Scheduling Self-Suspending Periodic Real-Time Tasks Using Model Checking
Yasmina Abdeddaïm, Damien Masson

To cite this version:

HAL Id: hal-00636122
https://hal-upec-upem.archives-ouvertes.fr/hal-00636122
Submitted on 26 Nov 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Scheduling Self-Suspending Periodic Real-Time Tasks Using Model Checking

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
Email: {y.abdeddaim/d.masson}@esiee.fr

Abstract—In this paper, we address the problem of scheduling periodic, possibly self-suspending, real-time tasks. We provide schedulability tests for PFP and EDF and a feasibility test using model checking. This is done both with and without the restriction of work-conserving schedules. We also provide a method to test the sustainability w.r.t the execution and suspension durations of the schedulability and feasibility properties within a restricted interval. Finally we show how to generate an on-line scheduler for such systems when they are feasible.

I. 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 [1]. Models that explicitly consider suspension durations exist but their analysis is proved difficult: in [2], 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 online 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 instance of the problem, i.e. for each task sets. Result 2) implies that traditional on-line schedulers are not optimal, whereas our approach is. 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. We prove that our approach is sustainable w.r.t execution and suspension durations.

Related work: In [3], 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, a pseudo-polynomial response-time test. The rest of the literature on self-suspending tasks focuses on the multiprocessor context [4], [5]. Our approach addresses the problem for periodic tasks, and is not restricted to RM. The timed automata approach has been already used to solve job shop scheduling problems [6], [7]. In [8], the authors present a model based on timed automata to solve real-time scheduling problems, but this model cannot be applied to self-suspending tasks because of the way the preemptions are modeled. Moreover, it intrinsically considers work-conserving schedules and cannot handle uncertain times.

Contributions: In this paper, we propose a timed-automata-based model for hard real-time periodic self-suspending tasks. We show how to use model checking to obtain both a necessary and sufficient feasibility test, and a schedulability test for classical scheduling policy (RM, DM, EDF). When these algorithms fail to schedule a feasible system, we show how to generate an appropriate scheduler. We also consider the cases where both the execution time and the suspension time of each task are uncertain. The generated schedulers then have an important property: the feasibility of a task set is sustainable w.r.t the execution and suspension durations.

Sect. II presents the task model. Sect. III introduces the self-suspending task automaton. Sect. IV exposes how to check the feasibility and the schedulability with PFP and EDF, and how to generate a sustainable scheduler. Sect. V presents experiments and finally we conclude.

II. SELF-SUSPENDING TASK MODEL

Task Model and assumptions: We consider the problem of scheduling a system $\Sigma = \{\tau_1, ..., \tau_n\}$ of $n$ independent possibly self-suspending periodic tasks synchronously activated on one processor. A task which does not suspend itself is characterized by the tuple $\tau_i = (C_i, T_i, D_i)$ where $C_i$ is its execution time, $T_i$ its period and $D_i$ its relative deadline with $D_i \leq T_i$. A task which suspends itself is characterized by the tuple $\tau_i = (P_i, T_i, D_i)$ where $P_i$ is its execution pattern. An execution pattern is a tuple $\mathcal{P}_i = (C_i^1, E_i^1, C_i^2, E_i^2, ..., C_i^m)$ where each $C_i^j$ is an execution time and each $E_i^j$ a suspension time. When these parameters can take values within an interval, we refer to the tasks as uncertain tasks and the execution pattern becomes $\mathcal{P}_i = ([C_i^1, C_i^m], [E_i^1, E_i^m], [C_i^2, C_i^m], ..., [C_i^m, C_i^m])$. To simplify the notations, when there is no ambiguity, the task id is not mentioned and the tuples becomes $\tau = (C, T, D)$ etc. This model and notations are inspired by existing literature on self suspending tasks [4], [1], [3], [2].

Sustainability: 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 is changed in a positive way. The sustainability is an important property since it permits to study the worst case scenario. In our task model, execution and suspension durations can be bounded within intervals. In the remaining of the paper, we
is labeled with an output action $T!$ and synchronizes with
the transition from waiting state ($f$) to active state ($s$) of
the automaton $A_r$. Finally, we introduce a third automaton $A_D$
which models the deadline of the task using an output action
$D!$ fired every $D$ time units. We call the tuple $(A_r,A_T,A_D)$
a self-suspending automata model.

IV. SCHEDULABILITY USING MODEL CHECKING

Model checking is an automatic verification technique used
to prove formally whether a model satisfies a property or
not. In this section, we present how we use CTL [12] model
checking to test the feasibility of a task set, its schedulability
with PFP and EDF, and the sustainability of a schedule.

A. Feasibility and Schedulability

Let $\Sigma = \{\tau_1 \ldots \tau_n\}$ be a finite set of self-suspending
tasks. We associate to every task $\tau_i$ a self-suspending automata
model $(A^i_r,A^i_T,A^i_D)$. We use a global variable $proc$ to ensure,
using a guard, that only one task is executing at once. We
introduce a new state $STOP_i$ for each automaton $A^i_r$, which
is reached if a task misses its deadline. For every non final
state of $A^i_r$, a transition labeled by an input action $D_i$? leads to
state $STOP_i$. These transitions synchronize with the deadline
automaton. Thus, if an instance of a task does not terminate
before its deadline, the automaton goes to the state $STOP_i$.

Proposition 1 (feasibility): Let $\Sigma = \{\tau_1 \ldots \tau_n\}$ be a set
of self-suspending tasks. $\Sigma$ is feasible iff CTL Formula 1 is
satisfied.

$$\phi_{Sched} : EG(\bigvee_{i \in [1,n]} STOP_i)$$

Sketch of Proof: Proposition 1 states that the self-suspending
problem is feasible iff there exists a feasible scheduling run i.e a run that never reached a STOP state.
Suppose that the scheduling problem $\Sigma$ 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 scheduling 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 scheduling
runs are not feasible i.e all the scheduling run lead to a STOP
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 by simply reading sequentially
the configurations of the feasible run until reaching the
one where all active tasks have terminated their execution
without missing their deadline.

Proposition 2 (PFP Schedulability): Let $\Sigma$ be a set of self-
suspending tasks sorted according to a priority function, with
$\tau_1 \leq \ldots \leq \tau_n$. $\Sigma$ is feasible according to a fixed priority
scheduler iff CTL Formula 2 is satisfied.

$^1$The operator $A$ means for all paths, $E$ there exists a path and $G$ globally
in the future [12].
Figure 2. Uncertain Self-Suspending Task. UPPAAL-TGA model is available at [11]. Uncontrollable transitions are represented using dashed lines.

\[
\phi_{FP} := \neg \left( \bigvee_{i \in \{1, \ldots, n\}} (s_i \wedge e_j) \lor \bigvee_{i \in \{1, \ldots, n\}} (p_i \wedge e_j) \lor \left( \bigvee_{i \in \{1, \ldots, n\}} \neg \text{STOP}_i \right) \right) \tag{2}
\]

Formula 2 states that there exists a feasible run where, in all the configurations, a task cannot be in its execution state if a less priority task is active i.e. in state \( s \) or \( p \). The idea of the proof is similar to the one of Proposition 1.

Proposition 3 (EDF Schedulability): Let \( \Sigma \) be a set of self-suspending tasks. \( \Sigma \) is schedulable according to EDF iff CTL Formula 3 is satisfied.

\[
\phi_{edf} := \neg \left( \bigvee_{i \in \{1, \ldots, n\}} \left( s_i \wedge e_j \wedge P_{ij} \right) \lor \left( \bigvee_{i \in \{1, \ldots, n\}} (p_i \wedge e_j \wedge P_{ij}) \lor \left( \bigvee_{i \in \{1, \ldots, n\}} \neg \text{STOP}_i \right) \right) \right) \tag{3}
\]

The formula \( P_{ij} \) is a state of an observer automaton reachable when \( x_{D_i} - x_j > D_i - D_j \) with \( x_{D_i} \) and \( x_j \) the clocks of the deadline automata \( A^i_{D_i} \) and \( A^j_{D_j} \).

Formula 3 states that there exists a run where, in all the configurations, a task cannot be in its execution state if a task with a closer deadline is active. The idea of the proof is similar to the one of Proposition 1.

B. Sustainable Scheduler

In a timed game automaton (TGA)[10], the set of transitions is split into controllable (\( \Delta_c \)) and uncontrollable (\( \Delta_u \)) ones. Solving a timed game consists in finding a strategy \( f \) s.t. a TGA supervised by \( f \) always satisfies a given formula. Fig. 2 represents our model adapted to uncertain self-suspending tasks. The start and preemption transitions are controlled by the scheduler, while the transitions from state \( e \) to \( f \), from \( e \) to \( id \) and from \( sp \) to \( p \) are controlled by the environment. The guard \( c_r \geq C_i - 1 \) on transitions \( e \) to \( id \) and \( e \) to \( f \) models that the task can terminate after \( C_i \) time units and the guard \( c_r < C_u - 1 \) that the task terminates before \( C_u \) time units. The invariant \( x_r \leq C_r \) and the guard \( x_r \geq E_j \) from \( sp \) to \( p \) models a suspension of a duration within \( [E_j, E_u] \).

Definition 1 (Scheduling strategy): Let \( A \) be a set of \( n \) uncertain self-suspending task automata. A scheduling strategy is a function \( f \) from the set of configurations of \( A \) to the set \( \Delta_c^i \cup \ldots \Delta_c^u \cup \{ \lambda \} \). s.t. if \( f(q, v) = e \in \Delta_c \), then execute the controllable transition \( e \) and if \( f((q, v)) = \lambda \) then wait in the configuration \((q, v)\).

Proposition 4 (Sustainability): Let \( \Sigma = \{\tau_1 \ldots \tau_n\} \) be a finite set of uncertain self-suspending tasks. A sustainable scheduling algorithm exists for \( \Sigma \) iff there exists a scheduling strategy \( f \) s.t. the self-suspending timed game automata model of \( \Sigma \) supervised by \( f \) satisfies the safety CTL Formula 4.

\[
\phi_{\text{sust}} := AG \neg \left( \bigvee_{i \in \{1, \ldots, n\}} \text{STOP}_i \right) \tag{4}
\]

Sketch of Proof: Formula 4 states that there exists a strategy function \( f \) s.t. for every configuration of the task set and every possible execution duration or suspension duration, there is a way to avoid the STOP states. Let \( \Sigma \) be a set of uncertain self-suspending tasks. Suppose that there exists a sustainable feasible scheduler for \( \Sigma \). Then, the strategy \( f \) can be simply computed by defining a strategy that mimics the decisions of the sustainable scheduler. Consider now that there exists a feasible scheduling strategy for the timed game satisfying Formula 4, then there exists a way for all the tasks to not miss their deadline whatever are the execution and the suspension durations. Thus this strategy can be used as a sustainable scheduling algorithm.

Algorithm 1 Scheduling Strategy Algorithm

1: \( (q_0, v_0) \leftarrow (q_i, v_i), \epsilon \leftarrow 0 \)
2: while \( q \neq q_0 \) and \( v \neq 0 \) do
3: \( \text{while } f((q, v)) = \lambda \text{ or no task finished or no end of suspension do} \)
4: \( \text{Wait: increase } v \)
5: \( \text{end while} \)
6: \( \text{if } f((q, v)) = \tau_r \text{ then} \)
7: \( \text{if } q_{\tau_r} \neq q \text{ then} \)
8: \( \text{execute task } \tau_r \text{ at time } t \leftarrow t_k \)
9: \( \text{end if} \)
10: \( \text{if } q_{\tau_r} = q \text{ then} \)
11: \( \text{if } q_{\tau_r} = q \text{ then} \)
12: \( \text{execute task } \tau_r \text{ at time } t \leftarrow t_k \)
13: \( \text{end if} \)
14: \( \text{end if} \)
15: \( \text{end if} \)
16: \( \text{if } \tau_r \text{ has finished then} \)
17: \( \text{if } \tau_r \text{ finished then} \)
18: \( \text{end if} \)
19: \( (q_{\tau_r}, v_{\tau_r}) \text{ is the configuration } (q, v) \text{ where } q_{\tau_r} \leftarrow f, v_{\tau_r}(x_{\tau_r}) \leftarrow 0 \)
20: \( \text{end if} \)
21: \( \text{if } \tau_r \text{ has terminated the suspended then} \)
22: \( (q_{\tau_r}, v_{\tau_r}) \text{ is the configuration } (q, v) \text{ where } q_{\tau_r} \leftarrow p, v_{\tau_r}(x_{\tau_r}) \leftarrow 0 \)
23: \( \text{end if} \)
24: \( (q, v) \leftarrow (q_{\tau_r}, v_{\tau_r}) \)
25: \( \text{end while} \)

To provide a sustainable scheduling algorithm we need to construct a feasible strategy if one exists. Such a strategy is finite, because of the decidability of the timed game problem [13]. Then an on-line scheduler is an algorithm that executes the pre-computed strategy. This is formalized by Algorithm 1. 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 18-25). Note that if the presented methods generate possibly idle schedules, we can compute work-conserving schedules by specifying this property in CTL Formulas 1, 2, 3 and 4.
V. Experiments

We used UPPAAL[14] and UPPAAL-TIGA[15] to implement our model [11]. We tested it on two examples. The first is composed by two regular self-suspending tasks. The second is composed by three uncertain self-suspending tasks. Fig. 3 and Fig. 4 present the obtained results. White squares represent suspension durations and hatched ones execution durations.

Example 1 (regular self-suspending tasks) In this experiment, we have modeled the system $\Sigma = \{\tau_1, \tau_2\}$ with $\tau_1 = ([1, 4, 1], 7, 7)$ and $\tau_2 = ([1, 3, 1], 6, 6)$, where $\tau_1$ has the highest priority and $\tau_2$ the lowest. We first have verified Formula 2: the property is verified, and the system is then feasible with a fixed priority scheduler. We then have modeled the system $\Sigma^* = \{\tau_1^*, \tau_2, \tau_3\}$ with $\tau_1^* = ([1, 2], [1, 2], 4), 10, 10)$. We have verified Formula 4 restricted to 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 4(a) presents the schedule obtained with $\Sigma$. Sub-figure 4(b) presents the schedule with $\Sigma^*$ where the third instance of $\tau_1$ executes with the pattern $P_1 = \{(C_1^1, E_1^1, C_1^2)\}$. It results in a deadline missed for $\tau_2$ at time 49. However, we have tested Formula 4. 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 $[C_1^1, C_1^2]$ and $[E_1^1, E_1^2]$) 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. 4(c) shows the resulting schedule of this strategy. Fig. 4(d) presents a work-conserving feasible schedule which can be obtained using the strategy generated by UPPAAL-TIGA.

VI. Conclusion

In this paper, we present how to use model checking to solve a difficult scheduling problem: the scheduling of periodic self-suspending tasks. We provide a feasibility test and schedulability tests with PFP and EDF. We also provide 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. Finally our approach permits to generate a scheduler for such systems.

Work in progress:

We first have to implement the scheduler generation and to formalize the memory complexity of generated on-line schedulers. Then we have to improve the way our automata handle preemptions. Finally we have to extend our model to consider multiprocessor platforms and tasks sporadic activation and compare the results with the ones presented in [3].

References