Decidable and Undecidable Problems in Schedulability Analysis Using Timed Automata

Pavel Krčál and Wang Yi
Uppsala University
Department of Information Technology
P.O. Box 337, S-751 05 Uppsala, Sweden
Email: {pavelk,yi}@it.uu.se

Abstract. We study schedulability problems of timed systems with non-uniformly recurring computation tasks. Assume a set of real time tasks whose best and worst execution times, and deadlines are known. We use timed automata to describe the arrival patterns (and release times) of tasks. From the literature, it is known that the schedulability problem for a large class of such systems is decidable and can be checked efficiently.

In this paper, we provide a summary on what is decidable and what is undecidable in schedulability analysis using timed automata. Our main technical contribution is that the schedulability problem will be undecidable if these two conditions hold: (1) the execution times of tasks are intervals and (2) a task is allowed to reset clocks. We show that if one of the above two conditions is dropped, the problem will be decidable again. Thus our result can be used as an indication in identifying classes of timed systems that can be analysed efficiently.

1 Introduction

Timed automata has been developed as a basic semantic model for real time systems. Recently it has been applied to solve scheduling problems, such as job-shop scheduling [AM01,AM02,Abd02,Feh02,HLP01] and real time scheduling [MV94,FPY02,AFM+02,FMPY03,WH03]. The basic idea behind these works is to model real time tasks (or jobs) and scheduling strategies of a system as variants of timed automata and then check the reachability of pre-specified states. As the reachability problem of such automata is decidable, the scheduling problems can be solved automatically and in many cases efficiently (e.g. for fixed priority scheduling [FMPY03]) using a model-checker such as Kronos [Yov97], Uppaal [LPY97] or HyTech[HHWT97]. For preemptive scheduling, stop-watch automata have been used to model preemption [MV94,CL00,Cor94,AM02]. But since the reachability problem for this class of automata is undecidable [ACH+95] there is no guarantee for termination in the general case.

Recent work [FPY02,FMPY03] show that some of the schedulability problems related to preemptive scheduling, which were conjectured to be undecidable due to the nature of preemption that may need the power of stop-watch to model, are shown to be decidable. The decidability results are based on the following assumptions. In [FPY02] the completion time of tasks has no influence on the future behaviour of a system. However, the execution times of tasks can be intervals though it is not stated clearly in this work. In [FMPY03] tasks can communicate with a system (data-dependent control) and announce thus their completion, but the computation time should be a known constant for each task. These are severe restrictions for practical applications where it is often the case that the computation time of a task can vary within an interval representing the best and the worst case execution times of a task, and the future behaviour of a system may depend on the completion time of a task (e.g. exception handling).

In this paper, we study systems with data-dependent control and with tasks whose computation times may be intervals. We show that the schedulability problem is undecidable even for a very restricted form of communication between the control unit of a system and the tasks. Our goal is to identify as closely as possible the borderline between decidable and undecidable problems in schedulability analysis using timed automata. Hopefully, our result can be used as an indication in determining which classes of real-time models can be analysed efficiently.

We adopt the model presented in [EWY98]. Assume a set of tasks whose best and worst execution times, and deadlines are known. The essential idea behind the model is to use timed automata as the arrival (or release) patterns of the tasks. Released tasks are stored in a task queue and executed according to the scheduling strategy. Moreover, tasks reset one distinguished clock upon their completion. Even this restricted type of communication allows the exact completion time of a task to be tested by an automaton.

As the main technical contribution of this paper, we show that (1) the interval execution time of tasks and (2) the ability of the automaton to test the exact completion time of tasks are sufficient and necessary to code the halting problem for two-counter machines. If one of the two conditions is dropped, the problem will be decidable again. We shall also summarise previous decidability results and discuss other variants of the problem.

The rest of the paper is organised as follows. In Section 2 we formally introduce our model, define the schedulability problem, and summarise previous decidability results. Section 3 contains the undecidability proof of the schedulability problem for our model. We discuss several variants of this model in Section 4. Section 5 concludes the paper with a summary and future work.

2 Preliminaries

2.1 Timed Automata with Tasks

A timed automaton [AD94] is a standard finite-state automaton extended with a finite collection of real-valued clocks. One can interpret timed automata as an abstract model of a running real-time system that describes the possible events occurring during its execution. The arrival times of the events must satisfy given timing constraints. To model how events accepted by a timed automaton should be handled or computed, we extend timed automata with asynchronous processes [EWY98,FPY02], i.e. tasks triggered (released) by the events. The idea is to associate each location of a timed automaton with an executable program called a task.

In this paper, we study the model of timed automata extended with tasks whose computation may vary within a closed interval bounded by non-negative integers where the lower bound is the best case execution time and the upper bound is the worst case execution time. Moreover, hard deadline of each task is known. To describe dependence between task completion and release times, we assume that each automaton has a distinguished clock, that is reset whenever a task finishes. This allows each task to announce its completion to the automaton.

Syntax. Let $\mathcal{P}$ ranged over by P, Q denote a finite set of task types. A task type may have different instances that are copies of the same program with different inputs. Each task P is characterised as a pair $([B, W], D)$, where $[B, W]$ is the execution time interval and D is the deadline for P, $B \leq W \leq D \in \mathcal{N}_0$ and $W \neq 0$. The deadline D is relative, meaning that when task P is released, it should finish within D time units. We use $B(P)$, $W(P)$, and $D(P)$ to denote the best execution time, the worst execution time, and the relative deadline of the task P.

As in timed automata, assume a finite set of real-valued variables $\mathcal{C}$ for clocks. 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. We use $\mathcal{B}_I(\mathcal{C})$ for the subset of $\mathcal{B}(\mathcal{C})$ where all atomic constraints are of the form $x {\sim} C$ and $\sim \in \{<, \leq\}$. The elements of $\mathcal{B}(\mathcal{C})$ are called *clock constraints* or *guards*.

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

- *$\langle N, l_0, E, I\rangle$ is a timed automaton where*
  - *N is a finite set of locations,*
  - *$l_0 \in N$ is the initial location,*
  - *$E \subseteq N \times \mathcal{B}(\mathcal{C}) \times 2^{\mathcal{C}} \times N$ is the set of edges, and*
  - *$I : N \mapsto \mathcal{B}_I(\mathcal{C})$ 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 a task type,[1] and*
- *$check \in \mathcal{C}$ is the clock which is reset whenever a task finishes.*

As a simplification we will use $l \xrightarrow{g,r} m$ to denote $(l, g, r, m) \in E$.

For convenience, in the rest of the paper we use extended timed automata (ETA) or simply automata when it is understood from the context instead of timed automata extended with tasks.

[1] Note that M is a partial function meaning that some of the locations may have no task.

Operational Semantics. Extended timed automata may perform two types of transitions just as standard timed automata. Intuitively, a discrete transition in an automaton denotes an event triggering a task. Whenever a task is released, it will be put in a scheduling (or task) queue for execution. A delay transition corresponds to the execution of the running task with the highest priority and idling for the other tasks waiting to run.

We represent the values of clocks as functions (called clock assignments) from $\mathcal{C}$ to the non-negative reals. A state of an automaton is a triple (l, σ, q) where l is the current control location, σ the clock assignment, and q is the current task queue. We assume that the task queue takes the form: $[P_1(b_1, w_1, d_1), \ldots, P_n(b_n, w_n, d_n)]$ where $P_i(b_i, w_i, d_i)$ denotes a released instance of task type P_i with remaining best computing time b_i, remaining worst computing time w_i, and relative deadline d_i denoted by $b(P_i), w(P_i)$ and $d(P_i)$ respectively.

A scheduling strategy Sch, e.g. FPS (fixed priority scheduling) or EDF (earliest deadline first), is a sorting function which changes the ordering of the task queue elements according to the task parameters. For example, $\mathsf{EDF}([P(3.1, 4.9, 10), Q(4, 4.5, 5.3))] = [Q(4, 4.5, 5.3), P(3.1, 4.9, 10))]$. We call such sorting functions (preemptive or non-preemptive) scheduling strategies. A non-preemptive strategy will never change the position of the first element in a queue.

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 on a processor. The result of $\mathsf{Run}(q, t)$ for $q = [P_1(b_1, w_1, d_1), P_2(b_2, w_2, d_2), \ldots, P_n(b_n, w_n, d_n)]$ is defined as $q' = [P_1(b_1 - t, w_1 - t, d_1 - t), P_2(b_2, w_2, d_2 - t), \ldots, P_n(b_n, w_n, d_n - t)]$. For example, let $q = [Q(2, 3, 5), P(4, 7, 10)]$. Then $\mathsf{Run}(q, 3) = [Q(-1, 0, 2), P(4, 7, 7)]$ in which the first task has been executed for 3 time units (and it will be removed from the queue).

A task P in the queue may finish when $b(P) = 0$ and $w(P) \geq 0$, and it must finish when $w(P) = 0$. Finished tasks are removed from the queue.

Further, for a non-negative real number t, we use $\sigma + t$ to denote the clock assignment which maps each clock x to the value $\sigma(x) + t$, $\sigma \models g$ to denote that the clock assignment σ satisfies the constraint g and $\sigma[r]$ for $r \subseteq \mathcal{C}$, to denote the clock assignment which maps each clock in r to 0 and agrees with σ for the other clocks (i.e. $\mathcal{C} \backslash r$). We omit braces when r is a singleton.

Definition 2. *Given a scheduling strategy* Sch*, the semantics of an extended timed automaton* $\langle N, l_0, E, I, M, \mathit{check} \rangle$ *with initial state* (l_0, σ_0, q_0) *is a transition system defined by the following rules:*

- $(l, \sigma, q) \longmapsto_{\mathsf{Sch}} (m, \sigma[r], \mathsf{Sch}(M(m) :: q))$ *if* $l \xrightarrow{g,a,r} m$, $\sigma \models g$, *and* $\sigma[r] \models I(m)$
- $(l, \sigma, []) \xrightarrow{t}_{\mathsf{Sch}} (l, \sigma + t, [])$ *if* $(\sigma + t) \models I(l)$
- $(l, \sigma, P :: q) \xrightarrow{t}_{\mathsf{Sch}} (l, \sigma + t, \mathsf{Run}(P :: q, t))$ *if* $t \leq w(P)$ *and* $(\sigma + t) \models I(l)$
- $(l, \sigma, P :: q) \xrightarrow{0}_{\mathsf{Sch}} (l, \sigma[\mathit{check}], q)$ *if* $b(P) \leq 0 \leq w(P)$ *and* $\sigma[\mathit{check}] \models I(l)$

where $P :: q$ *denotes the queue with the process* P *inserted in* q *(at the first position), and* $[]$ *denotes the empty queue.*

2.2 Schedulability and Decidability

In this subsection we define the schedulability problem for ETA and give a summary of the previous decidability results. Undecidability is discussed in the following section. We first mention that we have the same notion of reachability as for ordinary timed automata.

Definition 3. *We shall write* $(l, \sigma, q) \longrightarrow_{\mathsf{Sch}} (l', \sigma', q')$ *if* $(l, \sigma, q) \longmapsto_{\mathsf{Sch}} (l', \sigma', q')$ *or* $(l, \sigma, q) \xrightarrow{t}_{\mathsf{Sch}} (l', \sigma', q')$ *for a delay* t*. For an automaton with initial state* (l_0, σ_0, q_0) *and for a scheduling strategy* Sch*, we say that* (l, σ, q) *is reachable iff* $(l_0, \sigma_0, q_0)(\longrightarrow_{\mathsf{Sch}})^*(l, \sigma, q)$*.*

Now we can formalise the notion of schedulability.

Definition 4. *(Schedulability) A state* (l, σ, q) *where* $q = [P_1(b_1, w_1, d_1), \ldots, P_n(b_n, w_n, d_n)]$ *is a failure denoted* $(l, \sigma, \mathsf{Error})$ *if there exists* i *such that* $w_i \geq 0$ *and* $d_i < 0$*, that is, a task failed in meeting its deadline. Naturally an automaton* A *with initial state* (l_0, σ_0, q_0) *is non-schedulable with* Sch *iff* $(l, \sigma, \mathsf{Error})$ *is reachable for some* l *and* σ*. 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 following decidability results apply to some simpler variants of the schedulability problem for extended timed automata. By this we show, that whenever preemption, clock resets, or interval execution times are not allowed, the problem becomes decidable.

Theorem 1 ([EWY98]). *The problem of checking schedulability for extended timed automata with non-preemptive scheduling strategy is decidable.*

Proof. The proof in [EWY98] handles only tasks with worst case execution time and tasks without clock resets. But it can be easily modified for our model. □

The schedulability problem is decidable even for preemptive scheduling strategies when tasks are not allowed to communicate with an automaton (to reset clocks). The crucial point of allowing tasks to reset clocks and the reset clocks appearing in guards on an automaton is that the exact finishing time of a task will influence the release time of new tasks.

Theorem 2 ([FPY02]). *The problem of checking schedulability for extended timed automata where tasks do not reset clocks is decidable.*

Proof. It is easy to observe that we can consider only the worst case execution time of each task and the proof is given in [FPY02]. □

The schedulability is decidable even for extended timed automata with data-dependent control when the computation time is a known constant for each task and a task may to reset clocks by the end of their execution.

Theorem 3 ([FMPY03]). *The problem of checking schedulability for extended timed automata is decidable if $B(P) = W(P)$ for all tasks P.*

Proof. The proof is given in [FMPY03]. □

3 Undecidability

Our main result in this paper is that the schedulability problem with fixed priority scheduling strategy for the automata defined in the previous section is undecidable. However, the proof does not depend on fixed priority scheduling strategy and it can be easily modified for almost all preemptive scheduling strategies (e.g. the proof holds for EDF without any modification).

Theorem 4. *The problem of checking whether extended timed automaton (defined in Definition 1) is schedulable with fixed priority scheduling strategy is undecidable.*

The proof is done by reduction of the halting problem for two-counter machine to the schedulability problem for ETA. A *two-counter machine* consists of a finite state control unit and two unbounded non-negative integer counters. Initially, both counters contain the value 0. Such a machine can execute three types of instructions: incrementation of a counter, decrementation of a counter, and branching based upon whether a specific counter contains the value 0. Note that decrementation of a counter with the value 0 leaves this counter unchanged. After execution of an instruction, a machine changes deterministically its state. One state of a two-counter machine is distinguished as *halt state*. A machine halts if and only if it reaches this state.

We present an encoding of a two-counter machine M using extended timed automaton $\mathcal{A}_M$ such that M halts if and only if $\mathcal{A}_M$ is non-schedulable, based on the undecidability proofs of [HKPV98]. In the construction, the states of M correspond to specific locations of $\mathcal{A}_M$ and each counter is encoded by a clock. We shall show how to simulate the two-counter machine operations. First, we adopt the notion of W-wrapping of [HKPV98].

Definition 5. *An extended timed automaton over set of clocks $\mathcal{C}$ is* W-wrapping *if for all states (l, σ, q) reachable from its initial state and for all clocks $c \in \mathcal{C}$: $\sigma \models c \leq W$. A* W-wrapping edge *for a clock c and a location l is an edge from l to itself that is labelled with the guard $c = W$ and which resets the clock c. A clock that is reset only by wrapping edges is called* system clock.[2] *Each time period between two consecutive time points at which any system clock contains value* 0 *is called* W-wrapping period.

[2] Note that all system clocks contain the same value.

We use wrapping to simulate discrete steps of two-counter machine. Each step is modelled by several W-wrapping periods. We define the wrapping-value of a clock to be the value of the clock when the system clock is 0. Note that a clock is carrying the same wrapping value if it is not reset by another edge than the wrapping edges. This principle is shown in Figure 1, where a is a system clock and clock t contains the same wrapping-value when the automaton takes transitions e_1 and e_3.

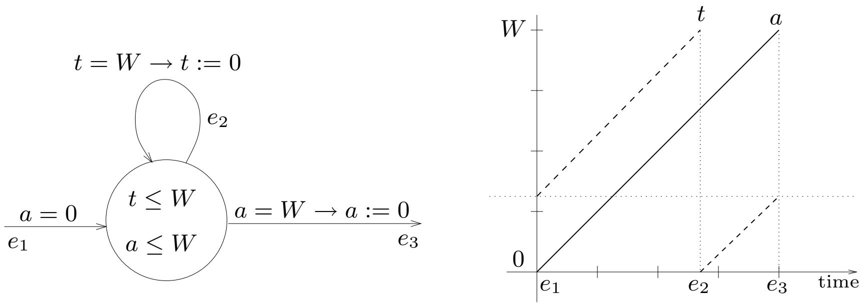

Fig. 1. The wrapping edge e_2 makes clock t carry the same wrapping-value when the transitions e_1 and e_3 are taken.

We shall encode a two-counter machine M with counters C and D using a 4-wrapping automaton $\mathcal{A}_M$ with one system clock denoted a and five other clocks c, d, h, t and *check*. In particular, we encode counters C and D of M by clocks c and d like this: counter value v corresponds to the clock wrapping-value 2^{1-v}. We use the density of the continuous domain to encode arbitrarily large values of the counters. Decrementation (incrementation) of a counter corresponds to doubling (halving) the wrapping-value of the corresponding clock. Test for zero corresponds to the check whether the clock wrapping-value equals to 2.

Now we show how to simulate the decrementation operation by doubling the wrapping-value of the clock d. To do this, we use two tasks: *short* and *long*. The task *short* has execution time within interval $[0, 1]$ and deadline 50; the task *long* has execution time within interval $[8, 8]$ and deadline 100. The tasks reset clock *check* by the end of their execution. Moreover, the priority of *short* is higher than the priority of *long*, i.e. *short* always preempts *long*. Notice that the execution time of task *short* can vary and the execution time of the task *long* is fixed.

The basic idea of doubling a wrapping-value $v \in (0, 1]$ of clock d is as follows: we assume that the current wrapping-value of d is v. We copy it to clock t (that is, to make the wrapping-value of clock t to be v). We release the task *long* non-deterministically and reset d. The idea is to use d to record the response time for *long*. We release two instances of *short* before *long* finishes, that is preempt *long* twice by *short*. We assume that the execution time of each of these two

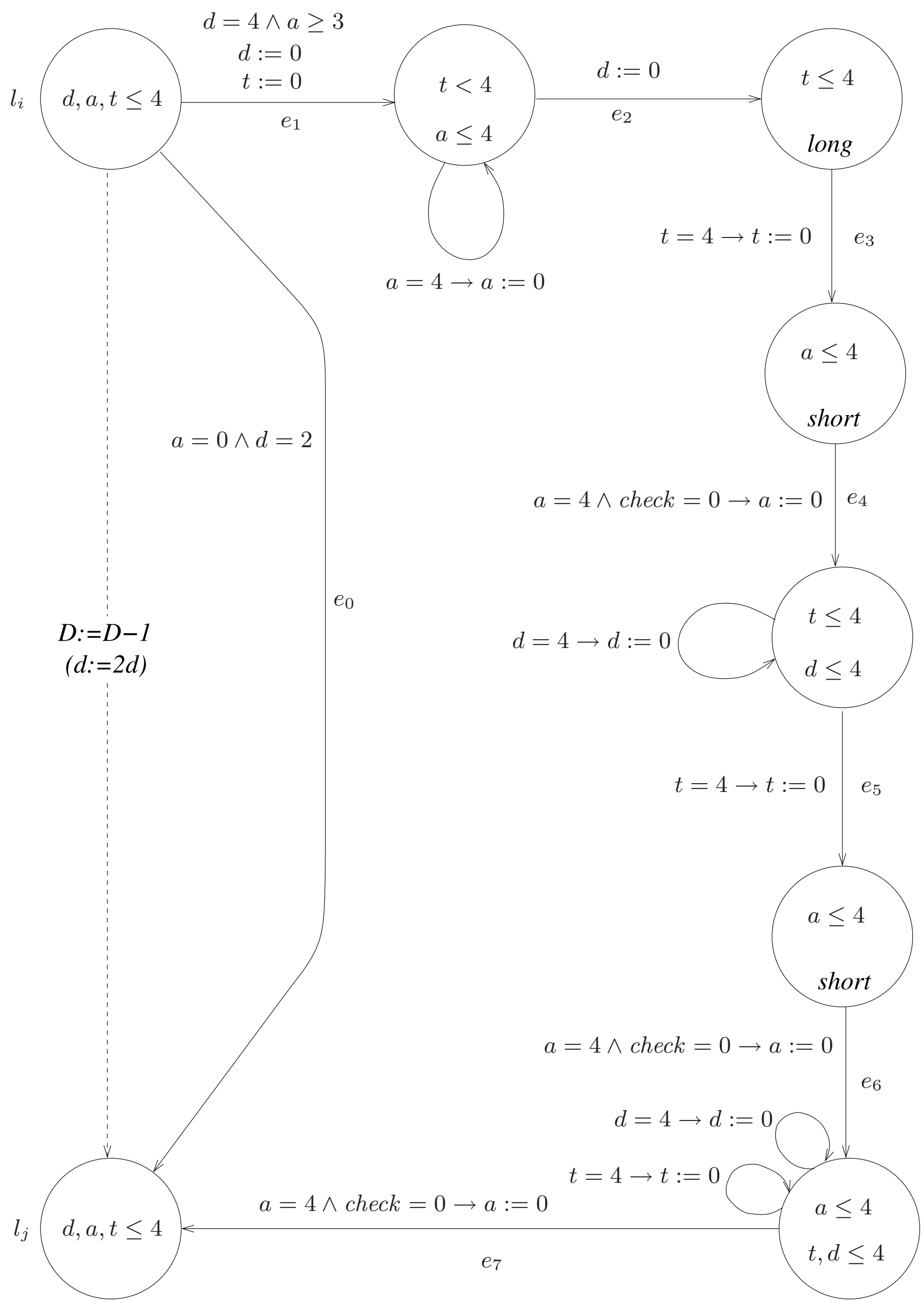

Fig. 2. A part of reduction automaton corresponding to a decrementation of D. The wrapping edges for clocks c, h, *check*, and for all clocks in locations l_i, l_j are omitted. The location invariants $c \leq 4, h \leq 4$, and $check \leq 4$ are also omitted.

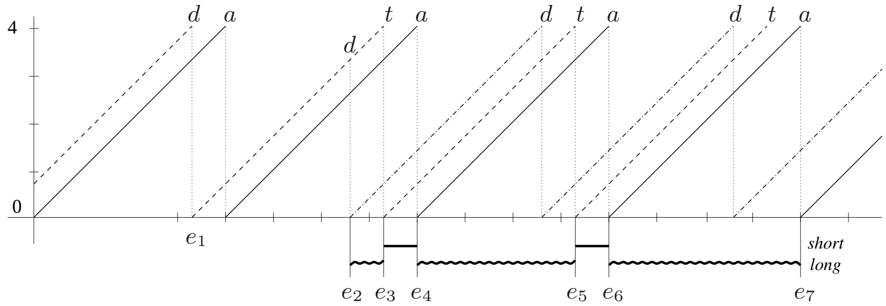

Fig. 3. Time chart of the doubling procedure

instances of *short* is exactly v time units. Note that v can be any real number within the interval $(0, 1]$. Then the response time for *long* is exactly $8+2v$. Note that if *long* finishes at a time point when the system clock a is reset to 0, the wrapping-value of d is $2v$. As *long* is released non-deterministically, there will be surely one such computation.

In Figure 2, we show the part of $\mathcal{A}_M$ that doubles the wrapping-value of clock d. Figure 3 illustrates the time chart of the doubling process. Assume that a two-counter machine M is currently in state s_i and that it wants to decrease the counter D and then move to state s_j. The locations l_i and l_j of $\mathcal{A}_M$ correspond to the states s_i and s_j respectively. Note that the dashed edge shows the transition of the two-counter machine (it is not a transition of $\mathcal{A}_M$). Note also that the decrementation operation leaves a counter with value 0 unchanged; the automaton can move from l_i directly to l_j through the transition e_0 when d contains the wrapping-value 2 (which corresponds to the counter value 0). Otherwise, the following steps are taken to double the wrapping-value of d.

Firstly, the wrapping-value of d is copied to clock t (by transition e_1), that is, t carries the same wrapping-value as d. Then the automaton non-deterministically guesses the doubled wrapping-value of d (note that when d is reset, it will carry a new wrapping-value). It resets d at nondeterministically chosen time instant and at the same time it releases the task *long* (transition e_2).

The automaton waits until clock t reaches time 4, then resets t and releases *short* (transition e_3), which preempts *long*. Note that the wrapping-value of t will remain to be v and at this time point the value of the system clock a is $4 - v$. Therefore a will reach 4 in v time units.

The next transition e_4 is guarded by two constraints: $a = 4$, $check = 0$. To satisfy these constraints, the automaton has to wait in this location for v time units, and task *short* must finish at this time point, which resets the clock *check*.[3]

[3] We have to make sure that *check* is not reset by a wrapping edge when it is tested by a guard of the automaton. This causes no technical difficulties and it is omitted from Figure 2.

By this we make *short* run (and prevent *long* from running) exactly for v time units. Now we repeat this procedure again. That is, the automaton waits until $t = 4$. Then it releases the task *short* and forces it to run exactly for v time units (transitions e_5 and e_6).

Now, if the non-deterministic guess of the doubled wrapping-value of d was correct, task *long* must finish when $a = 4$, which makes the guard on e_7 become true and the automaton moves to location l_j.

So if the location l_j is reachable, the wrapping-value of d is $2v$. This is stated in the following lemma.

Lemma 1. *Let (l_i, σ, q) be an arbitrary state of the automaton shown in Figure 2 where $\sigma(d) = v$ and $v \in (0, 1]$, and q is empty. Then (l_j, σ', q') is reachable for some σ' and q' and if (l_j, σ', q') is reachable, it must be the case that $q' = []$, and $\sigma'(d) = 2v$.*

Proof. The proof is obvious from the construction in Figure 2.

To increment a counter we need to halve a wrapping-value of a clock, say c. For this, we use the clock h to copy the wrapping-value of c. The new wrapping-value v of c is nondeterministically guessed and it is checked by the above doubling procedure. If the wrapping-value of h (the original wrapping-value of c) is $2v$, then the automaton can proceed to the location corresponding to the destination state in an increment instruction.

To simulate branching, we construct two transitions outgoing from a location with guards $a = 0 \wedge c = 2$ and $a = 0 \wedge c \neq 2$. The initial state of M corresponds to a location where both c and d contain the wrapping-value 2. This can be achieved by integer guards and resets.

The halt state corresponds to the location *halt* with unguarded self-loop releasing the task *long* whenever it is visited. It follows that the automaton $\mathcal{A}_M$ is schedulable if and only if the location *halt* is unreachable, i.e. the two-counter machine M does not halt.

4 Variants of the Problem

The proof can be easily modified even for some variants of the original setting. By this we want to show that the only sufficient conditions for undecidability are the following: the execution times of tasks are within intervals and an automaton can test the exact completion time of tasks.

The schedulability problem is undecidable if we use data variables as described in [FMPY03] for task completion announcement instead of a distinguished clock. An automaton $\mathcal{A}_M$ uses interleaving at the time instant when a task finishes to enforce and to measure correct execution time. Tasks assign the value 1 to the data variable A upon completion. Moreover, the automaton can use this variable in guards. The edges e_4 and e_6 in the original construction are substituted by the edges e', e'' and by the location l_{int} from Figure 4. The location where the task *short* is released is left exactly after t time units, but *short* is not finished yet ($A = 0$). However, the time does not flow in the location l_{int}. Just the task *short* can finish here. When this happens the edge e'' becomes enabled and the automaton can proceed.

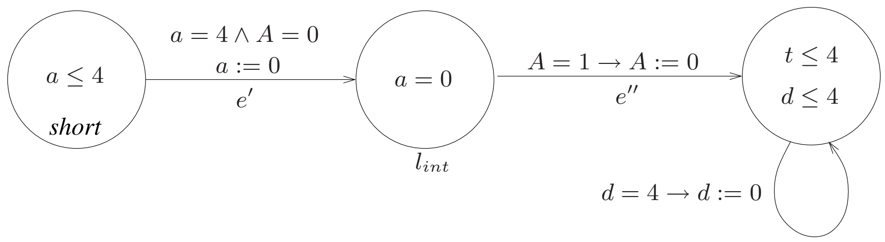

Fig. 4. Substitution of the clock resets by interleaving and data updates.

The schedulability problem is also undecidable if the tasks can only be released at integer time points. In this case, we encode a counter value v as a clock wrapping-value $4 - 2^{1-v}$. Decrementing (incrementing) of a counter does not correspond to doubling (halving) a clock wrapping-value anymore. Both instructions correspond to more complicated operations. Otherwise, the construction becomes even simpler. We do not need auxiliary clock t. Both *long* and *short* are released when $a = 0$. The task *short* should finish when $d = 0$ and we reset the clock d to obtain the doubled wrapping-value when the task *long* finishes. Synchronisation can be forced either by clock resets or by data variable updates. Figure 5 shows the time chart of the doubling procedure.

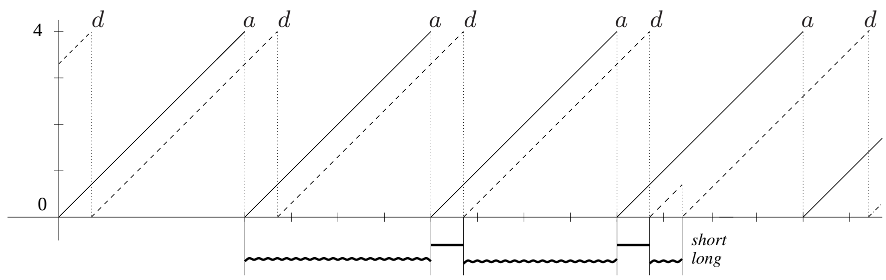

Fig. 5. The time chart of the doubling procedure for integer release points

Schedulability will also remain undecidable if we prohibit $B(P) = W(P)$, i.e. no task is allowed to have a constant computation time. Then we use the task $long_1$

with the execution interval $[7,8]$ instead of $long$. The guessed wrapping-value of d can be less or equal to the correctly doubled value, because the task $long$ can finish sooner. However, we repeat the whole doubling procedure with the task $long_2$ which has the execution interval $[8,9]$. Now the automaton does not guess new wrapping-value of d, but uses the wrapping-value from the previous step. Therefore, this verifying procedure can succeed only if the wrapping-value of d was guessed correctly in the first doubling procedure.

Moreover, we present yet another variant of the extended timed automata for which we suspect the schedulability problem to be decidable.

Consider the extended timed automata with the following modification. The clock used for the task completion announcement (*check*) can appear only in the guards of the form $C{\sim}check{\sim}D$ or $C{\sim}check$ where $\sim\,\in\{\leq,<\}$, and $C\neq D$. This means that we prohibit equality checking for this clock. The automaton can only decide upon whether the value of *check* lies in a non-singular interval. For this setting, it is an open problem whether the schedulability checking is decidable or not.

5 Conclusions and Future Work

We have studied timed systems where preemption can occur at any real time point. For these systems, the schedulability checking problem is decidable if either the computation time of each task is a known constant or the control unit of a system can not test the exact completion time of the tasks [FMPY03]. We have showed that the scheduling problem becomes undecidable if both of these restrictions are dropped. By comparing this result with known decidability results, we try to identify the borderline between decidable and undecidable problems in schedulability analysis for these systems. Hopefully, these results can be used as an indication in determining which classes of real-time systems can be analysed efficiently.

As future work, we will try to identify a class of systems where only partial information about completion time of a task can be obtained by the control unit such that the schedulability problem will become decidable. We have presented a model of such class of systems in Section 4 as a candidate.

References

[Abd02] Y. Abdeddaïm. *Scheduling with Timed Automata*. PhD thesis, Verimag, 2002.

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

[AFM+02] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times - a tool for modelling and implementation of embedded systems. In *Proc. TACAS'02*, volume 2280 of *LNCS*, pages 460–464. Springer, 2002.

[AM01] Y. Abdeddaïm and O. Maler. Job-shop scheduling using timed automata. In *Proc. CAV01*, volume 2102 of *LNCS*, pages 478–492. Springer, 2001.

[AM02] Y. Abdeddaïm and O. Maler. Preemptive job-shop scheduling using stopwatch automata. In *Proc. TACAS02*, volume 2280 of *LNCS*, pages 113–126. Springer, 2002.

[CL00] F. Cassez and F. Laroussinie. Model-checking for hybrid systems by quotienting and constraints solving. In *Proc. CAV'00*, volume 1855 of *LNCS*, pages 373–388. Springer, 2000.

[Cor94] J. Corbett. Modeling and analysis of real-time ada tasking programs. In *Proc. IEEE RTSS'94*, pages 132–141, 1994.

[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.

[Feh02] A. Fehnker. *Citius, Vilius, Melius - Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems.* PhD thesis, KU Nijmegen, 2002.

[FMPY03] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Schedulability analysis using two clocks. In *Proc. TACAS 2003*, volume LNCS 2619, pages 224–239. Springer–Verlag, 2003.

[FPY02] E. Fersman, P. Pettersson, and W. Yi. Timed automata with asynchronous processes: Schedulability and decidability. In *Proc. TACAS'02*, volume 2280 of *LNCS*, pages 67–82. Springer, 2002.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2):123–133, 1997.

[HKPV98] T.A. Henzinger, P.W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57:94–124, 1998.

[HLP01] Thomas Hune, Kim G. Larsen, and Paul Pettersson. Guided Synthesis of Control Programs using Uppaal. *Nordic Journal of Computing*, 8(1):43–64, 2001.

[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.

[MV94] J. McManis and P. Varaiya. Suspension automata: A decidable class of hybrid automata. In *Proc. CAV'94*, volume 818, pages 105–117. Springer, 1994.

[WH03] L. Waszniowski and Z. Hanzálek. Analysis of real time operating system based applications. In *Proc. FORMATS'03*, 2003.

[Yov97] S. Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2):110–122, 1997.