the choices of the environment no task will miss its deadline and knowing that a state $stop_i$ is reached if a task mises its deadline, we conclude that if f_S is used to execute the network $A_{\mathcal{P}}$, none of the automata of $A_{\mathcal{P}}$ will reach a state $stop_i$, this contradicts the hypothesis (2).

($\Leftarrow$) Suppose now that: (3) No sustainable scheduling algorithm exists for Σ and (4) there exists f a scheduling strategy of Σ. According to hypothesis (4), f is a scheduling strategy of Σ, thus the network $A_{\mathcal{P}}$ controlled by f never reaches a configuration with a *stop* state. The strategy f is then a partial mapping from the set of finite runs of $A_{\mathcal{P}}$ to the set $\{\Delta_c, \lambda\}$ where Δ_c is the set of controllable transitions of $A_{\mathcal{P}}$ i.e. the transitions representing an execution, a preemption, an activation, a suspension or a termination of a task. For a finite run ξ:

- if $f(\xi) = tr \in \Delta_c$ then the controller has to: execute, preempt, suspend or terminate a task from the last configuration of ξ,
- if $f(\xi) = \lambda$ then the controller has to wait in the last configuration of ξ i.e continue the execution of a task or let the processor idle.

Using this stategy, one can compute a schedule as presented in Subsection 6.4. This schedule is feasible (it never reaches a state stop) whatever are the durations of the suspensions, so this schedule is sustainable and thus it contradicts the hypothesis (3).

∎

Using Formula 6 rather than Formula 5 one can prove that there exits a work-conserving sustainable scheduling algorithm. This Formula is similar to Formula 2.

$$\begin{aligned}\phi_6 : AG\neg\Big(\bigvee_{i\in[1,n]} stop_i \bigvee_{i\in[1,n]} \big((act_i \wedge d_i > 0 \wedge c_i > 0)\\ \bigwedge_{j\neq i\in[1,n]} ((act_j \wedge d_j > 0 \wedge c_j > 0) \vee (susp_j \wedge c_j > 0)\\ \vee(ini_j \wedge c_j > 0))\big)\Big)\end{aligned} \tag{6}$$

6.2 Sustainable PFP Scheduler

Definition 2 (PFP Strategy). *Let $\Sigma = \{\tau_1 \dots \tau_n\}$ be a set of self-suspending tasks sorted according to the task priorities. A scheduling strategy f of Σ is called a PFP strategy iff: if $f(\xi) = e \in \Delta_c$ and e is a transition from state act_i to state exe_i, then the task τ_i is the highest priority active task.*

Proposition 5 (PFP Sustainability). *Let $\Sigma = \{\tau_1 \dots \tau_n\}$ be a set of self-suspending tasks sorted according to the priorities of the tasks. A PFP work-conserving algorithm is sustainable for Σ iff there exists a PFP strategy f s.t the timed game automata network $A_{\mathcal{P}}$ modeling Σ supervised by f satisfies the safety CTL Formula 7.*

$$\begin{aligned}\phi_7 : AG\neg(\bigvee_{i\in[1,n-1]} \bigvee_{j\in[i+1,n]} (act_i \wedge c_i > 0 \wedge d_i > 0 \wedge exe_j)\\ \bigvee_{i\in[1,n-1]} \bigvee_{j\in[i+1,n]} (pre_i \wedge exe_j)\big) \wedge \phi_6\end{aligned} \tag{7}$$

As for Formula 3, in Formula 7, the configurations where a task τ_j is in its execution state exe_j and a highest priority task τ_i $(i < j)$ is active since a time $t > 0$ $(act_i \wedge c_i > 0 \wedge d_i > 0)$ or preempted (pre_i) are forbidden. If a strategy f satisfies Formula 7 then it is a PFP strategy because this strategy cannot choose a transition from a state act_i to a state exe_i if τ_i has not the highest priority among the active tasks, otherwise a forbidden state is reached. In addition to the fact that the produced strategy is a PFP one, this strategy is a work-conserving strategy where no configuration reaches a stop state because of the part ϕ_6 of the Formula. So, using this work-conserving PFP strategy we can compute a PFP work-conserving algorithm. The policy of this algorithm is to execute the highest priority task if a new task is active and no task will misses its deadline because the strategy using the same policy never reached a stop state whatever are the suspension durations. Thus, this algorithm is sustainable.

In the other sens, if a work-conserving PFP algorithm S is sustainable (w.r.t given intervals of possible suspension durations) we can derive f_{PFP} a PFP work-conserving strategy satisfying Formula 7. This strategy f_{PFP} is a partial mapping from the set of finite runs of $A_{\mathcal{P}}$ to the set $\{\Delta_c, \lambda\}$ where Δ_c is the set of controllable transitions of $A_{\mathcal{P}}$. Let q_e be the last configuration of ξ:

- if a set of tasks is active in q_e then $f_{PFP}(\xi) = tr \in \Delta_c$ where tr is the controllable transition reaching the execution state of the highest priority task among all the active or executing tasks,
- if no task is active in q_e then $f_{PFP}(\xi) = \lambda$.

Note that all the other controllable transitions are taken when they are enabled because of the guards and invariants constraining these transitions.

This strategy satisfies Formula 7 because (1) due to the definition of the strategy f_{PFP}, states forbidden by Formula 7 are not reached and forbidden states cannot be reached when the environment fixes the termination of a suspension, this can only create new configurations with a new active task, (2) if a task is active and has the highest priority, the defined strategy f_{PFP} moves to the execution state of this task, so the strategy is work conserving because no idle times are inserted if a task is active and (3) no configuration with a state stop is reached because no task will mises its deadline whatever are the duration of suspension because the scheduling algorithm S fixing the choices of f_{PFP} is sustainable.

As a remark, the results of this subsection can be extended to EDF algorithm by fixing the priorities according to an EDF policy instead of fixed priority policy.

6.3 Sustainable Schedulability w.r.t Duration of Suspension and Execution

Let us consider now an uncertain task where both execution and suspension durations can be uncertain. In this case, the model of Section 3 is no more applicable. More precisely, the modeling of preemption is no more valid.

Indeed, the execution step duration is not known before the execution of the system but fixed by the environment. Therefore, the response time w_i of a preempted task τ_i can not be calculated, since we do not know precisely the duration of its preempting task. Preemption could however be modeled using stopwatch automata, a model where clocks can be stopped. In this model, the clock c_i is used to measure the duration of a task, and if the task is preempted,

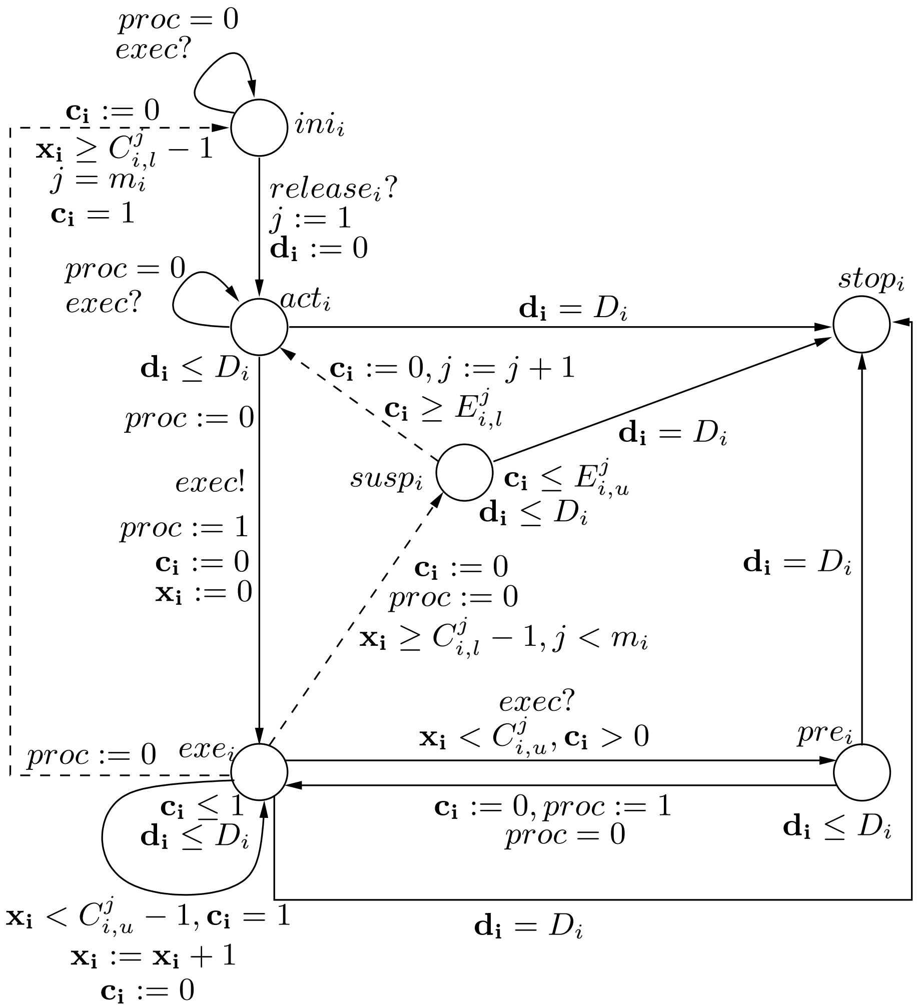


Figure 5: Uncertain Self-Suspending Task TGA. Uncontrollable transitions are represented using dashed lines.

the clock c_i is stopped. Unfortunately, model checking is known to be undecidable on this model in the general case [14, 15].

Thus we propose a new model to deal with preemption where the duration of a task is discretized as shown in Figure 5.

In this timed game automaton, to compute the execution time of a task we use the clock c_i plus an integer variable x_i as follows. The automaton can stay in the execution state exe_i exactly one time unit and the variable x_i keeps track of how many time units have been performed. This is done using an invariant $c_i \leq 1$ on state exe_i and a loop transition that increments x_i. The guard $x_i < C^j_{i,u} - 1$ on the loop transition restricts the duration of an execution step j to be upper bounded by $C^j_{i,u}$. The automaton can leave state exe_i if the guard $x_i < C^j_{i,l} - 1$ is satisfied, thus the duration of an execution step j is lower bounded by $C^j_{i,l}$.

The termination of execution steps is controlled by the environment, hence the transitions from exe_i to $susp_i$ and from exe_i to ini_i.

6.4 Sustainable Scheduler

If a task set Σ has been proven to be not PFP nor EDF sustainable, we can nevertheless define a sustainable scheduling algorithm if one exits.

Given a network of timed game automata modeling Σ, finding a sustainable scheduling algorithm consists in the construction of a feasible strategy if one exists. Such strategy is finite [12], but can be very huge since the upper bound complexity of reachability games on timed game automata has been proved to be EXPTIME [12].

The strategy can be stored as a table of possible reachable configurations, where possible transitions are mentioned for every configuration. The set of configurations is infinite but a finite representation of the state space of the transition system can be obtained using clock zones [9, 13].

Then an on-line sustainable scheduler is an algorithm that executes the pre-computed strategy.

This is formalized by Algorithm 1 where:

- $q_0 = (ini_1, \ldots ini_n)$ is the initial configuration,
- q^j_i a state of the automaton of task τ_j in the configuration $(q_i, \mathbf{v}_i, t_i)$,
- t_i is the the valuation of a global clock t in the configuration $(q_i, \mathbf{v}_i, t_i)$,
- Δ^j_c is a controllable transition in the automaton of task τ_j.

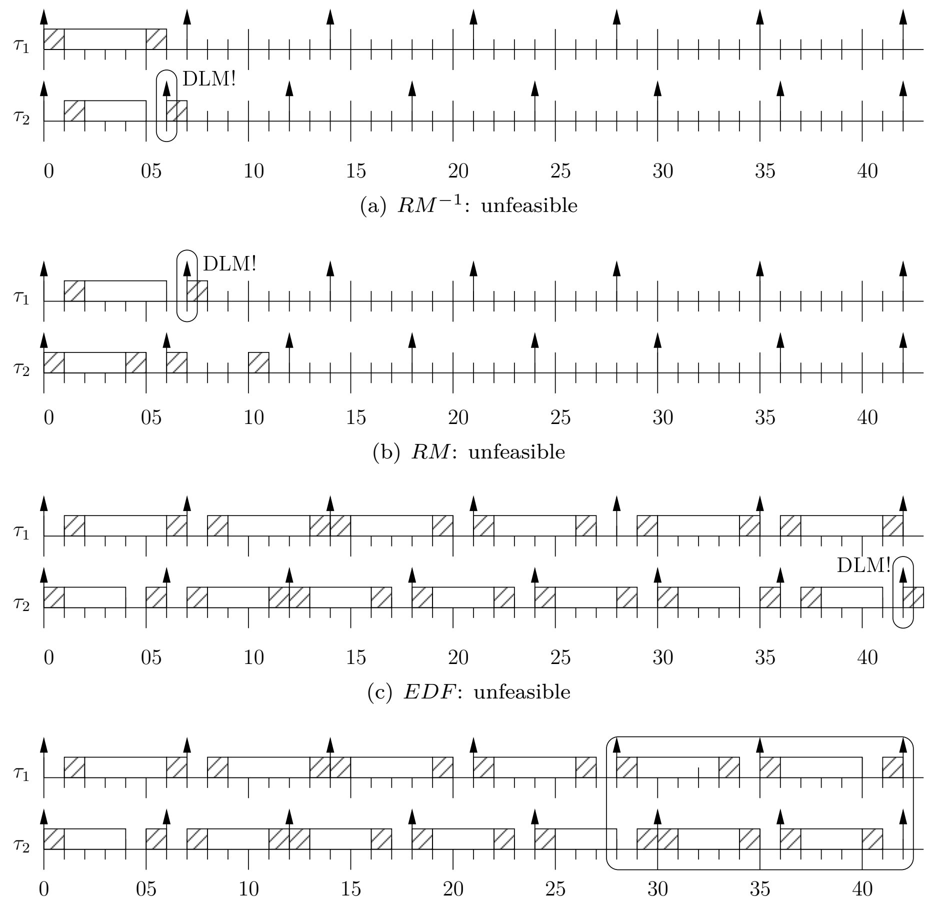
According to the actual configuration, the scheduling algorithm can decide 1) to stay in this configuration, i.e to continue the execution of a task or let the processor idle (lines 3-5) ; or 2) to execute, preempt or suspend a task (lines 6-17). Finally when an execution or a suspension terminates, the algorithm computes the new configuration (lines 18-24).

7 Experiments

We used Uppaal [21] and Uppaal-Tiga [6] to implement our model [2]. We present in this section two examples. The first is composed by two regular self-

Algorithm 1 Scheduling Strategy Algorithm

$(q, \mathbf{v}, v(t)) \leftarrow (q_0, \mathbf{v}_0, t_0), t_0 \leftarrow 0$
while $q \neq q_0$ or $v(t) = 0$ **do**
  while $f((q, \mathbf{v}, v(t))) = \lambda$ or no task finished or no end of suspension **do**
    Wait: increase $\mathbf{v}$ and $v(t)$
  end while
  if $f((q, \mathbf{v}, v(t))) = tr \in \Delta_c^j$ **then**
    $(q_k, \mathbf{v}_k, t_k)$ is the successor of $(q, \mathbf{v}, v(t))$ while taking the transition tr
    if $\exists q_k^j \neq q^j$ and $q_k^j = exe_j$ **then**
      execute task τ_j at time t_k
    end if
    if $\exists q_k^j \neq q^j$ and $q_k^j = pre_j$ **then**
      preempt task τ_j at time t_k
    end if
    if $\exists q_k^j \neq q^j$ and $q_k^j = susp_j$ **then**
      suspend task τ_j at time t_k
    end if
  end if
  if a task τ_j has terminate an execution step **then**
    $(q_k, \mathbf{v}_k, t_k)$ is the configuration $(q, \mathbf{v}, v(t))$ where $q_k^j \leftarrow ini_j, v_k(c_j) \leftarrow 0$
  end if
  if a task τ_j has terminate a suspension step **then**
    $(q_k, \mathbf{v}_k, t_k)$ is the configuration $(q, \mathbf{v}, v(t))$ where $q_k^j \leftarrow act_j, v_k(c_j) \leftarrow 0$
  end if
  $(q, \mathbf{v}, v(t)) \leftarrow (q_k, \mathbf{v}_k, t_k)$
end while



(a) RM^{-1}: unfeasible

(b) RM: unfeasible

(c) EDF: unfeasible

(d) $TAAS$: feasible

Figure 6: Feasible schedule exists but neither pfp or EDF is able to find it

suspending tasks. The second is composed by three uncertain self-suspending tasks. Fig. 6 and Fig. 7 present the obtained results. White squares represent suspension durations and hatched ones execution durations.

7.1 Experiment 1 (Regular tasks)

In this experiment, we have modeled the system $\Sigma = \{\tau_1, \tau_2\}$ with $\tau_1 = (0, (1,4,1), 7,7)$ and $\tau_2 = (0,(1,3,1),6,6)$. We have first used Formula 3 with RM priority assignments. The property is not verified, this result permits us to conclude that the task set is not schedulable according to RM. The same result is obtained with the inversed priority assignment. Sub-Fig. 6(a) and 6(b) validate these results: we see that τ_2 effectively misses a deadline at time 6 with inverse RM, and that τ_1 misses a deadline at time 7 with RM. We have then used Formula 4 to test the feasibility with EDF. The property is not verified, this can be confirmed by Sub-Fig. 6(c) that shows that τ_2 misses a deadline at time 42 under EDF. Finally we have used Formula 1 to test the unconstrained feasibility. The property is verified, thus a feasible schedule exists for this task problem. Using the produced feasible scheduling run, we are effectively able to produce the schedule presented by Sub-Fig. 6(c) (TAAS stands for *Timed-Automata-Assisted Scheduler*).

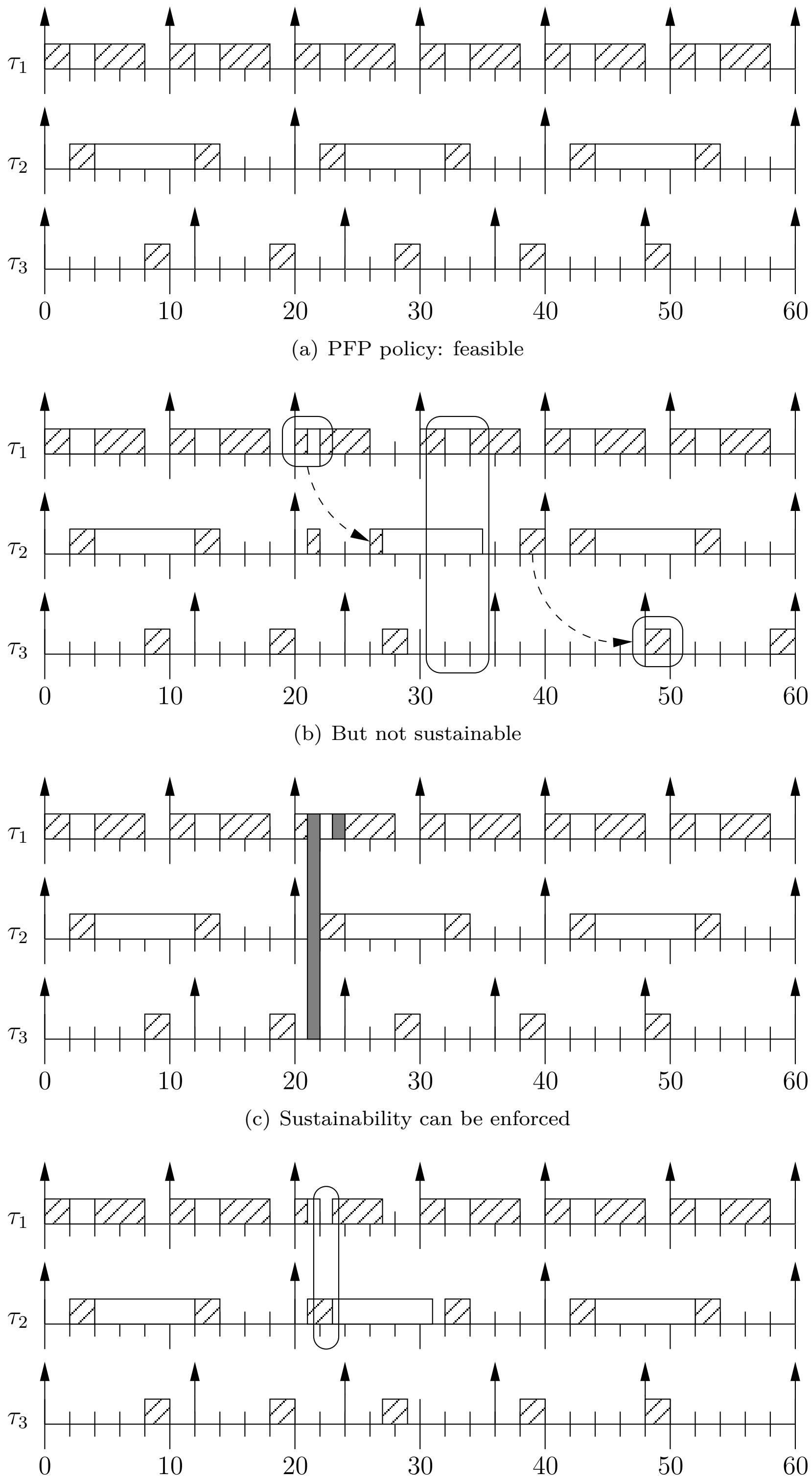
7.2 Experiment 2 (uncertain tasks)

In this experiment, we have modeled the system $\Sigma = \{\tau_1, \tau_2, \tau_3\}$ with $\tau_1 = (0, (2, 2, 4), 10, 10)$, $\tau_2 = (0, (2, 8, 2), 20, 20)$ and $\tau_3 = (0, (2), 12, 12)$, where τ_1 has the highest priority and τ_3 the lowest. We first have verified Formula 3: the property is verified, the system is then feasible with a fixed priority scheduler. We then have modeled the system $\Sigma^* = \{\tau_1^*, \tau_2, \tau_3\}$, with $\tau_1^* = (0, ([1, 2], [1, 2], 4), 10, 10)$. We have verified Formula 7 for fixed priority schedulers on our model, the property is not verified. We conclude that feasibility with a fixed priority scheduler is not sustainable for this system. Sub-figure 7(a) presents the schedule obtained with Σ. Sub-figure 7(b) presents the schedule with Σ^* where the third instance of τ_1 executes with the pattern $\mathcal{P}_1 = (C_1^1/2, E_1^1/2, C_1^2)$. It results in a deadline missed for τ_3 at time 49. However, we have tested Formula 5 and Formula 6. The outcome is positive in both cases: valid schedules restricted and non restricted to work-conserving ones. The feasibility of the system is then sustainable (within the intervals $[C_{i,l}^j, C_{i,u}^j]$ and $[E_{i,l}^j, E_{i,u}^j]$) in the general case and with a work-conserving scheduler. Indeed, there exists a simple way to enforce the sustainability: forcing the system to insert idle times when a task completes earlier than it was supposed to. Fig. 7(c) shows the resulting schedule of this strategy. Fig. 7(d) presents a work-conserving feasible schedule which can be obtained using the strategy generated by Uppaal-Tiga.

8 Conclusion

In this paper, we presented a method to solve the scheduling problem of periodic self-suspending tasks. We provided a feasibility test and schedulability tests with PFP and EDF. We proposed a method to test the sustainability of schedules w.r.t the execution and suspension durations. This is done both with the restriction of work-conserving schedules and in the general case. If the problem is unfeasible with PFP and EDF but proved to be feasible, our approach permits to generate a scheduler. The approach has been tested using the tools Uppaal and Uppaal-Tiga.

As future work, we first have to implement the scheduler generation and to formalize the memory complexity of generated on-line schedulers. We also have to extend our model to consider multiprocessor platforms and tasks sporadic activation and compare the results with the ones presented in [19]. Moreover, in this paper we supposed a synchronous activation of the tasks at time instant 0. If we consider task systems with offsets, it has to be proved that cyclicity result for such systems presented in [8] still holds for self-suspending tasks. The proof given by the authors to extend the property to interacting tasks system [8, Section 4.3] seams cover the self suspending case, providing a redefinition of the function $Waiting(t)$ as the sum of remaining computation times of tasks released before or at t and being under a self-suspension. Anyway, Algorithm 1 and proof



(a) PFP policy: feasible

(b) But not sustainable

(c) Sustainability can be enforced

(d) TAAS finds a work-conserving schedule

Figure 7: Sustainability

of Proposition 4 have to be adapted to extend this work to tasks with offset.

References

[1] Y. Abdeddaïm, E. Asarin, and O. Maler. On optimal scheduling under uncertainty. In *TACAS*, 2003.

[2] Yasmina Abdeddaïm and Damien Masson. Uppaal implementations. `http://igm.univ-mlv.fr/~masson/Softwares/SelfSuspending`.

[3] Yasmina Abdeddaïm and Damien Masson. Scheduling Self-Suspending Periodic Real-Time Tasks Using Model Checking. In *WIP RTSS*, 2011.

[4] Rajeev Alur and David Dill. Automata for modeling real-time systems. In *ICALP*, 1990.

[5] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[6] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. Larsen, and D. Lime. Uppaal-tiga: Time for playing games! In *CAV*, 2007.

[7] Ya-Shu Chen and Li-Pin Chang. A real-time configurable synchronization protocol for self-suspending process sets. *Real-Time Syst.*, 42(1-3):34–62, 2009.

[8] Annie Choquet-Geniet and Emmanuel Grolleau. Minimal schedulability interval for real-time systems of periodic tasks with offsets. *Theor. Comput. Sci.*, 310(1-3):117–134, January 2004.

[9] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, 1989.

[10] Ansgar Fehnker. Scheduling a steel plant with timed automata. In *RTCSA*, pages 280–286, 1999.

[11] Elena Fersman, Pavel Krcál, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Inf. Comput.*, 205(8):1149–1172, 2007.

[12] Thomas A. Henzinger and Peter W. Kopke. Discrete-time control for rectangular hybrid automata. *Theor. Comput. Sci.*, 221:369–392, June 1999.

[13] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. In *LICS*, 1992.

[14] K.Cerans. *Algorithmic Problems in Analysis of Real Time System Specifications*. PhD thesis, University of Latvia, 1992.

[15] Yonit Kesten, Amir Pnueli, Joseph Sifakis, and Sergio Yovine. Decidable integration graphs. *Inf. Comput.*, 150(2):209–243, 1999.

[16] In-Guk Kim, Kyung-Hee Choi, Seung-Kyu Park, Dong-Yoon Kim, and Man-Pyo Hong. Real-time scheduling of tasks that contain the external blocking intervals. In *RTCSA*, 1995.

[17] D. Kozen, editor. *Logics of Programs, Workshop*, volume 131 of *Lecture Notes in Computer Science*. Springer, 1982.

[18] Karthik Lakshmanan, Dionisio de Niz, and Ragunathan Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS*, 2009.

[19] Karthik Lakshmanan and Ragunathan Rajkumar. Scheduling self-suspending real-time tasks with rate-monotonic priorities. In *RTAS*, 2010.

[20] Karthik Lakshmanan, Ragunathan Rajkumar, and John Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *ECRTS*, 2009.

[21] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.

[22] Cong Liu and James H. Anderson. Task scheduling with self-suspensions in soft real-time multiprocessor systems. In *RTSS*, 2009.

[23] Cong Liu and James H. Anderson. Improving the schedulability of sporadic self-suspending soft real-time multiprocessor task systems. In *RTCSA*, 2010.

[24] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[25] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *STACS*, 1995.

[26] J. C. Palencia and M. González Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *RTSS*, 1998.

[27] Ragunathan Rajkumar. Dealing with suspending periodic tasks. Technical report, IBM Thomas J. Watson Research Center, 1991.

[28] Frederic Ridouard and Pascal Richard. Worst-case analysis of feasibility tests for self-suspending tasks. In *RTNS*, 2006.

[29] Frederic Ridouard, Pascal Richard, and Francis Cottet. Negative results for scheduling independent hard real-time tasks with self-suspensions. In *RTSS*, 2004.

[30] Jun Sun and Jane Liu. Synchronization protocols in distributed real-time systems. In *ICDCS*, 1996.

[31] Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocess. Microprogram.*, 40(2-3):117–134, 1994.