A Generic Task Model Using Timed Automata to Design and Analyze Real-Time Applications

Yasmina Abdeddaïm, Damien Masson

A Generic Task Model Using Timed Automata to Design and Analyze Real-Time Applications

Yasmina Abdeddaïm[1] and Damien Masson[1,2]

[1] Université Paris-Est, Département Systèmes Embarqués, ESIEE Paris, 2 bld Blaise Pascal, BP 99, 93162 Noisy-le-Grand CEDEX, France
[2] Laboratoire d'informatique Gaspard-Monge (LIGM), UMR CNRS 8049
{y.abdeddaim,d.masson}@esiee.fr

Abstract. In this work, we propose an approach for automated analysis of real-time scheduling problems based on timed automata. Tasks are modeled using timed automata while schedulers correspond to feasible runs computed using model checking algorithms. Using this model, we present a method to prove on one hand the feasibility of a scheduling problem and to compute, on the other hand, an hybrid off-line/on-line scheduling policy. We expose how to use it to re-demonstrate well known results, to produce efficient schedule regarding of various properties such as minimal preemption number, and to address the problem of scheduling real-time task set composed by possibly self-suspending tasks which is known to be NP-hard.

Keywords: Timed Automata, Real-Time Scheduling, Self-Suspending Tasks.

1 Introduction

In real-time systems, the *scheduling problem* consists in producing a feasible schedule for a real-time task set whenever one exists. A slightly different problem is, given a task set and a scheduling policy, to answer the question: did all the tasks will respect all their deadlines? This is the *feasibility problem*. The aim of this paper is to answer to these two problems: to provide a framework to both analyze the properties of a given *schedule* and to generate a feasible schedule (optionally with some parameterizable properties).

The scheduling theory often concentrates on the search for an *optimal* scheduler. An optimal scheduling policy is a policy that, given a task set, produces a non-feasible schedule if and only if it does not exist any other scheduling policy in the same class of algorithm that can produce a feasible schedule. The precision *in the same class of algorithm* is of importance. Indeed, scheduling algorithms can be characterized in many ways: *off-line* or *on-line* (*priority driven*), *work-conserving* or *idle*, *preemptive* or not. To illustrate the ambiguity of the word *optimal*, we can cite the RM, DM and EDF algorithms [1], which are all three said *optimal*. In fact, RM is optimal among fixed priority driven preemptive schedulers, and only for traffic composed by periodic tasks with implicit deadlines (i.e. $D_i = T_i$). DM is optimal among the same kind of schedulers, but for

traffic composed by periodic tasks with constrained deadlines (i.e. $D_i \leq T_i$). Finally EDF is optimal among (not necessarily fixed) priority driven schedulers for traffic composed by periodic tasks with constrained deadlines. Any optimality-related result on a scheduling policy (or comparison between several algorithms) so should be considered in a well defined referential, composed by the kind of scheduler considered for comparison, and the kind of scheduling problem considered (hypotheses on the traffic model).

Our approach is to clearly separate the behavior of a real-time task from specific constraints of a scheduling policy. Using a timed automaton [2], we model a task as a transition system between possible configurations of a task (activated, preempted, running, blocked). Feasible schedules then correspond to specific runs on the model. Different scheduling and feasibility problems can be solved using CTL [3] model checking on the generic model without constraining the model. While changing the property to verify, we can consider feasible runs only the ones which follow specific rules (e.g. the ones leading to an EDF schedule) or just ask if there exists a way to respect real-time constraints without restriction of the scheduling policy. This design permits to use the model either to test properties of a task set (e.g. schedulable, unschedulable, ...), of a specific scheduling policy on a task set (e.g. schedulable with RM, ...) or even to generate a feasible schedule.

One important characterization of a scheduling policy is the separation between off-line schedulers in one side, and on-line schedulers in the other side. A scheduling policy is said on-line if the scheduling decisions (task start, preemption, resume...) are taken at run time. It is said off-line if the actions at runtime are limited to read a table where all scheduling decisions are store. On-line scheduling is mostly represented by a sub-category: the priority-driven schedulers. Such a scheduler maintains up-to-date a sorted-by-priority queue of active tasks, and the processor is allocated to the task in head of this queue. A Distinction can also be made between fixed priority schedulers for which a priority is assigned to each task once and for all, and dynamic priority schedulers for which the priority of a task can change at any time. Off-line scheduling policies are easy to implement and have a very low overhead cost. However, they suffer from a possibly high space complexity issue, and are not suitable on systems that have to react dynamically to external and/or unpredictable behavior such as aperiodic event happening, cost under-run or over-run of a task or deadline missed. Moreover, the generation of the table can have a high time complexity, depending of the scheduling problem considered.

In the case of scheduling problem without uncertainties, the timed automaton model permits to produce off-line a scheduling table that can be read on-line. In scheduling under temporal uncertainties, the model produces a kind of hybrid scheduling policy. Decisions are taken on-line according to data computed off-line. It is suitable to schedule highly dynamics systems even if the space complexity should be examined in details. Such a scheduling policy can be classified among *clairvoyant algorithms*. This is an interesting property since

a lot of negative results for numbers of scheduling-related problems, such as NP-completeness, are applicable only to non-clairvoyant algorithms.

To illustrate the interest of our approach, we focus on one open problem in real-time systems theory: the scheduling of possibly self-suspending real-time tasks. This problem has recently been shown to be NP-hard in the strong sense [4], i.e. there is no optimal on-line algorithm to schedule it which takes its decisions in a polynomial time (unless $P = NP$). Using our approach to model self-suspending tasks, we can both determine if a schedule exists, and provide the scheduler with extra data to drive it to an existing schedule. Moreover, the use of the model permits us to conjecture that the schedulability of such task set is sustainable regarding the variations either on execution duration and suspension duration for work-conserving scheduler (this property can be trivially proved true with an idle scheduler). The sustainability is an important property of a scheduling problem solution: it says that if a task set is feasible, then the same task set with positive modifications (such as reduced execution time or augmented period) is still guaranteed feasible. This property permits to answer the feasibility problem on the based of the worst case execution times for tasks with uncertain durations.

After a review of related works in the remainder of this Section, we introduce timed automata in Section 2. Section 3 presents our model. Section 4 exposes how to use it to test systems schedulability. In Section 5 we present the problem of schedule self-suspending tasks and how to handle it with our model. Finally we conclude in Section 6.

Task Model. In this paper, we consider real-time systems built from a set of n periodic real-time tasks $\tau_1, \tau_2, ..., \tau_n$. Each task τ_i is characterized by a period T_i, a worst-case execution time C_i and a relative deadline D_i.

Related Work. In [5,6] timed automata are used to solve mainly non preemptive job shop scheduling problem. In these approaches, reachability analysis algorithms are used to construct optimal schedules in the sense of minimal total execution time. Handling preemptible tasks using timed automata is not possible mainly because clock variables cannot be stopped and modeling such tasks can only be done using stopwatch automata. Unfortunately, reachability analysis of stopwatch automata is an undecidable problem in the general case [7]. However, there exits methods [8,9] by which we can model preemptible tasks in timed automata with over-approximation and modeling with stopwatches is recently possible using the tool Uppaal 4.1 [10] with the support of a zone based over-approximation state exploration. In [11], authors proposed a schedulability analysis algorithm where the scheduler is modeled using an extension of timed automata called suspension automata [12]. Unfortunately the number of clocks needed is proportional to the maximal number of schedulable task instances associated with the model. The authors then presents an efficient algorithm in [13] for the problem of fixed priority scheduling, as the problem is undecidable when the execution times are given as intervals, they propose an over-approximation

technique. The control synthesis approach has been used to solve the job scheduling problem in the case of uncertain execution times [14], and in [15,16] control synthesis is used to construct a scheduler by adding control invariant modeling the schedulability constraints.

2 Preliminaries

Timed Automata A *timed automaton* [2] is a finite automaton augmented with a finite set of real-valued variables evolving continuously and synchronously with absolute time. 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 the following grammar: $\phi ::= x \sharp c \mid x - y \sharp c \mid \phi \wedge \phi$ where $c \in I\!N$, $x, y \in \mathcal{X}$, and $\sharp \in \{<, \leq, \geq, >\}$. A *clock valuation* is a function $v : \mathcal{X} \to \mathcal{R}_+ \cup \{0\}$ which associates with every clock x its value $v(x)$. Given a real $d \in \mathcal{R}$ 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,* $\mathcal{I} : Q \to \mathcal{C}(\mathcal{X})$ *if 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 augmented with the action* $\perp$ *that represents the empty action*

A configuration of a timed automaton is a pair $(q, \mathbf{v})$ consisting of a state q and a $dim(\mathcal{X})$ vector $\mathbf{v}$ of clock valuations. The semantic of a timed automaton is given as a timed transition system with two types of transition 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 $(q, \mathbf{v}) \xrightarrow{d} (q, \mathbf{v} + d\mathbf{1})$ $d \in \mathcal{R}_+$ with $\mathbf{v}$ and $\mathbf{v} + d$ satisfying $\mathcal{I}(q)$ the invariant of state q.

A *run* of the timed automaton starting from a configuration $(q_0, \mathbf{v}_0)$ is a sequence of time and discrete transitions $\xi : \quad (q_0, \mathbf{v}_0) \xrightarrow{t_1} (q_1, \mathbf{v}_1) \xrightarrow{t_2} \cdots$.

The reachability problem for timed automata is decidable and PSPACE-complete [2,17]. Since this result, many model checking algorithms [18,19,20] and timed model checkers [21,22] have been developed.

This basic model of timed automata can be extended to allow the use of integer variables. Let $\mathcal{V}$ be a set of integer variables and $\mathcal{B}(\mathcal{V})$ a set of integer constraints over $\mathcal{V}$ generated by the following grammar: $\phi_V ::= x < c \mid x \leq c \mid x > c \mid x \geq c \mid \phi \wedge \phi$ where $c \in I\!N$ and $x \in \mathcal{V}$. We define a timed automaton using integer variables as a tuple $\mathcal{A} = (\mathcal{Q}, q_0, \mathcal{X}, \mathcal{V}, \mathcal{I}, \Delta, \Sigma)$ where a transition $(q, \phi_x, \phi_v, a, r_x, r_v, q') \in \Delta$ is defined, in addition to the basic transitions of a timed automaton, by the integer guard $\phi_v \in \mathcal{B}(\mathcal{V})$ and the set r_v of linear update functions over integer variables.

A network of timed automata is the parallel composition $\mathcal{A}_1||\ldots||\mathcal{A}_n$ of a set of timed automata $\mathcal{A}_1,\ldots,\mathcal{A}_n$. Parallel composition used an interleaving semantic and synchronous communication can be done using input actions denoted $a?$ and output actions denoted $a!$ (this hand shake synchronization is used in the tool UPPAAL [22]) while asynchronous communication is done using shared variables.

Timed Game Automata A *timed game automaton* model [23] is an extension of the timed automaton model, which has been introduced for the synthesis of timed controllers.

Definition 2 (Timed Game Automaton). *A* timed game automaton *(TGA) is a timed automaton where the set of transitions Δ is split into controllable (Δ_c) and uncontrollable (Δ_u) transitions.*

The timed game automaton model defines the rules of a two players game with on one side the controller (mastering the controllable transitions) and on the other side 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. the automaton starting at the initial configuration and supervised by f always satisfies the formula whatever actions are chosen by the environment. More precisely, a strategy is a partial mapping f 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 execute the controllable transition e from the last configuration of the run ξ and if $f(\xi) = \lambda$ then wait in the last configuration of the run ξ. A strategy is *memory-less* whenever its result depends only on the last configuration of the run. It has been shown that solving a timed game is a decidable problem [23,24].

3 Real-Time Task Automata Model

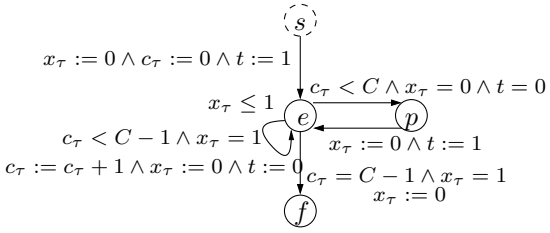


Fig. 1. Real-time task automaton

We present our method by introducing a generic model for scheduling using timed automata. For the sake of clarity, we assume a single processor scheduling

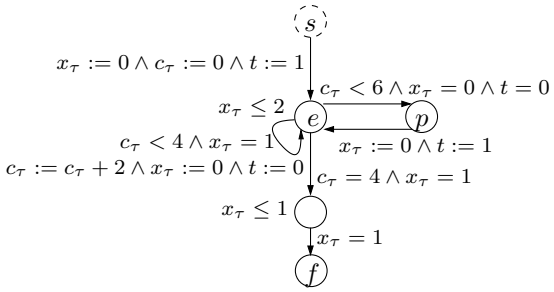


Fig. 2. Real-time task automaton, $C = 7$ and preemption every 2 time units

problem and all tasks periodic with their deadlines equal to their periods. However, the proposed approach can be easily generalized to less specific scheduling problems.

A real-time task can be among one of the possible configurations: active, running, waiting for the next activation, or preempted. The timed automaton model is used in our approach to model a real-time task as a transition system where the set of states represents the possible configurations of a task and the set of transitions the possible moves between different configurations of a task.

For every real-time task $\tau = (C, D, T)$, we associate a 4-states timed automaton $\mathcal{A}_\tau$ with one clock x_τ, one integer variable c_τ and a set of states $\mathcal{Q} = \{s, e, f, p\}$. State s is the waiting state where the task is active and not yet executed, e is the active state where the task is running, p is the state where the task is preempted and f is the one where the task is waiting for the next activation. An example is given by Figure 1. To compute the execution time of a task, we use the clock x_τ and the 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 unit has been performed. This is represented by an invariant $x_\tau \leq 1$ on state e and a loop transition that increments c_τ. The transition from e to f is enabled when the total time spent in the active state is equal to C (the completion time of the task τ). The preemption of the task is modeled using transitions from e to p and from p to e. Note it is supposes in this model that a task can be preempted only at integer times, moreover, if we consider that preemptions can occur only at certain known integer points we can arise the staying condition of state e as shown in Figure 2.

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 unit. This transition is labeled with an output action $T!$ and synchronize with the transition from waiting to active state of the automaton $\mathcal{A}_\tau$.

Definition 3 (A real-time task automata model). *Let $\tau(C, T, D)$ be a real-time periodic task with $T = D$. A real-time task automata model for τ is a tuple $(A_\tau, \mathcal{A}_T)$ where A_τ and $\mathcal{A}_T$ are two timed automata defined as follows:*

1. $\mathcal{A}_\tau(\mathcal{Q}_\tau, s, x_\tau, c_\tau, \mathcal{I}_\tau, \Delta_\tau, \Sigma_\tau)$:
   - $\mathcal{Q}_\tau = \{s, e, f, p\}$,
   - $\mathcal{I}_\tau(e) = x_\tau \leq 1$ *and* $\forall q \in \mathcal{Q}_\tau$, *if* $q \neq e$ $\mathcal{I}_\tau(q) = true$,
   - Δ_τ *is composed of the following transitions*
     - $(s, true, true, \bot, x_\tau, c_\tau = 0, e)$,
     - $(e, x_\tau = 1, c_\tau = C - 1, \bot, \emptyset, \emptyset, f)$,
     - $(e, x_\tau = 0, c_\tau < C, \bot, \emptyset, \emptyset, p)$,
     - $(p, true, true, \bot, x_\tau, \emptyset, e)$,
     - $(e, x_\tau = 1, c_\tau < C - 1, \bot, x_\tau, c_\tau = c_\tau + 1, e)$,
     - $(f, true, true, T?, \emptyset, \emptyset, s)$.
   - $\Sigma_\tau = \{T?\}$.
2. $\mathcal{A}_T(s_T, s_T, x_T, \emptyset, \mathcal{I}_T, \Delta_T, \Sigma_T)$:
   - $\mathcal{I}_T(s_T) = x_T \leq T$,
   - $\Delta_T = \{(s_T, x_T = T, true, T!, x_T, \emptyset, s_T)\}$,
   - $\Sigma_\tau = \{T!\}$.

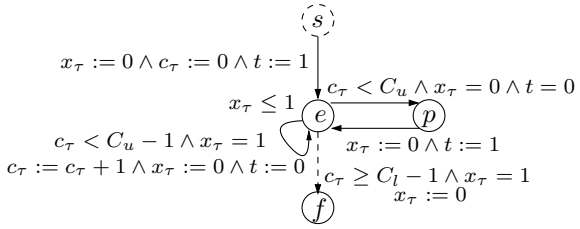


Fig. 3. Uncertain real-time task automaton

In the treated problem, we consider that all the task parameters (execution time, period and deadline) are known precisely in advance. In real-time scheduling, task processing can take more or less time than expected, thus the system is evaluated according to its worst case behavior. Let us consider a non deterministic scheduling problem where the execution time of a task is not given in advance, but restricted to be bounded within an interval. Let us call a real-time task τ an uncertain real-time task if at every activation of the task, the execution time can vary within an interval $[C_l, C_u]$. An uncertain real-time task is modeled using a timed game automata noted $\mathcal{TGA}_\tau$. The modeling of an uncertain task τ is similar to the model of definition 3, with a distinction between controllable and uncontrollable transitions. The start and preemption transition are controlled by the scheduler, while the end transition, from state e to f, is controlled by the environment and can be taken within $c \in [C_l, C_u]$. This is modeled using a guard $c_\tau \geq C_l - 1$ on the end transition and a guard $c_\tau < C_u - 1$ on the loop execution transition. Figure 3 represents an uncertain real-time task automaton where the uncontrollable transition is represented using a dashed line.

4 Schedulability Using Real-Time Task Automata

This section introduces some of the properties we are interested to verify from a scheduling point of view on real-time systems and exposes how to check these properties on our model using CTL [3] properties.

The first and more natural property we want to verify on the model is the feasibility of the task set. Second, one can be interested to know if the system can be scheduled with a specific scheduling policy (e.g. RM, DM, EDF). However, the feasibility of the system is not the only property of importance in the choice of the scheduling policy. One can be interested in a lot of other characterizations for the obtained schedule: minimizing average response times, the number of preemptions, the maximum jitter or ensure the sustainability of the schedule. A feasible schedule is said sustainable regarding to a parameter if it remains feasible when this parameter is changed in a positive way. For example, if a sustainable scheduling policy regarding to tasks' cost succeed to schedule a task set Σ, it should feasibly-schedule any task set where all the tasks has the same parameters as the ones of Σ except their cost which can be lesser or equal[3]. Sustainability of scheduling policies can be studied on a lot of parameters, such as the periods, the number of tasks or the system load. We will expose how these properties can be checked on the model.

A scheduling algorithm defines the rules which control when and how transitions between different task states occur. In our approach, the scheduling algorithm corresponds to a run in the product automaton of real-time task automata.

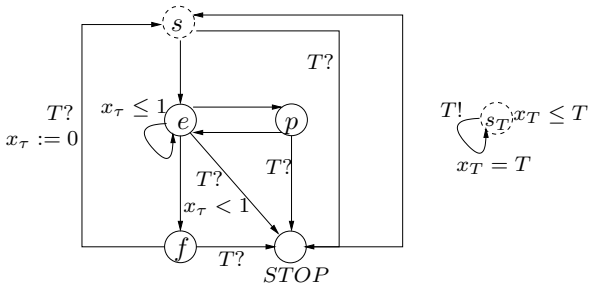


Fig. 4. Real-time task automata model. To be more readable, we omit the guards and updates represented in Figure 1.

Let $\Sigma = \{\tau_1(C_1, D_1, T_1) \dots \tau_n(C_n, D_n, T_n)\}$ be a finite set of real-time tasks. We associate to every task τ_i a real-time task automata model $(A^i_\tau, \mathcal{A}^i_T)$ with $\mathcal{A}^i_\tau(\mathcal{Q}^i_\tau, s^i, x^i_\tau, c^i_\tau, \mathcal{I}^i_\tau, \Delta^i_\tau, \Sigma^i_\tau)$ and $\mathcal{Q}^i_\tau = \{s_i, e_i, f_i, p_i\}$. We use a global variable *proc* indicating if the processor is idle ($proc = 1$) or not ($proc = 0$). This variable is tested for every transition leading to the execution state and reset to one for

[3] e.g. this property is not verified for most of the traditional multiprocessor scheduling algorithms

every transition starting from it. In the case of multi-processor scheduling, one can use several global variables, each of them corresponding to a processor.

We introduce a new state $STOP_i$ for each automaton A^i_τ, which is reached if a task misses its deadline. For every state of A^i_τ, a transition labeled by an input action $T_i?$ leads to state $STOP_i$. Thus, if a new instance of the task is active before the execution of the previous one, the automata goes to the state $STOP_i$ as shown by Figure 4.

Let us note $(q_i, \mathbf{v}_i)$ a configuration of the parallel composition of $\mathcal{A}^1_\tau || \dots || A^n_\tau$ s.t. $q_i = (q^1_i, q^2_i, \dots, q^n_i)$ where q^j_i is a state of the automaton A^j_τ. We note q_0 the tuple where $\forall j \in 1 \dots n \ q^j_0 = s_j$. The set of clocks is augmented with a global clock t which is never reset. We note t_i the valuation of the clock t in the configuration $(q_i, \mathbf{v}_i)$ and $\mathbf{v}_0$ the vector of valuations where all the clocks are set to 0.

Definition 4 (Scheduling run). *Let $\Sigma((A_{\tau_1}, \mathcal{A}_{T_1}), \dots, (A_{\tau_n}, \mathcal{A}_{T_n}))$ be a set of real-time task automata. A* scheduling run $\xi : \ (q_0, \mathbf{v}_0) \longrightarrow (q_1, \mathbf{v}_1) \longrightarrow \cdots$ *is a run of the parallel composition of $\mathcal{A}_{\tau_1} || \dots || A_{\tau_n}$. A scheduling run is* feasible *iff $\forall (q_i, \mathbf{v}_i) \in \xi \ \forall j \in \{1 \dots n\} \ q^j_i \neq STOP_j$.*

Thus, a set $\tau = \{\tau_1(C_1, D_1, T_1) \dots \tau_n(C_n, D_n, T_n)\}$ is schedulable if and only if there exists a feasible scheduling run in the corresponding real-time task automata model. Then, schedulability can be verified using the CTL formula 1.

$$\phi_{Sched} : EG\neg(\bigvee_{i=1\dots n} STOP_i) \tag{1}$$

A scheduling algorithm can be computed using a feasible trace satisfying this formula if it exists using Algorithm 1. This algorithm takes as input a feasible scheduling run and defines a scheduling policy. The proof of termination is based on the fact that if ξ is a feasible schedule, then it exists a configuration $(q_i, \mathbf{v}_i)$ with $q_i = q_0$ and $t_i > 0$ where all active tasks have terminated their execution without missing their deadline. In the case where $D \leq T$, 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.

Work-Conserving Scheduler Our model generates possibly idle schedules, i.e. schedules where the processor can be idle at any time. Scheduling theory often implicitly addresses problems for work-conserving schedulers (that produce schedule where the processor can be idle only when there is no ready task) because leaving the processor idle when tasks are ready seems to result in a resource wasting. But it is important to not restrict the model to work-conserving schedules since it exists task sets only feasible with idle schedulers. Moreover, the scheduling problem in idle-scheduling environment was proved undecidable without a clairvoyant algorithm.

Algorithm 1 Scheduling Run Algorithm

SchedRun[ξ]
  $i \leftarrow 0, t_i \leftarrow 0$
  while $q_i \neq q_0$ or $t_i == 0$ **do**
    if $\exists q_i^j \neq q_{i+1}^j$ **then**
      if $q_{i+1}^j = e^j$ **then**
        execute task τ_j at time $t = t_i$
      end if
      if $q_{i+1}^j = p^j$ **then**
        preempt task τ_j at time $t = t_i$
      end if
    end if
    $i \leftarrow i + 1$
  end while

Definition 5 (Work-Conserving Scheduling Run). *A scheduling run ξ is* work conserving *iff $\forall (q_i, \mathbf{v}) \in \xi$ if $\exists q_i^j \in \{s_i, p_i\}$ and $q_{i+1}^j = q_i^j$ then $\exists k \neq j$ s.t. $q_i^k \in \{e_k\}$*

Work conserving runs corresponds to work-conserving scheduling algorithms.

Let $\mathcal{P}$, be the set of n permutations of the set $\{p, s, f\}$ where the tuple $(f, f, \ldots, f)$ is excluded. We note $k_i \in \mathcal{P}$ the tuple $(k_i^1, k_i^2 \ldots k_i^n)$ where $\forall j \in 1 \ldots n$ $k_i^j \in \{p_j, s_j, f_j\}$ the preemptive, start and final states of the task τ_j.

To compute work-conserving runs, we use the CTL formula 2

$$\phi_{WC} : EG\neg(\bigvee_{k_i \in \mathcal{P}} (k_i^1 \wedge k_i^2 \wedge \ldots \wedge k_i^n \wedge POS)) \tag{2}$$

where POS is a state of an observer automaton, which is in state POS if $\exists j \in 1 \ldots n$ s.t. $x_\tau^j > 0$ and in state $NPOS$ if $\forall j \in 1 \ldots n$ $x_\tau^j = 0$. This formula states that it exists a run where in all the configurations containing an active task, it exists a task in its execution state.

Fixed-Priority Scheduler We consider that the tasks τ_i are sorted according to a priority function, such that $\tau_1 \leq \tau_2 \ldots \leq \tau_n$. A system of task is schedulable according to a fixed priority algorithm if and only if the CTL formula 3 is satisfied.

$$\phi_{FP} : EG\neg(\bigvee_{i=1\ldots n-1} (s_i \bigwedge_{j=i\ldots n} e_j) \vee \bigvee_{i=1\ldots n-1} (p_i \bigwedge_{j=i\ldots n} e_j) \vee (\bigvee_{i=1\ldots n} STOP_i)) \tag{3}$$

This formula states that it exists a run where in all the configurations, a task cannot be in its execution state if a less priority task is active.

Earliest-Deadline-First Scheduler Earliest deadline first (EDF) is a dynamic priority scheduler where, if two tasks are active, the scheduler assigns the

processor to the task the closest to its deadline. A system of task is schedulable according to EDF if and only if the CTL formula 4 is satisfied.

$$\phi_{edf} : EG\neg(\bigvee_{i=1..n}\bigvee_{j\neq i=1..n}(s_i \wedge e_j \wedge P_{ij}) \bigvee_{i=1..n}\bigvee_{j\neq i=1..n}(p_i \wedge e_j \wedge P_{ij}) \vee (\bigvee_{i=1..n} STOP_i)) \tag{4}$$

In this formula, P_{ij} and NP_{ij} are the states of an observer automaton, the automaton is in state P_{ij} if $x_{T_i} - x_{T_j} > T_i - T_j$ and in NP_{ij} if $x^i_T - x^i_T \leq T_i - T_j$ where x^i_T and x^j_T are the clocks of the period automata $\mathcal{A}^i_T$ and $\mathcal{A}^j_T$. This formula states that it 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.

Minimal Preemption Scheduler Our approach can be used to find scheduling algorithms that minimizes the number of preemptions. This can be done simply by adding a variable that counts the number of preemptions. A task cannot be preempted no more than its duration, so a guard is used to limit the number of preemptions. A global variable *preemp* is used to count the total number of preemptions. To verify if a task set is schedulable with less then c preemptions, we use the CTL formula:

$$\phi_{Preemp} : EG\neg(\bigvee_{i=1...n} STOP_i \vee preemp > c) \tag{5}$$

Sustainable Scheduler Let $\tau = \{\tau_1([C^1_l, C^1_u], D^1, T^1) \ldots \tau_n([C^n_l, C^n_u], D^n, T^n)\}$ be a finite set of uncertain real-time tasks. We associate to every task τ_i an uncertain real-time task automaton TGA^i_τ. A scheduling algorithm is sustainable in $[C^i_l, C^i_u] \forall i \in 1 \ldots n$ if all tasks do not miss there deadline whatever are the execution times.

Definition 6 (Scheduling strategy). *Let A be a network of n uncertain real-time task automata. A* scheduling strategy *is a strategy f from the set of configurations of A to the set $\Delta^1_c \cup \ldots \Delta^n_c \cup \{\lambda\}$. A scheduling strategy is feasible iff the automata A starting at the initial configuration and supervised by f never reach a state $STOP_i$ $\forall i \in 1 \ldots n$.*

In CTL, this means that the network supervised by f satisfies the safety formula:

$$\phi_{sust} : AG\neg(\bigvee_{i=1...n} STOP_i) \tag{6}$$

Given a network of uncertain real-timed automata and a formula ϕ_{sust}, finding a sustainable scheduling algorithm (in $[C_l, C_u]$) consists in the construction of a feasible strategy if one exists. The scheduler can be computed using Algorithm 2.

Algorithm 2 Scheduling Strategy Algorithm

SchedStrategy[f]
  $(q, \mathbf{v}) \leftarrow (q_0, \mathbf{v}_0), t \leftarrow 0$
  while $q \neq q_0$ or $t == 0$ **do**
    while $f((q, \mathbf{v})) = \lambda$ or no task has finished **do**
      $\mathbf{v} = \mathbf{v} + 1$
    end while
    if $f((q, \mathbf{v})) = tr \in \Delta_c^j$ **then**
      $(q_k, \mathbf{v}_k)$ is the successor of $(q, \mathbf{v})$ while taking the transition tr
      if $\exists q_k^j \neq q^j$ and $q_k^j = e^j$ **then**
        execute task τ_j at time $t = t_k$
      end if
      if $\exists q_k^j \neq q^j$ and $q_k^j = p^j$ **then**
        preempt task τ_j at time $t = t_k$
      end if
    end if
    if a task τ_j has finished **then**
      let be $(q_k, \mathbf{v}_k)$ s.t.:
      $\forall i = 1 \dots n \; i \neq j \; q_k^i = q^i, v_k(x_\tau^i) = v_k(x_\tau^i)$ and
      $q_k^j = f^j, v_k(x_\tau^j) = 0$
    end if
    $(q, \mathbf{v}) = (q_k, \mathbf{v}_k)$
  end while

5 Self-Suspending Tasks

Traditional real-time scheduling theory is build on the hypothesis that the tasks cannot suspend themselves [1]. However, real-world tasks have to communicate, to synchronize and to perform external input/output operations. Each of these actions can result in a significant suspension interval for a task. It is especially true when modern architectures are considered: a task executing on a multi-core processor can have to synchronize with another one which is executing on a different core ; a task executing on an heterogeneous ISA multiprocessor system can have to migrate temporarily in order to execute specific instructions ; in the context of embedded systems where some accelerated co-processors such as Digital Signal Processors (DSPs) or Graphics Processing Units (GPUs) can be available, executing a part of a task on one of them results in a self-suspension of the task from the main processor point of view. Moreover, the durations of the suspension intervals are mostly independent of the processor (CPU) speed and so of tasks' worst case execution times. The easiest way to deal with such tasks is to account these suspension intervals as a part of the task computation time itself, and to enforce busy-waiting, wasting the gain-time offered by the hardware parallelism.

Task Model with Self-Suspensions Two task models are considered in the literature. In the simplest one, each task is characterized by the tuple $\tau_i =$

(C_i, E_i, T_i, D_i) where C_i is its worst case execution time and E_i its worst case suspension time (T_i and D_i being as usual the period and the deadline). There is no assumption on the number and time arrival of suspensions. The only information is the sum of their durations. This model was used in [25,26]. The second is an extension where a task is characterized by the tuple $\tau_i = (\mathcal{P}_i, T_i, D_i)$ with $\mathcal{P}_i$ its execution pattern. An execution pattern is a tuple $\mathcal{P}_i = (C_i^1, E_i^2, C_i^2, E_i^2, ..., C_i^m)$ Where each C_i^j is a worst case execution time and each E_i^j a worst case suspension time. This model was used in [27,4].

In [4], authors expose three negative results on these kind of systems scheduled on-line with a non-clairvoyant algorithm. These three points can be solved using our framework:

The scheduling problem for self-suspending tasks is NP-hard in the strong sense. However, with our model we can say if a task set is feasible.

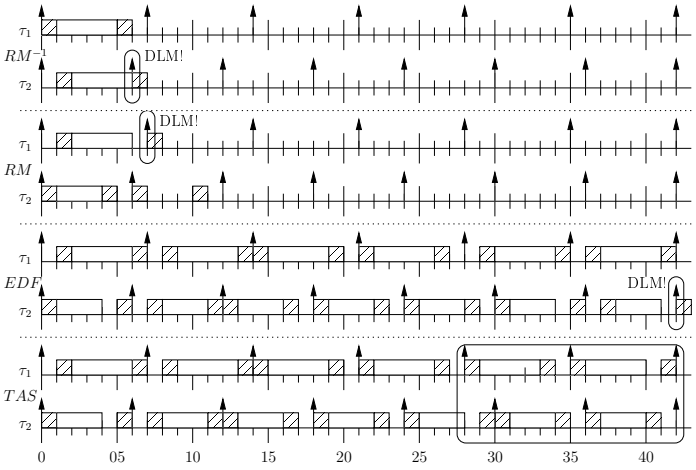
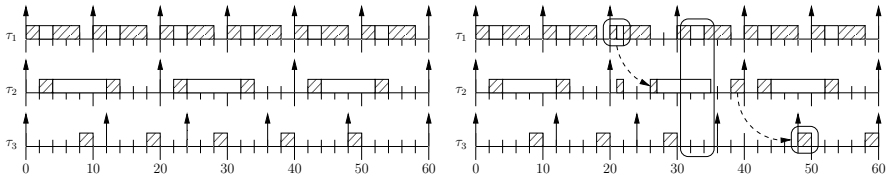


Fig. 5. Feasible schedule exists but neither pfp or EDF is able to find it

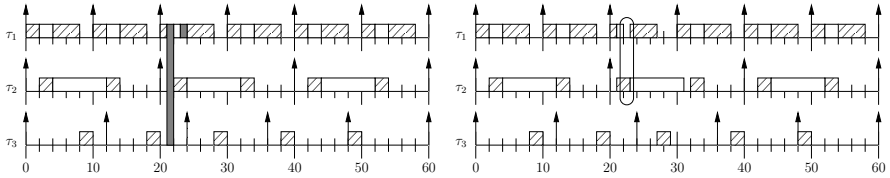
Classical algorithms do not maximize tasks completed by their deadlines. Figure 5 shows that it exists systems for which feasible schedules exist whereas neither PFP (with all priority assignments possibilities) nor EDF are able to produce one of them. However our timed-automata-assisted-scheduler (TAS on the Figure) permits to find a feasible schedule whenever one exists.

Scheduling Anomalies can occur at run-time. Figure 6 exhibits the non sustainability of fixed priority schedulers when tasks can suspend themselves. Sub-figure 6(a) presents the schedule of task set $\{\tau_1, \tau_2, \tau_3\} = \{((2, 2, 4), 10, 10, High), ((2, 8, 2), 20, 20, Medium), (2, 12, 12, Low)\}$ when the tasks always execute and



(a) With A fixed priority preemptive policy, the system is feasible

(b) But can be unfeasible if a working and/or a self-suspension duration is decreased



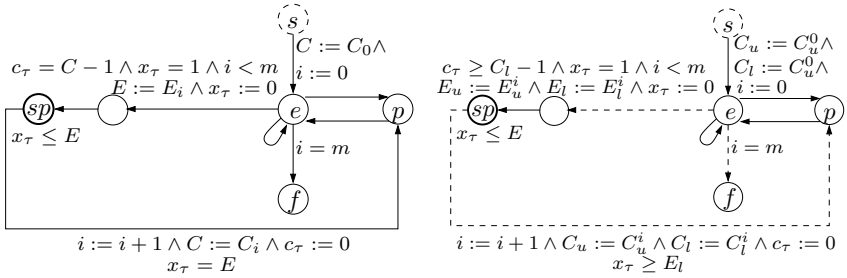
(c) Sustainability can be enforced

(d) But a more efficient schedule can be found with the model

Fig. 6. PFP is not sustainable for self-suspending tasks: counter example on system $\{\tau_1, \tau_2, \tau_3\} = \{((2, 2, 4), 10, 10, High), ((2, 8, 2), 20, 20, Medium), (2, 12, 12, Low)\}$

suspend exactly for their worst case execution time. We can see that the system is feasible all over its hyper period. Sub-figure 6(b) exposes the schedule 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 the τ_3 at time 49. Note that it exists a simple way to enforce the sustainability: forcing the system to simulate activity when a task completes earlier than it was supposed to. Figure 6(c) exposes the resulting schedule of this strategy on the previous example. However, this results in resource wasting. The gain time resulting from the earlier completion of a task could better be used for increase system dynamism or robustness, e.g. to compensate a cost overrun. Figure 6(d) produces a feasible schedule for the previous example where the system is never idle when work is pending. It is possible at the expense of a priority inversion at time 21. The time automata assisted scheduling can easily find such feasible schedule, where the problem is probably hard for a pure non-clairvoyant scheduler. To conclude with this example, note that an EDF schedule suffers from the exactly same anomaly. The system is feasible when the costs are fixed, but deadlines are missed (by τ_2 at time 40 and τ_1 at time 50) if the third instance of τ_1 executes according to the modified pattern.

Self-Suspending Automata To handle self-suspending scheduling problem with timed automata, we had to modify our model by adding a suspended state for tasks, however, all results presented in the previous section will remain valid with the modified model.



(a) Worst case execution time (b) Uncertain durations

Fig. 7. Self suspending automata

Let $\tau = (\mathcal{P}, T, D)$ be a self suspending task with an execution pattern $\mathcal{P} = (C^1, E^1, C^2, E^2, ..., C^m)$ composed of m steps. To model a self suspending task using timed automata, we add to the real-time task automaton a new state *sp* where the execution of a task can be suspended (see Figure 7(a)). The automaton can stay in the state exactly E time units, this is modeled using an invariant $x_\tau \leq E$ and a guard $x_\tau = E$. At each step i of the execution pattern, the variables E and C are settled to E^i and C^i respectively and the task finishes when all the steps has been computed (see the guard $i = m$ on the end transition of Figure 7(a)). The timed game automaton of Figure 7(b) represents the model for an uncertain self suspending task. In this model, the execution pattern $\mathcal{P}$ is a set of intervals $([C_l^1, C_u^1], [E_l^1, E_u^1], [C_u^2 C_u^2], [E_l^2, E_u^2], ..., [C_l^m C_u^m])$ representing uncertain execution time and suspension time.

6 Conclusion

We presented a real-time application model using timed automata and exposed how to use it to handle classical and known-unfeasible scheduling problems. The approach has been tested using the tools Uppaal [22] and Uppaal-Tiga [28]. In our experiments, we focused on the efficiency of the model to construct different schedules by changing the properties to verify. We managed to use our model to test the feasibility of scheduling problem according to fixed priority and dynamic priority schedulers and to produce feasible schedules when these methods are not optimal. We tested also the model for minimal preemption scheduling and to generate sustainable schedules when they exists. All the tests have been done in both cases of work-conserving and idle scheduling. Actually, the schedules are generated by hand using the feasible trace and the winning strategy generated by the tools. Future works must propose an automatic generation of the scheduler and must examine other solutions to model the preemption or focus on optimization algorithm for state space exploration. Indeed, we used a discretization of the execution time of a task to handle preemption. This can leads to a combinatoric explosion of the model when the execution time of tasks grows.

References

1. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard real time environment. Journal of the Association for Computing Machinery **20** (1973) 46–61
2. Alur, R., Dill, D.: Automata for modeling real-time systems. In: International Colloquium on Automata, Languages and Programming (ICALP). (1990)
3. Kozen, D., ed.: Logics of Programs, Workshop. In Kozen, D., ed.: Logics of Programs. Volume 131 of Lecture Notes in Computer Science., Springer (1982)
4. Ridouard, F., Richard, P., Cottet, F.: Negative results for scheduling independent hard real-time tasks with self-suspensions. In: Proceedings of the 25th IEEE International Real-Time Systems Symposium. (2004)
5. Abdeddaim, Y., Maler, O.: Job-shop scheduling using timed automata. In: International Conference on Computer Aided Verification (CAV'01). (2001)
6. Fehnker, A.: Scheduling a steel plant with timed automata. In: 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99). (1999)
7. Kesten, Y., Pnueli, A., Sifakis, J., Yovine, S.: Decidable integration graphs. Inf. Comput. **150** (1999)
8. Waszniowski, L., Hanzalek, Z.: Over-approximate model of multitasking application based on timed automata using only one clock. In: 19th IEEE International Parallel and Distributed Processing Symposium. IPDPS '05 (2005)
9. Gabor, M., Nikil, D., Sherif, A.: A conservative approximation method for the verification of preemptive scheduling using timed automata. In: 15th IEEE Symposium on Real-Time and Embedded Technology and Applications. RTAS '09 (2009)
10. David, A., Illum, J., Larsen, K., Skou, A.: Model-Based Framework for Schedulability Analysis Using UPPAAL 4.1. In: Model-Based Design for Embedded Systems. CRC Press (2010)
11. Fersman, E., Pettersson, P., Yi, W.: Timed automata with asynchronous processes: schedulability and decidability. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02). (2002)
12. McManis, J., Varaiya, P.: Suspension automata: A decidable class of hybrid automata. In: 6th International Conference on Computer Aided Verification. CAV '94 (1994)
13. Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: Schedulability analysis of fixed-priority systems using timed automata. Theoretical Computer Science **354** (2006) 301–317
14. Abdeddaim, Y., Asarin, E., Maler, O.: On optimal scheduling under uncertainty. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03). (2003)
15. Altisen, K., Gler, G., Sifakis, J.: A methodology for the construction of scheduled systems. In: FTRTFT'02. (2000)
16. Altisen, K., Gössler, G., Sifakis, J.: Scheduler modeling based on the controller synthesis paradigm. Real-Time Syst. **23** (2002)
17. Alur, R., Dill, D.: A theory of timed automata. Theoritical Computer Science **2** (1994) 183–235
18. R.Alur, C.Courcoubetis, Dill, D.: Model-checking in dense real-time. Inf. Comput. **104** (1993) 2–34
19. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Information and Computation **111** (1992) 394–406

20. G.Larsen, K., Pettersson, P., Yi, W.: Model-checking for real-time systems. In: FCT. (1995)
21. Yovine, S.: Kronos: A verification tool for real-time systems. STTT **1** (1997) 123–133
22. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. STTT **1** (1997) 134–152
23. O.Maler, Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems (an extended abstract). In: Annual Symposium on Theoretical Aspects of Computer Science (STACS). (1995)
24. de Alfaro, L., Henzinger., A., Majumdar, R.: Symbolic algorithms for infinite-state games. In: 12th International Conference on Concurrency Theory (CONCUR'01). (2001)
25. Liu, C., Anderson, J.H.: Task scheduling with self-suspensions in soft real-time multiprocessor systems. In: Proceedings of the 2009 30th IEEE Real-Time Systems Symposium. RTSS '09 (2009)
26. Liu, J.W.S.W.: Real-Time Systems. 1st edn. Prentice Hall PTR, Upper Saddle River, NJ, USA (2000)
27. Lakshmanan, K., Rajkumar, R.: Scheduling self-suspending real-time tasks with rate-monotonic priorities. In: IEEE Real-Time and Embedded Technology and Applications Symposium. (2010)
28. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K., Lime, D.: Uppaal-tiga: Time for playing games! In: International Conference on Computer Aided Verification (CAV). (2007)