Timed Automata as Task Models for Event-Driven Systems

Christer Norström[1] and Anders Wall[1]
[1]Mälardalen University
Department of Computer Engineering
P.O. Box 883, S-721 23 Västerås, Sweden
{awl,cen} @mdh.se

Wang Yi[1,2]
[2]Uppsala University
Department of Computer Systems
P.O. Box 325, S-751 05 Uppsala, Sweden
yi@docs.uu.se

Abstract

In this paper, we extend the classic model of timed automata with a notion of real time tasks. The main idea is to associate each discrete transition in a timed automaton with a task (an executable program). Intuitively, a discrete transition in an extended timed automaton denotes an event releasing a task and the guard on the transition specifies all the possible arriving times of the event (instead of the so–called minimal inter-arrival time). This yields a general model for hard real-time systems in which tasks may be periodic and non-periodic.

We show that the schedulability problem for the extended model can be transformed to a reachability problem for standard timed automata and thus it is decidable. This allows us to apply model-checking tools for timed automata to schedulability analysis for event-driven systems. In addition, based on the same model of a system, we may use the tools to verify other properties (e.g. safety and functionality) of the system. This unifies schedulability analysis and formal verification in one framework. We present an example where the model–checker UPPAAL *is applied to check the schedulability and safety properties of a control program for a turning lathe.*

1. Introduction

The traditional approach to the development of hard real-time system is often based on scheduling theory. There are various methods [5, 12, 7] e.g. rate monotonic scheduling, which have been very successful for the analysis of time-driven systems as tasks are *periodic.* To deal with *non-periodic* tasks in event–driven systems, the standard method is to consider non-periodic tasks as periodic using the *minimal* inter-arrival times as *task periods.* Clearly, the analysis result based on such a task model would be pessimistic in many cases, e.g. a task set which is schedulable may be considered as non-schedulable as the inter-arrival times of the tasks may vary over time, that are not necessary minimal.

In recent years, in the area of formal methods, there have been several advances in formal modeling and analysis of real time systems based the theory of timed automata due to the pioneering work of Alur and Dill [2]. Notably, a number of verification tools have been developed (e.g. KRONOS and UPPAAL [6, 4]) in the framework of timed automata, that have been successfully applied in industrial case studies (e.g. [3, 13, 11]). Timed automata have proved expressive enough for many real-life examples, in particular, for event-driven systems. The advantage with timed automata is that one may specify very relaxed timing constraints on events (i.e. discrete transitions) than the traditional approach in which events are often considered to be periodic. However, it is not clear how the model of timed automata can be used for schedulability analysis. In this paper, we present an extended version of timed automata with real-time tasks to provide a model for event-driven systems. We show that the extended model can be used for both schedulability analysis and verification of other properties, e.g. safety and liveness properties of timed systems. This unifies schedulability analysis and formal verification in one framework.

The main idea is to associate each discrete transition in a timed automaton with a task (or several tasks in the general case). A task is assumed to be an executable program with two given parameters: its worst execution time and deadline. Intuitively, a discrete transition in an extended timed automaton denotes an event releasing a task and the guard (clock constraints) on the transition specifies all the possible arrival times of the associated task. Whenever a task is released, it will be put in the scheduling queue for execution. We assume that the tasks will be executed according to a given scheduling strategy e.g. earliest deadline first. Then a delay transition of the timed automaton corresponds to the execution of the task with earliest deadline and idling for the other waiting tasks.

Thus, the sequences of discrete transitions of an extended timed automaton will correspond to the sequences of *arrivals* of non-periodic tasks. We say that such a sequence of tasks is schedulable if all the tasks can be executed within their deadlines. Naturally an automaton is *schedulable* if all the task sequences are schedulable. We shall show that under the assumption that the tasks are non-preemptive, the schedulability problem can be transformed to a reachability problem for ordinary timed automata and thus it is decidable. This allows us to apply model-checking tools for timed automata to schedulability analysis for event-driven systems. We present an example where the model–checker UPPAAL is applied to check the schedulability and safety properties of a control program in control applications.

The rest of this paper is organized as follows: Section 2 presents the syntax and semantics of the extended timed automata with tasks. Section 3 shows how to transform the scedulability analysis problem for extended model to a reachability problem for ordinary timed automata, and thus schedulability analysis may be performed by the existing verification tools for timed automata. Section 4 provides an example to illustrate our approach. Section 5 concludes the paper with summarized results and future work.

2. Timed Automata with Real-Time Tasks

The theory of timed automata was first introduced in [2] and has since then established as a standard model for real time systems. We first give an brief review to fix the terminology and notation and then present an extended version of the model with tasks.

2.1. Timed Automata

A timed automaton is a standard finite-state automaton extended with a finite collection of real-valued clocks. The transitions of a timed automaton are labelled with a *guard* (a condition on clocks), an *action*, and a *clock reset* (a subset of clocks to be reset). Intuitively, a timed automaton starts execution with all clocks set to zero. Clocks increase uniformly with time while the automaton is within a node. A transition can be taken if the clocks fulfill the guard. By taking the transition, all clocks in the clock reset will be set to zero, while the remaining keep their values. Thus transitions occur instantaneously. Semantically, a state of an automaton is a pair of a control node and a *clock assignment*, i.e. the current setting of the clocks. Transitions in the semantic interpretation are either labelled with an action (if it is an instantaneous switch from the current node to another) or a positive real number i.e. a time delay (if the automaton stays within a node letting time pass).

For the formal definition, we assume a finite set of alphabets Act for actions and a finite set of real-valued variables C for clocks. We use a, b etc to range over Act and X_1, X_2 etc. to range over C. We use $\mathcal{B}(C)$ ranged over by g and later by ϕ etc, denote the set of conjunctive formulas of atomic constraints in the form: $X_i \sim m$ or $X_i - X_j \sim n$ where $X_i, X_j \in C$ are clocks, $\sim \in \{\leq, <, \geq, >\}$, and m, n are natural numbers. The elements of $\mathcal{B}(C)$ are called *clock constraints.*

Definition 1. *A timed automaton over actions Act and clocks C is a tuple $\langle N, l_0, E\rangle$ where*

- *N is a finite set of nodes,*
- *$l_0 \in N$ is the initial node, and*
- *$E \subseteq N \times \mathcal{B}(C) \times Act \times 2^C \times N$ is the set of edges.*

When $\langle l, g, a, r, l'\rangle \in E$, we write $l \xrightarrow{g,a,r} l'$. $\square$

Formally, we represent the values of clocks as functions (called clock assignments) from C to the non–negative reals $\mathrm{R}_{\geq 0}$. We denote by $\mathcal{V}$ the set of clock assignments for C. A semantical *state* of an automaton is now a pair (l, u), where l is a node of the automaton and u is a clock assignment and the semantics of the automaton is given by a transition system with the following two types of transitions (corresponding to delay–transitions and action–transitions):

- $(l, u) \xrightarrow{d} (l, u + d)$
- $(l, u) \xrightarrow{a} (l', u')$ if $l \xrightarrow{g,a,r} l'$, $u \in g$ and $u' = [r \mapsto 0]u$

where for $d \in \mathrm{R}_{\geq 0}$, $u + d$ denotes the clock assignment which maps each clock X in C to the value $u(X) + d$, and for $r \subseteq C$, $[r \mapsto 0]u$ denotes the assignment for C which maps each clock in r to the value 0 and agrees with u over $C \backslash r$. By $u \in g$ we denote that the clock assignment u satisfies the constraint g.

2.2. Extended Timed Automata with Tasks

We shall view a timed automaton as an abstract model of a running process. The model describes the possible events (alphabets accepted by the automaton) that may occur during the execution of the process and the occurrence of the events must follow the timing constraints (given by the clock constraints). But the model gives no information on how these events should be handled. In many cases, for example in a control system, when an external event occurs, some computation must be performed to handle the event. A more concrete example is an interrupt handling system.

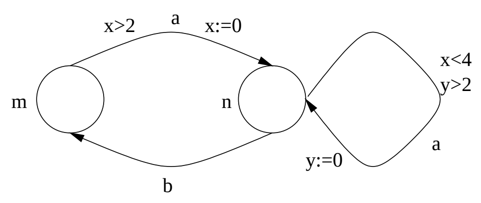

Figure 1. An Example Timed Automaton with Tasks.

Whenever an interrupt signal occurs, the associated interrupt handling program will be executed.

Now, assume that each action symbol in a timed automaton is associated with a program called *task*. Let P ranged over by p etc, denote the set of tasks. We further assume that the *worst case execution time* and *hard deadline* of the tasks in P are known. We shall use clock constraints to specify the arrival times of the tasks. Thus, each task p in P is characterized as a pair (c, d) of natural numbers with $c \leq d$ where c is the execution time of p and d is the relative deadline for p.

The deadline d is a relative deadline meaning that when task p is released, it should finish within d time units.

Definition 2. *An extended timed automaton with tasks (TAT), over actions Act, clocks C and tasks P is a tuple $\langle N, l_0, E, T\rangle$ where*

- *$\langle N, l_0, E, T\rangle$ is a standard timed automaton,*
- *$T : Act \hookrightarrow P$ is a partial function assigning tasks to actions.*

$\square$

Semantically, an extended automaton may perform two types of transitions just as an ordinary timed automaton. In addition, an action transition will release a new instance of the task associated with the action. Assume that there is a queue holding all the task instances generated by action transitions and ready to run. The queue corresponds to the ready queue in an operating systems. A semantic state of an extended automaton is a triple consisting of a *node* (the current control node), a *clock assignment* (the current setting of the clocks) and a *task queue* (the current status of the ready queue).

Consider the automaton of Figure 1. Let $p1$ and $p2$ be tasks handling the interrupt signals a and b respectively. Assume that the initial state is $(m, [x = 0, y = 0], [])$ where the clocks are 0 and the task queue is empty. Then the automaton may demonstrate the following sequence of transitions:

$$\begin{aligned}(m, [x = 0, x = 0], []) &\xrightarrow{3} (m, [x = 3, y = 3], [])\\ &\xrightarrow{a} (n, [x = 0, y = 3], [p1])\\ &\xrightarrow{a} (n, [x = 0, y = 0], [p1, p1])\\ &\xrightarrow{3} (n, [x = 3, y = 3], [p1, p1])\\ &\xrightarrow{a} (n, [x = 3, y = 0], [p1, p1, p1])\\ &\xrightarrow{1} (n, [x = 4, y = 1], [p1, p1, p1])\\ &\xrightarrow{b} (m, [x = 4, y = 1], [p1, p1, p1, p2])\\ &\ldots\end{aligned}$$

Note that several instances of the same task may be released. However, the number of copies may be bounded by the clock constraints. For example, in state $(n, [x = 4, y = 1], [p1, p1, p1])$, no more instance of $p1$ will be released because the clock values will not satisfy the constraint $x < 4$ and $y > 2$, but an instance of $p2$ may be released by the b-transition (which has no timing constraint).

In the above example, we have only shown that the task queue is growing due to action transitions. Now we discuss the effect of delay transitions on task queue. We shall see that the queue will be shrinking due to delay transitions. Let $p1 = p2 = (2, 8)$ i.e. the computation time of both $p1$ and $p2$ is 2 and the deadline is 8. We assume that there is a processor running the task instances according to a certain scheduling strategy. A delay transition with t time units is to execute the tasks in the queue with t time units. After the transition, a task will be removed from the queue (shrinking) if its computation time becomes 0 and the deadlines of all tasks in the queue will be decreased by t (since time has progressed by t). Now we have a precise description on the state changes for the above transition sequence:

$$\begin{aligned}(m, [x = 0, x = 0], []) &\xrightarrow{3} (m, [x = 3, y = 3], [])\\ &\xrightarrow{a} (n, [x = 0, y = 3], [(2, 8)])\\ &\xrightarrow{a} (n, [x = 0, y = 0], [(2, 8), (2, 8)])\\ &\xrightarrow{3} (n, [x = 3, y = 3], [(1, 5)])\\ &\xrightarrow{a} (n, [x = 3, y = 0], [(1, 5), (2, 8)])\\ &\xrightarrow{1} (n, [x = 4, y = 1], [(2, 7)])\\ &\xrightarrow{b} (n, [x = 4, y = 1], [(2, 7), (2, 8)])\\ &\ldots\end{aligned}$$

More precisely we have the following assumptions on the underlining execution model:

1. A ready queue holding the task instances released and waiting for execution. A task instance will be removed from the queue when its computation time becomes 0.
2. An on-line scheduler Sch sorting the queue according to a given scheduling strategy. It will report $\perp$ if the queue becomes non-schedulable when a new task instance is added.

3. A single processor executing the tasks according to the ordering of the queue. It will always execute the task in the first position. The tasks are executed non-preemtive.

Further we use $\mathrm{Run}(q, t)$ to denote the resulted task queue after t time units of execution. The meaning of $\mathrm{Run}(q, t)$ should be obvious. For example, let $q = [(2, 7), (2, 8)]$ and $t = 3$ then $\mathrm{Run}(q, t) = [(1, 5)]$ in which the first task is finished and the second has been executed for 1 time unit. Now we are ready to present the transitional rules for extended timed automata.

Definition 3. *The semantics of an extended automaton is a transition system defined by the following transition rules (corresponding to release of new task and execution of existing tasks):*

- $(l, u, q) \xrightarrow{a} (l', u', \mathrm{Sch}(q'))$ *if* $l \xrightarrow{g,a,r} l'$*,* $u \in g$*,* $u' = u[r \mapsto 0]$*, and* $q' = q :: T(a)$
- $(l, u, q) \xrightarrow{t} (l, u + t, \mathrm{Run}(q, t))$

We shall write $(l, u, q) \longrightarrow (l', u', q')$ *if* $(l, u, q) \xrightarrow{a} (l', u', q')$ *for an action* a *or* $(l, u, q) \xrightarrow{d} (l', u', q')$ *for a delay* d*.* □

Finally, to handle concurrency and synchronization, parallel composition of extended timed automata may be introduced in the same way as for ordinary timed automata (e.g. see [10]) using the notion of synchronization function [8]. For example, consider the parallel composition $A||B$ of A and B over the same set of actions Act. The set of nodes of $A||B$ is simply the product of A's and B's nodes, the set of clocks is the (disjoint) union of A's and B's clocks, the edges are based on synchronizable A's and B's edges with enabling conditions conjuncted and reset-sets unioned. Note that due to the notion of synchronization function [8], the action set of the parallel composition will be Act and thus the task assignment function for $A||B$ is the same as for A and B.

3. Schedulability Analysis as Reachability Analysis

Traditionally, the temporal attributes for a real-time computer systems are derived from their environment, e.g. period times, etc. These attributes are used for constructing a model of the system in terms of its temporal behavior. Such a temporal model is often called a task model, which is used to verify whether the system is schedulable or not, but other properties such as functional and safety properties can not be verified based on such a model. In our approach, we may construct a model for the whole system including the environment and tasks in the control system. The parallel composition of these models give us the possibility of not only verifying temporal constraints, but also its other aspects such as synchronization between tasks and simple computations within tasks etc.

Normally, a system is said to be schedulable if all tasks can always be executed within their deadlines, i.e. no deadlines are violated. The objective of the schedulability analysis is to verify that there are no violation of deadlines in all situations where the system may evolve to. Now we formalize the notion of schedulability for extended timed automata.

Definition 4. *An extended timed automaton* A *is non-schedulable if it may reach a non-schedulable state, that is:* $(l_0, u_0, q_0) \longrightarrow^* (l, u, \perp)$ *where* (l_0, u_0, q_0) *is the initial state of* A*, and* $\longrightarrow^*$ *is the transitive closure of* $\longrightarrow$*. We say that* A *is schedulable if and only if all its reachable states are schedulable.* □

Thus, the schedulability of extended automata can be checked by reachability analysis, to prove that $(l, u, \perp)$ is not reachable in the automaton. However, it is not obvious that the reachability problem for extended automata is decidable. In fact, the decidability of this problem is closely related to the preemptiveness of the tasks P. The following is one of our main results in this paper.

Theorem 1. *The problem of checking schedulability for extended timed automata over non-preemptive tasks* P *is decidable.*

Proof idea: It is based on the fact that the problem of schedulability checking for extended timed automata can be transformed to the reachability problem for standard timed automata, which is known to be decidable [1]. See the following subsection for details on the transformation. □

3.1. Transformation from TAT to ordinary timed automata

The idea is to construct a timed automaton simulating a ready queue and a scheduler that code all possible scenarios of the system described by a TAT, including the tasks in the queue and schedules. For example, consider the temporal attributes of the two tasks p_a and task p_b, where p_a had a worst-case-execution time (wcet), of 4 time units (tu), and a deadline (d), of 7 tu. The second task p_b has a wcet of 3 tu and a deadline of 5 tu.

Intuitively for a system to be schedulable, the ready queue can contain only a finite number of task instances. More precisely, there can only be MNT_i instances of task i, where MNT_i is given by:

$$MNT_i = \left\lfloor \frac{d_i}{c_i} \right\rfloor$$

where d_i denotes the deadline for task i and c_i denotes the computation time.

By calculating the maximum length of the ready queue, we know that to be schedulable, the queue in our example can only contain one instance of p_a and one instance of p_b. If at any time point, there are more than one instances of a particular task in the ready queue waiting for execution, we know for sure that the system is non-schedulable and the error state should be reached. This ensures a finite number of states in our model of the scheduler and the ready queue. Now, we use the above example to present the algorithm for constructing the scheduler and queue automaton, which can be generalized easily to the general case.

1. Create three different nodes, one node in which the ready queue is empty, one for which there exists task instances in the ready queue and, finally an error node.

2. Create transitions from the empty node to the running node, one for every action associated with a task. Furthermore, tasks can arrive while in the run node, consequently we need one transition from run back to run for every possible task instance as well. In order to keep track of every new task instance, a unique semaphore for every instance is introduced (denoted as taska and taskb in Figure 2). We also need a unique deadline clock for every instance in order to know which task to execute and to detect deadline violations.

3. According to EDF, the task having least time left until its deadline should be executed. For all possible task instances, create a transition from run to run which compares its relative deadline to all the other ready tasks. In our example p_a should be executing if $7 - d_a < 5 - d_b$, and p_b if $5 - d_b < 7 - d_a$ where d_a and d_b are the deadline clocks. In order to keep track of execution time of the running task, a clock is reseted on every release of a task. In our example, this clock is denoted as c. Furthermore, as we consider the non-preemptive case, no task can start to execute while another task already is executing. Thus we need a semaphore to know whether the processor is idle or not (denoted *r* in Figure 2).

4. Introduce one transition from run to run for every possible instance which terminates the task whenever c becomes equal to its specified execution time and its deadline clock is less or equal to its specified deadline. Termination is modeled by resetting the instance semaphore.

5. If ready queue gets empty, i.e. no tasks instances are present in the queue a transition to the empty node should be taken.

6. For each possible task instance we introduce a transition from run to error if:

   - An action A occurs, making the number of instances of A exceeding MNT_a
   - The executing task has overrun its deadline
   - A task pending for execution in the ready queue has exceeded its deadline

Figure 2 shows the result from transforming our example system shown. This is an ordinary timed automata for which decidability has been proven in [1].

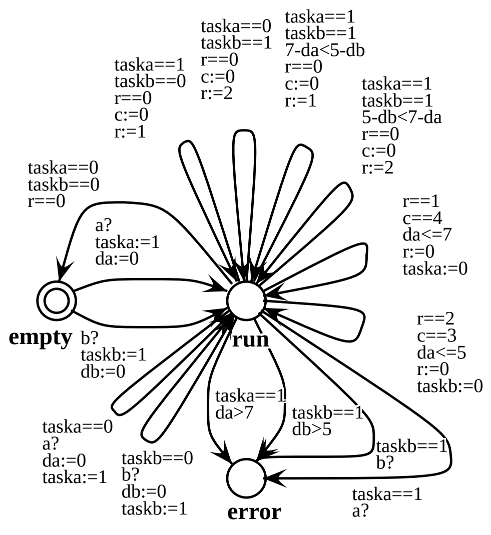

Figure 2. A model of the ready queue and the scheduler using ordinary timed automata

For the general case, the scheduler and queue automata is illustrated in Figure 3 where q denotes a queue, r is the executing task, c measures how long time the executing task has been running and d(i) is a vector keeping track of the time elapsed since the tasks entered the ready queue. C(i) is a vector holding the worst case execution time of all tasks. Both are vectors are finite as been discussed above. Moreover, the function sch() returns the instance among all tasks residing in the queue having least time left until its deadline. Task *i* is returned by sch() if the predicate $\bigwedge_{\forall m \in q : m \neq i} d(i) - d(m) \leq D_i - D_m$ is true, where D_i denotes the relative deadline specified for task *i*.

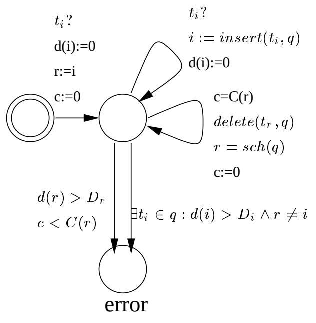

Figure 3. A general model of the scheduler using ordinary timed automata

4. A Case Study with UPPAAL

UPPAAL is a model-checker for timed automata [9]. As shown in the previous section, the scheduler and ready queue can be modeled as an ordinary timed automaton. In this section, we present an example showing how to use UPPAAL for schedulability checking.

Our example system is a event-driven application controlling the speed of the shaft in a turning lathe. The objectives of the formal verification is to verify that the system is schedulable and the safety requirement that the engine is not turned on by the control task while the emergency stop is active. An event reports the current speed of the shaft and a control task is checking that the speed is within the speed limits (in our example speed=3). If the speed is to high (over 3), the engine is turned off and if the speed is to low (below 3), the engine is turned on. There is also an emergency stop function which is implemented in software. The setup is shown in Figure 4.

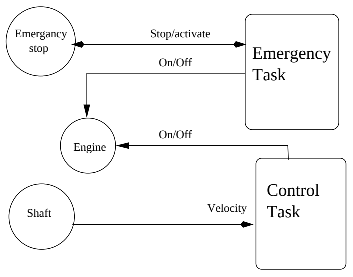

Figure 4. The setup for our example system

As shown in Figure 4, the parts belonging to the systems environment are the shaft having an optical sensor generating an event on every complete revolution, the emergency stop button having two states: up or down and the engine, being either on or off. Consequently, we have to model all these parts as a network of TATs. Moreover, we have two software tasks, the control task and the emergency stop handler. These parts also have to be modeled in TATs belonging to the network constituting the complete system.

4.1. Modeling the system

We start by modeling the environment, i.e. the shaft, the emergency stop button and the engine. This can for instance be done as shown in Figure 5, 6.

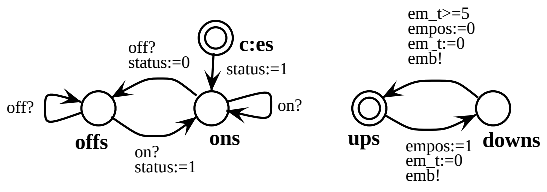

Figure 6. A model of the engine and the emergency stop button

If the engine is on, the shaft makes a complete revolution in between 4-8 time units, and an event is generated every time the optical sensor detects a complete revolution.

Next to model is the emergency stop handler and the control task. The control task has a calculated wcet of 2 tu and a hard deadline of 3 tu (Figure 8).

As for the control task, a deadline and a wcet must be specified for the emergency stop handler. According to our imagined requirement specification, it must respond within 2 tu, i.e. it has a deadline at 2 tu. The wcet estimation result in a wcet of 1 tu (see Figure 7). Furthermore, two subsequent activations/deactivation of the emergency stop can not be less than 5 tu in between. This gives us a minimum interarrival time for the emergency stop handler of 5 tu.

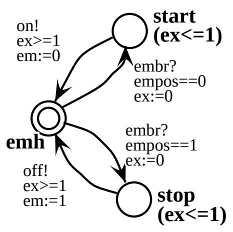

Figure 7. A model of the emergency handler in timed automata

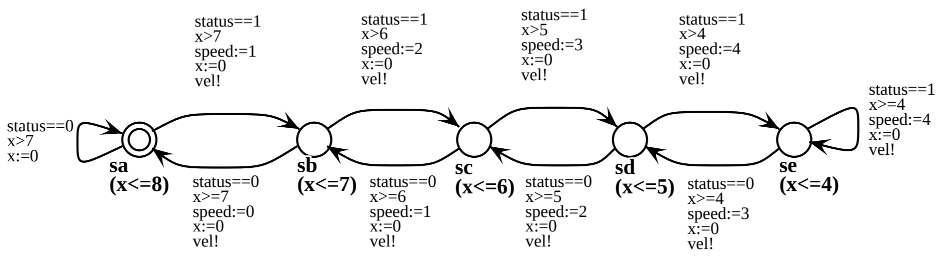

Figure 5. A model of the shaft in timed automata

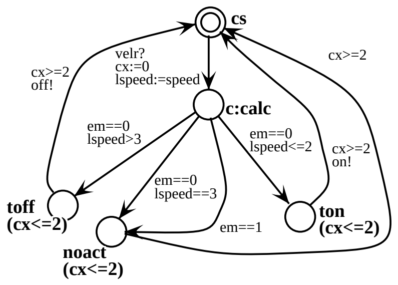

Figure 8. A model of the control task in timed automata

The model of the scheduler is omitted in the paper. However, this process will be generated automatically by UPPAAL according to the algorithm given in Section 3.1 and will be invisible for the designer.

4.2. Verifying schedulability and safety

We use model checking and reachability analysis on our network of TAT for this purpose. UPPAAL uses a timed CTL language for specifying properties to verify. To verify that the system is schedulable, we must show that the error state is never reachable. We will use the *always* predicate in our example as *always* not α is equivalent to *never*. This property is specified as shown in the formula below, *scheduler.error* means the state error in the process named scheduler:

$$\forall\Box not\ scheduler.error$$

For the safety property we need to verify that the system never reach a state where the control task is in position to turn the engine on while the emergency stop has been activated. For our model, such an expression looks like the formula given below:

$$\forall\Box not(control\_task.ton\ and\ em = 1)$$

First we will verify the schedulability property. As a result UPPAAL tells us that the property is not satisfied by giving a counter example. Consequently, the system is not schedulable. In order to obtain a schedulable system, the temporal constraints on the tasks have to be modified. The counter example given by UPPAAL, shows that the emergency handler task misses its deadline if this event happens just after the control task has been invoked. By changing the deadlines for the control task and the emergency stop handler to 4 tu, the system becomes schedulable. This is verified by the same property, but with an updated scheduler model. The model of the scheduler must be updated since now there can exist two instances of the control task and four instances of the emergency handler simultaneously in the ready queue.

Next to verify is our safety property, i.e. the control task should not be able to turn the engine on as long as the emergency stop is activated. In this case UPPAAL reports that the property is satisfied and consequently, the safety requirement is fulfilled.

It is of course possible to verify other functional properties. For instance, we can verify that the shaft eventually will rotate with the set value. In our model, the set value is the speed of 3, i.e. *the speed is eventually equal to 3*. The corresponding formula given in UPPAAL logic is:

$$\exists\Diamond speed = 3$$

5. Conclusions

An important step in the development of embedded real-time systems is ”schedulability analysis” that is to check whether all tasks in a system can be executed within the given deadlines in all possible scenarios. The traditional approach to schedulability analysis is often based on scheduling theory and a task model, which has been very successful for periodic tasks, but less successful for event-driven tasks.

In this paper, we have developed an extended version of timed automata with real-time tasks to provide a model for event-driven systems, which can be used for modeling, schedulability analysis, formal verification, and code generation. The main idea is to associate each discrete transition in a timed automaton with a task (an executable program e.g. written in C) with its worst case execution time. Intuitively, a discrete transition in an extended timed automaton denotes an event releasing a task and the guard on the transition specifies all the possible arriving times of the event (instead of the so–called minimal inter-arrival time). This yields a general model for hard real-time systems in which tasks are non-periodic. In this model, an automaton is used to model control structure of a systems and associated tasks are used to perform computation. Thus, code generation for such a model is reduced to transform the automaton into a runnable program with procedure-call. However, a critical problem is to guarantee that all the tasks associated with the automaton can be executed within their deadlines. This is the so-called schedulability checking problem. As the main result of this paper, we have shown that the schedulability checking problem for the extended timed automata with real time tasks can be transformed to a reachability problem for standard timed automata and thus it is decidable. This result allows us to apply model-checking tools for timed automata to schedulability analysis for event-driven systems. In addition, based on the same model of a system, we may use the tools to verify other properties (e.g. safety and functionality) of the system. This unifies schedulability analysis and formal verification in one framework.

As future work, we plan to extend the UPPAAL model checker for schedulability analysis. Future work also include code generation which is to translate extended timed automata with tasks into executable programs.

References

[1] R. Alur. Model-checking in dense real-time. *Information and computing*, 1993.

[2] R. Alur and D. Dill. Automata for modelling real-time systems. In *Proceedings of ICALP'90*, volume 443 of *Lecture Notes in Computer Science*. Springer, 1990.

[3] Bengtsson, Griffioen, Kristoffersen, Larsen, L. an d Pettersson, and Yi. Verification of an audio protocol with bus collision using uppaal. In *Proceedings of CAV'96*, volume 1102, 1996.

[4] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL in 1995. In *Proc. of the 2*nd *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1055 in Lecture Notes in Computer Science, pages 431–434. Springer–Verlag, Mar. 1996.

[5] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997.

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

[7] M. L. Dertouzos. Control robotics: The procedural control of physical processes. *Information Processing*, 1974.

[8] H. Hüttel and K. G. Larsen. The use of static constructs in a modal process logic. In *Logic at Botik'89*, number 363, pages 163–180. Springer–Verlag, 1989.

[9] K. G. Larsen, P. Pattersson, and Y. Wang. UPPAAL in a nutshell. *Springer International Journal of Software Tools for Technology Transfer*, 1, 1997.

[10] K. G. Larsen, P. Petterson, and Y. Wang. Compositional and symbolic model-checking of real-time systems. In *Proceedings of the 16th Real-Time Systems Symposium*, pages 76–87. IEEE Computer Society Press, 1995.

[11] M. Lindahl, P. Pettersson, and W. Yi. Formal design and analysis of a gear controller. *Lecture Notes in Computer Science*, 1384:281–297, 1998.

[12] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the Association for Computing Machinery*, 2, 1973.

[13] H. Lönn, P. Pettersson, and W. Yi. Formal Verification of a TDMA Protocol Start-Up Mechanism. In *Proceedings of 1997 IEEE Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 235–242, 1997.