Scheduling Self-Suspending Periodic Real-Time Tasks Using Model Checking

Yasmina Abdeddaïm, Damien Masson

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 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 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 focus 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 $\tau_i = (\mathcal{P}_i, T_i, D_i)$ where $\mathcal{P}_i$ 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 $\mathcal{P}_i = ([C_{i\,l}^1, C_{i\,u}^1], [E_{i\,l}^1, E_{i\,u}^1], [C_{i\,l}^2, C_{i\,u}^2], ..., [C_{i\,l}^m, C_{i\,u}^m])$. 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

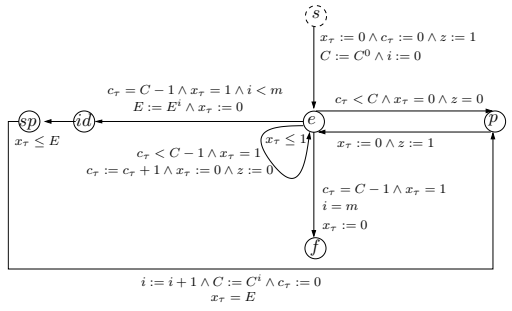


Figure 1. Self-Suspending Task. UPPAAL model is available in [11]

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.

III. The Self-Suspending Timed Automaton Model

A timed automaton [9], is an automaton augmented with a set of real variables $\mathcal{X}$, called clocks. The firing of a transition can be controled by a clock constraint called guard and a location can be constrained by a staying condition called invariant. A *clock valuation* is a function that associates to a clock x its value $v(x)$ and a *configuration* is a pair $(q, \mathbf{v})$ where q is a state of the automaton and $\mathbf{v}$ a vector of clock valuations. A *run* is a sequence of timed and discrete transitions, timed transitions representing the elapse of time in a state, and discrete ones the transitions between states. Synchronous communication between timed automata can be done using input actions ($a?$) and output actions ($a!$).

Self-Suspending Timed Automaton: Let $\tau = (\mathcal{P}, T, D)$ be a self-suspending task. We associate to τ a timed automaton $\mathcal{A}_\tau$ (Fig. 1) with one clock x_τ and of states $\mathcal{Q} = \{s, e, f, p, sp, id\}$. State s is the waiting state where the task is active, e the execution state, p the state where the task is preempted, f is the one where the task has finished its execution, sp the suspension state and id is an idle state used to model a possible idle step. To compute the execution time of a task, we use the clock x_τ and a variable c_τ as follows: the automaton can stay in the execution state exactly one time unit and the variable c_τ keeps track of how many time units have been performed. This is represented by an invariant $x_\tau \leq 1$ on state e and a loop transition that increments c_τ. The guard $c_\tau < C - 1$ restricts the number of loop transitions. The preemption of the task is modeled using transitions from e to p and from p to e. Note that the modeling of preemption at any time is not possible using timed automata because clock variables cannot be stopped. Thus, it is supposed in this model that a task can be preempted only at integer times. The transition from e to id is enabled when the total time spent in the active state is equal to C and the number of performed step is less than m. To model the suspension of a task we use the state sp. The automaton can stay in the state sp exactly E time units. This is modeled using an invariant $x_\tau \leq E$ on the state and a guard $x_\tau = E$. The task finishes when all the steps have been computed. This is modeled using the guard $i = m$ from state e to state f. To make this task periodic of period T, we use a second timed automaton $\mathcal{A}_T$ with one state and a loop transition enabled every T time units. This transition is labeled with an output action $T!$ and synchronizes with the transition from waiting state (f) to active state (s) of the automaton $\mathcal{A}_\tau$. Finally, we introduce a third automaton $\mathcal{A}_D$, which models the deadline of the task using an output action $D!$ fired every D time units. We call the tuple $(A_\tau, \mathcal{A}_T, \mathcal{A}_D)$ a 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 $\Sigma = \{\tau_1 \dots \tau_n\}$ be a finite set of self-suspending tasks. We associate to every task τ_i a self-suspending automata model $(A^i_\tau, \mathcal{A}^i_T, \mathcal{A}^i_D)$. 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_τ, which is reached if a task misses its deadline. For every non final state of A^i_τ, 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 $\Sigma = \{\tau_1 \dots \tau_n\}$ be a set of self-suspending tasks. Σ is feasible iff CTL Formula 1 is satisfied.

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

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 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 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 Σ be a set of self-suspending tasks sorted according to a priority function, with $\tau_1 \leq \dots \leq \tau_n$. Σ 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].

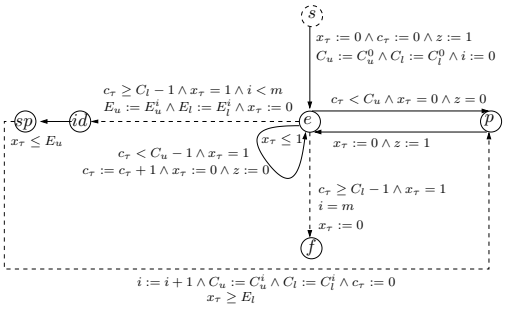


Figure 2. Uncertain Self-Suspending Task. UPPAAL-TIGA model is available at [11]. Uncontrollable transitions are represented using dashed lines.

$$\phi_{FP} : EG\neg(\bigvee_{i\in[1,n-1]} (s_i \bigwedge_{j\in[i,n]} e_j) \vee \bigvee_{i\in[1,n-1]} (p_i \bigwedge_{j\in[i,n]} e_j) \vee (\bigvee_{i\in[1,n]} STOP_i)) \quad (2)$$

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 Σ be a set of self-suspending tasks. Σ is schedulable according to EDF iff CTL Formula 3 is satisfied.

$$\phi_{edf} : EG\neg(\bigvee_{i\in[1,n]} \bigvee_{j\neq i\in[1,n]} (s_i \wedge e_j \wedge P_{ij}) \bigvee_{i\in[1,n]} \bigvee_{j\neq i\in[1,n]} (p_i \wedge e_j \wedge P_{ij}) \vee (\bigvee_{i\in[1,n]} STOP_i)) \quad (3)$$

P_{ij} is a state of an observer automaton reachable when $x_{D_i} - x_{D_j} > D_i - D_j$ with x_{D_i} and x_{D_j} the clocks of the deadline automata $\mathcal{A}_D^i$ and $\mathcal{A}_D^j$.

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 (Δ_c) and uncontrollable (Δ_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 $c_\tau \geq C_l - 1$ on transitions e to id and e to f models that the task can terminate after C_l time units and the guard $c_\tau < C_u - 1$ models that the task terminates before C_u time units. The invariant $x_\tau \leq C_u$ and the guard $x_\tau \geq E_l$ from sp to p models a suspension of a duration within $[E_l, E_u]$.

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 $\Delta_c^1 \cup \ldots \Delta_c^n \cup \{\lambda\}$. 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 Σ iff there exists a scheduling strategy f s.t the self-suspending timed game automata model of Σ supervised by f satisfies the safety CTL Formula 4.

$$\phi_{sust} : AG\neg(\bigvee_{i\in[1,n]} STOP_i) \quad (4)$$

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 Σ be a set of uncertain self-suspending tasks. Suppose that there exists a sustainable feasible scheduler for Σ. Then, the strategy f can be simply computed by defining a strategy that mimic 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. ■

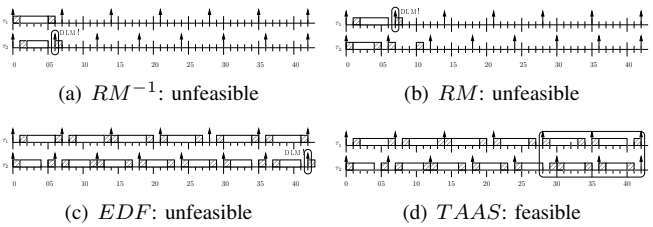
Algorithm 1 Scheduling Strategy Algorithm

1: $(q, \mathbf{v}) \leftarrow (q_0, \mathbf{v}_0), t \leftarrow 0$
2: **while** $q \neq q_0$ or $t = 0$ **do**
3: **while** $f((q, \mathbf{v})) = \lambda$ or no task finished or no end of suspension **do**
4: Wait: increase $\mathbf{v}$
5: **end while**
6: **if** $f((q, \mathbf{v})) = tr \in \Delta_c^j$ **then**
7: $(q_k, \mathbf{v}_k)$ is the successor of $(q, \mathbf{v})$ while taking the transition tr
8: **if** $\exists q_k^j \neq q^j$ and $q_k^j = e^j$ **then**
9: execute task τ_j at time $t \leftarrow t_k$
10: **end if**
11: **if** $\exists q_k^j \neq q^j$ and $q_k^j = p^j$ **then**
12: preempt task τ_j at time $t \leftarrow t_k$
13: **end if**
14: **if** $\exists q_k^j \neq q^j$ and $q_k^j = sp^j$ **then**
15: suspend task τ_j at time $t \leftarrow t_k$
16: **end if**
17: **end if**
18: **if** a task τ_j has finished **then**
19: $(q_k, \mathbf{v}_k)$ is the configuration $(q, \mathbf{v})$ where $q_k^j \leftarrow f^j$, $v_k(x_\tau^j) \leftarrow 0$
20: **end if**
21: **if** a task τ_j has terminate the suspension **then**
22: $(q_k, \mathbf{v}_k)$ is the configuration $(q, \mathbf{v})$ where $q_k^j \leftarrow p^j$, $v_k(x_\tau^j) \leftarrow 0$
23: **end if**
24: $(q, \mathbf{v}) \leftarrow (q_k, \mathbf{v}_k)$
25: **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[2]. 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.

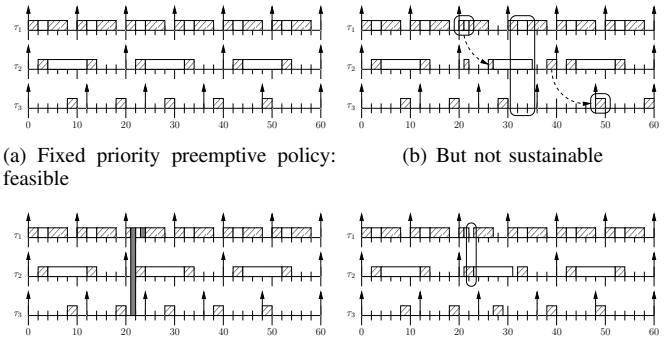
[2] q_i^j is the state of the automaton j in the configuration $(q_i, \mathbf{v}_i)$; q_0 is the initial configuration ; t is an additional global clock which is never reset ; t_i the valuation of the clock t in the configuration $(q_i, \mathbf{v}_i)$.



(a) RM^{-1}: unfeasible (b) RM: unfeasible

(c) EDF: unfeasible (d) $TAAS$: feasible

Figure 3. Feasible schedule exists but neither pfp or EDF is able to find it



(a) Fixed priority preemptive policy: feasible (b) But not sustainable

(c) Sustainability can be enforced (d) TAAS found a work-conserving schedule

Figure 4. Sustainability

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 $\tau_2 = ((1,3,1),6,6)$. We have first used Formula 2 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. 3(a) and 3(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 3 to test the feasibility with EDF. The property is not verified, this can be confirmed by Sub-Fig. 3(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. 3(c) (TAAS stands for *Timed-Automata-Assisted Scheduler*).

Example 2 (uncertain self-suspending tasks) In this experiment, we have modeled the system $\Sigma = \{\tau_1, \tau_2, \tau_3\}$ with $\tau_1 = ((2,2,4),10,10)$, $\tau_2 = ((2,8,2),20,20)$ and $\tau_3 = ((2),11,11)$, where τ_1 has the highest priority and τ_3 the lowest. We first have verified Formula 2: 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 $\tau_1^* = (([1,2],[1,2],4),10,10)$. 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 Σ. Sub-figure 4(b) presents the schedule with Σ^* where the third instance of τ_1 executes with the pattern $\mathcal{P}_1 = (\frac{C_1^1}{2}, \frac{E_1^1}{2}, C_1^2)$. It results in a deadline missed for τ_3 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 $[C_l^i, C_u^i]$ and $[E_l^i, E_u^i]$) 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

[1] J. W. S. W. Liu, *Real-Time Systems*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[2] F. Ridouard, P. Richard, and F. Cottet, "Negative results for scheduling independent hard real-time tasks with self-suspensions," in *RTSS*, 2004.

[3] K. Lakshmanan and R. Rajkumar, "Scheduling self-suspending real-time tasks with rate-monotonic priorities," in *RTAS'10*, 2010.

[4] C. Liu and J. H. Anderson, "Task scheduling with self-suspensions in soft real-time multiprocessor systems," in *RTSS'09*, 2009.

[5] ——, "Improving the schedulability of sporadic self-suspending soft real-time multiprocessor task systems," in *RTCSA'10*, 2010.

[6] Y. Abdeddaïm, E. Asarin, and O. Maler, "On optimal scheduling under uncertainty," in *TACAS'03*, 2003.

[7] A. Fehnker, "Scheduling a steel plant with timed automata," in *RTCSA'99*, 1999.

[8] E. Fersman, P. Krcal, P. Pettersson, and W. Yi, "Task automata: Schedulability, decidability and undecidability," *Inf. Comput.*, vol. 205, pp. 1149–1172, August 2007.

[9] R. Alur and D. Dill, "Automata for modeling real-time systems." in *ICALP'90*, 1990.

[10] O. Maler, A. Pnueli, and J. Sifakis, "On the synthesis of discrete controllers for timed systems," in *STACS'95*, 1995.

[11] Y. Abdeddaïm and D. Masson, "Uppaal and uppaal-tiga implementations." [Online]. Available: http://igm.univ-mlv.fr/~masson/Softwares/SelfSuspending/

[12] D. Kozen, Ed., *Logics of Programs, Workshop*, ser. Lecture Notes in Computer Science, vol. 131. Springer, 1982.

[13] T. A. Henzinger and P. W. Kopke, "Discrete-time control for rectangular hybrid automata," *Theor. Comput. Sci.*, vol. 221, pp. 369–392, June 1999.

[14] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *STTT*, vol. 1, no. 1-2, 1997.

[15] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. Larsen, and D. Lime, "Uppaal-tiga: Time for playing games!" in *CAV'07*, 2007.