Schedulability Analysis Using Two Clocks

Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi
Uppsala University
Department of Information Technology
P.O. Box 337, S-751 05 Uppsala, Sweden
Email: {elenaf,leom,paupet,yi}@it.uu.se

Abstract. In classic scheduling theory, real-time tasks are usually assumed to be periodic, i.e. tasks arrive and compute 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 in many cases is huge.
In this paper, we show that for 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 classic Rate-Monotonic Scheduling. We believe that this is the optimal solution to the problem, a problem that was suspected undecidable previously. We also extend the result to systems in which the timed automata and the tasks may read and update shared data variables. Then the release time-point of a task may depend on the values of the shared variables, and hence on the time-point at which other tasks finish their exection. We show that this schedulability problem can be encoded as timed automata using $n+1$ extra clocks, where n is the number of tasks.

1 Introduction

In the area of real time scheduling methods such as rate monotonic scheduling are widely applied in the analysis of periodic tasks with deterministic behaviours. For non-periodic tasks with non-deterministic behaviours, there are no satisfactory procedures. In reality control tasks are often triggered by sporadic events coming from the environment. The common approach to analyze schedulability of such systems with non-periodic tasks is to consider the minimal inter-arrival time of a task as its period and then follow the ordinary technique used for periodic tasks. Obviously such an approximate method is quite pessimistic since the task control structures are not considered. A major advantage can be gained using timed automata to specify relaxed timing constraints on events and model

other behavioural aspects such as concurrency and synchronization. In order to perform schedulability analysis with timed automata the model of Extended Timed Automata (ETA) has been suggested in [FPY02]. It unifies timed automata [AD94] with the classic 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 has been proven to be decidable for any scheduling policy and the algorithm for schedulability analysis was presented. It is based on translation of the schedulability problem into reachability for the decrementation automata [MV94]. A remaining challenge is to make the result applicable for schedulability analysis of systems with non-uniformly recurring tasks that scale up to industrial systems. In this paper we present an efficient algorithm for schedulability analysis of systems with relaxed timing constraints, which uses only two additional clocks. The algorithm also allows to compute the worst-case response time for non-periodic tasks.

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

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 system that describes the possible events occurring during its execution. Those events must satisfy given timing constraints. To clarify how events, accepted by a timed automaton, should be handled or computed we extend timed automata with asynchronous processes [FPY02], i.e. tasks triggered by events asynchronously. 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].

Syntax. Let $\mathcal{P}$ ranged over by P, Q, R, denote a finite set of task types. A task type may have different instances that are copies of the same program with different inputs. 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

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

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.

As in timed automata, assume a finite set of alphabets Act for actions and a finite set of real-valued variables $\mathcal{C}$ for clocks. We use a, b etc. to range over Act and x_1, x_2 etc. to range over $\mathcal{C}$. We use $\mathcal{B}(\mathcal{C})$ ranged over by g to denote the set of conjunctive formulas of atomic constraints in the form: $x_i \sim C$ or $x_i - x_j \sim D$ where $x_i, x_j \in \mathcal{C}$ are clocks, $\sim \in \{\leq, <, \geq, >\}$, and C, D are natural numbers. The elements of $\mathcal{B}(\mathcal{C})$ are called *clock constraints*.

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

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

Operational Semantics. 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 corresponds to the arrival of new task instances.

We represent the values of clocks as functions (called clock assignments) from $\mathcal{C}$ to the non–negative reals. 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, $\mathsf{EDF}([P(3.1, 10),$

[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.

$Q(4, 5.3)]) = [Q(4, 5.3),\ P(3.1, 10)])$. We call such sorting functions scheduling strategies that may be preemptive or non-preemptive [3].

Run is a function which given a real number t and a task queue q returns the resulted task queue after t time units of execution according to available computing resources. For simplicity, we assume that only one processor 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 non-negative a 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 with* $M(m)$ *inserted in* q*.*

2.2 Schedulability and Decidability

In this section we briefly review the verification problems of ETA. For more details, we refer the reader to [FPY02]. 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.*

[3] 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.

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

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

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

Proof. The proof is given in [FPY02]. □

3 Main Result: Two Clocks Encoding

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.

Our analysis technique is inspired by Joseph and Pandya's rate-monotonic analysis of periodic tasks [JP86], 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. Similar to Joseph and Pandya, we check for each task type 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.

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 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. The check is performed in priority order for each task in $\mathcal{P}$, starting with the task of highest priority.

To construct the $E(A)$, the automaton A is annotated with distinct synchronization actions $\mathsf{release}_i$ on all edges leading to locations labeled 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. □

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:

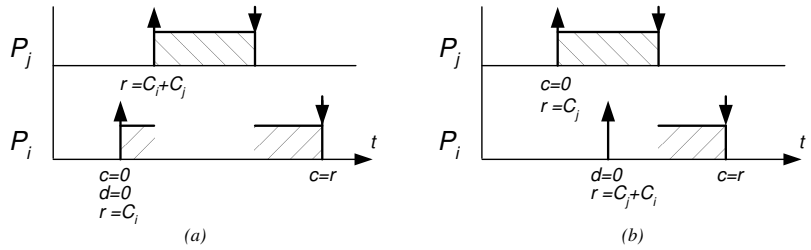

Fig. 1. Task execution schemes for tasks P_i and P_j with $\mathsf{Prio}(j) > \mathsf{Prio}(i)$. The symbols $\uparrow$ and $\downarrow$ indicate release and completion of tasks, respectively.

- 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 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 clock d is reset when the analysis of a task instance begins, and will be used to check that 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 how the clocks c and d are reset, and 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.
- 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:

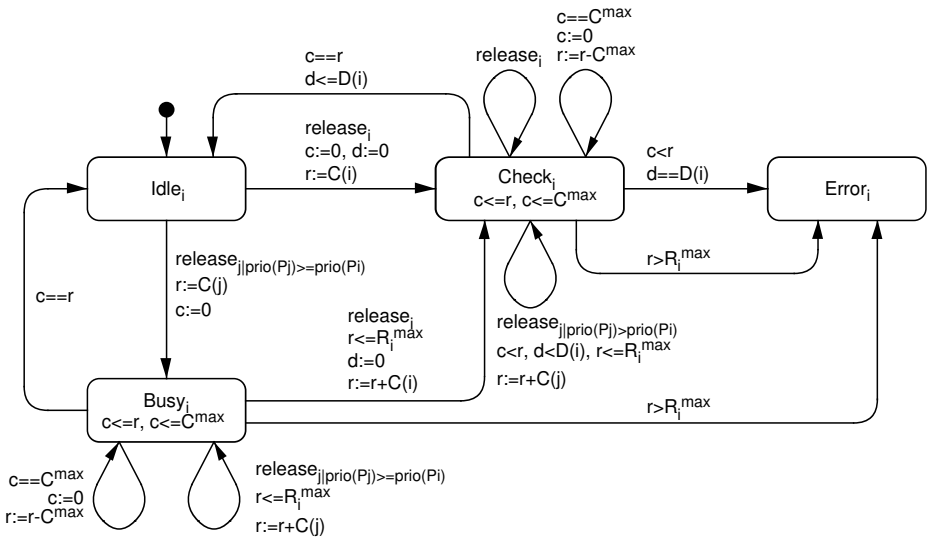

Fig. 2. Encoding of schedulability problem.

- 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 $\mathsf{c}{=}\mathsf{r}$ 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} 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 task sets in $E_i(\mathsf{Sch})$.

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 clock assignments assigning all clocks with* 0 *and* q_0 *is the empty task queue. Then the following holds:*

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

for some l, u, l', v, i*.*

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

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 [AFP+03].

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.

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 $C(\mathsf{Hd}(q, t)) > t$
- $(l, \sigma, q) \xrightarrow{t}_{\mathsf{Sch}} (l, (\sigma[A(\mathsf{Hd}(q))]) + t, \mathsf{Run}(q, t))$ if $(\sigma + t) \models I(l)$ and $C(\mathsf{Hd}(q)) = t$

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

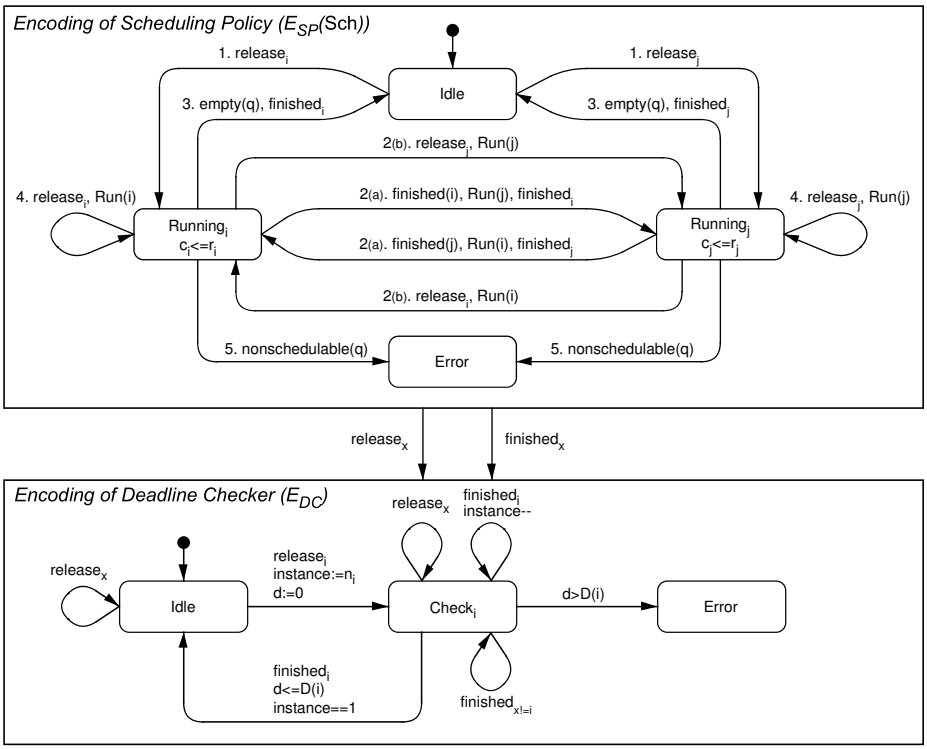

Fig. 3. Encoding of schedulability problem.

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

The construction of $E(\mathsf{Sch})$ is illustrated in Fig.3. It is 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 $E_{\mathsf{SP}}(\mathsf{Sch})$. We first introduce some notation. 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

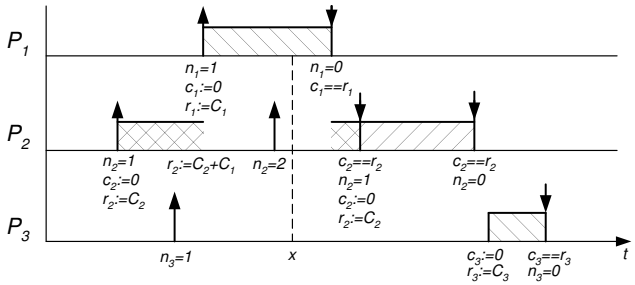

Fig. 4. Task execution scheme where $\mathsf{Prio}(1) > \mathsf{Prio}(2) > \mathsf{Prio}(3)$.

$\mathsf{status}(k,l) = \mathsf{running}$ (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).

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

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

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 labeled as follows:

1. Idle to $\mathsf{Running}_i$: edges labeled 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 Idle: edges labeled with guard $\mathsf{empty}(q)$ and reset $\{\mathsf{n}_i := 0, R(P_i)\}$.
3. $\mathsf{Running}_i$ to $\mathsf{Running}_m$: two types of edges:
   (a) the running task P_{ij} is finished and P_{mn} is scheduled to run by $\mathsf{Run}(m,n)$. There are two cases:
      i. P_{mn} was preempted earlier: 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).
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 labeled 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 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 the following holds:*

$$(l_0, u_0, q_0)(\longrightarrow)^*(l, u, \mathsf{Error}) \text{ iff } (\langle l_0, \mathsf{Idle}\rangle, v_0)(\longrightarrow)^*(\langle l', \mathsf{Error}, m\rangle, v) \text{ or } (\langle l_0, \mathsf{Idle}\rangle, v_0)(\longrightarrow)^*(\langle l'', m', \mathsf{Error}\rangle, v')$$

for some $l, u, l', l'', m, m', v, v'$.

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

5 Implementation

The algorithm described in Section 3 has been implemented in Times, a tool for modeling and schedulability analysis of embedded real-time systems [AFM+02]. The modeling language of Times is ETA as described in Section 4.1 of this paper. The tool currently supports simulation, schedulability analysis, checking of safety and liveness properties, and synthesis of executable C-code [AFP+03].

A screenshot of the Times tool analysing a simple control system with data-dependent control consisting of tasks with fixed priorities is shown in Fig.5. The schedulability analysis is performed as described in Section 3.

The system analysed in Fig.5 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

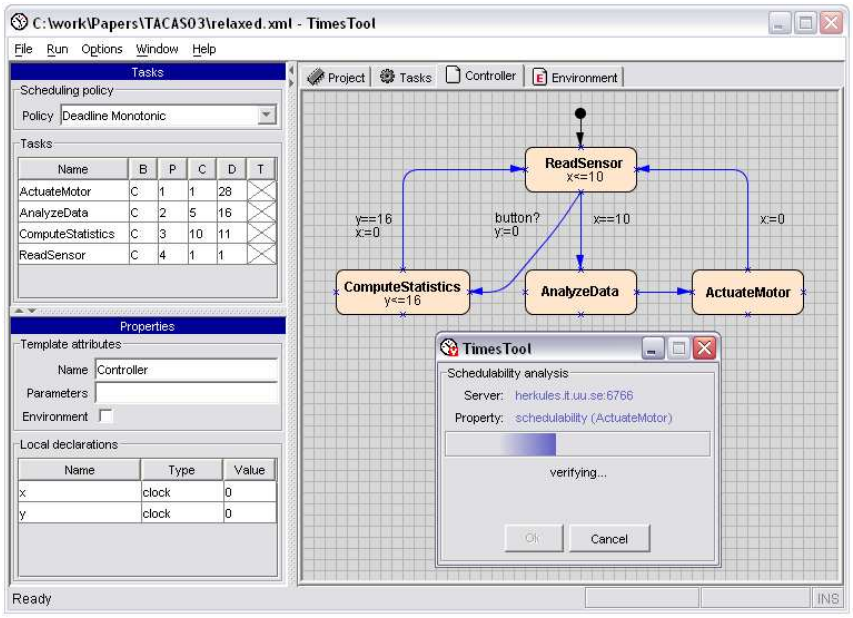

Fig. 5. The TIMES tool performing schedulability analysis.

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 [FPY02] 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 machine[6]. Thus, the time consumption is reduced significantly for this system.

In addition to schedulability analysis, it is possible to adjust the algorithms presented in this paper, and implemented in TIMES, to compute the worst-case response time of tasks in a schedulable system. In general, the response time of a task is a non-integer value. We take the worst-case response time to be the lowest integer value 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 [7] in the symbolic states generated during the schedulability analysis of task P_i (i.e. in the reachability analysis). In Fig.5 the numbers in the task table column D are the worst-case response times of the tasks in the system. Thus, if any of them is decreased, the system becomes non-schedulable.

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

[7] Here we refer to clock d described in Section 3.

6 Conclusions and Related Work

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. In this case the encoding into reachability problem for standard timed automata uses $n + 1$ clocks, where n is the number of tasks types. 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 reduce the computation time needed to analyse an example systems with fixed priority scheduling strategy.

Related work. Well established scheduling theory and scheduling algorithms are described in various publications. In the area of real time scheduling methods such as rate monotonic scheduling [But97] are widely applied in the analysis of systems with deterministic behaviours restricted to periodic tasks. However, for systems with non-periodic tasks and non-deterministic behaviours, there are still no satisfactory procedures to perform schedulability analysis. One of the approaches to achieve schedulability is based on controller synthesis paradigm [AGS02,AGP+99]. The methodology described in [AGS02] relies on the idea that one can build schedulable system successively restricting guards of the controllable actions in its model in an appropriate way. However, concepts related to implementation description are not addressed in this work. In the area of non-preemptive scheduling timed automata has been used mainly for job-shop scheduling [Feh99,AM01,HLP01]. The idea is to get schedules out of traces produced during reachability analysis for pre-defined locations specifying scheduling goal. Stop-watch automata have been used to solve preemptive scheduling problem [MV94,Cor94,CL00]. But since reachability analysis problem for this class of automata is undecidable in general there is no guarantee of termination for the analysis without the assumption that task preemptions occur only at integer points. Tools have been developed to support the design and analysis of embedded systems with tasks. Examples hereof include TAXYS [CPP+01] and Times [AFM+02].

References

[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 *In Proc.TACAS'02*, volume 2280 of *LNCS*, pages 460–464. Springer, 2002.

[AFP+03] T. Amnell, E. Fersman, P. Pettersson, H. Sun, and W. Yi. Code synthesis for timed automata. To appear in Nordic Journal of Computing, 2003.

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

[AGS02] 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.

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

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

[CL00] F. Cassez and F. Laroussinie. Model-checking for hybrid systems by quotienting and constraints solving. In *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 *In Proc. IEEE RTSS'94*, pages 132–141, 1994.

[CPP+01] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys: a tool for the development and verification real-time embedded systems. In *In Proc. CAV'01*, volume 2102 of *LNCS*. Springer, 2001.

[Feh99] A. Fehnker. Scheduling a steel plant with timed automata. In *In Proc. IEEE RTCSA'99*, 1999.

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

[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.

[JP86] M. Joseph and P. Pandya. Finding response times in a real-time system. *BSC Computer Journal*, 29(5):390–395, October 1986.

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