Timed Automata with Asynchronous Processes: Schedulability and Decidability

Elena Fersman, Paul Pettersson and Wang Yi*
Uppsala University
Department of Computer Systems
P.O. Box 325, S-751 05 Uppsala, Sweden
{elenaf,paupet,yi}@docs.uu.se

Abstract. We adopt a model of timed automata extended with asynchronous processes i.e. tasks triggered by events. A task is an executable program characterized by its worst execution time and deadline, and possibly other parameters such as priorities etc. for scheduling. The main idea is to associate each location of an automaton with a task (or a set of tasks). A transition leading to a location denotes an event triggering the tasks and the clock constraint on the transition specifies the possible arrival times of the event. This yields a model for real-time systems expressive enough to describe concurrency and synchronization, and tasks which may be periodic, sporadic, preemptive and (or) non-preemptive. An automaton is schedulable if there exists a (preemptive or non-preemptive) scheduling strategy such that all possible sequences of events accepted by the automaton are schedulable in the sense that all associated tasks can be computed within their deadlines.

Our main result is that the schedulability checking problem is decidable. The problem has been conjectured to be undecidable due to the nature of preemptive scheduling. To our knowledge, this is the first decidability result for preemptive scheduling in dense-time models. The proof is based on a class of suspension automata, that is timed automata with subtraction in which clocks may be updated by subtraction operations. We show that if each clock is bounded with a maximal constant and subtraction operations are performed on clocks only in the bounded zone, the reachability problem is decidable. The crucial observation is that subtraction preserves region equivalence in the bounded clock zone, and the schedulability checking problem can be encoded as a reachability problem for such automata. Based on the proof, we have developed a symbolic technique, which has been implemented as a prototype tool for schedulability analysis.

1 Introduction

Real time systems are often designed as a set of tasks imposed with hard time constraints such as deadlines. The tasks may be triggered periodically or sporadically (non-periodically) by either time or events. Hard time constraints mean that whenever a task is triggered (released), it must be computed within the given deadline. One of the most important issues in the development of real time systems is *schedulability analysis* prior to the implementation. It is to check, based on models of the environment (controlled systems) and control software under development, whether the tasks can be guaranteed to meet their deadlines with the available computing resources such as processor time.

* Contact author.

In the area of real time scheduling, researchers have developed various methods [But97] e.g. rate monotonic scheduling, which are widely applied in the analysis of periodic tasks in time-driven systems. To deal with *non-periodic* tasks in event–driven systems, the standard method is to consider non-periodic tasks as periodic using the estimated *minimal* inter-arrival times as *task periods*. Clearly, the analysis based on such a task model would be pessimistic in many cases, e.g. a task set which is schedulable may be considered as non-schedulable as the inter-arrival times of the tasks may vary over time, that are not necessary minimal. To achieve more precise analysis, we need task models that allow more precise and relaxed timing constraints.

In recent years, in the area of formal methods, there have been several advances in formal modeling and analysis of real time systems based the theory of timed automata due to the pioneering work of Alur and Dill [AD94]. Notably, a number of verification tools have been developed (e.g. Kronos and UPPAAL [DY95,BLL+96,LPY97]) in the framework of timed automata, that have been successfully applied in industrial case studies (e.g. [BGK+96,LP97,LPY98]). Timed automata have proved expressive enough to model many real-life examples, in particular, for event-driven systems. The advantage with timed automata in modeling systems is that one may specify more relaxed timing constraints on events (i.e. discrete transitions) than the traditional approach in which events are often considered to be periodic. Moreover, timed automata also allow for modelling other behavioral aspects of systems such as synchronization and concurrency. However, it is not clear how timed automata can be used for schedulability analysis because there is no support for specifying resource requirements and hard time constraints on computations.

Following the work of [EWY98], we propose to extend timed automata with asynchronous processes i.e. tasks triggered by events. A task is an executable program characterized by its worst case execution time and deadline, and possibly other parameters such as priorities etc for scheduling. The main idea is to associate each location of an automaton with a task (or a set of tasks in the general case). Intuitively a transition leading to a location in the automaton denotes an event triggering the task and the guard (clock constraints) on the transition specifies the possible arrival times of the event. Semantically, an automaton may perform two types of transitions. Delay transitions correspond to the execution of running tasks (with highest priority) and idling for the other waiting tasks. Discrete transitions correspond to the arrival of new task instances. Whenever a task is triggered, it will be put in the scheduling queue for execution (i.e. the ready queue in operating systems). We assume that the tasks will be executed according to a given scheduling strategy e.g. FPS (fixed priority scheduling) or EDF (earliest deadline first). Thus during the execution of an automaton, there may be a number of processes (released tasks) running logically in parallel.

For example, consider the automaton shown in Figure 1. It has three locations l_0, l_1, l_2, and two tasks P and Q (triggered by a and b) with computing time and relative deadline in brackets $(2, 10)$, and $(4, 8)$ respectively. The automaton models a system starting in its initial location may move to location l_1 by event a at any time, which triggers the task P. In location l_1, as long as the constraints $x \geq 10$ and $y \leq 40$ are satisfied and event a occurs, a copy (instance) of task P will be created and put in the scheduling queue. However, in location l_1, it can not create more than 5 instances of task P because the constraint $y \leq 40$ will be violated after 40 time units. In fact, every copy will be computed before the next instance arrives and the scheduling queue may contain at most one task instance and no task instance will miss its deadline in location l_1. In location l_1, the system is also able to accept event b, trigger the task Q and then switch to location l_2. In l_2, because there is no constraints labelled on the

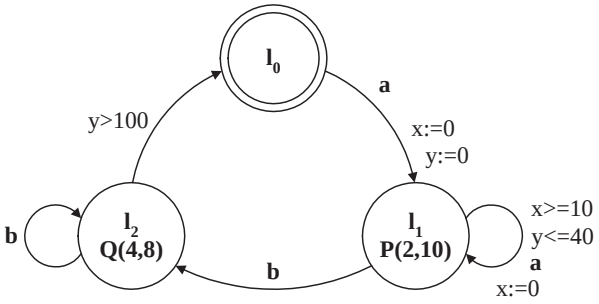

Fig. 1. Timed automaton with asynchronous processes.

b-transition, it may accept an unbounded number of b's, and create an unbounded number of copies of task Q in 0 time. in zero time because there is no constraint on the b-transition. This is the so-called zeno behavior. However, after more than two copies of Q, the queue will be non schedulable[1]. This means that the system is non-schedulable. Thus, zeno-behaviour will correspond to non schedulability in our setting, which is a natural property of the model.

We shall formalize the notion of schedulability in terms of reachable states. A state of an extended automaton will be a triple (l, u, q) consisting of a location l, a clock assignment u and a task queue q. The task queue contains pairs of remaining computing times and relative deadlines for all released tasks. Naturally, a state (l, u, q) is schedulable if q is schedulable in the sense there there exists a scheduling strategy with which all tasks in q can be computed within their deadlines. An automaton is schedulable if all reachable states of the automaton are schedulable. Note that the notion of schedulablity above is relative to the scheduling strategy. A task queue which is not schedulable with one scheduling strategy, may be schedulable with another strategy. In [EWY98], we have shown that under the assumption that the tasks are non-preemptive, the schedulability checking problem can be transformed to a reachability problem for ordinary timed automata and thus it is decidable. The result essentially means that given an automaton it is possible to check whether the automaton is schedulable with any *non-preemptive* scheduling strategy. For *preemptive scheduling* strategies, it has been conjectured that the schedulability checking problem is undecidable because in *preemptive scheduling* we must use stop-watches to accumulate computing times for tasks. It appears that the computation model behind preemptive scheduling is stop-watch automata for which it is known that the reachability problem is undecidable.

Surprisingly the above conjecture is wrong. In this paper, we establish that the schedulability checking problem for extended timed automata is decidable for preemptive scheduling. Note that the preemptive earliest deadline first algorithm (EDF) is optimal in the sense if EDF can not schedule a task set, no other algorithms can. Thus our result applies to not only preemptive scheduling, but any scheduling strategy. That is, for a given extended timed automata, it is checkable if there exists a scheduling strategy (preemtive or non preemtive) with which the automaton is schedulable. The main idea in the proof is to model scheduling strategies with variants of timed automata (not stop-watch automata), and then encode the schedulability analysis

[1] According to the optimal scheduling strategy EDF, no scheduling strategy will be able to schedule the queue $[Q(4,8), Q(4,8), Q(4,8)]$ so that the deadline for the last instance of Q can be met.

problem as a reachability problem. We identify a variant of timed automata (a class of suspension automata): *timed automata with subtraction* in which clocks may be updated by subtraction. We show that if each clock is bounded with a maximal constant and subtraction operations are performed on clocks only in the bounded zone, the reachability problem is decidable[2]. The crucial observation is that the schedulability checking problem can be translated to a reachability problem for bounded timed automata with subtraction.

The rest of this paper is organized as follows: Section 2 presents the syntax and semantics of timed automata extended with tasks. Section 3 describes scheduling problems related to the extended model. Section 4 is devoted to the main proof that the schedulability checking problem for preemptive scheduling is decidable. Section 5 concludes the paper with summarized results and future work, as well as a brief summary and comparison with related work.

2 Timed Automata with Tasks

Recall that a timed automaton is a standard finite-state automaton extended with a finite collection of real-valued clocks. The transitions of a timed automaton are labelled with a *guard* (constraints on clocks), an *action*, and a *clock reset* (a subset of clocks to be reset to zero). We may view a timed automaton as an abstract model of a running system. The model describes the possible events (alphabets accepted by the automaton) that may occur during the operation of the system and the occurrence of the events must obey the timing constraints (given by the clock constraints).

However it is not clear how events or action symbols accepted by a timed automaton should be handled or computed. In fact, there is no way in timed automata to specify resource requirements and hard time constraints on computations. We propose to extend timed automata with asynchronous processes i.e. tasks triggered by events asynchronously. The main idea is to associate each location of a timed automaton with an executable program (or a set of programs in the general case) called a *task type* or simply a task.

2.1 Syntax

Let $\mathcal{P}$ ranged over by P, Q, R, denote a finite set of task types. A task type may have different instances that are copies of the same program with different inputs. We further assume that the *worst case execution times* and *hard deadlines* of tasks in $\mathcal{P}$ are known [3]. Thus, each task P is characterized as a pair of natural numbers denoted $P(C, D)$ with $C \leq D$, where C is the worst case execution time of P and D is the relative deadline for P. The deadline D is a relative deadline meaning that when task P is released, it should finish within D time units. We shall use $C(P)$ and $D(P)$ to denote the worst case execution time and relative deadline of P respectively.

As in timed automata, assume a finite set of alphabets Act for actions and a finite set of real-valued variables $\mathcal{C}$ for clocks. We use a, b etc to range over Act and x_1, x_2

[2] Because the subtraction preserves region equivalence in the bounded clock zone, and therefore the state space of a bounded automaton with subtraction can be partitioned according to region equivalence

[3] Note that tasks may have other parameters such as fixed priority for scheduling and other resource requirements e.g. on memory consumption. For simplicity, in this paper, we only consider computing time and deadline.

etc. to range over $\mathcal{C}$. We use $\mathcal{B}(\mathcal{C})$ ranged over by g to denote the set of conjunctive formulas of atomic constraints in the form: $x_i \sim C$ or $x_i - x_j \sim D$ where $x_i, x_j \in \mathcal{C}$ are clocks, $\sim \in \{\leq, <, \geq, >\}$, and C, D are natural numbers. The elements of $\mathcal{B}(\mathcal{C})$ are called *clock constraints*.

Definition 1. *A timed automaton extended with tasks, over actions Act, clocks $\mathcal{C}$ and tasks $\mathcal{P}$ is a tuple $\langle N, l_0, E, I, M\rangle$ where*

- *$\langle N, l_0, E, I\rangle$ is a timed automaton where*
  - *N is a finite set of locations ranged over by l, m, n,*
  - *$l_0 \in N$ is the initial location, and*
  - *$E \subseteq N \times \mathcal{B}(\mathcal{C}) \times Act \times 2^{\mathcal{C}} \times N$ is the set of edges.*
  - *$I : N \mapsto \mathcal{B}(\mathcal{C})$ is a function assigning each location with a clock constraint (a location invariant).*
- *$M : N \hookrightarrow \mathcal{P}$ is a partial function assigning locations with tasks* [4].

Intuitively, a discrete transition in an automaton denotes an event triggering a task and the guard (clock constraints) on the transition specifies all the possible arrival times of the event (or the associated task). Whenever a task is triggered, it will be put in the scheduling (or task) queue for execution (corresponding to the ready queue in operating systems).

Clearly extended timed automata is at least as expressive as timed automata; for example, if M is the empty mapping, we will have ordinary timed automata. It is a rather general and expressive model. For example, it may model time-triggered periodic tasks as a simple automaton as shown in Figure 2(a) where P is a periodic task with computing time 2, deadline 8 and period 20. More generally it may model systems containing both periodic and sporadic tasks as shown in Figure 2(b) which is a system consisting of 4 tasks as annotation on locations, where P_1 and P_2 are periodic with periods 20 and 40 respectively (specified by the constraints: x=20 and x=40), and Q_1 and Q_2 are sporadic or event driven (by event a and b respectively).

In general, there may be a number of processes (released tasks) running logically in parallel. For example, an instance of task Q_2 may be released before the preceding instance of task P_1 has been computed because there is no constraint on when b_2 will arrive. This means that the scheduling queue may contains at least P_1 and Q_2. In fact, instances of all four task types may appear in the queue at the same time.

To have a more general model, we may allow data variables shared between automata and tasks. For example. a boolean can be used to denote the completion of a task. The arrivals of events may be effected by the completion of certain tasks. This will not add any technical difficulty. However for simplicity, we will not consider synchronization between an automaton and the associated tasks in this paper. The only requirement on the completion of a task is given by the deadline. When a task is finished does not effect the control behavior specified in the automaton.

To handle concurrency and synchronization, parallel composition of extended timed automata may be introduced in the same way as for ordinary timed automata (e.g. see [LPY95]) using the notion of synchronization function [HK89]. For example, consider the parallel composition $A||B$ of A and B over the same set of actions Act. The set

[4] Note that M is a partial function meaning that some of the locations may have no task. Note also that we may also associate a location with a set of tasks instead of a single one. It will not introduce technical difficulties.

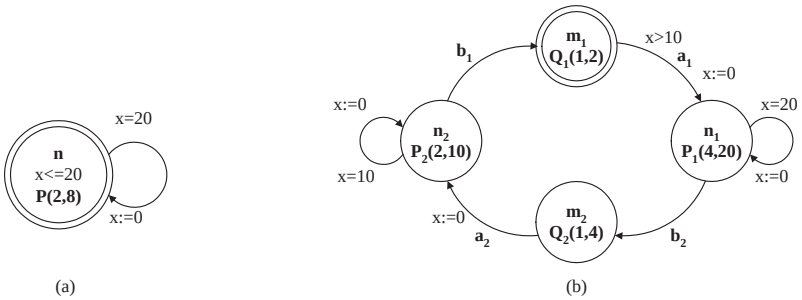

Fig. 2. Modeling Periodic and Sporadic Tasks.

of nodes of $A||B$ is simply the product of A's and B's nodes, the set of clocks is the (disjoint) union of A's and B's clocks, the edges are based on synchronizable A's and B's edges with enabling conditions conjuncted and reset-sets unioned. Note that due to the notion of synchronization function [HK89], the action set of the parallel composition will be Act and thus the task assignment function for $A||B$ is the same as for A and B.

2.2 Operational Semantics

Semantically, an extended timed automaton may perform two types of transitions just as standard timed automata. But the difference is that delay transitions correspond to the execution of running tasks with highest priority (or earliest deadline) and idling for the other tasks waiting to run. Discrete transitions corresponds to the arrival of new task instances.

We represent the values of clocks as functions (called clock assignments) from $\mathcal{C}$ to the non–negative reals $\mathcal{R}_{\geq 0}$. We denote by $\mathcal{V}$ the set of clock assignments for $\mathcal{C}$. Naturally, a semantic state of an automaton is a triple (l, u, q) where l is the current control location, u denotes the current values of clocks, and q is the current task queue. We assume that the task queue takes the form: $[P_1(c_0, d_0), P_2(c_1, d_1)...P_n(c_n, d_n)]$ where $P_i(c_i, d_i)$ denotes a released instance of task type P_i with remaining computing time c_i and relative deadline d_i.

Assume that there are a number of processors running the released task instances according to a certain scheduling strategy Sch e.g. FPS (fixed priority scheduling) or EDF (earliest deadline first) which sorts the task queue whenever new tasks arrives according to task parameters e.g. deadlines. In general, we assume that a scheduling strategy is a sorting function which may change the ordering of the queue elements only. Thus an action transition will result in a sorted queue including the newly released tasks by the transition. A delay transition with c time units is to execute the task in the first positon of the queue with c time units. Thus the delay transition will decrease the computing time of the first task with c. If its computation time becomes 0, the task should be removed from the queue (shrinking). We adopt the structural equivalence over queues respecting $[P_1(0, d), P_2(c_1, d_1)...P_n(c_n, d_n)] = [P_2(c_1, d_1)...P_n(c_n, d_n)]$. Moreover, after a delay transition with c time units, the deadlines of all tasks in the queue will be decreased by c (since time has progressed by c). To summarize the above intuition, we introduce the following functions on task queues:

- Sch is a sorting function for task queues (or lists), that may change the ordering of the queue elements only. For example, $\mathsf{EDF}([P(3.1,10),Q(4,5.3)]) = [Q(4,5.3),P(3.1,10)])$. We call such sorting functions scheduling strategies that may be preemptive or non-preemptive [5].
- Run is a function which given a real number t and a task queue q returns the resulted task queue after t time units of execution according to available computing resources. For simplicity, we assume that only one processor available [6]. Then the meaning of $\mathsf{Run}(q,t)$ should be obvious and it can be defined inductively. For example, let $q = [Q(4,5),P(3,10)]$. Then $\mathsf{Run}(q,6) = [P(1,4)]$ in which the first task is finished and the second has been executed for 2 time units.

Further, for a real number $t \in \mathcal{R}_{\geq 0}$, we use $u+t$ to denote the clock assignment which maps each clock x to the value $u(x)+t$, $u \models g$ to denote that the clock assignment u satisfies the constraint g and $u[r \mapsto 0]$ for $r \subseteq \mathcal{C}$, to denote the clock assignment which maps each clock in r to 0 and agrees with u for the other clocks (i.e. $\mathcal{C}\backslash r$).

Now we are ready to present the operational semantics for extended timed automata by transition rules:

Definition 2. *Given a scheduling strategy* Sch [7]*, the semantics of an extended timed automaton* $\langle N, l_0, E, I, M\rangle$ *with initial state* (l_0,u_0,q_0) *is a transition system defined by the following rules:*

- $(l,u,q) \xrightarrow{a}_{\mathsf{Sch}} (m, u[r\mapsto 0], \mathsf{Sch}(M(m) :: q))$ *if* $l \xrightarrow{g,a,r} m$ *and* $u \models g$
- $(l,u,q) \xrightarrow{t}_{\mathsf{Sch}} (l, u+t, \mathsf{Run}(q,t))$ *if* $(u+t) \models I(l)$

where $M(m) :: q$ *denotes the queue with* $M(m)$ *inserted in* q*.*

We shall omit Sch from the transition relation whenever it is understood from the context.

Consider the automaton in Figure 2(b). Assume that preemptive earliest deadline (EDF) is used to schedule the task queue i.e. Sch is sorting the queue in EDF order, (and Run is as defined earlier). Then the automaton with initial state $(m_1, [x = 0], [Q_1(1,2)])$ may demonstrate the following sequence of typical transitions:

$$
\begin{aligned}
(l_0,[x=0],[Q_1(1,2)]) &\xrightarrow{0.5} (m_1,[x=0.5],[Q_1(0.5,1.5)])\\
&\xrightarrow{10} (m_1,[x=10.5],[Q_1(0,0)]) = (m_1,[x=10.5],[])\\
&\xrightarrow{a_1} (n_1,[x=0],[P_1(4,20)])\\
&\xrightarrow{0.5} (n_1,[x=0.5],[P_1(3.5,19.5)])\\
&\xrightarrow{b2} (m_2,[x=0.5],[Q_2(1,4),P_1(3.5,19.5)])\\
&\xrightarrow{0.3} (m_2,[x=0.8],[Q_2(0.7,3.7),P_1(3.5,19.2)])\\
&\xrightarrow{a_2} (n_2,[x=0],[Q_2(0.7,3.7),P_2(2,10),P_1(3.5,19.2)])\\
&\xrightarrow{b_1} (m_1,[x=0],[Q_2(0.7,3.7),Q_1(1,2),P_2(2,10),P_1(3.5,19.2)])\\
&\ldots
\end{aligned}
$$

[5] As in scheduling theory, we adopt the standard assumptions on scheduling strategies: A non-preemptive strategy will never change the position of the first element of a queue. A preemtive strategy may change the ordering of task types only, but never change the ordering of task instances of the same type.

[6] However, the semantics may be extended to multi-processor setting. We only need to modify the function Run according to available resources (i.e. number of processors in this case).

[7] Note that we fixed Run to be the function that represents a one-processor system.

Note that the queue is always ordered by EDF with increasing deadlines and though the queue is growing, in location m_1, the generation of new task instances will be slowed down by the constraint $x > 10$ labeled on the edge m_1 to n_1 and the queue will be reduced by the following transition:

$$(m_1, [x = 0], [Q_2(0.7, 3.7), Q_1(1, 2), P_2(2, 10), P_1(3.5, 19.2)]) \xrightarrow{10} (n_1, [x = 10], [])$$

The automaton may also perform other sequences of transitions triggering copies of tasks. For example, it may stay in location n_1 or n_2 where instances of the periodic tasks P_1 and P_2 may be released and executed.

3 Schedulability Analysis

In this section we study verification problems related to the model presented in previous section. First, we have the same notion of reachability as for timed automata.

Definition 3. *We shall write* $(l, u, q) \longrightarrow (l', u', q')$ *if* $(l, u, q) \xrightarrow{a} (l', u', q')$ *for an action* a *or* $(l, u, q) \xrightarrow{t} (l', u', q')$ *for a delay* t*. For an automaton with initial state* (l_0, u_0, q_0)*,* (l, u, q) *is reachable iff* $(l_0, u_0, q_0)(\longrightarrow)^*(l, u, q)$*.*

Note that the reachable state space of an extended timed automaton is infinite because of not only the real-valued clocks, but also the unbounded size of the task queue. However, for certain analysis, e.g. safety properties (that are not related to the task queue), we may only be interested in the reachability of locations. A nice property of our extension is that the location reachability problem can be checked by the same technique as for timed automata [DOTY95,BLL+96,ACH+95,YPD94]. So we may view the original timed automaton (without task assignment) as an abstraction of its extended version, which preserves location reachability. The existing model checking tools such as [DOTY95,BLL+96] can be applied directly to verify the abstract models.

But if properties related to the task queue are of interests, we need to develop a new verification technique. One of the most interesting properties of extended automata related to the task queue is schedulability.

According to the transition rules, the task queue is growing with action transitions and shrinking with delay transitions. Multiple copies (instances) of the same task type may appear in the queue [8]. To illustrate the changing size of the task queue, we consider again the automaton shown in Figure 1 in Section 1. In location l_1, it can never generate more than 5 instances of P due to the constraint $y \leq 40$, namely the number of copies may be bounded by the clock constraints [9]. This is illustrated by the following transitions:

[8] In practice, copies of the same task type may have different input to compute.

[9] In fact, in location l_1, the size of the queue will be bounded by 1 because every released instance will be computed before the next release that will not arrive within 10 time units due to the constraint $x \geq 10$.

$$
\begin{aligned}
(l_0, [x=0, y=0], []) &\xrightarrow{3} (l_0, [x=3, y=3], []) \\
&\xrightarrow{a} (l_1, [x=0, y=0], [P(2,10)]) \\
&\xrightarrow{2.5} (l_1, [x=2.5, y=2.5], [P(0,7.5)]) = (l_1, [x=2.5, y=2.5], []) \\
&\xrightarrow{7.5} (l_1, [x=10, y=10], []) \\
&\xrightarrow{a} (l_1, [x=0, y=10], [P(2,10)]) \\
&\ldots \\
&\xrightarrow{a} (l_1, [x=0, y=40], [P(2,10)]) \\
&\xrightarrow{3.4} (l_1, [x=3.4, y=43.4], [P(0,6.6)]) = (l_1, [x=3.4, y=43.4], [])
\end{aligned}
$$

In fact, the queue size in location l_1 is bounded by 1. But in location l_2, an infinite number of copies of Q may be released in zero time because there is no constraint on the b-transition, which demonstrates the so-called zeno behavior:

$$
\begin{aligned}
(l_1, [x=3.4, y=43.4], []) &\xrightarrow{b} (l_2, [x=3.4, y=43.4], [Q(4,8)]) \\
&\xrightarrow{b} (l_2, [x=3.4, y=43.4], [Q(4,8), Q(4,8)]) \\
&\xrightarrow{b} (l_2, [x=3.4, y=43.4], [Q(4,8), Q(4,8), Q(4,8)]) \\
&\ldots
\end{aligned}
$$

But note that after more than two copies of Q, the queue will be non schedulable[10]. We notice that zeno-behaviour will correspond to non schedulability in our setting which is a nice property of the model. We shall see that non-schedulability can be checked by reachability analysis. However zeno-freeness does not necessarily implies schedulability.

Now we formalize the notion of schedulability.

Definition 4. *(Schedulability) A state (l, u, q) where $q = [P_1(c_1, d_1) \ldots P_n(c_n, d_n)]$ is a failure denoted (l, u, Error) if there exists i such that $c_i \geq 0$ and $d_i < 0$, that is, a task failed in meeting its deadline. Naturally an automaton A with initial state (l_0, u_0, q_0) is non-schedulable with Sch iff $(l_0, u_0, q_0)(\longrightarrow_{\mathsf{Sch}})^*(l, u, \mathsf{Error})$ for some l and u. Otherwise, we say that A is schedulable with Sch.*

More generally, we say that A is schedulable iff there exists a scheduling strategy Sch with which A is schedulable.

The schedulability of a state may be checked by the standard schedulability test. We say that (l, u, q) is schedulable with Sch if $\mathsf{Sch}(q) = [P_1(c_1, d_1) \ldots P_n(c_n, d_n)]$ and $(\sum_{i \leq k} c_i) \leq d_k$ for all $k \leq n$. Alternatively, an automaton is schedulable with Sch if all its reachable states are schedulable with Sch.

Note that checking schedulability of a state is a trivial task according to the definition. But checking the relative schedulability of an automaton with respects to a given scheduling strategy is not easy, and checking the general schedulability (equivalent to finding a scheduling strategy to schedule the automaton) is even more difficult.

Fortunately the queues of all schedulable states of an automaton are bounded. First note that a task instance that has been started can not be preempted by another instance of the same task type. This means that there is only one instance of each task type in the queue whose computing time can be a real number and it can be arbitrarily

[10] According to the optimal scheduling strategy EDF, no scheduling strategy will be able to schedule the queue $[Q(4,8), Q(4,8), Q(4,8)]$ to meet the deadline for the last instance of Q.

small. Thus the number of instances of each task type $P \in \mathcal{P}$, in a schedulable queue is bounded by $\lceil D(P)/C(P) \rceil$ and the size of schedulable queues is bounded by

$$\sum_{P \in \mathcal{P}} \lceil D(P)/C(P) \rceil$$

We will code schedulability checking problems as reachability problems. First, we consider the case of non-preemptive scheduling to introduce the problems we are aiming at in this paper. We have the following positive result.

Theorem 1. *The problem of checking schedulability relative to non-preemptive scheduling strategy for extended timed automata is decidable.*

Proof. A detailed proof is given in [EWY98]. We scketch the proof idea here. It is to code the given scheduling strategy as a timed automaton (called the scheduler) denoted $E(\mathsf{Sch})$ which uses clocks to remember computing times and relative deadlines for released tasks. Note that for non-preemptive scheduling, only one instance is running, which means that only one clock, denoted c is needed to remember the computing time for the running task and one clock, denoted d_i for each released task instance P_i to remember the deadline.

The scheduler automaton is constructed as follows: Whenever a task instance P_i is released by an event $\mathsf{release}_i$, d_i is reset to 0. Whenever a task is started to run[11], c is reset to 0. Whenever the constraint $d_i = 0$ is satisfied, and P_i is not running, an error-state (non-schedulable) should be reached.

We also need to transform the original automaton A to $E(A)$ to synchronize with the scheduler that P_i is released whenever a location, say l to which P_i is associated, is reached. This is done simply by replacing actions labeled on transitions leading to l with $\mathsf{release}_i$.

Finally we construct the product automaton $E(\mathsf{Sch})||E(A)$ in which both $E(\mathsf{Sch})$ and $E(A)$ can only synchronize on identical action symbols namely $\mathsf{release}_i$'s. It can be proved that if an error-state of the product automaton is reachable, the original extended timed automaton is non-schedulable.

For *preemptive scheduling* strategies, it has been conjectured that the schedulability checking problem is undecidable. The reason is that if we use the same ideas as for non-preemptive scheduling to encode a preemptive scheduling strategy, we must use stop-watches (or integrators) to add up computing times for suspended tasks. It appears that the computation model behind preemptive scheduling is stop-watch automata for which it is known that the reachability problem is undecidable. Surprisingly this conjecture is wrong.

Theorem 2. *The problem of checking schedulability relative to preemptive scheduling strategy for extended timed automata is decidable.*

The rest of this paper will be devoted to the proof of this theorem. It follows from Lemma 3, 4, and 5 established in the following section.

Before we go further, we state a more general result follows from the above theorem.

[11] Initially or the running task P_i is finished i.e. the constraint $c = C(P)$ is satisfied.

Theorem 3. *The problem of checking schedulability for extended timed automata is decidable.*

From scheduling theory[But97], we know that the preemptive version of Earliest Deadline First scheduling (EDF) is optimal in the sense that if a task queue is non schedulable with EDF, it can not be schedulable with any other scheduling strategy (preemptive or non preemptive). Thus, the general schedulability checking problem is equivalent to the relative schedulability checking with respects to EDF.

4 Decidability and Proofs

We shall code the schedulability checking problem as a reachability problem. Note that in the case of non preemptive scheduling, we are able to do so using timed automaton only. For preemptive scheduling, we need a more expressive model.

4.1 Timed Automata with Subtraction

Definition 5. *A timed automaton with subtraction is a timed automaton in which clocks may be updated by subtraction in the form $x := x - C$ in addition to reset of the form: $x := 0$, where C is a natural number.*

Note that this is the so called suspension automata [MV94] and also called updatable automata in [BDFP00]. It is known that the reachability problem for this class of automata is undecidable. However, for the following class of suspension automata, location reachability is decidable.

Definition 6. *(Bounded Timed Automata with Subtraction) A timed automaton is bounded iff for all its reachable states (l, u, q), there is a maximal constant C_x for each clock x such that*

1. *$u(x) \geq 0$ for all clocks x, i.e. clock values should not be negative* [12]*, and*
2. *$u(x) \leq C_x$ for all l' such that $l \xrightarrow{gar} l'$ and $(x := x - C) \in r$ for some C.*

Note that in general, it may be difficult to compute the maximal constants from the syntax of an automaton. But we shall see that we can compute the constants for our encoding of scheduling problems.

Because subtraction operations on clocks are performed only within a bounded area. It preserves the region equivalence. We adopt the standard definition due to Alur and Dill [AD94].

Definition 7. *(Region Equivalence denoted $\sim$) For a clock $x \in \mathcal{C}$, let C_x be a constant (the ceiling of clock x). For a real number t, let $\{t\}$ denote the fractional part of t, and $\lfloor t \rfloor$ denote its integer part. For clock assignments $u, v \in \mathcal{V}$, u, v are region-equivalent denote $u \sim v$ iff*

1. *for each clock x, either $\lfloor u(x) \rfloor = \lfloor v(x) \rfloor$ or $(u(x) > C_x$ and $v(x) > C_x)$, and*

[12] This condition may be relaxed (e.g. the clock values may be negative, but bounded with a lower bound). But this is precisely what we need for the main proof.

2. *for all clocks* x, y *if* $u(x) \leq C_x$ *and* $u(y) \leq C_y$ *then*
   (a) $(\{u(x)\} = 0$ *iff* $\{v(x)\} = 0$ *and*
   (b) $(\{u(x)\} \leq \{u(y)\}$ *iff* $\{v(x)\} \leq \{v(y)\})$

It is known that region equivalence is preserved by the delay (addition) and reset. In the following, we establish that region equivalence is also preserved by subtraction for clocks that are bounded as defined in Definition 6. For a clock assignment u, let $u(x - C)$ denote the assignment: $u(x - C)(x) = u(x) - C$ and $u(x - C)(y) = u(y)$ for $y \neq x$. The following results states that $\sim$ is a bisimulation with respect to the operations: addition, reset and the conditional subtraction.

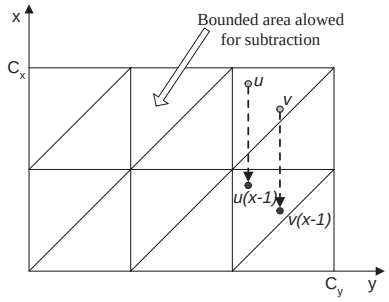

Fig. 3. Region equivalence preserved by subtraction when clocks are bounded.

Lemma 1. *Let* $u, v \in \mathcal{V}$. *Then* $u \sim v$ *implies*

1. $u + t \sim v + t$ *for a positive real number* t, *and*
2. $u[x \mapsto 0] \sim v[x \mapsto 0]$ *for a clock* x *and*
3. $u(x - C) \sim v(x - C)$ *for all natural numbers* C *such that* $C \leq u(x) \leq C_x$. *for a natural number* C.

Proof. The proof for the first two items can be found in the literature e.g. [LW97].

We check the third according to the definition of regioin equivalence. It is illustrated in Figure 3 that the equivalence is preserved by subtraction in the bounded area. Assume that $u \sim v$, and for each clock x we also have $C \leq u(x) \leq C_x$. Note that because $u \sim v$, and $C \leq u(x)$, we have $C \leq v(x)$. Then we have $u(x - C)(y) \geq 0$ and $v(x - C)(y) \geq 0$ for any clock y. Now we have two cases to check:

1. If $\lfloor u(x) \rfloor = \lfloor v(x) \rfloor$, obviously we have $\lfloor u(x) - C \rfloor = \lfloor v(x) - C \rfloor$ for a natural number C and then $\lfloor u(x - C) \rfloor = \lfloor v(x - C) \rfloor$ by definition.
2. Note that the subtraction operation on clocks does not change the fractional parts of clock values. Therefore for all clocks y and z, we have:
   (a) $(\{u(x - C)(y)\} = 0$ iff $\{v(x - C)(y)\} = 0$ and
   (b) $(\{u(x - C)(y)\} \leq \{u(x - C)(z)\}$ iff $\{v(x - C)(y)\} \leq \{v(x - C)(z)\})$

In fact, region equivalence over clock assignments induces a bisimulation over reachable states of automata, which can be used to partition the whole state space as a finite number of equivalence classes.

Lemma 2. *Assume a bounded timed automaton with subtraction, a location l and clock assignments u and v. Then $u \sim v$ implies that*

1. *whenever* $(l,u) \longrightarrow (l',u')$ *then* $(l,v) \longrightarrow (l',v')$ *for for some* v' *s.t.* $u' \sim v'$ *and*
2. *whenever* $(l,v) \longrightarrow (l',v')$ *then* $(l,u) \longrightarrow (l',u')$ *for for some* u' *s.t.* $u' \sim v'$.

Proof. It follows from Lemma 1.

Note that the above lemma essentially states that if $u \sim v$ then (l,u) and (l,v) are bisimular, which implies the following result.

Lemma 3. *The location reachability problem for bounded timed automata with subtraction, whose clocks are bounded with known maximal constants is decidable.*

Proof. Because each clock of the automaton is bounded by a maximal constant, it follows from lemma 2 that for each location l, there is a finite number of equivalence classes of states which are equivalent in the sense that they will reach the same equivalence classes of states. Because the number of locations of an automaton is finite, the whole state space of an automaton can be partitioned into finite number of such equivalence classes.

4.2 Encoding of Schedulability as Reachability

Assume an automaton A extended with tasks, and a *preemptive scheduling* strategy Sch. The aim is to check if A is schedulable with Sch. As for the case of *non-preemptive scheduling* (Theorem 1), we construct $E(A)$ and $E(\mathsf{Sch})$, and check a pre-defined error-state in the product automaton of the two. The construction is illustrated in figure 4.

$E(A)$ is constructed as a timed automaton which is exactly the same as for the non-preemptive case (Theorem 1) and $E(\mathsf{Sch})$ will be constructed as a timed automaton with subtraction.

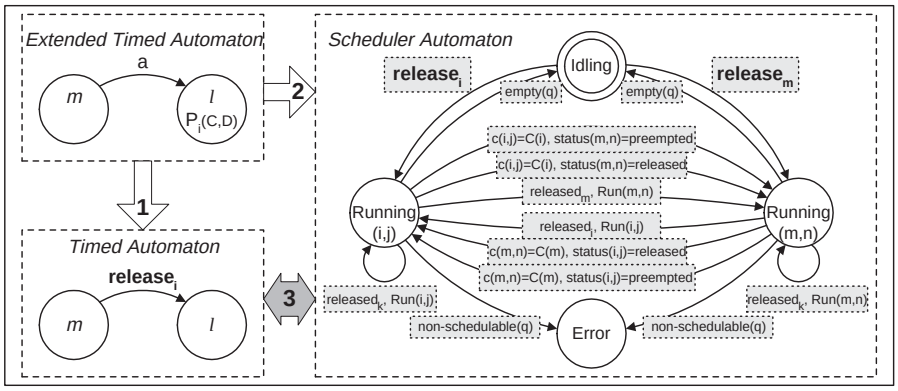

Fig. 4. Encoding of schelulability problem.

We introduce some notation. We use $C(i)$ and $D(i)$ to stand for the worst case execution time and relative deadline respectively for each task type P_i. We use P_{ij} to denote the jth instance of task type P_i.

For each task instance P_{ij}, we have the following state variables:

$\mathsf{status}(i,j)$ initialized to free. Let $\mathsf{status}(i,j) = \mathsf{running}$ stand for that P_{ij} is executing on the processor, $\mathsf{preempted}$ for that P_{ij} is started but not running, and $\mathsf{released}$ for that P_{ij} is released, but not started yet. We use $\mathsf{status}(i,j) = \mathsf{free}$ to denote that P_{ij} is not released yet or position (i,j) of the task queue is free.

Note that according to the definition of scheduling strategy, for all i, there should be only one j such that $\mathsf{status}(i,j) = \mathsf{preempted}$ (only one instance of the same task type is started), and for all i,j, there should be only one pair (k,l) such that $\mathsf{status}(k,l) = \mathsf{running}$ (only one is running for a one-processor system).

We need two clocks for each task instance:

1. $c(i,j)$ (a computing clock) is used to remember the accumulated computing time since P_{ij} was started (when $\mathsf{Run}(i,j)$ became true) [13], and subtracted with $C(k)$ when the running task, say P_{kl}, is finished if it was preempted after it was started.
2. $d(i,j)$ (a deadline clock) is used to remember the deadline and reset to 0 when P_{ij} is released.

We use a triple $\langle c(i,j), d(i,j), \mathsf{status}(i,j)\rangle$ to represent each task instance, and the task queue will contain such triples. We use q to denote the task queue. Note that the maximal number of instances of P_i appearing in a schedulable queue is $\lceil D(i)/C(i)\rceil$. We have a bound on the size of queue as claimed earlier, which is $\sum_{P_i \in \mathcal{P}} \lceil D(i)/C(i)\rceil$. We shall say that queue is empty denoted $\mathsf{empty}(q)$ if $\mathsf{status}(i,j) = \mathsf{free}$ for all i,j.

For a given scheduling strategy Sch, we use the predicate $\mathsf{Run}(m,n)$ to denote that task instance P_{mn} is scheduled to run according to Sch. For a given Sch, it can be coded as a constraint over the state variables. For example, for EDF, $\mathsf{Run}(m,n)$ is the conjunction of the following constraints:

1. $d(k,l) \leq D(k)$ for all k,l such that $\mathsf{status}(k,l) \neq \mathsf{free}$: no deadline is violated yet
2. $\mathsf{status}(m,n) \neq \mathsf{free}$: P_{mn} is released or preempted
3. $D(m) - d(m,n) \leq D(i) - d(i,j)$ for all (i,j): P_{mn} has the shortest deadline

$E(\mathsf{Sch})$ contains three type of locations: Idling, $\mathsf{Running}$ and Error with $\mathsf{Running}$ being parameterized with (i,j) representing the running task instance.

1. Idling denotes that the task queue is empty.
2. $\mathsf{Running}(i,j)$ denotes that task instance P_{ij} is running, that is, $\mathsf{status}(i,j) = \mathsf{running}$. We have an invariant for each $\mathsf{Running}(i,j)$: $c(i,j) \leq C(i)$ and $d(i,j) \leq D(i)$.
3. Error denotes that the task queues are non-schedulable with Sch.

There are five types of edges labeled as follows:

1. Idling to $\mathsf{Running}(i,j)$: there is an edge labeled by

[13] In fact, for each task type, we need only one clock for computing time because only one instance of the same task type may be started.

- guard: none
- action: $\mathsf{release}_i$
- reset: $c(i,j) := 0$, $d(i,j) := 0$, and $\mathsf{status}(i,j) := \mathsf{running}$

2. $\mathsf{Running}(i,j)$ to Idling: there is only one edge labeled with
   - guard: $\mathsf{empty}(q)$ that is, $\mathsf{status}(i,j) = \mathsf{free}$ for all i,j (all positions are free).
   - action: none
   - reset: none
3. $\mathsf{Running}(i,j)$ to $\mathsf{Running}(m,n)$: there are two types of edges.
   (a) The running task P_{ij} is finished, and P_{mn} is scheduled to run by $\mathsf{Run}(m,n)$. There are two cases:
      i. P_{mn} was preempted earlier:
         - guard: $c(i,j) = C(i)$, $\mathsf{status}(m,n) = \mathsf{preempted}$ and $\mathsf{Run}(m,n)$
         - action: none
         - reset: $\mathsf{status}(i,j) := \mathsf{free}$, $\{c(k,l) := c(k,l) - C(i) | \mathsf{status}(k,l) = \mathsf{preempted}\}$, and $\mathsf{status}(m,n) := \mathsf{running}$
      ii. P_{mn} was released, but never preempted (not started yet):
         - guard: $c(i,j) = C(i)$, $\mathsf{status}(m,n) = \mathsf{released}$ and $\mathsf{Run}(m,n)$
         - action: none
         - reset: $\mathsf{status}(i,j) := \mathsf{free}$, $\{c(k,l) := c(k,l) - C(i) | \mathsf{status}(k,l) = \mathsf{preempted}\}$, $c(m,n) := 0, d(m,n) := 0$ and $\mathsf{status}(m,n) := \mathsf{running}$
   (b) A new task P_{mn} is released, which preempts the running task P_{ij}:
      - guard: $\mathsf{status}(m,n) = \mathsf{free}$, and $\mathsf{Run}(m,n)$
      - action: $\mathsf{released}_m$
      - reset: $\mathsf{status}(m,n) := \mathsf{running}$, $c(m,n) := 0$, $d(m,n) := 0$, and $\mathsf{status}(i,j) := \mathsf{preempted}$
4. $\mathsf{Running}(i,j)$ to $\mathsf{Running}(i,j)$. There is only one edge representing the case when a new task is released, and the running task P_{ij} will continue to run:
   - guard: $\mathsf{status}(k,l) = \mathsf{free}$, and $\mathsf{Run}(i,j)$
   - action: $\mathsf{released}_k$
   - reset: $\mathsf{status}(k,l) := \mathsf{released}$ and $d(k,l) := 0$
5. $\mathsf{Running}(i,j)$ to Error: for each pair (k,l), there is an edge labeled by $d(k,l) > D(k)$ and $\mathsf{status}(k,l) \neq \mathsf{free}$ meaning that the task P_{kl} which is released (or preempted) fails in meeting its deadline.

The third step of the encoding is to construct the product automaton $E(Sch) \parallel E(A)$ in which both $E(Sch)$ and $E(A)$ can only synchronize on identical action symbols. Now we show that the product automaton is bounded.

Lemma 4. *All clocks of* $E(\mathsf{Sch})$ *in* $E(Sch) \parallel E(A)$ *are bounded and non negative.*

Proof. All computing clocks $c(k,l)$ are bounded by $D(k)$. This is due to the fact that all edges labeled with a subtraction is guarded by the constraint: $\mathsf{Run}(m,n)$ which requies $d(k,l) \leq D(k)$ (no deadline is violated in order to stay in $\mathsf{Running}$). $d(k,l) \leq D(k)$ implies that $c(k,l) \leq D(k)$ because $c(k,l)$ and $d(k,l)$ are always reset to zero at the same time when a new instance of P_k is released. Thus $c(k,l)$ is bounded.

Secondly, the only possibility for a computing clock, say $c(k,l)$ for task P_{kl}, to become negative is by subtractions. But a subtraction is done on $c(k,l)$ only when a task, say P_{ij}, is finished i.e. $c(i,j) = C(i)$ holds. As $c(i,j)$ is reset to zero before $c(k,l)$ is reset to zero ($\mathsf{status}(k,l) = \mathsf{preempted}$ implying that P_{kl} was released and preeempted earlier), we have $c(i,j) \geq c(k,l)$ implying that $c(i,j) - C(k) \geq 0$. Thus all clocks are non-negative.

Now we have the correctness theorem for our encoding. Assume, without losing generality, that the initial task queue of an automaton is empty.

Lemma 5. *Let A be an extended timed automaton and* Sch *a scheduling strategy. Assume that (l_0, u_0, q_0) and $(\langle l_0, \mathsf{Idling}\rangle, u_0)$ are the initial states of A and the product automaton $E(A)||E(\mathsf{Sch})$ respectively where l_0 is the initial location of A, u_0 and v_0 are clock assignements assigning all clocks with 0 and q_0 is the empty task queue. Then for all l, u, v:*

$$(l_0, u_0, q_0)(\longrightarrow)^*(l, u, \mathsf{Error}) \text{ iff } (\langle l_0, \mathsf{Idling}\rangle, u_0 \cup v_0)(\longrightarrow)^*(\langle l, \mathsf{Error}\rangle, u \cup v)$$

Proof. It is by induction on the length of transition sequence (i.e. reachability steps).

The above lemma states that the schedulability analysis problem can be solved by solving a reachability problem for timed automata extended with subtraction. From Lemma 4, we know that $E(\mathsf{Sch})$ is bounded. Because the reachability problem is decidable due to Lemma 3, we complete the proof for our main result stated in Theorem 2.

5 Conclusions and Related Work

We have studied a model of timed systems (timed automata extended with tasks), which unifies timed automata with the classic task models from scheduling theory. The model can be used to specify resource requirements and hard time constraints on computations, in addition to features offered by timed automata. It is general and expressive enough to describe concurrency and synchronization, and tasks which may be periodic, sporadic, preemptive and (or) non-preemptive. The classic notion of schedulability is nicely extended to automata model. Our main technical contribution is the proof that the schedulability checking problem is decidable. The problem has been conjectured to be undecidable for years. To our knowledge, this is the first decidability result for preemptive scheduling in dense-time models. Based the results, we have developed a symbolic schedulability checking algorithm based using the DBM techniques. It has been implemented in a tool for modeling and scheduling of timed systems. As future work, we shall study the schedule synthesis problem. More precisely given an automaton, it is desirable to characterize the set of schedulable traces accepted by the automaton.

Related work. Scheduling is a well-established area. A numerous number of analysis methods have been published in the literature. For systems restricted to periodic tasks, algorithms such as rate monotonic scheduling and earliest-deadline first are widely used and exact analysis methods exist, see e.g. [KS97]. These methods can be extended to handle non-periodic tasks by considering them as periodic with the minimal inter-arrival time as the task periods. Our work is more related to work on using automata to model and solve scheduling problems. In several papers, e.g. [CL00,Cor94,AGS00], stopwatch automata [ACH+95] are applied to model scheduling algorithms with sporadic tasks. However, as reachability analysis for stopwatch automata is undecidable, this approach can not be applied in general to solve scheduling problems. Approximative and semi-decision algorithms exits but they can only be used to prove that a system is not schedulable (inconclusive for schedulability) and they do not guarantee termination. In [MV94], McManis and Varaiya presents a restricted class of stopwatch automata, called suspension automata, and give a set of conditions for

which reachability analysis of suspension automata is decidable. Unfortunately, the given conditions restrict the applicability of the result as scheduling algorithms such as rate-monotonic and earliest-deadline first, can not be analysed. The work presented in this paper is more general and we show that the presented analysis is guaranteed to terminate for all inputs. Another model for timed system is presented in [AGS00]. It is designed to be compositional and suitable for describing priority-driven scheduling algorithms. The authors show that a number of scheduling algorithms, with or without preemption, can be modeled. However, the problem of schedulability analysis is not addressed in the paper. Related to this is the work on integrating specification and scheduler generation of real-time systems, presented in [AGP+99]. The modeling language, which can be used to describe both sporadic and preemptive tasks, is a Petri-net model with time. The authors presents a controller synthesis technique that can be used to construct an online non-preemptive scheduler that satisfies all given constraints in the model. The problem addressed is restricted to non-preemptive schedulers. In this paper, we use the idea of replacing suspensions of timers by subtraction of clock values, as suggested in [MV94]. It has been shown in [BDFP00] that updating clock variables by subtraction of integer values in timed automata is undecidable in general. We identify a decidable class of such updatable automata, which is precisely what we need to solve scheduling problems.

References

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AGP+99] Karine Altisen, Gregor Gößler, Amir Pnueli, Joseph Sifakis, Stavros Tripakis, and Sergio Yovine. A framework for scheduler synthesis. In *IEEE Real-Time Systems Symposium*, pages 154–163, 1999.

[AGS00] K. Altisen, G. Gößler, and J. Sifakis. A methodology for the construction of scheduled systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, Pune, India*, 2000.

[BDFP00] P. Bouyer, C. Dufourd, E. Fleury, and A. Petit. Expressiveness of updatable timed automata. In *Proc. 25th Int. Symp. Math. Found. Comp. Sci. (MFCS'2000), Bratislava, Slovakia, Aug. 2000*, volume 1893, pages 232–242. Springer, 2000.

[BGK+96] J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using UPPAAL. In *Computer Aided Verification*, pages 244–256, 1996.

[BLL+96] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL: a tool suite for automatic verification of real-time systems. *Lecture Notes in Computer Science*, 1066:232–243, 1996.

[But97] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems. Predictable Scheduling Algorithms and Applications.* Kulwer Academic Publishers, 1997.

[CL00] F. Cassez and F. Laroussinie. Model-checking for hybrid systems by quotienting and constraints solving. In *Proceedings of the twelfth International Conference on Computer-Aided Verification*, volume 1855, pages 373–388, Standford, California, USA, 2000. Springer-Verlag.

[Cor94] J. Corbett. Modeling and analysis of real-time ada tasking programs. In *Proceedings Real-Time Systems Symposium, IEEE Computer Society Press*, pages 132–141, 1994.

[DOTY95] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III: Verification and Control*, volume 1066, pages 208–219, Rutgers University, New Brunswick, NJ, USA, 22–25 October 1995. Springer.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proceedings of the 16th IEEE Real-Time Systems Symposium,* Pisa, Italy, pages 66–75, 1995.

[EWY98] C. Ericsson, A. Wall, and W. Yi. Timed automata as task models for event-driven systems. In *Proceedings of Nordic Workshop on Programming Theory*, 1998.

[HK89] Hans Hüttel and Kim G. Larsen. The use of static constructs in a modal process logic. *Logic at Botik*, 363:163–180, 1989.

[KS97] C. M. Krishna and K. G. Shin. *Real-Time Systems.* McGraw-Hill, 1997.

[LP97] H. Lönn and P. Pettersson. Formal verification of a tdma protocol startup mechanism. In *IEEE Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 235–242, 1997.

[LPY95] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Compositional and symbolic model-checking of real-time systems. In *IEEE Real-Time Systems Symposium*, pages 76–89, 1995.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[LPY98] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal design and analysis of a gear controller. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 281–297, 1998.

[LW97] Kim Guldstrand Larsen and Yi Wang. Time-abstracted bisimulation: Implicit specifications and decidability. *Information and Computation*, 134(2):75–101, 1997.

[MV94] J. McManis and P. Varaiya. Suspension automata: A decidable class of hybrid automata. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification*, volume 818, pages 105–117, Standford, California, USA, 1994. Springer-Verlag.

[YPD94] W. Yi, P. Pettersson, and M. Daniels. Automatic verification of real-time communicating systems by constraint-solving. In *Proc. of the 7th International Conference on Formal Description Techniques*, 1994.