Schedulability Analysis of Fixed-Priority Systems Using Timed Automata

Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi *

Division of Computer Systems

Department of Information Technology

Uppsala University

Abstract

In classic scheduling theory, real-time tasks are usually assumed to be periodic, i.e. tasks are released and computed with fixed rates periodically. To relax the stringent constraints on task arrival times, we propose to use timed automata to describe task arrival patterns. In a previous work, it is shown that the general schedulability checking problem for such models is a reachability problem for a decidable class of timed automata extended with subtraction. Unfortunately, the number of clocks needed in the analysis is proportional to the maximal number of schedulable task instances associated with a model, which is in many cases huge. In this paper, we show that for fixed priority scheduling strategy, the schedulability checking problem can be solved using standard timed automata with *two* extra clocks in addition to the clocks used in the original model to describe task arrival times. The analysis can be done in a similar manner to response time analysis in classic Rate-Monotonic Analysis (RMA). The result is further extended to systems with data-dependent control, in which the release time of a task may depend on the time-point at which other tasks finish their execution. For the case when the execution times of tasks are constants, we show that the schedulability problem can be solved using $n + 1$ extra clocks, where n is the number of tasks. The presented analysis techniques have been implemented in the Times tool. For systems with only periodic tasks, the performance of the tool is comparable with tools implementing the classic RMA technique based on equation-solving, without suffering from the exponential explosion in the number of tasks.

Key words: Real Time Systems, Schedulability Analysis, Timed Automata, Modeling and Verification, Tool

* Corresponding author: Wang Yi, P.O. Box 337, S-751 05 Uppsala, Sweden.
Email address: {elenaf,leom,paupet,yi}@it.uu.se (Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi).

1 Introduction

In the area of real time scheduling methods such as rate monotonic scheduling are widely applied for the analysis of periodic tasks with deterministic behaviours. For systems with non-uniformly recurring tasks, it is a known fact that there are no satisfactory techniques. In reality, control tasks are often triggered by sporadic events coming from the environment. The common approach to analyse the schedulability of such systems is to consider the minimal inter-arrival time of a task as its period and then adopt the ordinary technique used for periodic tasks. Obviously such an approximate method will be in many cases pessimistic since the task control structures are not considered. In order to specify more relaxed timing constraints on events and model other behavioural aspects such as concurrency and synchronization, Extended Timed Automata (ETA) have been suggested in [14] as a generic model for timed systems. It unifies timed automata [7] with the classical task models from scheduling theory allowing to execute tasks asynchronously and specify hard time constraints on computations. Furthermore, the problem of schedulability analysis for this model is proven to be decidable for any scheduling policy. An algorithm for schedulability analysis is presented in [14] based on a translation of the schedulability problem into a reachability problem for the decrementation automata [22]. Unfortunately, the number of clocks needed in the analysis is proportional to the maximal number of schedulable task instances associated with a model, which is often huge. A remaining challenge is to make the result applicable to industrial systems.

In this paper we present an efficient algorithm for the case when fixed priority scheduling strategies are adopted. We show that for a fixed priority scheduling strategy, the schedulability checking problem can be solved by reachability analysis on standard timed automata using only *two* extra clocks in addition to the clocks used in the original model to describe task arrival times. The analysis can be done in a similar manner to response time analysis in classical Rate-Monotonic Scheduling, which calculates the worst-case response time for tasks one by one according to the fixed-priority order. We believe that this is the optimal solution to the problem, a problem that was suspected undecidable previously.

We shall extend the result to systems with data-dependent control, in which the timed automata and the tasks may read and update shared data variables i.e. the release time-point of a task may depend on the values of the shared variables, and hence on the time-point at which the released tasks finish their execution. When the execution times of tasks are known constants, we show that the schedulability problem can be encoded as a reachability problem for timed automata using $n + 1$ extra clocks, where n is the number of tasks. Unfortunately when the execution times of tasks are intervals, the problem is

undecidable [19]. For the undecidable case, we shall present a solution using over-approximation. The techniques presented for all the decidable cases, have been implemented in the TIMES tool based on reachability analysis of timed automata. For periodic tasks, the performance of our tool is comparable with tools implementing the classical RMA technique based on equation-solving. For systems with only periodic tasks, the analysis is insensitive to the number of tasks as the worst case scenario is assumed to appear in the first periods of the tasks.

Related Work. For schedulability analysis of systems restricted to periodic tasks, a large number of techniques are available, see e.g. [10, 18, 21]. 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. For fixed priority periodic tasks with offsets and release jitters, techniques for schedulability analysis have been developed in [25, 23, 24]. Our work is more related to work on using automata to model and solve scheduling problems, aiming at systems whose tasks have complex control structures and arrival patterns. In [12, 11], stopwatch automata [6] are applied to model scheduling algorithms with sporadic tasks and semi-decision algorithms are presented. Timed automata [7] have been used to solve non-preemptive scheduling problems mainly for job-shop scheduling [1, 13, 16]. Similarly, stopwatch automata have been used to solve preemptive job-shop scheduling problems e.g. [2]. These techniques specify pre-defined locations of an automaton as goals to achieve by scheduling and use reachability analysis to construct traces leading to the goal locations. The traces are used as schedules. A work on relating classical scheduling theory to timed systems is the controller synthesis approach [3, 4, 5]. The idea is to achieve schedulability by construction. The authors present a controller synthesis technique that can be used to construct a scheduler to control the system so that all given scheduling constraints in the model are satisfied. An alternative approach is presented in [26] in which the schedulability of a system is established by proving that the specification (formalised in the temporal logic TLA) of the system and the scheduler satisfy the given scheduling constraint.

The rest of this paper is organized as follows: Section 2 describes the syntax and semantics of ETA and defines scheduling problems related to the model. In Section 3, we present the main result of this paper – an algorithm to perform schedulability analysis of systems with relaxed timing constraints. Section 4 is devoted to schedulability analysis of systems with fixed priorities and data-dependent control. In Section 5, we describe implementation issues and how to perform worst-case response time analysis. Section 6 concludes the paper with summary and related work.

2 Preliminaries

In this section, we review the model of extended timed automata with real time tasks and the notion of schedulability as well as the decidability result presented in [14]. A timed automaton [7] 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 system that describes the possible events occurring during its execution. The arrival times of the events must satisfy the given timing constraints. To specify how events, accepted by a timed automaton, should be handled or computed we extend timed automata with asynchronous processes [14], i.e. tasks triggered by events synchronously and computed asynchronously (i.e. buffered). The idea is to associate each location of a timed automaton with an executable program called a task. We assume that the execution times and hard deadlines of the tasks are known [1].

Let $\mathcal{P}$ ranged over by P and P_1, P_2 etc, denote a finite set of task types. A task type may have different instances that are copies of the same program with different inputs. Throughout the paper, we shall not distinguish tasks from task instances when it is understood from the context. Each task P is characterized as a pair of natural numbers denoted $P(C, D)$ with $C \leq D$, where C is the execution time (or computation time) of P and D is the deadline for P. The deadline D is relative, 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. We shall use $C(i)$ and $D(i)$ instead of $C(P_i)$ and $D(P_i)$ when it is understood from the context.

As in timed automata, assume a finite alphabet 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 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.*

[1] Task may have other parameters such as fixed priority for scheduling and other resource requirements, e.g. memory requirement.

- $I : N \mapsto \mathcal{B}(\mathcal{C})$ *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* [2].

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 release times of the associated task). Whenever a task is triggered, it will be put in a scheduling (or task) queue for execution (corresponding to the ready queue in operating systems).

Extended timed automata may perform two types of transitions just as standard timed automata. The difference is that delay transitions correspond to the execution of running tasks with highest priority and idling for the other tasks waiting to run. Discrete transitions correspond 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. A state of an automaton is a triple (l, u, q) where l is the current control location, u the clock assignment, and q is the current task queue. We assume that the task queue takes the form: $[P_1(c_1, d_1), \ldots, P_n(c_n, d_n)]$ 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

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, $\mathsf{EDF}([P(3.1, 10), Q(4, 5.3)]) = [Q(4, 5.3), P(3.1, 10)])$. We call such sorting functions scheduling strategies that may be preemptive or non-preemptive [3]. Thus an action transition will result in a sorted queue including the tasks released by this transition. A delay transition with c time units is to execute the task in the first position of the queue with c time units. Thus the delay transition will decrease the computing time of the first task by c. If its computation time becomes 0, the task should be removed from the queue (shrinking). We adopt the structural equivalence over queues respecting $[P_1(0, d), P_2(c_2, d_2), \ldots, P_n(c_n, d_n)] = [P_2(c_2, d_2), \ldots, P_n(c_n, d_n)]$. Moreover, after a delay transition with c time units, the deadlines of all tasks in the queue will be decreased by c.

Run is a function which given a real number t and a task queue q returns

---

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

[3] 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 preemptive strategy may change the ordering of task types only, but never change the ordering of task instances of the same type.

the resulted task queue after t time units of execution according to available computing resources. For simplicity, we assume that only one processor is available. 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 non-negative real number t, 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. $\mathcal{C} \backslash r$).

Definition 2 *Given a scheduling strategy* Sch[4]*, 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* q *with* $M(m)$ *inserted into it.*

Now, we briefly review the verification problems of ETA. For more details, we refer the reader to [14]. We first mention that we have the same notion of reachability as for ordinary 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 ETA is infinite not only because of the real-valued clocks, but also unbounded size of the task queue.

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 $\mathsf{Sch}(q) = [P_1(c_1, d_1) \ldots P_n(c_n,$

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

$d_n)]$ 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.

Theorem 1 *The problem of checking schedulability for extended timed automata is decidable.*

PROOF. The proof is given in [14]. □

3 Encoding of Fixed-Priority Schedulers

In this section we present the main result of this paper. It shows that for timed automata extended with tasks executed according to fixed priorities, the scheduling problem can be encoded into a reachability problem of ordinary timed automata using only two additional clocks.

Assume an ETA A and a fixed priority scheduling strategy Sch. To solve the scheduling problem, for each $P_i \in \mathcal{P}$ we construct timed automata $E_i(\mathsf{Sch})$ and $E(A)$, and check for reachability of a predefined error state in the product automaton of the two. If the error state is reachable, task P_i of automaton A is not schedulable with Sch, i.e. P_i will eventually miss its deadline. The check is performed following the given fixed priority order for each task in $\mathcal{P}$, starting with the task of highest priority.

Our analysis technique is inspired by Joseph and Pandya's rate-monotonic analysis of periodic tasks [17], where the worst-case response time of each task is calculated as the sum of the task's execution time, and the blockings imposed by other tasks. Following Joseph and Pandya's work, for each task type we check independently that it meets its deadline. However, the model of ETA gives rise to a more general scheduling problem than systems with periodic tasks only. As a result, we can not base our analysis on the existence of an a priori known worst-case scenario for a given task. Instead, it will be part of the analysis to find all situations in which a task may execute.

To construct the $E(A)$, the automaton A is annotated with distinct synchronization actions $\mathsf{release}_i$ on all edges leading to locations labelled with the task name P_i. The actions will allow the scheduler to observe when tasks are released for execution in A. The rest of this section is devoted to show that $E_i(\mathsf{Sch})$ can be constructed as a timed automaton using only two clocks.

Theorem 2 *Given a fixed priority scheduling strategy* Sch, $E_i(\mathsf{Sch})$ *can be encoded as a timed automaton containing two clocks.*

PROOF. Follows from Lemma 1 and 2 shown later in this section.

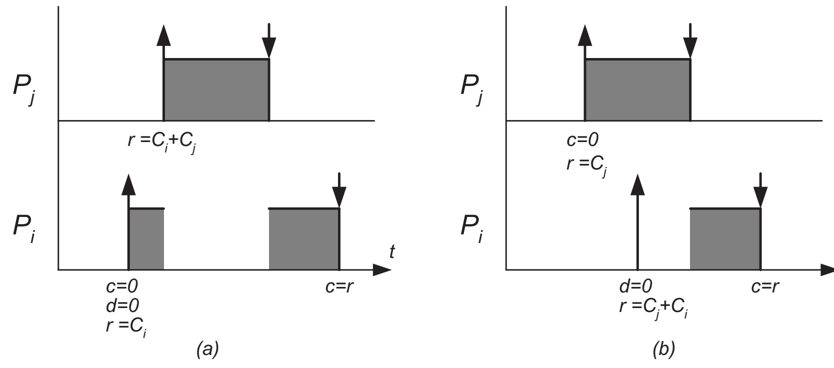

Fig. 1. Task execution schemes for tasks P_i and P_j with $\mathsf{Prio}(j) > \mathsf{Prio}(i)$. The symbols $\uparrow$ and $\downarrow$ indicate release and completion of tasks, respectively.

In the encoding of $E_i(\mathsf{Sch})$, we shall use $C(i)$, $D(i)$ and $\mathsf{Prio}(i)$ to denote the worst-case execution time, the deadline, and the priority of task type P_i, respectively. $E_i(\mathsf{Sch})$ uses the following variables:

- d - a clock measuring the time since the analysed task instance of P_i was released for execution,
- c - a clock accumulating the time since the task queue was last empty (or containing only tasks P_k with $\mathsf{Prio}(k) < \mathsf{Prio}(i)$).
- r - a data variable used to sum up the time needed to complete all tasks released since the processor was last idle (i.e. not executing instances of P_i and all higher priority tasks). The boundedness of r will be stated in Lemma 1.

The clock d is reset when the analysis of a task instance begins, and will be used to check if it completes before its deadline. The clock c is used to compute the time point when the analysed task instance of P_i completes. The variable r will be assigned so that P_i completes when $\mathsf{c} = \mathsf{r}$. Fig.1 shows in two Gantt charts how the variables are used in $E_i(\mathsf{Sch})$. In Fig.1(a) task P_i executes immediately but is preempted by P_j. In Fig.1(b) task P_i is released when task P_j is already executing. Note that the clocks c and d are reset, and the variable r is updated in the two scenarios so that task P_i is completed when the condition $\mathsf{c} = \mathsf{r}$ is satisfied. Note also that the deadline of P_i is reached when $\mathsf{d} = D(i)$ (as d is reset when P_i is released for execution).

The encoding of $E_i(\mathsf{Sch})$ is shown in Fig.2. Intuitively, the locations have the following interpretations:

- Idle_i - denotes a situation where no task P_j with $\mathsf{Prio}(j) \geq \mathsf{Prio}(i)$ is being executed (or ready to be executed).
- Check_i - an instance of task type P_i is currently ready for execution (possibly executing) and is being analysed for schedulability.
- Busy_i - a task of type P_j with priority $\mathsf{Prio}(j) \geq \mathsf{Prio}(i)$ is currently executing.

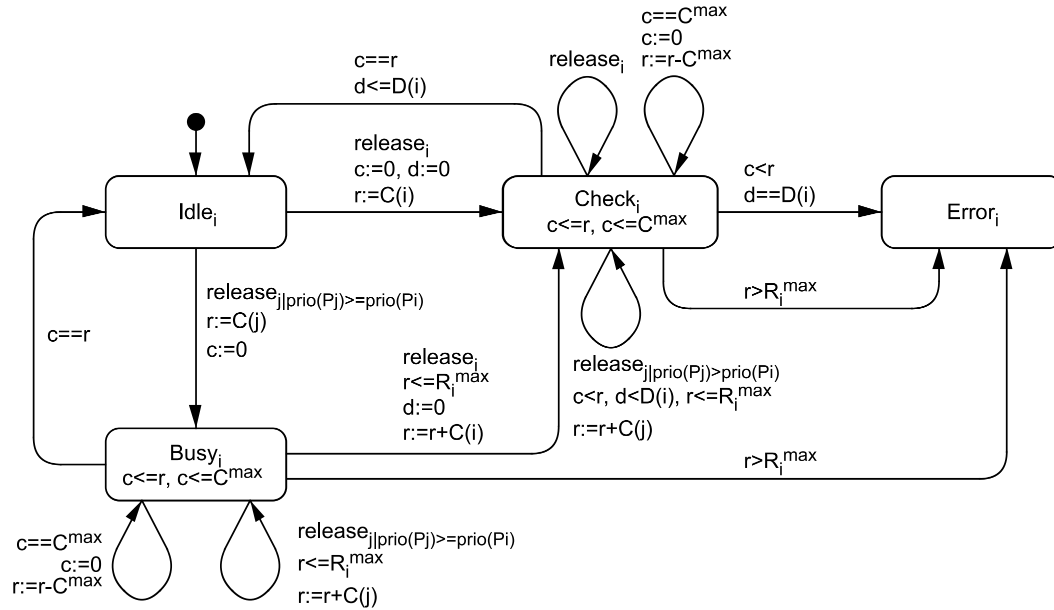

Fig. 2. Encoding of schedulability problem.

- Error_i - the analysed task queue is not schedulable with Sch.

The analysis of an instance of P_i starts when a transition from Idle_i or Busy_i to Check_i is taken. The transitions in $E_i(\mathsf{Sch})$ have the following intuitive interpretations:

- Idle_i - is (re-)entered when the task instance being checked in Check_i, or a sequence of tasks arrived in Busy_i, has finished execution. In both cases the enabling condition c=r ensures that the location is reached when all tasks P_j with $\mathsf{Prio}(j) \geq \mathsf{Prio}(i)$ have finished their executions.
- Busy_i - the ingoing transitions to Busy_i are taken when a task P_j such that $\mathsf{Prio}(j) \geq \mathsf{Prio}(i)$ is released. The additional self-loop, is taken to decrement both c and r with the constant value C^{max}. This does not change the truth-value of any of the guards in which c and r appear, as the values are always compared to each other.
- Check_i - transitions entering Check_i from Idle_i or Busy_i are taken when a task instance of P_i is (non-deterministically) chosen for checking. Self-loops in Check_i are taken to update r at the release of higher-priority tasks. New instances of P_i in Check_i are ignored as they are considered by the non-deterministic choice in location Busy_i.
- Error_i - is reached when the analysed task instance reaches its deadline (encoded $\mathsf{d} = \mathsf{D}(i)$) before completion (encoded $\mathsf{c} < \mathsf{r}$). In addition, Error_i is entered if the set of released tasks is guaranteed to be non-schedulable (encoded $\mathsf{r} > \mathsf{R}_i^{max}$, the value of R_i^{max} is discussed below).

In addition to these transitions, in Fig 2 we have omitted self-loops in all locations, which synchronize with $E(A)$ whenever a task of priority lower

than $\mathsf{Prio}(i)$ is released. They can be ignored as these tasks do not affect the response time of P_i.

The constant C^{max} mentioned above can be any value greater than 0. We use $\mathsf{C}^{max} = \max_i(C(i))$. To find a value for R_i^{max}, we need the result of the previous analysis steps. Recall that the analysis of all $P_i \in \mathcal{P}$ is performed in priority order, starting with the highest priority. Thus, when P_i is analysed we can find the maximum value assigned to r in the previous analysis steps. Let r^{max} denote this value. Recall that $\mathsf{r} - \mathsf{c}$ is always the time remaining until the released tasks complete their executions (except in location Idle_i and Error_i where r is not updated). For the set of released tasks to be schedulable we have that $\mathsf{r} - \mathsf{c} < \mathsf{r}^{max} + D(i)$. It follows that $\mathsf{r} < \mathsf{r}^{max} + D(i) + \mathsf{C}^{max}$ since $\mathsf{c} \leq \mathsf{C}^{max}$. We set the constant $\mathsf{R}_i^{max} = \mathsf{r}^{max} + D(i) + \mathsf{C}^{max}$ and use $\mathsf{r} > \mathsf{R}_i^{max}$ to detect non-schedulable tasks sets in $E(\mathsf{Sch})$.

The last step of the encoding is to construct the product automata $E(A)||E_i(\mathsf{Sch})$ for each $P_i \in \mathcal{P}$, and check by reachability analysis that location Error_i is not reachable in the product automaton. We now show that $E(A)||E_i(\mathsf{Sch})$ is bounded.

Lemma 1 *The clocks* c *and* d *and the data variable* r *of* $E_i(\mathsf{Sch})$ *in* $E(A)|| E_i(\mathsf{Sch})$ *are bounded.*

PROOF. The clocks d and c are bounded by the constants $D(i)$ and C^{max} respectively. The data variable r is bounded by $\mathsf{R}_i^{max} + \max_{\{j\ :\ \mathsf{Prio}(j) > \mathsf{Prio}(i)\}} C(j)$.

Lemma 2 *Let A be an extended timed automaton and* Sch *a fixed-priority scheduling strategy. Assume that (l_0, u_0, q_0) and $(\langle l_0, \mathsf{Idle}_i\rangle, v_0)$ are the initial states of A and the product automaton $E(A)||E_i(\mathsf{Sch})$ respectively where l_0 is the initial location of A, u_0 and v_0 are assigning all clocks with 0 and q_0 is the empty task queue. Then for any l the statement*

$$(l_0, u_0, q_0)(\longrightarrow)^*(l, u, \mathsf{Error})$$

holds for some u if and only if the following holds for some v and i:

$$(\langle l_0, \mathsf{Idle}_i\rangle, v_0)(\longrightarrow)^*(\langle l, \mathsf{Error}_i\rangle, v)$$

PROOF. We assume that the task queue takes the form: $[P_1 ... P_n]$ where P_i denotes a released instance of task type i with remaining computing time $c(P_i)$ and relative deadline $d(P_i)$, and the variable (and clock) assignment v in the product automaton takes the form (u, v), where u is an assignment for A and v is an assignment for $E_i(\mathsf{Sch})$. Whenever it is understood, we shall write c, d, r to denote $v(c), v(d), v(r)$.

We show the existence of simulations between the states of A and $E(A)|| E_i(\mathsf{Sch})$. Let $S_1 = \{(l,u,q), (\langle l, \mathsf{Idle}_k\rangle, (u,v)) \mid \mathit{empty}(q)\}$, $S_2 = \{(l,u,q), (\langle l, \mathsf{Busy}_k\rangle, (u,v)) \mid (\sum_{i=1}^{k} c(P_i)) = r - c\}$, $S_3 = \{(l,u,q), (\langle l, \mathsf{Check}_k\rangle, (u,v)) \mid (\sum_{i=1}^{k} c(P_i)) = r - c,\ d = D(P_k) - d(P_k)\}$, $S_4 = \{(l,u,q), (\langle l, \mathsf{Check}_k\rangle, (u,v)) \mid (\sum_{i=1}^{k} c(P_i)) > d(P_k),\ r - c > D(P_k) - d\}$, $S_5 = \{(l,u,q), (\langle l, \mathsf{Error}_k\rangle, (u,v)) \mid c(P_k) > 0,\ d(P_k) = 0\}$ and $S = S_1 \cup S_2 \cup S_3 \cup S_4 \cup S_5$.

We prove that S and S^{-1} are simulations. First we prove that S is a simulation.

Assume that $((l,u,q), (\langle l, \mathsf{Idle}_k\rangle, (u,v))) \in S_1$. We consider the two types of transitions:

- (Action) Assume $(l,u,q) \xrightarrow{a} (l', u', \mathsf{Sch}(M(l') :: q))$. Further assume that this transition is induced by $l \xrightarrow{g,a,r} l'$ and $u \models g$. Then the product automaton can also make the following a-transitions: $(\langle l, \mathsf{Idle}_i\rangle, (u,v)) \xrightarrow{a} (\langle l', \mathsf{Busy}_i\rangle, (u', v[c := 0, r := C(M(l'))]))$. Here a new task is inserted into the queue, and the variable r is set to the WCET of the new task. Therefore, $(\sum_{i=1}^{k} c(P_i)) = C(M(l')) = r - c$, and then $((l', u', \mathsf{Sch}(M(l') :: q)), (\langle l', \mathsf{Busy}_i\rangle, (u', v[c := 0, r := C(M(l'))]))) \in S_2$. Alternatively, if $M(l') = P_k$, the product automaton can also have the transition: $(\langle l, \mathsf{Idle}_i\rangle, (u,v)) \xrightarrow{a} (\langle l', \mathsf{Check}_i\rangle, (u', v[c := 0, r := C(P_k), d := 0]))$. Here P_k is inserted into the queue, and the variable r is set to the $C(P_k)$. Therefore, $(\sum_{i=1}^{k} c(P_i)) = C(P_k) = r - c$, and then $((l', u', \mathsf{Sch}(M(l') :: q)), (\langle l', \mathsf{Check}_i\rangle, (u', v[c := 0, r := C(P_k), d := 0]))) \in S_3$
- (Delay) Assume $(l,u,q) \xrightarrow{t}_{\mathsf{Sch}} (l, u+t, \mathsf{Run}(q,t))$, where $(u+t) \models I(l)$. Then the product automaton can make the following delay transition: $(\langle l, \mathsf{Idle}_i\rangle, (u, c, r, d)) \xrightarrow{t} (\langle l, \mathsf{Idle}_i\rangle, (u+t, c+t, r, d+t))$. Running the empty queue for t time units results in an empty queue, therefore $((l, u+t, \mathsf{Run}(q,t)), (\langle l, \mathsf{Idle}_i\rangle, (u+t, v+t))) \in S_1$

The rest of the proof that S as well as S^{-1} is a simulation is similar.

Thus, we have shown that the scheduling problem can be solved by a reachability problem for timed automata, and from Lemma 1 we know that the reachability problem is bounded. This completes the proof of Theorem 2.

4 Analysing Data-Dependent Control

In this section we extend the result of the previous section to handle extended time automata in which the tasks may use (read and update) data variables, shared between the tasks and the automata. This results in a model with *data-dependent control* in the sense that the behaviour of the control automaton, and the release time-point of tasks may depend on the values of the shared

variables, and hence on the time-points at which other tasks complete their executions. We first present the model of ETA extended with data variables [9].

4.1 Extended Timed Automata with Data Variables

Syntax. Assume a set of variables $\mathcal{D}$ ranged over by u, which takes their values from finite data domains, and are updated by assignments in the form $u := \mathcal{E}$, where $\mathcal{E}$ is a mathematical expression. We use $\mathcal{R}$ to denote the set of all possible assignments. A task P is now characterized by a triple $P(C, D, R)$, where C and D are the execution time and the deadline as usual, and $R \subseteq \mathcal{R}$ is a set of assignments. We use $R(P)$ to denote the set of assignments of P, and we assume that a task assigns the variables according to $R(P)$ by the end of its execution.

The data variables assigned by tasks may also be updated and tested (or read) by the extended timed automata. Let $\mathcal{A} = \mathcal{R} \cup \{x := 0 \mid x \in \mathcal{C}\}$ be the set of updates. We use r to stand for a subset of $\mathcal{A}$. To read and test the values of the data variables, let $\mathcal{B}(\mathcal{D})$ be a set of predicates over $\mathcal{D}$. Let $\mathcal{B} = \mathcal{B}(\mathcal{D}) \cup \mathcal{B}(\mathcal{C})$ be ranged over by g called guards.

Unfortunately the analysis of data-dependent control structures can not be based on the WCET of tasks only for the obvious reason that if a task updates shared variables by the end of its execution (before WCET) it can trigger releases of the other tasks and lead to a negative schedulability result. This means that a system may be schedulable when all the tasks actually consume the WCET and it may not be schedulable when some of the tasks consume less than the WCET.

In the following, we present a solution for the schedulability analysis problem for the case when the execution times, denoted $C(P)$ for task P, are constants. The case when the execution times are intervals i.e. best and worst execution times is considered later.

Operational Semantics. To define the semantics, we use valuations to denote the values of variables. A valuation is a function mapping clock variables to the non-negative reals, and data variables to the data domain. We denote by $\mathcal{V}$ the set of valuations ranged over by σ. For a non-negative real number t, we use $\sigma + t$ to denote the valuation which updates each clock x with $\sigma(x) + t$, and $\sigma[r]$ to denote the valuation which maps each variable α to the value of $\mathcal{E}$ if $\alpha := \mathcal{E} \in r$ (note that $\mathcal{E}$ is zero if α is a clock) and agrees with σ for the other variables. We are now ready to present the semantics of extended timed automata with data variables by the following rules:

- $(l,\sigma,q)\xrightarrow{a}_{\mathsf{Sch}}(m,\sigma[r],\mathsf{Sch}(M(m)::q))$ if $l\xrightarrow{g,a,r}m$ and $\sigma\models g$
- $(l,\sigma,q)\xrightarrow{t}_{\mathsf{Sch}}(l,\sigma+t,\mathsf{Run}(q,t))$ if $(\sigma+t)\models I(l)$ and $C(\mathsf{Hd}(q,t))>t$
- $(l,\sigma,q)\xrightarrow{t}_{\mathsf{Sch}}(l,(\sigma[R(\mathsf{Hd}(q))])+t,\mathsf{Run}(q,t))$ if $(\sigma+t)\models I(l)$ and $C(\mathsf{Hd}(q))=t$

where $M(m)::q$ denotes the queue q with $M(m)$ inserted into it and $\mathsf{Hd}(q)$ denotes the first element of q.

4.2 Schedulability Analysis

As in the previous section, we shall encode the ETA A and the fixed-priority scheduling strategy Sch into timed automata and check for reachability of predefined error states. The encoding $E(A)$ is the same as in the previous section. However, the encoding of Sch will be different with data-depended control, as the result of the schedulability analysis depends on the data-variables that may be updated whenever a task completes its execution. In the rest of this section we describe how to construct $E(\mathsf{Sch})$:

Theorem 3 *For an extended timed automaton A with data variables, and a fixed priority scheduling strategy Sch, $E(\mathsf{Sch})$ can be constructed as timed automaton containing $n+1$ clocks, where n is a number of task types used in A.*

PROOF. Follows from Lemma 3 and 4 shown later in this section.

The construction of $E(\mathsf{Sch})$ is illustrated in Fig.3. It consists of two parallel automata: $E_{\mathsf{SP}}(\mathsf{Sch})$ - encoding the scheduling policy (containing n clocks), and E_{DC} - encoding a generic deadline checker (containing one clock). As in the previous section, the two scheduling automata (in this case both $E_{\mathsf{SP}}(\mathsf{Sch})$ and E_{DC}) synchronize with $E(A)$ on the action $\mathsf{release}_i$ when an instance of task P_i is released. In addition, $E_{\mathsf{SP}}(\mathsf{Sch})$ and E_{DC} synchronize on $\mathsf{finished}_i$ whenever an instance of P_i finishes its execution.

Encoding of Scheduling Policy $E_{\mathsf{SP}}(\mathsf{Sch})$. Let P_{ij} denote instance j of task P_i. For each P_{ij}, $E_{\mathsf{SP}}(\mathsf{Sch})$ has a state variable $\mathsf{status}(i,j)$ that is initially set to free. Let $\mathsf{status}(i,j)=\mathsf{running}$ denote that P_{ij} is executing on the processor, $\mathsf{preempted}$ that P_{ij} is started but not running, and $\mathsf{released}$ that P_{ij} is released but not yet started. We use $\mathsf{status}(i,j)=\mathsf{free}$ to denote that P_{ij} is not released yet. Note that for all (i,j) there can be only one j such that $\mathsf{status}(i,j)=\mathsf{preempted}$ (i.e. only one instance of the same task type is started), and for all (i,j) there can only be one pair (k,l) such that

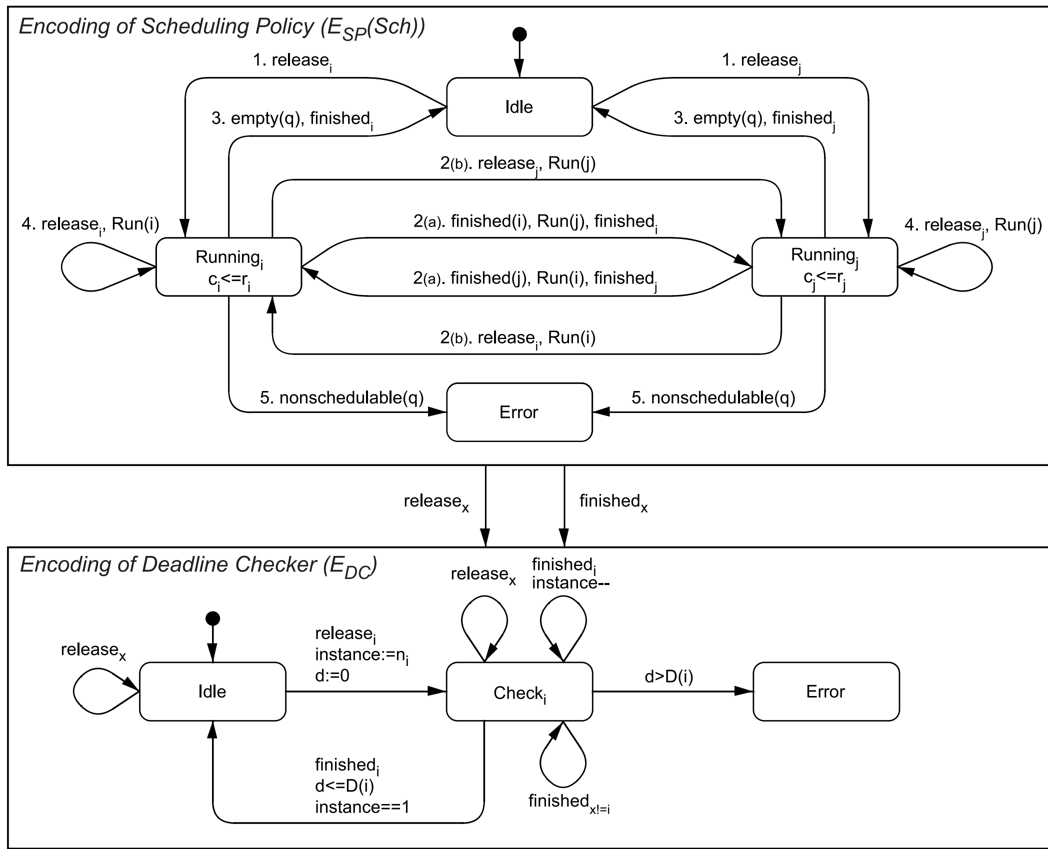

Fig. 3. Encoding of schedulability problem.

$\mathsf{status}(k, l) = \mathsf{running}$ (i.e. only one task is running in a one-processor system). For each task type P_i we use three variables:

- c_i - clock measuring the time passed since P_i started its execution. We reset c_i whenever an instance of P_i is started.
- r_i - data variable accumulating the response time of P_i from the moment it starts to execute. r_i is set to $C(i)$ when an instance of P_i is started, and updated to $\mathsf{r}_i + C(j)$ when a higher-priority task P_j is released.
- n_i - data variable keeping track of the number of P_i currently released.

In Fig. 4, we show how the above variables are used in $E_{\mathsf{SP}}(\mathsf{Sch})$. At time point x state variable status has the values $\mathsf{status}(1, 1) = \mathsf{running}$, $\mathsf{status}(2, 1) = \mathsf{preempted}$, $\mathsf{status}(2, 2) = \mathsf{released}$, and $\mathsf{status}(3, 1) = \mathsf{released}$.

To represent each task instance in $E_{\mathsf{SP}}(\mathsf{Sch})$ we use a triple $\langle c_i, r_i, \mathsf{status}(i, j)\rangle$, and the task queue q will contain such triples. Note that the maximal number of instances of P_i appearing in a schedulable queue is $\lceil D(i)/C(i)\rceil$. Thus, the size of the queue is bounded to $\sum_{P_i \in \mathcal{P}} \lceil D(i)/C(i)\rceil$. We shall say that queue is empty, denoted $\mathsf{empty}(q)$, if $\mathsf{status}(i, j) = \mathsf{free}$ for all (i, j).

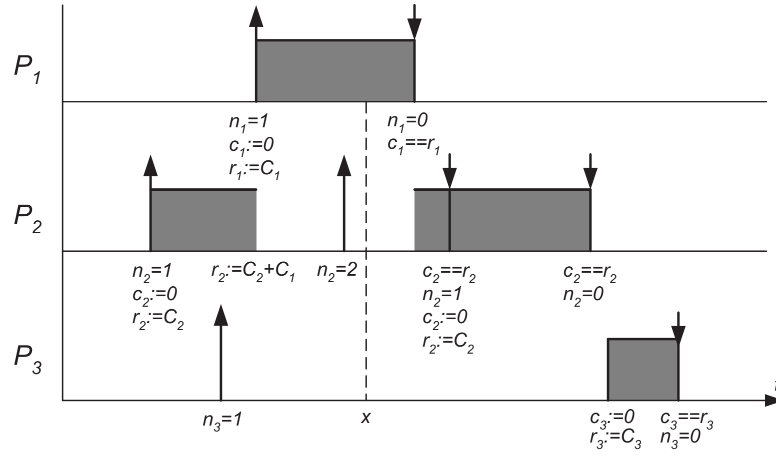

Fig. 4. Task execution scheme where $\mathsf{Prio}(1) > \mathsf{Prio}(2) > \mathsf{Prio}(3)$.

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 fixed priority scheduling policy Sch, it can be coded as a constraint over the state variables. For example, for deadline-monotonic scheduling [5], $\mathsf{Run}(m, n)$ is the conjunction of the following constraints:

- $\mathsf{r}_k \leq D(k)$ for all k, l such that $\mathsf{status}(k, l) \neq \mathsf{free}$: all response time integers are less than deadlines
- $\mathsf{status}(m, n) \neq \mathsf{free}$: P_{mn} is released or preempted
- $D(m) \leq D(i)$ for all i: P_m has the highest priority

We use $\mathsf{Run}(m)$ to denote that a task instance of P_m is scheduled to run according to Sch. The predicate $\mathsf{finished}(m, n)$ denotes that P_{mn} has finished its execution. We define $\mathsf{finished}(m, n)$ to $(c_m = r_m) \wedge (\mathsf{status}(m, n) \neq \mathsf{free})$. Finally, we use $\mathsf{nonschedulable}(q)$ to denote that the queue q is non-schedulable in a sense that there exists a pair (i, j) for which $r_i > D(i)$ and $\mathsf{status}(i, j) \neq \mathsf{free}$.

The automaton $E_{\mathsf{SP}}(\mathsf{Sch})$ contains three type of locations: Idle, $\mathsf{Running}_i$ and Error. Note that $\mathsf{Running}_i$ is parameterized with i representing the running task type. Location Idle denotes that the task queue is empty. $\mathsf{Running}_i$ denotes that task instance of type P_i is running, that is, for some j $\mathsf{status}(i, j) = \mathsf{running}$. For each $\mathsf{Running}_i$ we have the location invariant $c_i \leq r_i$. Error denotes that the task queue is non-schedulable with Sch. There are five types of edges labelled as follows:

(1) Idle to $\mathsf{Running}_i$: edges labelled with action $\mathsf{release}_i$, and reset $\{r_i := C(i), c_i := 0, \mathsf{n}_i := 1, \mathsf{status}(i, j) := \mathsf{running}\}$.
(2) $\mathsf{Running}_i$ to $\mathsf{Running}_m$: two types of edges:

[5] In deadline-monotonic scheduling, task priorities are assigned according to deadlines, such that $\mathsf{Prio}(i) > \mathsf{Prio}(j)$ iff $D(i) < D(j)$.

(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: encoded by guard $\mathsf{finished}(i,j) \wedge \mathsf{status}(m,n) = \mathsf{preempted} \wedge \mathsf{Run}(m,n)$, action $\mathsf{finished}_i$, and reset $\{\mathsf{status}(i,j) := \mathsf{free}, \mathsf{n}_i := \mathsf{n}_i - 1, \mathsf{status}(m,n) := \mathsf{running}, R(P_i)\}$
   (ii) P_{mn} was released, but never preempted (not started yet): encoded by guard $\mathsf{finished}(i,j) \wedge \mathsf{status}(m,n) = \mathsf{released} \wedge \mathsf{Run}(m,n)$ action $\mathsf{finished}_i$, and reset $\{\mathsf{status}(i,j) := \mathsf{free}, \mathsf{n}_i := \mathsf{n}_i - 1, r_m := C(m), c_m = 0, \mathsf{status}(m,n) := \mathsf{running}, R(P_i)\}$

(b) a new task P_{mn} is released, which preempts the running task P_{ij}: encoded by guard $\mathsf{status}(m,n) = \mathsf{free} \wedge \mathsf{Run}(m,n)$, action $\mathsf{release}_m$, and reset $\{\mathsf{status}(m,n) := \mathsf{running}, \mathsf{n}_m := \mathsf{n}_m + 1, r_m := C(m), c_m := 0, \mathsf{status}(i,j) := \mathsf{preempted}\} \cup \{r_k := r_k + C(m) \mid \mathsf{status}(k,l) = \mathsf{preempted}\}$ (we increment the response times of all preempted tasks by the execution time of the released higher-priority task).

(3) $\mathsf{Running}_i$ to Idle: edges labelled with guard $\mathsf{empty}(q)$ and reset $\{\mathsf{n}_i := 0, R(P_i)\}$.

(4) $\mathsf{Running}_i$ to $\mathsf{Running}_i$: edges representing the case when a task release does not preempt the running task P_{ij}: encoded by guard $\mathsf{status}(k,l) = \mathsf{free} \wedge \mathsf{Run}(i,j)$, action $\mathsf{released}_k$, and reset $\{\mathsf{status}(k,l) := \mathsf{released}, \mathsf{n}_k := \mathsf{n}_k + 1\} \cup \{r_k := r_k + C(m) \mid \mathsf{status}(k,l) = \mathsf{preempted}\}$

(5) $\mathsf{Running}_i$ to Error: an edge labelled by the guard $\mathsf{nonschedulable}(q)$.

Encoding of Deadline Checker E_{DC}. It is similar to the encoding of $E_i(\mathsf{Sch})$ described in the previous section, in the sense that it checks for deadline violations of each task instance independently. The clock d is used in E_{DC} to measure the time since the analysed instance of P_i was released for execution. E_{DC} also uses a data variable, named $\mathsf{instance}$. From location Idle the automaton non-deterministically starts to analyse a task on the edge to Check_i, at which clock d is reset and $\mathsf{instance}$ is set to n_i, i.e. the current number of released instances of task P_i. In Check_i, $\mathsf{instance}$ is decremented whenever an instance of P_i finishes its execution. The analysed task finishes when $\mathsf{instance} = 1$ and the location Idle is reentered. However, if d is greater than $D(i)$, the task failed to meet its deadline and the location Error is reached.

The next step of the encoding is to construct the product automaton $E(A)|| E_{\mathsf{SP}}(\mathsf{Sch})||E_{\mathsf{DC}}$ in which the automata can only synchronize on identical action symbols. We now show that the product automaton is bounded.

Lemma 3 *The clocks c_i and d, and the data variables r_i and n_i of $E_{\mathsf{SP}}(\mathsf{Sch})|| E_{\mathsf{DC}}$ in $E(A)||E_{\mathsf{SP}}(\mathsf{Sch})||E_{\mathsf{DC}}$ are bounded.*

Proof. First note that the integers r_k are bounded by $D(k) + \max_i(C(i))$ due to the fact that all edges incrementing r_k (by some $C(i)$) are guarded by the

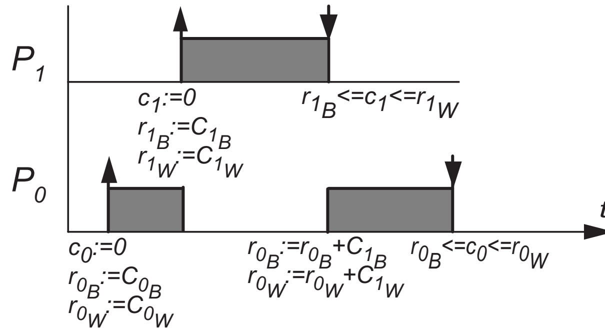

Fig. 5. Interval execution times.

constraint $\mathsf{Run}(m, n)$ requiring $\mathsf{r}_k \leq D(k)$. The bound for n_k is $\lceil D(k)/C(k) \rceil$. The clocks d and c_k are bounded by $\max_i(D(i))$ and r_k, respectively.

Lemma 4 *Let A be an extended timed automaton and Sch a fixed-priority scheduling strategy. Assume that (l_0, u_0, q_0) and $(\langle l_0, \mathsf{Idle}, \mathsf{Idle}\rangle, v_0)$ are the initial states of A and the product automaton $E(A)||E_{\mathsf{SP}}(\mathsf{Sch})||E_{\mathsf{DC}}$ respectively where l_0 is the initial location of A, u_0 and v_0 are clock assignments assigning all clocks with 0 and q_0 is the empty task queue. Then for any l the statement*

$$(l_0, u_0, q_0)(\longrightarrow)^*(l, u, \mathsf{Error})$$

holds for some u if and only if the statement

$$(\langle l_0, \mathsf{Idle}, \mathsf{Idle}\rangle, v_0)(\longrightarrow)^*(\langle l, m, n\rangle, v)$$

holds for some v, m and n where either m or n is Error.

Proof. The lemma can be proved by establishing the simulation between the states of A and $E(A)||\ E_{\mathsf{SP}}(\mathsf{Sch})||\ E_{\mathsf{DC}}$. It is similar to the proof for Lemma 2.

4.3 Systems with Interval execution times

We may extend the model to handle tasks whose execution time is an interval of the form $[C_{iB}, C_{iW}]$, where C_{iB} and C_{iW} denote the best and worst case execution times of task P_i respectively. Unfortunately the schedulability checking problem for such systems is undecidable [19].

In the following, we present an analysis method using over-approximation. The idea is to modify the scheduler automaton so that the variables are updated as shown in Figure 5. As before, we use c_i to keep track of the accumulated execution time of P_i, and a pair of data variables r_{iB} and r_{iW} to sum up the best and the worst completion time of P_i. Obviously r_{iB} and r_{iW} should be set to C_{iB} and C_{iW} respectively when task P_i starts to execute. Observe that each preemption will enlarge the difference $r_{0W} - r_{0B}$ for the preempted task P_0

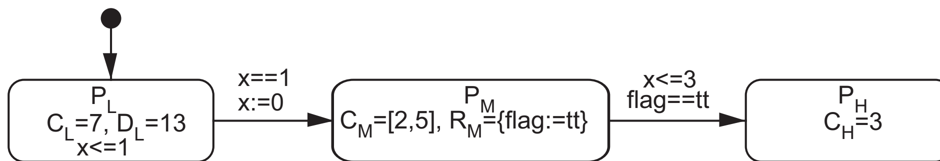

Fig. 6. An example when the over-approximation gives the wrong answer.

with lower priority by the difference $C_{1W} - C_{1B}$ for the finishing task P_1 with higher priority. Accordingly, we modify the scheduler automaton as follows: on edges labelled $\mathsf{finished}_j$ from locations $\mathsf{Running}(P_j)$ the guard should be $r_{jB} \leq c_j \leq r_{jW}$ and variable updating should be $r_{kB} := r_{kB} + C_{jB}, r_{kW} := r_{kW} + C_{jW}$ for all k such that $\mathsf{status}(P_k) = \mathsf{preempted}$. The rest of the scheduler automaton remains the same as before.

It is easy to see that the presented algorithm is an over-approximation. For example, consider the system shown in Figure 6. Tasks P_L, P_M and P_H have priorities low, medium and high respectively. Task P_L starts executing at time 0, and is being preempted by the task P_M at time 1. P_M has execution times in the interval $[2, 5]$ and by the end of its execution it sets the boolean variable $flag$ to true, which is initially set to false. If P_M completes its execution before 3 time units, it can trigger the higher priority task P_H that also preempts the execution of P_L. Obviously, the worst-case response time of P_L is 13, which means that it finishes its execution within its deadline. However, the algorithm will compute the worst-case response time of P_L as a sum of worst-case execution times of P_L, P_M and P_H, which equals 15 and exceeds the deadline of P_L.

5 Implementation and Experiments

Except the approximate method for the undecidable case, the presented analysis techniques have been implemented in TIMES, a tool for modeling and schedulability analysis of embedded real-time systems [8]. The modeling language of TIMES is ETA as described in Section 4.1. As the scheduling problems are modeled using timed automata, we are able to solve the related analysis problems using symbolic techniques based on DBM's (Difference Bound Matrices). The analysis module of TIMES is based on the verification engine of UPPAAL [20]. The tool currently supports symbolic simulation, schedulability analysis, and model checking of safety and bounded liveness properties. In addition, the tool can also be used to generate executable C-code [9] from the verified models.

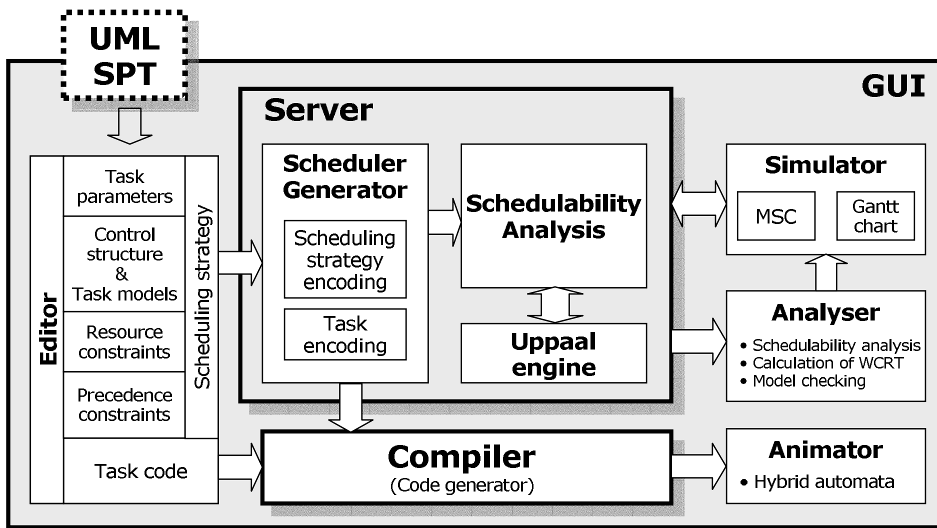

Fig. 7. The TIMES tool architecture.

5.1 An Overview of TIMES

The architecture of the TIMES tool is illustrated in Figure 7. The tool offers the following main features:

- **Editor** to graphically model a system and the abstract behaviour of its environment. A system description consists of a task set and a network of timed automata extended with tasks. A task is described by the task code (in C), its (worst-case) computation time and (relative) deadline, and if applicable optional parameters for priority (for fixed priority scheduling), period (for periodic tasks), and minimal inter-arrival time (for sporadic tasks). It is also possible to specify precedence constraints on the tasks using an editor for AND/OR precedence graphs, and resource access patterns using semaphores.
- **Simulator** to visualise the dynamic behaviour of a system model as Gantt charts and message sequence charts. The simulator can be used to randomly generate possible execution traces, or alternatively the user can control the execution by selecting the transitions to be taken. The simulator can also be used to visualise error traces produced in the analysis phase.
- **Analyser** to check that the tasks associated to a system model are guaranteed to always meet their deadline. In case schedulability analysis finds a task that may fail to meet its deadline, a trace is generated and visualised in the simulator. It is also possible to compute the worst-case response times of individual tasks. The schedulability analysis has also been extended to handle resource and precedence constraints [15]. In addition to schedulability, it is possible to analyse safety and liveness properties specified as temporal logic formulae.

- **Server** consisting of two parts: a scheduler generator, and a module for schedulabilty analysis based on the UPPAAL engine [20] with extensions. The scheduler generator produces a scheduler automaton based on input from the editor, which is composed in parallel with an annotated version of the original system automata. The parallel composition is analysed by on-the-fly reachability techniques in the schedulabilty analysis module. Currently supported scheduling policies are: rate monotonic, deadline monotonic, fixed priority scheduling (with user defined priorities), earliest deadline first (EDF), and first come first served (FCFS). All scheduling policies support preemptive or non-preemptive task sets.
- **Compiler** to generate executable C code from timed models i.e. timed automata with tasks. If a model is proven to be schedulable, the execution of the generated code will guarantee the timing constraints i.e. the deadlines of tasks under the assumption that the target platform ensures that the task code can be executed in the specified computation time.
- **Animator** to transform hybrid automata modeling the controlled environment into C code simulating the controlled objects in the environment of the embedded system. The simulated environment enables the designer to experiment with the design prior to implementation.

A screen-shot of the TIMES tool analysing a simple control system is shown in Fig. 8. In the main window, a control automaton is displayed. To the left, a table shows the specified task parameters. The task parameters currently supported are: behaviour (B) [6], priority (P), computation time (C), deadline (D), and period (T).

5.2 Calculation of Worst-Case Response Times

The schedulability analysis in TIMES is essentially performed by computing the worst-case response times of tasks, and then comparing with the respective deadlines. In practice, the worst-case response time is a system parameter that can be used not only for checking the schedulability of a system, but also for analysis of the system performance. Therefore, when schedulability analysis of a system is performed in TIMES, it is possible to show the worst-case response times of the tasks in the system.

The worst-case response time of a task is the time delay from the instant the task is released to the instant it finishes. In general, the response time of a task is a non-integer value due to the fact that a task can be released at any time point at which the task queue may already contain tasks with higher priorities, whose remaining computing times can be any reals. We take

[6] The behaviour field is one of: periodic (P), sporadic (S), or controllable (C) if the time points for the task release are specified by an automaton.

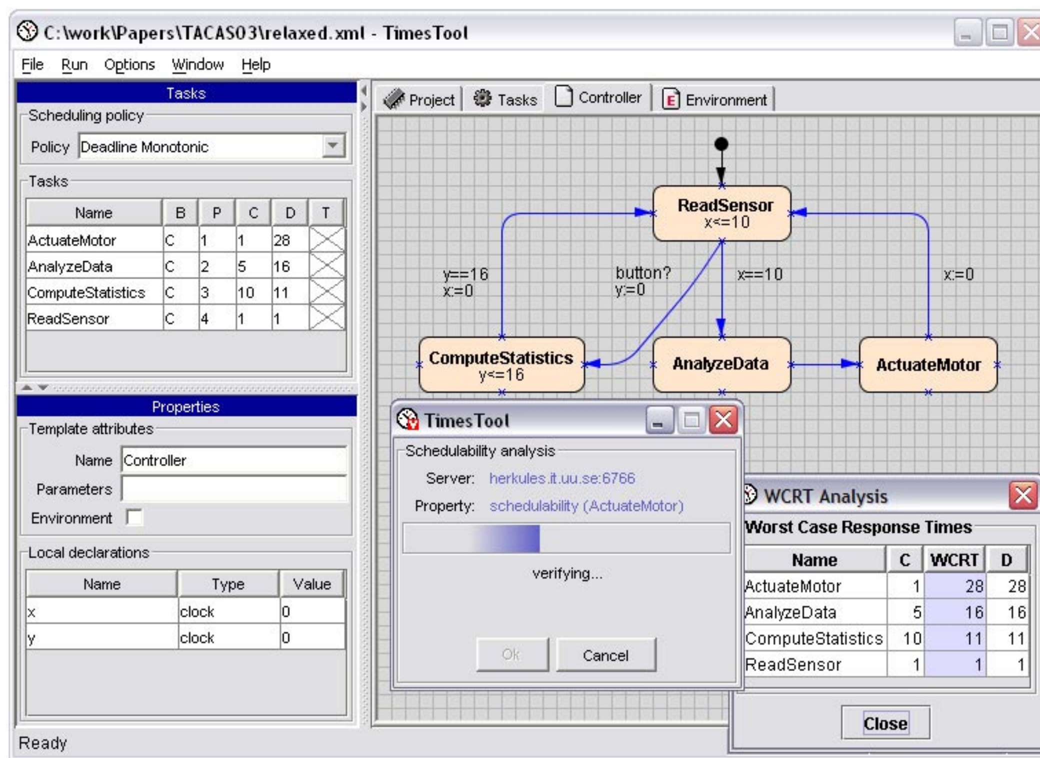

Fig. 8. The TIMES tool performing schedulability analysis.

the worst-case response time to be the least integer greater or equal to the longest response time of a task. The worst-case response time of task P_i can be obtained from the maximum value appearing in the upper bound on the clock d in the symbolic states generated during the schedulability analysis of task P_i (i.e. in the reachability analysis).

An example. We use an example to illustrate how the tool is used for schedulability analysis. Fig. 8 shows a system consisting of tasks with fixed priorities and data-independent control. It is a simple controller of a motor, periodically polling a sensor and at requests providing a user with sensor statistics. In the initial location, an instance of task ReadSensor is released. The controller waits 10 time units for a user to push the button. If the button is not pushed, the controller releases the two tasks AnalyzeData and ActuateMotor. If the button is pushed when the controller operates in its initial location, an instance of task ComputeStatistics is released for execution, and the controller waits 16 time units before releasing task ReadSensor again.

The system has been analysed with two algorithms implemented in the TIMES tool. An implementation based on the original decidability result described in [14] consumes 2.7 seconds, whereas an implementation of the algorithm presented in Section 3 of this paper terminates in 0.1 seconds on the same

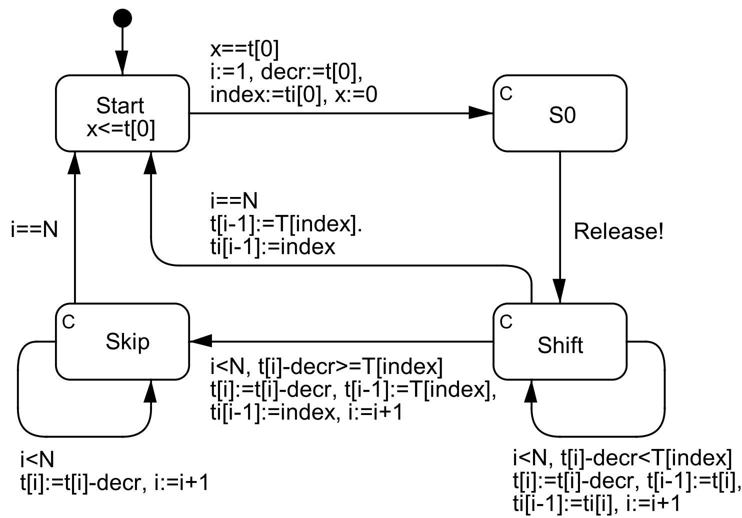

Fig. 9. Encoding of periodic task release pattern PeriodicTasks(const N).

machine[7]. Thus, the time consumption is reduced significantly for this system.

5.3 Periodic Tasks and Experiments

For the analysis of systems with only periodic tasks, the classical technique, Rate-Monotonic Analysis (RMA) using equation solving, is the best known technique to calculate the worst case response times. Based on the critical instant assumption that the worst case scenario will appear in the first period for all tasks, RMA calculates only the response time for the first period of each task. To benefit from this assumption, as in RMA, for the analysis of periodic tasks, Times is implemented to explore only the first periods of the tasks. Therefore, as RMA, Times does not suffer from the exponential explosion in the number of periodic tasks.

To further improve the performance of the tool for the analysis of periodic tasks, we use a one-clock timed automaton shown in Fig. 9 to describe the task arrival pattern of a task set containing only periodic tasks. The automaton PeriodicTasks encodes releases of all periodic task using only one clock. The automaton Scheduler shown in Fig. 10 is a slightly modified version of the scheduler automaton described in Section 3, schedules only the first periods of the released tasks. It is now parameterized with a constant ID that is the index of the currently checked task. The automaton PeriodicTasks is parameterized with a constant N that is a number of periodic tasks with indexes less than or equal to ID. The automaton PeriodicTasks uses the following variables:

[7] The measurements were made on a Sun Ultra-80 running SunOS 5.7. The UNIX program `time` was used to measure the time consumption.

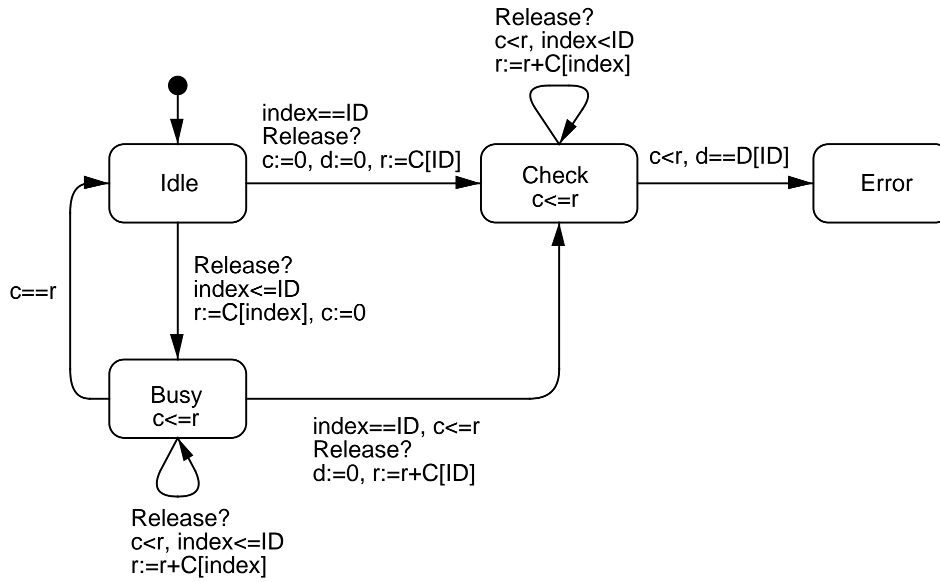

Fig. 10. Encoding of scheduler Scheduler(const ID).

- clock x keeps track of the moments when tasks are released,
- int t[N] is a sorted array of times remaining until next release for every task (initialized with task offsets),
- int[0,N] ti[N] is an array of task indexes corresponding to the elements of the array t[N],
- int decr is the time until next task release relative to the moment of the previous task release,
- int[0,N] i is a counter for enumerating elements of t and ti, and
- const T[N] is an array of task periods.

The two automata communicate through the channel Release and integer variable index which represents the index of the task released at the moment of synchronization. In the initial location the automaton PeriodicTasks waits for the next task release, sets the value of decr to the value of the first element of t, and index to the value of the first element of ti. Then the scheduler is signalled about a new task release. In the Shift and Skip locations all values of t are decremented by decr and those less than the period of currently released task are shifted one position to the beginning making room for the period of the newly released task. Then the automaton PeriodicTasks moves back to the Start location. The Scheduler automaton behaves similarly to the one described in section 3 except one channel and one integer are used instead of N channels. If in the system tasks with indexes $\leq$ ID are only periodic then it is sufficient to consider only the longest of their first periods, thus the transition leading from the location Busy to the location Idle is omitted.

To evaluate the performance of our two clock encoding for fixed priority scheduling strategy, we have studied several task sets containing up to 500 periodic tasks with randomly generated task parameters: periods, computing

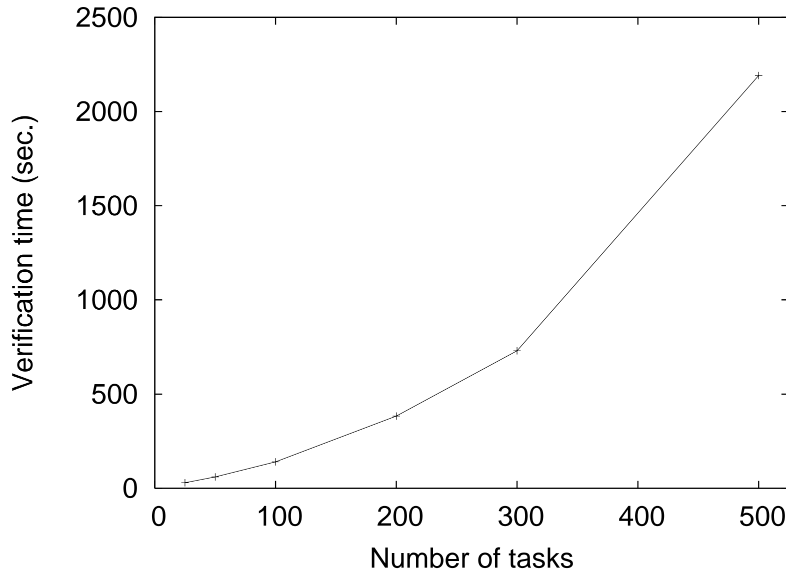

Fig. 11. Dependency between the number of tasks and the verification time.

times and deadlines equal to the periods. The task periods are generated according to the formula $n \pm random[0, n/10]$ where n is the number of tasks in the respective task set, and the computing times are random numbers in the interval $[1, 3]$.

The implementation based on the original decidability result described in [14] never terminated due to the large number of required clocks. The times taken by schedulability analysis using two-clocks encoding are shown in Fig. 11. One can see the dependency between the number of tasks in the task set and the time it takes to analyse the set to be quadratic. The performance is indeed comparable with the RMA technique based on equation solving for schedulability analysis of perodic tasks. However, our technique is able to check systems containing not only periodic tasks, but also tasks with non-uniformly recurring patterns.

6 Conclusions

In this paper we have shown that for fixed priority scheduling strategy, the schedulability checking problem of timed automata extended with tasks can be solved by reachability analysis on standard timed automata using only two additional clocks. We have also shown how to extend the result to systems with data-dependent control, i.e. systems in which the release time-points of a task may depend on the values of shared variables, and hence on the time-point at which other tasks finish their execution. For such systems we use additional clocks for keeping track of execution of the tasks that have shared variables

with the control automata. We need one additional clock for each task type that updates variables shared between control automata and/or other tasks. In this case the schedulability checking problem uses $n+1$ extra clocks, where n is the number of tasks types that update the shared variables. However, this result is applicable only when exact execution times of tasks are known. When the task execution times of tasks are given as intervals, an over-approximation technique can be used as the problem is undecidable.

Both these encodings use much fewer clocks than the analysis suggested in the original decidability result, and we believe that we have found the optimal solutions to the problems. The presented encodings seem to suggest that the general schedulability problem of ETA can be transformed into a reachability problem of standard timed automata, instead of timed automata with subtraction operation on clocks. This is indeed the case, but the number of clocks used in the standard timed automaton will be the same as in the encoding using timed automata with subtraction.

The schedulability checking algorithms described in this paper have been implemented in the Times tool. An experiment shows that the new techniques substantially reduces the computation time needed to analyse an example systems with fixed priority scheduling strategy.

Acknowledgements: We would like to thank the anonymous referees for their constructive comments.

References

[1] Y. Abdeddaïm and O. Maler. Job-shop scheduling using timed automata. In *In Proc. CAV'01*, volume 2102 of *Lecture Notes in Computer Science*, pages 478–492. Springer–Verlag, 2001.

[2] Y. Abdeddaïm and O. Maler. Preemptive job-shop scheduling using stop-watch automata. In *Proc. of TACAS'02*, pages 113–126, 2002.

[3] K. Altisen, G. Gößler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *Proc. of IEEE RTSS'99*, pages 154–163. IEEE Computer Society Press, 1999.

[4] K. Altisen, G. Gößler, and J. Sifakis. A methodology for the construction of scheduled systems. In *Proc. of FTRTFT'2000, LNCS 1926, pp.106-120*, 2000.

[5] K. Altisen, G. Gößler, and J. Sifakis. Scheduler modeling based on the controller synthesis paradigm. *Journal of Real-Time Systems, special issue on Control Approaches to Real-Time Computing*, 23:55–84, 2002.

[6] 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.

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

[8] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times - a tool for modelling and implementation of embedded systems. In *Proc. of the 8*th *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 460–464. Springer–Verlag, 2002.

[9] T. Amnell, E. Fersman, P. Pettersson, H. Sun, and W. Yi. Code synthesis for timed automata. Submitted to Nordic Journal of Computing, 2002.

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

[11] F. Cassez and F. Laroussinie. Model-checking for hybrid systems by quotienting and constraints solving. In *Proceedings of the 12*th *International Conference on Computer-Aided Verification*, volume 1855, pages 373–388, Standford, California, USA, 2000. Springer-Verlag.

[12] 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.

[13] A. Fehnker. Scheduling a steel plant with timed automata. In *Proc. of RTCSA'99*. IEEE Computer Society Press, 1999.

[14] E. Fersman, P. Pettersson, and W. Yi. Timed automata with asynchronous processes: Schedulability and decidability. In *Proc. of the 8*th *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 67–82. Springer–Verlag, 2002.

[15] E. Fersman and W. Yi. A generic approach to schedulability analysis of real time tasks. To appear in Nordic Journal on Computing, 2004.

[16] T. Hune, K. G. Larsen, and P. Pettersson. Guided Synthesis of Control Programs using UPPAAL. *Nordic Journal of Computing*, 8(1):43–64, 2001.

[17] M. Joseph and P. Pandya. Finding response times in a real-time system. *BSC Computer Journal*, 29(5):390–395, October 1986.

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

[19] P. Krčál and W. Yi. Decidable and undecidable problems in schedulability analysis using timed automata. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of TACAS'04, Barcelona, Spain.*, volume 2988 of *Lecture Notes in Computer Science*, pages 236–250. Springer-Verlag, 2004.

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

[21] J. W. S. Liu. *Real-Time Systems.* Prentice-Hall, 2000.

[22] 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.

[23] J.C. Palencia and M. Gonzlez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proc. of RTSS'98*, pages 26–38. IEEE Computer Society Press, 1998.

[24] O. Redell and M. Törngren. Calculating exact worst case response times for static priority scheduled tasks with offsets and jitter. In *Proc. of IEEE Real Time Technology and Applications Symposium*, 2002.

[25] K. Tindell. Adding time-offsets to schedulability analysis, January 1994.

[26] Liu Z. and Joseph M. Verification, refinement and scheduling of real-time programs. *Theoretical Computer Science*, 253(1):119–152, January 2001.