Adaptive Task Automata: A Framework for Verifying Adaptive Embedded Systems

Leo Hatvani, Paul Pettersson, and Cristina Seceleanu

Mälardalen University, 721 23, Västerås, Sweden
{leo.hatvani,paul.pettersson,cristina.seceleanu}@mdh.se

Abstract. We present a framework for modeling and analysis of adaptive embedded systems, based on the model of timed automata with tasks. The model is extended with primitives allowing modeling of adaptivity, by testing the potential schedulability of a given task, in the context of the set of currently enqueued tasks. This makes it possible to describe adaptive embedded systems, in which decisions to admit further tasks or take other measures of adaptivity is based on available CPU resources, external, or internal events. We show that this model can be encoded in the framework of timed automata, and hence that the problem is decidable. We also validate the framework, by using the UPPAAL tool.

1 Introduction

Adaptive embedded systems are embedded systems that must be capable of dynamic reconfiguration, to adapt to e.g., changes in available resources, user- or application-driven mode changes, or modified quality of service requirements. The possibility to adapt provides flexibility that extends the area of operation of embedded systems and potentially reduces resource consumption, but also poses challenges in many aspects of systems development, including system modeling, scheduling, and analysis.

In embedded systems, tasks are usually assumed to execute periodically according to classical real-time scheduling methods, such as rate monotonic scheduling, other fixed priorities, earliest deadline first, or first-in first-out [5]. For systems with non-periodic tasks or non-deterministic task behaviors fewer general results exist. Automata models have been proposed to relax some of the assumptions on the arrival patterns of tasks. In the model of *task automata* (or timed automata with tasks) [8,10], the release patterns of tasks are modeled using *timed automata* [1], such that a set of tasks with known parameters is released at the time point an automaton location is reached. It has been shown that the corresponding schedulability problem for this bigger class of possible release patterns is decidable, i.e., the problem of checking if, for all possible traces of a task automata, the tasks released are schedulable (or not), assuming a given scheduling policy. It has also been shown how to generate code from task automata, such that a modeled system can be realized on a hardware platform running e.g., WxWorks [3,4]. The theory is implemented in the TIMES tool [2].

On the another hand, many results exist for formal verification of adaptive embedded system models specified in high level languages such as UML Statecharts, as enumerated by Schaefer [13]. Another set of results describes application of formal verification of schedulability to: multiprocessor systems [14], satellite systems [11], or providing generalized frameworks for schedulability analysis [7]. All of these studies have one thing in common: the non-schedulability of the system can be determined only after a task misses its deadline, and thus the information is not present soon enough, such that it can be used to avoid entering such state.

In this work, we propose a framework for modeling and analysis of *adaptive real-time embedded systems*, based on the model of task automata, and assuming a single CPU preemptive environment. We extend the model with primitives allowing modeling of adaptivity based on the schedulability of the set of currently released tasks (i.e., the ready queue), if further tasks are released. In particular, we propose to add a schedulability predicate that can be used as a conjunct of a timed automaton guard. The predicate evaluates to true at a given time point, iff the current ready queue, extended with zero or more specified tasks, is schedulable with a given scheduling policy. This allows for modeling of e.g., adaptive embedded systems in which decisions to admit further tasks are based on available CPU resources, or systems in which tasks with high quality of service can occasionally be replaced with alternative lower quality tasks, when the CPU load is too high.

	P	T	D	C
t_0	3	10	10	5
t_1	2	15	6	4
t_1'	1	–	6	1

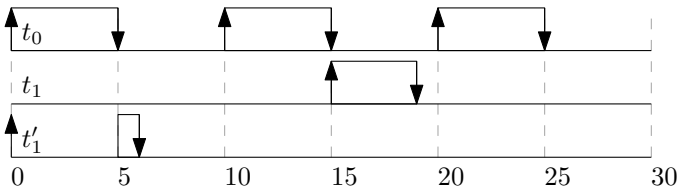

Fig. 1. A trace of a task set with adaptable task t_1.

As a small example of the proposed model, consider a system with two tasks t_0, t_1, and t_1', where t_1' is a version of t_1 with lower quality of service, which requires less CPU time. The task parameters are given in Fig. 1: P is priority, T is period, D is deadline, and C is computation time. Since $P_0 > P_1 > P_1'$, task t_0 will be executed periodically without being preempted. We assume t_1 will be admitted only if it has a chance to complete before deadline, otherwise t_1' is released. The system is schedulable, and will release t_0 every 10 ms, and will try to release t_1 every 15ms. If t_1 cannot be released at that time point, due to interference from t_0, task t_1' will be released. Modeled in our extended task automata model, we can check schedulability, verify how many times out of k task t_1' replaces t_1, and interpret a simulated trace as static cyclic scheduler for the system.

As our main result, we show that the schedulability problem and other reachability properties of the proposed model are decidable for fixed priority scheduling policies. Our encoding of the problem is based on previous results of Fersman

et.al. [8,10], in which it is shown how given task automata can be encoded and analyzed as a network of timed automata. However, in comparison to the previous work, our type of adaptive systems cannot rely completely on encoding the scheduler and explore the state space to check if the system is schedulable or not. Instead, we need to check in advance if a system is schedulable, or will be schedulable with the potential release of one or several additional tasks.

The rest of this paper is organized as follows: in the next section, we describe preliminaries, in Section 3 adaptive scheduling policies encompassed by the model, and in Section 4 our main result, the encoding. In Section 5, we give some examples, and conclude the paper in Section 6.

2 Preliminaries: Task Automata

Our model of *adaptive task automata* is based on the model of *task automata* (or *timed automata with tasks*) [8,10,12], which extends the model of timed automata with a notion of tasks. A *timed automata* [1] is simply a finite state automata extended with a finite set of real-valued clocks. The edges of timed automata are labeled with Boolean combinations of simple clock constraints, events, and a *reset set* of clocks, specifying a subset of the clocks to be reset when the edge is taken. In the model of task automata, the idea is to associate each location of a timed automaton with a an executable program, called *task*, which is assumed to be released when the location is reached. Each task is assumed to be associated with given parameters such as execution time, hard deadline, priority, etc. It is possible to interpret a task automaton as an abstract model of a running system, in which the underlying timed automata describes the time points at which possible events occur, and the location-associated tasks, triggered by the occurring event.

Syntax. Let $\mathcal{T}$ ranged over by $t_0, \ldots, t_n$ denote a finite set of task types. Each task type may have different instances over time, however, we will assume, without lack of generality, that at each time point there is at most one instance of each task type released. Each task type is associated with a a triple of natural numbers $t_i(C_i, D_i, P_i)$, where C_i is the task's computation time, D_i its relative deadline (relative from the release time point), and P_i its priority. Further, let *Act* ranged over by *a, b* etc, denote the set of action labels, and $\mathcal{C}$ ranged over by $x_0, \ldots, x_n$ the finite set of real-valued clocks. We use $\mathcal{B}(\mathcal{C})$ ranged over by g to denote the set of conjunctive formulas of constraints, called clock constraints, of the form $x_i \sim n$ and $x_i - x_j \sim m$, where $\sim \in \{\leq, <, >, \geq\}$, and n and m are natural numbers.

Definition 1. *[10] A* task automaton *over* Act, $\mathcal{C}$, *and* $\mathcal{T}$ *is a tuple* $\langle L, l_0, E, I, M \rangle$, *where* L *is a set of location ranged over by* $l_0, \ldots l_n$, $l_0 \in L$ *is the initial location,* $E \subseteq L \times \mathcal{B}(\mathcal{C}) \times \text{Act} \times 2^{\mathcal{C}} \times L$ *is the set of edges,* $I : L \mapsto \mathcal{B}(\mathcal{C})$ *is a function assigning each location with a location invariant, and* $M : L \hookrightarrow \mathcal{T}$ *is a partial function assigning locations with tasks.* □

Semantics. Like in standard timed automata, a task automaton may perform two types of actions. A *delay transition* corresponds to progression of time and execution of the released task with the highest priority, and idling lower priority tasks waiting to run. An *action transitions* corresponds to taking an enabled edge (one whose guard evaluates to true given the current clock values), and (possibly) releasing a task associated with the location reached.

A state of a task automaton is a triple $\langle l, u, q\rangle$, where l is the current control location, $u : \mathcal{C} \mapsto \mathbb{R}_{\geq 0}$ is a function mapping clocks to non-negative real values, and q is the current ready queue of tasks. The task queue is formed as: $[t_i(c_i, d_i), \ldots, t_j(c_j, d_j)]$, where t_i is the task type, c_i is the remaining computation time, and d_i the relative deadline. A scheduling function, such as fixed priority or earliest deadline first, is a function Sch sorting the task queue w.r.t. the task parameters. For instance, $[t_1(1,2), t_2(2.5,4)]$ is sorted according to fixed priority, if $P_1 > P_2$. Note that a scheduling policy can be either preemptive or non-preemptive, depending on whether the first queue position can be changed (preemptive) or not (non-preemptive).

To define the semantics, we also need a function $\mathsf{Run}_{\mathsf{Sch}}$ that takes a task queue q and a non-negative real-number δ, and returns the result of executing q for δ time units, with the given scheduling function Sch (e.g., $\mathsf{Run}_{\mathsf{FPS}}([t_1(1,2), t_2(2.5,4)], 2) = [t_2(1.5,2)]$, for a fixed priority scheduling fuction $\mathsf{Run}_{\mathsf{FPS}}$).

Definition 2. *[10] Given a task automata $\langle L, l_0, E, I, M\rangle$ with an initial state $\langle l_0, u_0, q_0\rangle$, and a scheduling strategy Sch, the semantics is a transition system defined as:*

- $\langle l, u, q\rangle \xrightarrow{a}_{\mathsf{Sch}} \langle l', r(u), \mathsf{Sch}(M(l') :: q)\rangle$ *if* $l \xrightarrow{g,a,r} l' \in E$ *and* $u \models g$
- $\langle l, u, q\rangle \xrightarrow{\delta}_{\mathsf{Sch}} \langle l, u \oplus \delta, \mathsf{Run}_{\mathsf{Sch}}(q, \delta)\rangle$ *if* $(u \oplus \delta) \models I(l)$

where $r(u)$ is 0 for all $x_i \in r$ and $u(x_i)$ otherwise, $t :: q$ is the result of merging t with q, and $u \oplus \delta$ is the result of adding δ to all clock values in u. □

Schedulability. Verification problems of the above model, with non-preemptive and preemptive tasks, have been already investigated in [10,12]. Here we briefly review the notions of reachability and schedulability. A state $\langle l, u, q\rangle$ is *reachable* with a given scheduling policy Sch, if $\langle l_0, u_0, q_0\rangle$ $(\longrightarrow_{\mathsf{Sch}})^* \langle l, u, q\rangle$, where $\longrightarrow_{\mathsf{Sch}}$ is $\xrightarrow{a}_{\mathsf{Sch}}$ or $\xrightarrow{\delta}_{\mathsf{Sch}}$. Further, a state $\langle l, u, q\rangle$ with $q = [t_0(c_0, d_0), \ldots, t_n(c_n, d_n)]$ is defined as *deadline-missed*, if there is some $0 \leq i \leq n$ such that $c_i > 0$ and $d_i \leq 0$. A task automaton A is defined to be *non-schedulable with* Sch iff a deadline-missed state is reachable with Sch. Otherwise, A is considered to be *schedulable with* Sch. In general, A is said to be *schedulable* if it is schedulable with some scheduling strategy Sch. The problem of checking schedulability of task automata with preemptive tasks is proven to be decidable in [10].

3 Adaptive Task Automata

In this section, we describe the model of *adaptive task automata*, which extends the model of timed automata for adaptivity. Our aim is to enable modeling

of adaptivity based on the schedulability of the set of currently released tasks, and the effect of potentially releasing additional tasks for execution. In terms of modeling, the extension consists of a set of predicates for schedulability test, which can be used in conjunction with other guards on edges of task automata. As a main result of this paper, we will also show how the resulting model can be encoded as timed automata, and hence, that reachability and schedulability checking are decidable.

Definition 3. *Given a task automaton state $\langle l, u, q\rangle$, with $q = [t_0(c_0, d_0), \ldots, t_n(c_n, d_n)]$, and two distinct tasks, t_i and t_j, let $\mathcal{P}$ be the set of predicates $\{inqueue/1, sched/1, sched/2\}$ satisfied as follows:*

$\langle l, u, q\rangle \models inqueue(t_i)$ *if* $t_i \in q$

$\langle l, u, q\rangle \models sched(t_i)$ *if* $(\sum_{j=0}^{i} c_j) \leq d_i \wedge inqueue(t_i) \vee \langle l, u, \mathsf{Sch}(t_i :: q)\rangle \models sched(t_i) \wedge \neg inqueue(t_i)$

$\langle l, u, q\rangle \models sched(t_i, t_j)$ *if* $inqueue(t_i) \wedge \langle l, u, \mathsf{Sch}(t_j :: q)\rangle \models sched(t_i)$

□

We say that t_i is *active* in state $\langle l, u, q\rangle$ if $\langle l, u, q\rangle \models inqueue(t_i)$. In the rest of the paper, we will omit $\langle l, u, q\rangle$ if the context is obvious. Intuitively, $sched(t_i)$ is true in a state, if t_i will meet its deadline, given that q is executed according to Sch. We say that t_i is *schedulable* if $sched(t_i)$. Similarly, $sched(t_i, t_j)$ is true in a state, if t_i is schedulable even if t_j is released (added to q).

We now define the model of adaptive task automata. Let $\mathcal{B}(\mathcal{P} \cup \mathcal{C})$ denote the set of conjunctive formulas of clock constraints in $\mathcal{B}(\mathcal{C})$, and predicates in $\mathcal{P}$.

Definition 4 (Adaptive task automata). *An* adaptive task automaton *over* Act, $\mathcal{C}$, *and* $\mathcal{T}$ *is a tuple* $\langle L, l_0, E', I, M\rangle$, *where* L, l_0, I, M *are defined as in task automata in Definition 1. The set of edges is defined as:* $E' \subseteq L \times \mathcal{B}(\mathcal{P} \cup \mathcal{C}) \times \text{Act} \times 2^{\mathcal{C}} \times L$. □

Hence, the set of guards of the edges is extended to conjunctions of clock constraints and the predicates of Definition 3.

Example 1. The adaptive task automaton shown in Fig. 2 describes the release pattern of the task t_1 and corresponding backup task t_1' from Fig. 1. The automaton consists of a clock x, and three states: Start, Release t_1, and Release t_1'. The edge from state Start to the states releasing tasks t_1 or t_1' is immediate, given the invariant $x \leq 0$ of state Start. The choice of the next state is regulated by the evaluation of the respective guards on the edges, $sched(t_1)$ or $sched(t_1') \wedge \neg sched(t_1)$, respectively. Once one of the Release $\{t_1, t_1'\}$ states is entered, the corresponding task is released, and the automaton spends the rest of the period in that state, before returning to start and resetting the clock x. Note that a third edge from Start to an error location, taken in case when none of the alternatives can be released, has been omitted from the figure for simplicity.

Derived predicates. The predicates defined above can be used to derive several other useful predicates, including:

- $sched\_all = (\bigwedge_i inqueue(t_i) \Rightarrow sched(t_i))$,

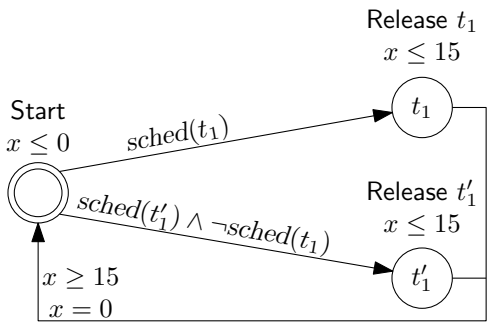

Fig. 2. Adaptive task automata for the task t_1 from the Example 1.

– $sched\_all(t_i) = (\bigwedge_j inqueue(t_j) \Rightarrow sched(t_j, t_i))$.

The predicate *sched_all* evaluates to *true*, in case all tasks in the queue are schedulable, assuming scheduling policy Sch. The second predicate holds if all the tasks in the queue are schedulable, if task t_i is released. We will make use of the above derived predicates in an example presented in Section 5.

4 Encoding of the Adaptive Task Automata

In this section, we present an encoding of the task release automata, the scheduler, and the task queue, as timed automata models. The encoding is presented in terms of the variables that are used to model the execution of tasks. Based on these variables, the predicate *sched*() is encoded, and finally, an encoding of a fixed priority scheduler is presented.

Modeling a task set execution in timed automata requires tracking of several values for each executed task instance. To establish if a task has executed in time, we keep track of the amount of time that the task has been executing, and the amount of time that has passed since the task has been released. By using these values, and comparing them to the computation times and relative deadlines of the tasks, we can establish if a task is able to complete successfully, or not.

Our encoding is based on, and combines ideas introduced by Fersman et al. [8,9]. The following variables are used for each task t_i:

– c_i - a clock that resets every time the predicate $(\exists t_j \mid inqueue(t_j) \wedge P_j \geq P_i)$ changes value from false to true, where P_i and P_j are priorities of tasks t_i and t_j respectively;
– d_i - a clock reset when the task t_i is released;
– r_i - an integer variable (of bounded domain) that contains a sum of the computation times C_i of all tasks of higher or equal priority to task t_i, which have been released since c_i has been last reset.

The use of these variables will be exemplified on the scenario illustrated in Fig. 3. Four task instances are released: t_1 (at time point 4), t_2 (at time point 1), and

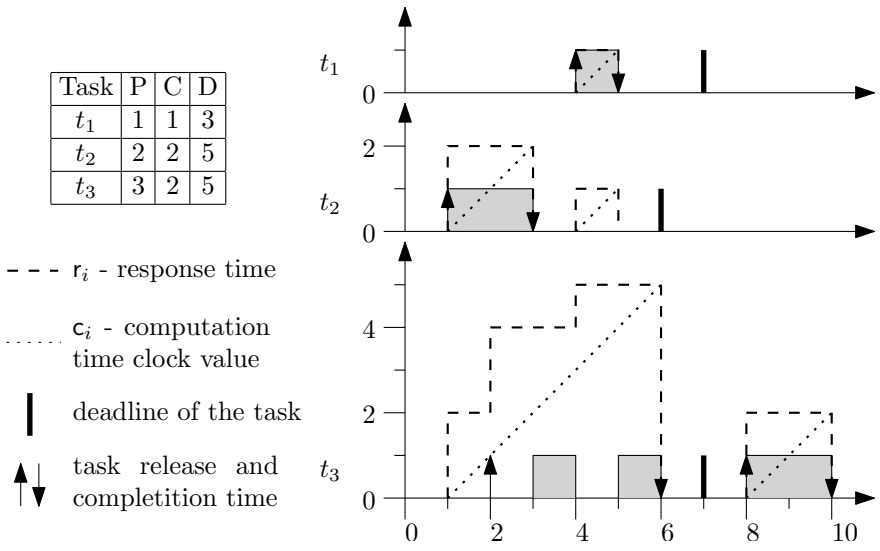

Fig. 3. Tracking of essential variables for each task.

t_3 (at time points 2 and 8). The task parameters and the values of variables r_i, and clocks c_i, over time, are also given in the figure. Clocks d_i are left out for clarity, but the point where they would become equal to the corresponding value D_i is marked with thick vertical bars.

The variables and clocks of all tasks are reset at the release of the first task t_2, at time point 1. As t_2 is released, its computation time (2) is added to all the r_i of tasks with lower or equal priority to t_2, i.e., $\mathsf{r}_2 = \mathsf{r}_2 + 2 = 2$ and $\mathsf{r}_3 = \mathsf{r}_3 + 2 = 2$.

A task completes its execution when $\mathsf{c}_i = \mathsf{r}_i$. In our case, this happens first at time point 3, when $\mathsf{r}_2 = \mathsf{c}_2$. However, before this, task t_3 is released at time point 2, so r_3 is increased by 2, the computation time of task t_3. The only clock reset at this time is d_3, to start measuring time until its relative deadline.

At time point 4, task t_1 is released, causing the reset of all its variables, and those of task t_2 (according to how c_i is updated). Variables r_1, r_2, and r_3 are increased by 1 (the computation time of task t_3), to 1, 1, and 5, respectively.

We now focus on task t_3. Observe that the difference $\mathsf{r}_3 - \mathsf{c}_3$ for task t_3 represents the time left until t_3 completes its execution (assuming no higher priority task is released). The time left to its deadline is given by $D_3 - \mathsf{d}_3$. Comparing the two values, we get the amount of time that the task can be delayed without missing its deadline, and hence, as long as the inequality holds, the task will meet its deadline. The values are illustrated in Fig. 4. In fact, at time x, there is enough time to execute a higher priority task for 2 time units, since $\mathsf{r}_3 - \mathsf{c}_3 + 2 \leq D_3 - \mathsf{d}_3$. When task t_1 is later released, we already know that task t_3 can finish at time 6, i.e., 1 time unit before its deadline.

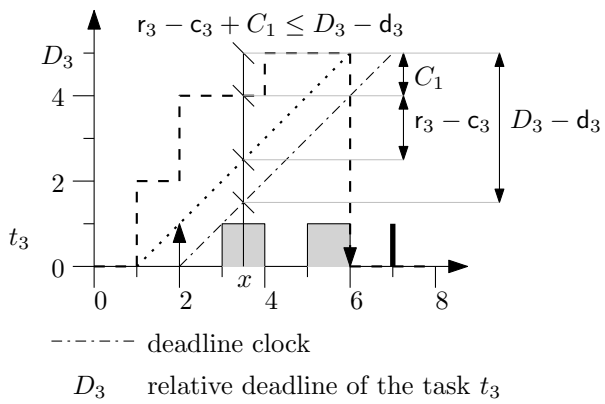

Fig. 4. Visual explanation of the schedulability predicate encoding.

4.1 Encoding the predicate *sched*()

Given the variables introduced above, and given that there is a possible scheduler model (introduced in the next section), we encode the predicate *sched*() as follows:

$$sched(t_i) = \begin{cases} \mathsf{r}_i - \mathsf{c}_i \leq D_i - \mathsf{d}_i & \text{if } inqueue(t_i) \\ \mathsf{r}_i - \mathsf{c}_i + C_i \leq D_i - \mathsf{d}_i & \text{if } \neg inqueue(t_i) \wedge P_{run} > P_i \\ C_i \leq D_i & \text{if } \neg inqueue(t_i) \wedge P_{run} < P_i \end{cases}$$

where t_{run} refers to the currently executing task.

The first case has been explained in the previous section, note that it covers all cases where $t_i = t_{run}$, since $inqueue(t_{run})$ is invariantly true. In case the task of interest (t_i) has not been released yet, its computation time is not included in the expression $\mathsf{r}_i - \mathsf{c}_i \leq D_i - \mathsf{d}_i$, so this gives rise to the second case. In case the task is not yet released, and it has higher priority than the currently running task, it will execute immediately, and its schedulability is then only depending on computation time being shorter than the deadline, hence the third case. This case cannot be covered by the second case, since the clocks are considered inactive at this point, and can only be reset and not read.

The implementation of the scheduler requires a strict ordering between the tasks. We have introduced that ordering by assuming unique task priorities. Together with the requirement of single task instance per task, this makes $P_i = P_j$ lead to an error state, and it is therefore not considered.

The derivation of the schedulability predicate that tests the schedulability of task t_i, based on the release of task t_j, can be done from the second case above, by replacing c_i with a new computation time C_j. This provides the following predicate that tests whether the task t_i is schedulable, if task t_j is released:

$$sched(t_i, t_j) = \begin{cases} \mathsf{r}_i - \mathsf{c}_i + C_j \leq D_i - \mathsf{d}_i & \text{if } P_i < P_j \wedge inqueue(t_i) \\ \mathsf{r}_i - \mathsf{c}_i \leq D_i - \mathsf{d}_i & \text{if } P_i > P_j \wedge inqueue(t_i) \\ \text{false} & \text{if } \neg inqueue(t_i) \end{cases}$$

The second case of this predicate holds when the task that we want to release will not influence the measured task.

4.2 Encoding the Fixed Priority Scheduler

We have devised a model of a fixed priority scheduler, to support our approach to the verification of adaptive embedded systems. This encoding enables us to simulate the passage of time in the model, and simultaneously, keep track of response times of tasks in the queue. This is required for an on-line analysis of schedulability. Next, we give the scheduler's encoding high-level description, yet omitting some details due to lack of space.

High level description. The model consists of three locations with identified, different roles: Idle, Busy and Error, as shown in the overview Fig. 5. The corresponding locations can also be found in the Fig. 6.

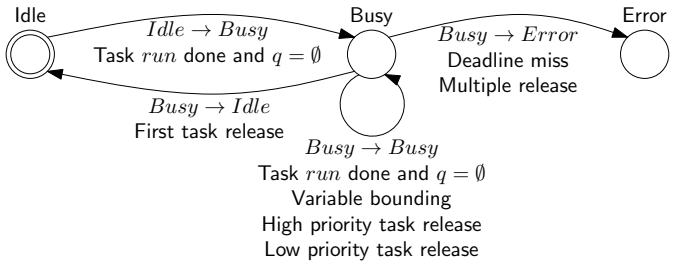

Fig. 5. A high level overview of the scheduler and queue encoding in timed automata.

The scheduler and queue timed automaton model starts in the Idle location. As soon as some task is released, the location changes to Busy, and if an error occurs, the model switches to the Error location. Otherwise, the model loops in the Busy location, for as long as there are tasks in the queue. The addition of the Error location makes it possible to easily distinguish between an error in the schedule, and a deadlock in the task release model.

The queue is implemented such that each task t_i has attribute $inqueue_i$. This attribute indicates whether or not the task is present in the ready queue and is therefore directly tied to the $inqueue(t_i)$ predicate.

The initial location of the model is Idle. The model can be in this location only when there are no tasks in the queue, and no task is being executed. As soon as one of the tasks is released (added to the queue), the model changes its location to Busy, via the First task release edge. The consequence of taking this edge is that all of the clocks and variables are reset, in order to initiate a new cycle of execution. After that, the variables related to the release of the first task are updated (detailed explanation of variable updates is presented in section 4.3).

When the automaton is in the Busy location, it means that a task instance is being executed on the CPU. Since the model does not implement any task blocking mechanism, the situation when there are tasks in the queue, but none is executing, cannot occur.

The Busy location wraps in on itself in multiple edges. Many of these edges are restricted to execute at the same time point. This is enforced by an invariant

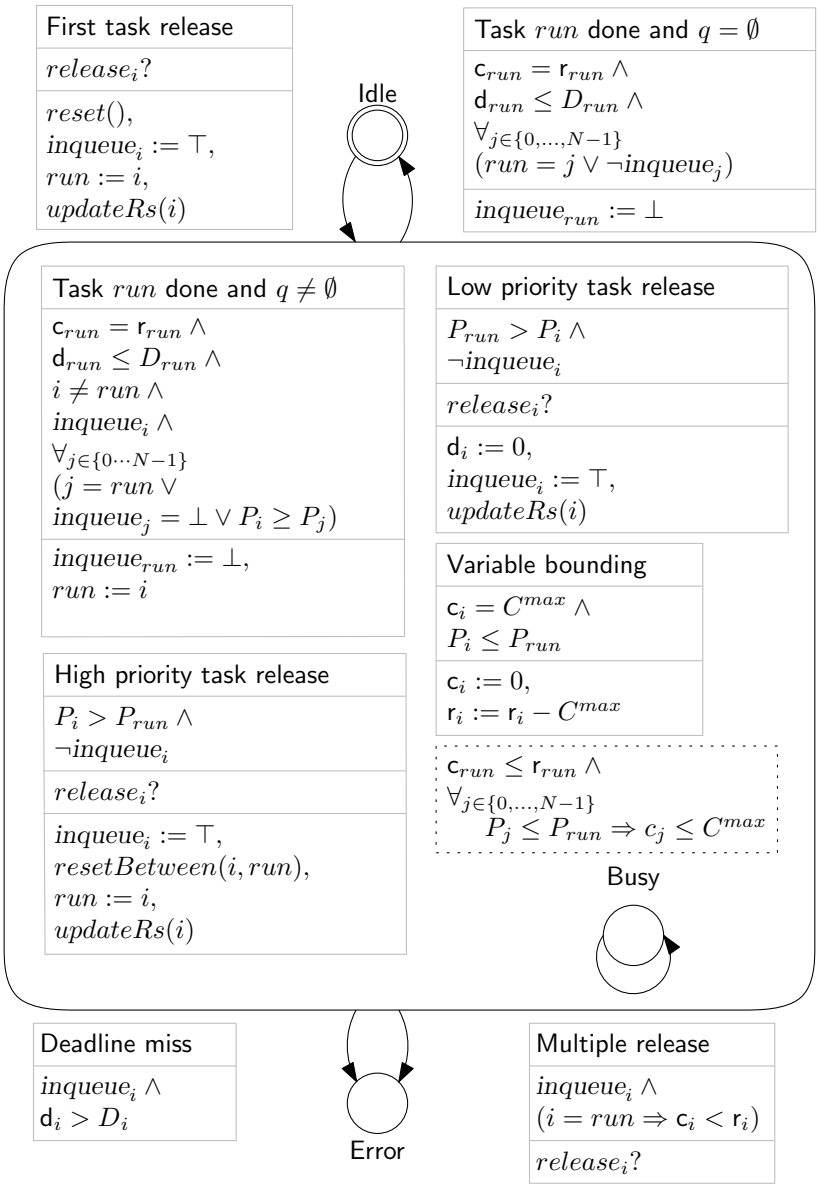

Fig. 6. The full model of scheduler and queue. The boxes represent transitions described by (in order from top to bottom): name, guard predicate, synchronization expression, and assignment. If one of the values is nil it is not shown.

on the Busy location (shown in dotted box in Fig. 6). The model uses variable i to represent classes of edges that are instantiated for every task in the task set. For instance, if there are five task types in the task set, there will be five Variable bounding edges, one for each task type. Below, we enumerate the classes of edges looping in the Busy location:

- Task run done and $q \neq \emptyset$ - After the current task has completed its execution, this current task, denoted by the value of the run variable, is removed from the queue, and a next task is chosen for execution, out of those currently in the queue. The choice of the next task is done by selecting the edge corresponding to a task that has higher priority than all of the other tasks.
- High priority task release - It releases a new task into the queue, which preempts the currently running task. The release changes the status of the currently executing task, sets a new value of the variable run, and resets the currently inactive variables that have lower or equal priority than the new task.
- Low priority task release - If the new task is not of higher priority then the currently running task, it is then just placed in the queue. Its variables are already active, so only the deadline clock d_i is reset.
- Variable bounding - Due to the nature of timed automata, it is required that the variables in the model have upper and lower bounds. This process is explained in detail in section 4.3.

Last but not least, we need to consider the possibilities for the model to switch to the Error location. In such a case, there are two classes of edges and, once again, they are iterated over all tasks:

- Deadline miss edge is taken when a task misses its deadline, that is, the deadline clock becomes greater then the value of the relative deadline.
- Multiple release edge is taken when a task is released, but it is already in the queue.

Finally, the edge "Task run done and $q = \emptyset$" is taken when the last task in the queue is completed, and there are no more tasks left. We remove the currently running task from the queue and return to the Idle location.

4.3 Variable Bounding

To be able to verify timed automata models, all of the variables, including clocks, have to be bound. To bound variables in this model, we have introduced a loop on the Busy location, named Variable bounding. This loop is executed for each individual task t_i, whenever its total computation time reaches a certain value C^{max}. It reduces the total computation time c_i to zero, and subtracts C^{max} from the corresponding response time variable r_i, thus not influencing the delta $\mathsf{r}_i - \mathsf{c}_i$. By doing this, we ensure that the total computation time is always lower or equal to C^{max}, and that the response time variable is kept bound to $C^{max} + D^{max}$, within a working system. C^{max} can be any value greater or equal to the maximum of computation times in the current task set, and D^{max} is the maximum of deadlines in the task set. If the response time becomes greater than $C^{max} + D^{max}$, we can guarantee that the task will breach its deadline, and the model becomes unschedulable.

Theorem 1 *The problem of checking the schedulability of the system, modeled using adaptive task automata, is decidable.*

Proof Sketch. Due to space limitation, we give only a proof sketch here. In this section, we have presented a way of encoding adaptive task automata using timed automata, featuring a fixed priority scheduler. Since all of the variables in the model are bounded, and the problem of decidability of bounded timed automata with subtraction has been already proved decidable [10], the problem of decidability of checking schedulability in this particular case follows straightforwardly. □

5 Examples

To further illustrate the benefits that the system designers could get from using our model, we have analyzed two example systems, one synthetic, and one based on real world ideas.

5.1 Admission Control - a synthetic example

This example demonstrates the usage of the *sched_all*(t_i) predicate, for a given task t_i. We assume a system with two tasks, t_1 and t_2, where each has an alternative version of itself, t'_1 and t'_2, respectively. The task parameters are shown in Fig. 7; parameter J represents the task's jitter value. For instance, the task t_1 will be released every 10 time units, but can be up to 2 time units late.

Task	P	C	D	T	J
t_1	4	5	5	10	2
t'_1	3	3	5	–	2
t_2	2	3	7	10	5
t'_2	1	2	7	–	5

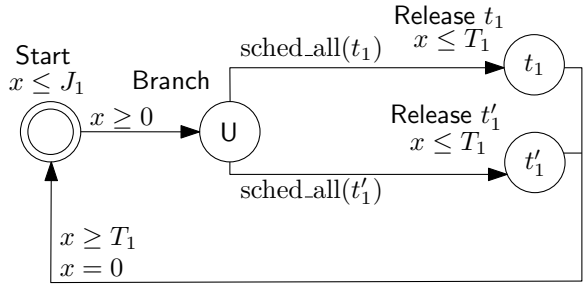

Fig. 7. Task set and adaptive task automata model for the synthetic example.

Fig. 7 shows the task automaton corresponding to t_1; the one of t_2 is similar, hence we omit it. For the task t_1, the task automaton checks whether all of the other tasks in the system are schedulable if the task t_1 is released. If the tasks are not schedulable, it tries to release the alternative variant of the task: t'_1. The two task automata instances are modeled as timed automata, and communicate with the scheduler via channels. The order between the preferred and alternative variant of the tasks, respectively, is ensured by using channel priorities [6]. For these models, we have proven that the system would never run into the **Error** state of the scheduler, and that (all of) the variants of the tasks will be eventually released. Proving that the system will never get into the Error state is the most demanding on the UPPAAL prover, and it required about 0.08 seconds CPU time, and 42MB memory on a dual-core 3.0GHz CPU, equipped with 4GB of RAM.

5.2 Smartphone task management example

The second example has been adapted from an idealized smartphone operating system. Modern smartphone devices support multitasking, yet have quite limited resources available for realizing their functionality. We propose a scheduler-level solution that enables a phone to adapt to the current situation fluently, by dynamically restricting the quality of service provided to the user.

The basic assumption is that the software in the smart phone is being executed in cycles. A series of short tasks that handle different applications are being executed each cycle. The applications that we have chosen for this example are: phone call, video call, and multimedia. The user can turn any of these applications on, or off, at arbitrary moments. The switch status of the application will not be immediately reflected in the active task set, but the task set will change during the next cycle, instead.

	P	T	D	C	Description
t_{cl}	5	10	10	4	Call
t_{vc}	4	10	10	3	Video Chat
t_{mm}	3	10	10	7	Multimedia: max quality
t'_{mm}	2	–	10	4	Multimedia: medium quality
t''_{mm}	1	–	10	3	Multimedia: low quality

Table 1. Set of tasks for the smartphone example.

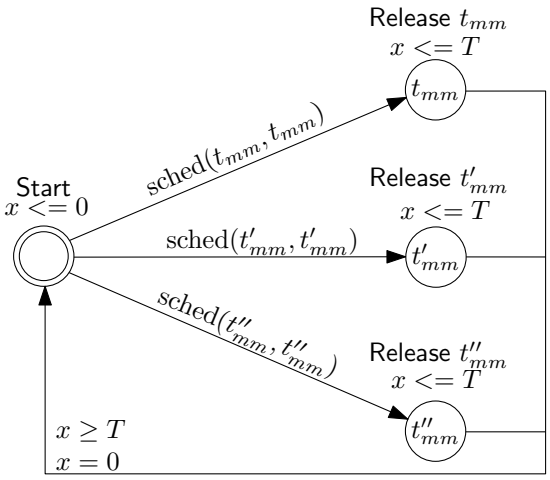

Fig. 8. Adaptive task automaton model for the smartphone example.

We have modeled the smartphone as an adaptive task automaton, and then implemented it as timed automata. The system model relies on a fixed priority scheduler. Tasks t_{cl} (phone call), and t_{vc} (video call), are described by "periodic release" automata, whereas task t_{mm} (multimedia) is modeled using the adaptive

task release automaton presented in Fig. 8. The automaton has been modeled using priorities [6], to remove nondeterminism from the execution.

Once the system has been modeled, a full verification of schedulability becomes possible. As previously, verification of not reaching the Error state has been the most demanding and, required about 0.03s and 34MB of RAM memory.

6 Conclusion

In this paper, we have proposed a framework for formal modeling and scheduling of adaptive embedded systems, which relies on a task automata description of the system (tasks and scheduler). In order to check at each task's release time point whether the system is schedulable, or will be with the potential release of other additional tasks, we have introduced a set of schedulability predicates to be used in the guards of the task automata model.

The encodings and on-line schedulability tests that we have devised can be seen as model-level means of predicting, at release time-moments, the timeliness behavior of real-time tasks with very general release patterns, which are stored in the ready queue. Our liberal adaptive task automata model, enhanced with predicates for schedulability test, lets one perform on-line adaptations that decide to admit or not certain tasks, depending on their respective adherence to the desired real-time requirements, that is, meeting their deadlines. The salient result of our work is the decidability of reachability and schedulability of adaptive task automata, by showing that the resulting model can be encoded in the timed automata framework.

The power of our approach resides exactly in the fact that the task selection strategy is specified as a predicate on clocks and integers. As it stands now, that is, assuming fixed priority schedulers, the model is compatible with any scheduler that has fixed ordering between the tasks, once the tasks are released. As with every formalized approach, there are some potentially useful-to-solve unexplored issues, which need further attention. For instance, it would be interesting to check on the consequences of allowing a task set to run, even if, based on our schedulability tests, we decide that it misses its deadline at the current time point. Another problem that deserves investigation is the possibility of releasing more than one task at a time, and verify the resulting model.

We also consider to extend the method to cater also for other schedulers than fixed-priority, for instance, Earliest-Deadline-First (EDF) schedulers. Nevertheless, although, as for now, our technique is restricted to fixed-priority schedulers, it can already decide on task executions at run-time, but has also the potential of manipulating the queue of released tasks, in the sense of switching ready tasks' priorities, if the case, removing certain tasks from the queue, etc., all based on possible further additions to the schedulability predicates.

The final avenue to explore would be along investigating the efficiency of our approach, when handling real-world industrial case study.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (April 1994)
2. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: Times: a tool for schedulability analysis and code generation of real-time systems. In: Proc. of International Workshop on Formal Modeling and Analysis of Timed Systems. Lecture Notes in Computer Science, Springer-Verlag (2003)
3. Amnell, T., Fersman, E., Pettersson, P., Sun, H., Yi, W.: Code synthesis for timed automata. Nordic Journal of Computing 9(4), 269–300 (2002)
4. Åsberg, M., Nolte, T., Pettersson, P.: Prototyping and code synthesis of hierarchically scheduled systems using times. Journal of Convergence (Consumer Electronics) 1(1), 77–86 (December 2010)
5. Buttazzo, G.C.: Hard Real-Time Computing Systems. Predictable Scheduling Algorithms and Applications. Kulwer Academic Publishers (1997)
6. David, A., Håkansson, J., Larsen, K., Pettersson, P.: Model checking timed automata with priorities using dbm subtraction. In: Asarin, E., Bouyer, P. (eds.) Formal Modeling and Analysis of Timed Systems, Lecture Notes in Computer Science, vol. 4202, pp. 128–142. Springer Berlin / Heidelberg (2006)
7. David, A., Illum, J., Larsen, K., Skou, A.: Model-Based Framework for Schedulability Analysis Using UPPAAL 4.1. CRC Press (2011/12/27 2009)
8. Fersman, E., Krcal, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. Information and Computation 205(8), 1149 – 1172 (2007)
9. Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: Schedulability analysis of fixed-priority systems using timed automata. Theor. Comput. Sci. 354, 301–317 (March 2006)
10. Fersman, E., Pettersson, P., Yi, W.: Timed automata with asynchronous processes: Schedulability and decidability. In: In Proceedings of TACAS 2002. pp. 67–82. Springer-Verlag (2002)
11. Mikučionis, M., Larsen, K., Rasmussen, J., Nielsen, B., Skou, A., Palm, S., Pedersen, J., Hougaard, P.: Schedulability analysis using uppaal: Herschel-planck case study. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification, and Validation, Lecture Notes in Computer Science, vol. 6416, pp. 175–190. Springer Berlin / Heidelberg (2010)
12. Norström, C., Wall, A., Yi, W.: Timed automata as task models for event-driven systems. In: Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on. pp. 182 –189 (1999)
13. Schaefer, I.: Integrating Formal Verification into the Model-Based Development of Adaptive Embedded Systems. Ph.D. thesis, TU Kaiserslautern, Kaiserslautern, Germany (Oct 2008), iSBN 978-3-89963-862-2
14. Yu, F., Li, G., Xiong, N.: Schedulability analysis of multi-processor real-time systems using uppaal. In: Information Science and Engineering (ICISE), 2010 2nd International Conference on. pp. 1 –6 (dec 2010)