Adaptive Task Automata with Earliest-Deadline-First Scheduling

Leo Hatvani*[1], Alexandre David†[2], Cristina Seceleanu‡[1], and Paul Pettersson§[1]

[1]Mälardalen University, Västerås, Sweden
[2]Aalborg University, Aalborg, Denmark

August 19, 2014

Abstract

Adjusting to resource changes, dynamic environmental conditions, or new usage modes are some of the reasons why real-time embedded systems need to be adaptive. This requires a rigorous framework for designing such systems, to ensure that the adaptivity does not result in invalidating the system's real-time constraints.

To address this need, we have recently introduced adaptive task automata, a framework for modeling, verification, and schedulability analysis in adaptive, hard real-time embedded systems, assuming a fixed-priority scheduler.

In this work, we extend the adaptive task automata framework to incorporate the earliest-deadline-first scheduling policy, as well as enable implementation of any other dynamic scheduling policy. To prove the decidability of our model, and at the same time maintain a manageable degree of conciseness, we show an encoding of our model as a network of timed automata with clock updates. To support this, we also show that reachability in our class of timed automata with updates is decidable. Our contribution helps to streamline the process of designing safety critical adaptive embedded systems.

1 Introduction

One way to enable real-time embedded systems to cope with environment, application, or platform changes is to introduce adaptivity at the design phase of system development. Adaptivity lets the system adjust to a new situation, but at the same time may introduce new errors such as breached timing constraints or other extra-functional requirements. Our goal is to propose a way to streamline modeling and verification of adaptive embedded systems (AES) in order to minimize the introduction of such errors at the design stage.

In the framework of *adaptive task automata* (ATA) that we have recently proposed [11], we have started to address this need by providing formal support for modeling the AES behavior, simulation of the system execution, and verification of the schedulability. By formally verifying the system's schedulability, we ensure that the system is going to meet its hard real-time specifications as well as satisfy any other extra-functional properties.

*leo.hatvani@mdh.se
†adavid@cs.aau.dk
‡cristina.seceleanu@mdh.se
§paul.pettersson@mdh.se

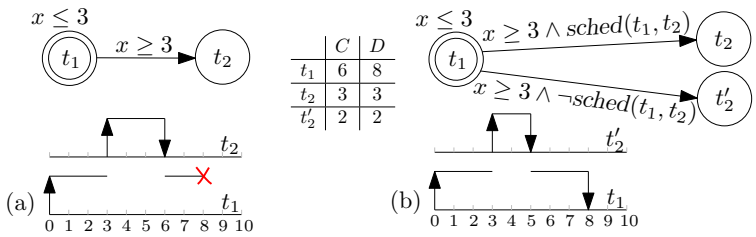


Figure 1: An adaptation example: (a) task automaton model, and (b) ATA model.

In our previous work on adaptive task automata, we have assumed fixed priority scheduling (FPS) policy. In this work we are extending the framework to support dynamic scheduling policies by incorporating the earliest-deadline-first (EDF) scheduling policy into the framework. Hereinafter we will refer to the specific variant of ATA with the EDF scheduling policy as ATA_{EDF}.

The main contribution of this work is to find solutions to the challenges of verifying the EDF schedulability of hard real-time tasks, in ATA. To tackle this, we show that verification of schedulability in ATA_{EDF}, described in section 2, is decidable, by proposing an encoding of the framework as a network of timed automata with (clock) updates (section 3). We present a summary of the proof of bisimilarity between the model and its encoding as well as decidability of reachability for our class of timed automata with updates (section 4).

2 Adaptive Task Automata

The adaptive task automata framework builds on top of task automata [8] by providing predicates that influence task release patterns based on the content of the ready queue. The task automata framework, in turn, is based on timed automata [2] extended with: tasks that can be released upon entering locations, a queue, and a scheduler to handle the released tasks and simulate their execution. Since the current work elaborates on ATA extensively, we refer the reader to the cited literature for more in-depth information.

In our model, we assume a uniprocessor system with independent, non-suspending tasks. For each task, computation time and relative deadline are known and are specified as natural numbers. At any point in time, there can be at most one task instance (job) per task in the queue and will be also referred to as task.

2.1 Introductory Example

As a simple example, consider the set of tasks in Figure 1. Each task is characterized by its execution time C and a relative deadline D. Figure 1(a) models the release of the task t_1 at time 0 by annotating the initial location (double concentric circle) with the task. Task t_2 is released in the second location after 3 time units. The delay is modeled by adding a zero-initialized clock (x) to the system, annotating the initial location with the invariant $x \leq 3$ that models that the location will be exited after at most 3 time units, and adding a guard $x \geq 3$ on the edge, denoting that the edge will not be taken until at least 3 time units have passed.

If we schedule the model in Figure 1(a) using EDF, the deadline of the task t_1 will be reached before the task has a chance to complete. Assuming that we have t'_2, a lower quality alternative to task t_2, having a lower computation time, we could release t'_2 instead. To be able to chose the variant of the task to be released, we have introduced the following predicates in our previous work [11]:

- *inqueue*(t_i) which is true iff the task t_i is waiting in the ready queue or currently executing.
- *sched*(t_i) evaluates whether the task t_i is going to complete its execution by the deadline.
- *sched*(t_i, t_j), assuming that the task t_i is already in the queue, evaluates whether it will complete in time if the task t_j is released into the queue.

By incorporating the predicate *sched*(t_i, t_j) into the model of Figure 1(a), we get the model presented in Figure 1(b). Here, task t_2 is released only if it will not disrupt task t_1, otherwise, task t_2' is released. With this modification, which can be seen as adaptive behavior, both tasks can successfully complete.

2.2 Overview of the Existing Framework

In ATA, the ready queue is a sequence of tasks ordered by the scheduling policy. Each task t_i in the ready queue is defined by two real values c_i and d_i. They represent the remaining execution time until completion (c_i) and the time until the task reaches its deadline (d_i).

Let us denote by T the set of tasks, and by $P(T)$, ranged over by p, the set of various Boolean combinations of the above predicates over the set of tasks. Utilizing this notation, an adaptive task automaton can be defined as follows.

Definition 1. [11] *An adaptive task automaton over actions Act, clocks X, invariants $\Phi(X)$, guard constraints $B(X)$, tasks T, and predicates over tasks $P(T)$ (Definition 3) is a tuple $\langle Act, X, L, l_0, E, I, M\rangle$ where L is a finite set of locations, $l_0 \in L$ is the initial location, $E \subseteq L \times B(X) \times P(T) \times Act \times 2^X \times L$ is the set of edges, $I : L \mapsto \Phi(X)$ is a function assigning each location an invariant, and $M : L \hookrightarrow T$ is a function annotating locations with tasks.*

Guard constraints $B(X)$ are a set of conjunctions of atomic constraints of the type $x \sim C$ or $x - y \sim C$ where $x, y \in X$ are clocks, C is a natural number, and $\sim \in \{<, \leq, =, \geq, >\}$. Invariants $\Phi(X)$ are a set of conjunctions of atomic constraints of the type $x \sim C$ where $x \in X$ is a clock, C is a natural number, and $\sim \in \{<, \leq\}$.

In the case of $(l, g, p, a, r, l') \in E$, we write $l \overset{g,p,a,r}{\longrightarrow} l'$, where $g \in B(X)$ is a guard constraint, $a \in Act$ is an action, and r is the subset of clocks that will be reset on taking the edge. □

We can represent the state of an adaptive task automaton as a triple $\langle l, u, q\rangle$, where $l \in L$ is the current location, $u \mapsto \mathbb{R}_{\geq 0}$ is a function mapping clocks to non-negative real values, and $q = [t_0(c_0, d_0), \ldots, t_n(c_n, d_n)]$ is the current ready task queue. $\mathsf{Sch}(q)$ is a function that returns the ready queue sorted according to the scheduling policy, and $\mathsf{Run}_{\mathsf{Sch}}(q, \delta)$ is a function that returns the ready queue after it was executed for δ time units.

Definition 2. [11] *Given an adaptive task automaton $\langle Act, X, L, l_0, E, I, M\rangle$ with an initial state $\langle l_0, u_0, q_0\rangle$, and a scheduling strategy Sch, its semantics is a transition system defined as:*

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

where $r(u)$ is 0 for all $x_i \in r$ and $u(x_i)$ otherwise, $t :: q$ is the result of releasing t into the queue q, and $u \oplus \delta$ is the result of adding $\delta \in \mathbb{R}_{\geq 0}$ to all clock values in u. If both transitions are enabled, the choice is non-deterministic. □

Intuitively, in the context of tasks, transitions are possibilities to release new tasks, while delays in locations correspond to the execution of tasks.

Definition 3. [11] *Given a task automaton state $\langle l, u, q\rangle$, with $q = [t_0(c_0, d_0), \ldots, t_n(c_n, d_n)]$, a scheduling policy Sch, and two distinct tasks, t_i and t_j, let P be the set of predicates $\{inqueue(t_i), sched(t_i), sched(t_i, t_j)\}$ satisfied as follows:*

$$\begin{array}{ll} \langle l,u,q\rangle \models & inqueue(t_i) \ \textit{if} \ \ t_i \in q \\ \langle l,u,q\rangle \models & sched(t_i) \ \textit{if} \ \ inqueue(t_i) \wedge (c_i + \sum_{j \in HP(t_i)} c_j) \leq d_i \vee \\ & \qquad \neg inqueue(t_i) \wedge \langle l,u,\mathsf{Sch}(t_i :: q)\rangle \models sched(t_i) \\ \langle l,u,q\rangle \models & sched(t_i,t_j) \ \textit{if} \ \ inqueue(t_i) \wedge \langle l,u,\mathsf{Sch}(t_j :: q)\rangle \models sched(t_i) \end{array}$$

where $HP(t_i)$ is the set of all tasks that have higher priority than t_i, and $\mathsf{Sch}(t_j :: q)$ is the queue ordered by the scheduling policy Sch after the release of the task t_j.

Boolean combinations of the above predicates over a set of tasks T give us the set of all possible combinations of predicates denoted by $P(T)$. □

3 Encoding of $\mathbf{ATA_{EDF}}$

In order to show the decidability of the $\mathrm{ATA_{EDF}}$ framework, we have encoded the universal $\mathrm{ATA_{EDF}}$ model as a network of timed automata with (clock) updates (TAU). First we present the framework of timed automata with updates. The framework was introduced previously by Bouyer et al. [6], yet we use a variant whose decidability has to be proven for our result to hold. Then the encoding itself is laid out in three steps. The first step shows a way to encode task releases, the second provides the intuition behind the encoding of the predicates used for adaptivity, and the third introduces the encoding of the scheduler. After we have encoded the system as timed automata with updates, we provide a proof that the reachability problem for our class of timed automata with updates is decidable and that the encoding is bisimilar to the original model. The $\mathrm{ATA_{EDF}}$ is more challenging than ATA as the task priorities are decided online.

3.1 Timed Automata with Updates

The timed automata framework, as defined by Alur and Dill [3], has served as the basis for several modeling variations proposed in order to fit specific design purposes [8, 6, 13]. Along the same line, our approach also relies on a variant of timed automata.

To concisely encode the scheduler model as timed automata, we need to allow for "clock to clock" assignments. Although such clock assignments are already present in the updatable timed automata framework [6], they are defined on models without invariants on locations. Since our work depends on location invariants, let us define the extension of timed automata that supports clock to clock assignments as well as location invariants.

Definition 4. *A* timed automaton with updates *(TAU) over clocks X and actions Act is a tuple $\langle Act, X, L, l_0, E, I\rangle$, where L is a finite set of locations, l_0 is the initial location, $E \subseteq L \times B(X) \times Act \times 2^X \times 2^{X^2} \times L$ is the set of edges, and $I : L \to \Phi(X)$ assigns invariants to locations. In the set of edges E, $B(X)$ is the set of guard constraints, 2^X represents the set of clock resets, and 2^{X^2} represents the set of clock assignments of the form $x := y$, where $x, y \in X$.*

The set of invariants $\Phi(X)$ is a set of conjunctions of atomic expressions of the type $x \sim C$ where $x \in X$ is a clock, C is a natural number, and $\sim \in \{<, \leq\}$. The set of guard constraints $B(X)$ can be defined as a set of Boolean combinations of atomic expressions of the type $x \sim C$ or $x - y \sim C$ where $x, y \in X$ are clocks, and $\sim \in \{<, \leq, =, \geq, >\}$.

In the case of $(l, g, a, r, s, l') \in E$, we write $l \xrightarrow{g,a,r,s} l'$, where r is the subset of clocks that will be reset on taking the edge, and s the set of clock assignments. □

The semantics of TAU is defined in terms of a timed transition system over states of the form (l, u), where l is a location, $u \mapsto \mathbb{R}_{\geq 0}$ is an assignment of clocks to non-negative real values, and the initial state is (l_0, u_0), where u_0 assigns all clocks in X to 0.

Definition 5. *Given a timed automaton with updates $\langle Act, X, L, l_0, E, I\rangle$ with an initial state $\langle l_0, u_0\rangle$, its semantics is a transition system defined as:*

- $\langle l, u\rangle \xrightarrow{a} \langle l', r(s(u))\rangle$ *if* $l \overset{g,a,r,s}{\longrightarrow} l' \in E$ *and* $u \models g$
- $\langle l, u\rangle \xrightarrow{\delta} \langle l, u \oplus \delta\rangle$ *if* $(u \oplus \delta) \models I(l)$

where $s(u)$ performs the assignments $x_i := x_j$ for every $(x_i, x_j) \in s$, $r(u)$ is 0 for all $x_i \in r$ and $u(x_i)$ otherwise, and $u \oplus \delta$ is the result of adding $\delta \in \mathbb{R}_{\geq 0}$ to all clock values in u. If both transitions are enabled, the choice is non-deterministic. □

A timed trace σ of a TAU, as is also the case with timed automata [3], is a sequence of delay and action transitions $\sigma = (l_0, u_0) \overset{a_1}{\rightarrow} (l_1, u_1) \overset{a_2}{\rightarrow} \ldots \overset{a_n}{\rightarrow} (l_n, u_n)$ where a_i can be either action ($\xrightarrow{a}$) or delay ($\xrightarrow{\delta}$) transition, and a location l is said to be *reachable* if there exists a timed trace ending in the state (l, u).

A *network* of TAU, $A_1 || \ldots || A_n$ over X and Act is defined as the parallel composition of n TAU over X and Act. Semantically, a network of TAU again describes a timed transition system obtained from those components, by requiring action transitions to synchronize on complementary actions (i.e., $a?$ is complementary to $a!$) [5].

3.2 Eearliest-Deadline-First Scheduling Policy

To encode the scheduler, we need to clearly define the EDF policy in the context of this paper. Since the strategy for choosing the next task between two or more tasks with equal deadlines does not impact the optimality of the EDF algorithm [10], we can give the following definition of EDF with deterministic tie resolution.

Definition 6. *According to the EDF scheduling policy with deterministic tie resolution, the priority P_i of task t_i is greater than the priority P_j of task t_j if the time left until the absolute deadline d_i of task t_i is smaller than the time left until the absolute deadline d_j of task t_j, or their absolute deadlines are equal and $i > j$ holds. This can be expressed as*

$$P_i > P_j \iff d_i < d_j \vee (d_i = d_j \wedge i > j)$$

where i and j represent strictly ordered task indices. □

3.3 Task Releases

In ATA, tasks are released on changing to locations that are annotated with sets of tasks. A straightforward method to realize instant task triggering upon entering a location is to use synchronization channels on the edges of the corresponding TAU representation. This is demonstrated in Figure 2.

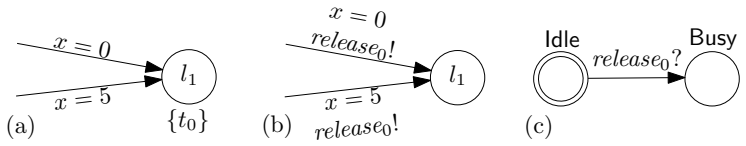


Figure 2: (a) task automaton, (b) (a)'s encoding, (c) part of (b)'s scheduler

In Figure 2(a), we have a basic task automaton location with two disjunctive edges leading to it. Location l_1 is annotated with the task set $\{t_0\}$. By entering the location via any of the edges, the task t_0 should be released and handled by the scheduler.

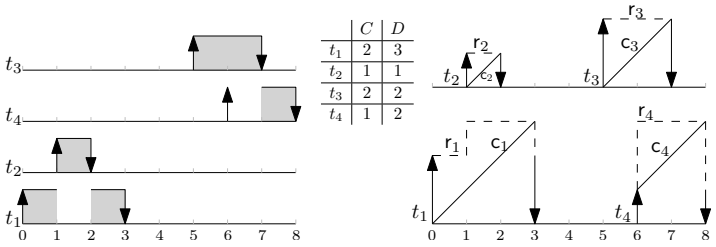


Figure 3: Gantt chart and the encoding specific representation of tasks

Modeling this behavior in TAU requires annotating every edge entering the location l_1 with a synchronization channel that creates a network of timed automata between the observed automaton presented in Figure 2(b) and the corresponding edges in the scheduler automaton as seen in Figure 2(c). In some cases, additional committed locations [4] might be needed to accomplish this.

3.4 Schedulability Predicates

The ATA model implements adaptivity via a set of scheduling predicates that may restrict edge guards: $sched(t_i)$, $sched(t_i, t_j)$, and $inqueue(t_i)$. All predicates are evaluated within the context of the current ready queue.

To express the predicates in timed automata with updates, we need to define an adequate encoding of the relevant variables that describe tasks in ATA models. The task automata model and consequently the adaptive task automata model define the task t_i in terms of remaining computation time c_i and time left until the deadline d_i. We encode the remaining computation time as the difference between the response time r_i and the computation time c_i: $c_i = \mathsf{r}_i - \mathsf{c}_i$.

To illustrate this encoding, let us observe Figure 3. The left side of the figure presents a Gantt chart of task releases, while the right side presents a graph of the values of the variables c and r for the same set of tasks. Note that, in the graph, the tasks t_2 and t_3, as well as t_1 and t_4 are presented on the same level to conserve vertical space.

At time 0, task t_1 is released. A higher priority task t_2 preempts it at time 1. At the moment of preemption, the response time r_1 is increased by C_2, the computation time of t_2, while the response time of task t_2 is equal to its computation time. Both tasks complete when their computation time becomes equal to their response time, respectively.

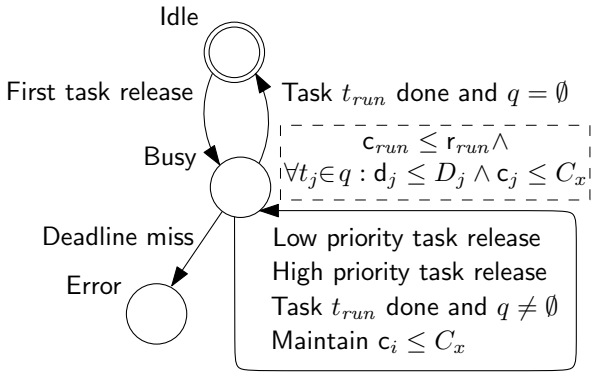
Two time units after task t_1 completes, task t_3 is released. It is already executing when task t_4 is released. Although task t_4 has computation time of only 1 time unit, its response time already accounts for t_3. Due to the continuous nature of timed automata clocks, we cannot extract information on how much of the computation time of task t_3 has been already used, so we have to use the full response time of task t_3 increased by the response time of task t_4. In order for this response time to be in context, we also need to copy the clock value of c_3 to c_4, hence the clock c_4 starts from 1.

The time until deadline is encoded by simply comparing an increasing clock to the relative deadline, but it is not shown here.

3.5 Scheduler and Queue

Next, we encode the EDF scheduling policy together with the queue as a single automaton, which we will hereafter refer to as the scheduler automaton.

Our scheduler is created assuming the encoding of predicates outlined in the previous section and the EDF policy presented earlier. These two constraints, addressed at the same time, have significantly increased the complexity of the encoding. In Figure 4,



First task release
Sync $release_i?$
Update $q := q \cup \{t_i\}; t_{run} := t_i; \mathsf{r}_i := C_i; \mathsf{c}_i := 0; \mathsf{d}_i := 0; P_i := N$

Task t_{run} done and $q = \emptyset$
Guard $\mathsf{c}_{run} = \mathsf{r}_{run} \wedge \mathsf{d}_{run} \leq D_{run} \wedge q = \{t_{run}\}$
Update $q := q \setminus \{t_{run}\}$

Task t_{run} done and $q \neq \emptyset$
Guard $\mathsf{c}_{run} = \mathsf{r}_{run} \wedge \mathsf{d}_{run} \leq D_{run} \wedge t_i \in q \wedge t_i \neq t_{run} \wedge P_i = P_{run} - 1$
Update $q := q \setminus \{t_{run}\}; t_{run} := t_i; \forall t_j \in q : P_j := P_j + 1$

Maintain $\mathsf{c}_i \leq C_x$
Guard $\mathsf{c}_i = C_x \wedge t_i \in q$
Update $\mathsf{c}_i := 0; \mathsf{r}_i := \mathsf{r}_i - C_x$

Deadline miss
Guard $t_i \in q \wedge \mathsf{c}_i < \mathsf{r}_i \wedge \mathsf{d}_i \geq D_i$

Low priority task release
Guard $t_{next}^{\mathsf{EDF}}(t_i) = t_j \wedge \mathsf{c}_{run} < \mathsf{r}_{run}$
Sync $release_i?$
Update $q := q \cup \{t_i\}; \forall k \in q | P_k < P_j : P_k := P_k - 1; P_i := P_j - 1; \mathsf{r}_i := \mathsf{r}_j;$
$\mathsf{c}_i := \mathsf{c}_j; \mathsf{d}_i := 0; \forall t_k \in q | P_k < P_j : \mathsf{r}_k := \mathsf{r}_k + C_i$

High priority task release
Guard $t_{next}^{\mathsf{EDF}}(t_i) = \emptyset \wedge \mathsf{c}_{run} < \mathsf{r}_{run}$
Sync $release_i?$
Update $q := q \cup \{t_i\}; \mathsf{c}_i := 0; \mathsf{d}_i := 0; \mathsf{r}_i := 0; \forall t_j \in q \setminus \{t_i\} : P_j := P_j - 1;$
$P_i := N; \forall t_j \in q : \mathsf{r}_j := \mathsf{r}_j + C_i; t_{run} := t_i$

Figure 4: Overview of the encoding $E(\mathsf{Sch})$.

we show the entire scheduler model encoded as a timed automaton with updates, using synchronization channels to release tasks.

To reduce the presentation complexity of the encoding and make it more accessible to human readers, we have used a number of shorthands. For example, the queue is encoded as the set q. Since this set is referenced in every location, we need to replicate each location for every possible value of q. Since the number of tasks in the system (N) is finite and known in advance, this means that there will be 2^N replications of every location to reflect the set q. Next, only those locations that imply values that satisfy the incoming guards are connected by the edges to the originating location. The same approach can be applied to all other integer variables and translate this representation into a pure TAU. The exception to this approach is the function $t_{next}^{\mathsf{EDF}}()$ that will be addressed later.

The scheduler consists of three locations: Idle, Busy, and Error. The edges are classes of edges that are instantiated by iterating the variable t_i over the set of tasks. Task identifiers such as t_i and i are used interchangeably to reduce the maximum subscript level.

Since a task can be in the queue or not, the queue is encoded as a set q. Tasks themselves are represented via a number of variables: t_i represents the i-th task, t_{run} keeps track of the currently running task, c_i represents task computation clock explained in subsection 3.4, r_i contains the current response time of the task, and c_i is compared to r_i to evaluate if the task has completed its execution; d_i is a clock that is reset when a task is released, and is compared to the natural D_i to check if the task's deadline has passed, P_i is the current priority of the task. The priority N, equal to the number of tasks in the system, is the highest priority and it corresponds to the currently executing task.

The scheduler starts in the location Idle. This location corresponds to an empty task queue and it will be reentered on any occasion when there are no tasks left in the queue.

The edge going out of the location Idle is First task release. This edge is taken whenever the encoding of the adaptive task automaton synchronizes on $release_i$ channel without any additional constraints. Consequently, the task t_i is added to the queue, the currently running task is set to t_i, the response time is set to the computation time, the deadline clock is reset, and the task is assigned the highest priority.

In Busy location, there are four edges looping in the state, one returning to Idle and one leading to Error location. The invariant on Busy location, shown in dashed rectangle in Figure 4, ensures that, in the Busy location, the currently running task will not execute longer than its computation time, and that all of the tasks in the system have not missed their deadlines.

In case that a deadline is missed, the edge Deadline miss is taken. The deadline is considered missed when the task is in the queue, still has some execution left, and has reached or exceeded its deadline. In such case, the system enters the Error location and deadlocks.

To explain the looping edges on the Busy location, let us first define the selector $t_{next}^{\mathsf{EDF}}()$.

Definition 7. *The selector $t_{next}^{EDF}(t_i) = t_j$ selects the task t_j that has the next higher priority in the queue relative to the task t_i, regardless of whether the task t_i is in the queue or not according to the deterministic EDF policy (Definition 6).*

The selector returns the empty set if it is invoked for the highest priority task in the queue or any not-yet-released task that would become the highest priority task if it were released. □

Due to the nature of the EDF algorithm, a pure TAU implementation of this selector requires replication of any edge annotated with this selector into several edges. For the current permutation of tasks in the queue (implied by the current pure TAU location, and expressed via P_i and q variables in the representation), edges are created to test whether the new task will fit into any of the given possible positions in the queue. During

verification, due to the determinism outlined in Definition 7, only one of those edges will be enabled at any time.

The edge High priority task release employs this selector to check if the newly released task has higher priority than any of the tasks in the queue. The edge guard also checks whether the currently running task is still running. This check ensures that whenever a task completes it is removed from the queue before any further actions are taken. Since the newly released task has a higher priority than any other task in the queue, its response time is equal to its computation time. All of the other tasks' response times need to be increased by the computation time of the newly released task. Priorities of other tasks are reduced and the newly released task acquires the highest priority.

On the other hand, when the newly released task has lower priority than the currently running task, it needs to be placed at the correct place in the queue via Low priority task release edge. This is where the determinism of our EDF implementation via t_{next}^{EDF} selector comes into play. We need to ensure that the tasks added to the queue via this edge will be executed in the same sequence as they are added to the queue. Otherwise, the computed response times would be invalidated. As with the previous edge, we add the task to the queue, but this time we need to copy the response time and computation clock from the higher priority task. Then, we increase the response time of the new task, as well as any of lower priority, with the computation time of the released task.

As the time passes in Busy state, tasks are executing and will be removed from the queue when they complete, by one of the Task t_{run} done edges. The edge Task t_{run} done and $q \neq \emptyset$ is taken if the task has completed its execution before the deadline and if the next task is present in the queue. To switch the currently running task, the latter is taken out of the queue, a new task is set to currently running task and all of the active tasks' priorities are increased by one to keep priority values bound between 1 and N.

If the task is the last task in the queue, the edge Task t_{run} done and $q = \emptyset$ is enabled and removes the task from the queue while moving the automaton into the Idle location.

To keep all clocks and response times bound the edge Maintain $c_i \leq C_x$, resets the clock c_i to 0 every time an active clock reaches the maximum clock value C_x and the corresponding response time is decreased by C_x. While this edge alters the value of clocks, it does not influence the relevant difference $\mathsf{r}_i - \mathsf{c}_i$. This mechanism resolves the potential unboundedness of the system caused by the inheritance of c_i and r_i values in Low priority task release. Without it, any system that repeatedly releases tasks of lower priority than the currently running task can become unbounded.

4 Decidability

The decidability of schedulability verification for our model depends on two things: decidability of reachability for our variant of timed automata with updates (subsection 4.1) and that the encoding of $\mathrm{ATA}_{\mathrm{EDF}}$ model into timed automata with updates represents the original model correctly (subsection 4.2).

4.1 Decidability of Timed Automata with Updates

Alur and Dill [3] observe that we can partition the state space of a timed automaton into a finite number of discrete regions that can be exhaustively explored in a finite amount of time. Hence, the location reachability problem is decidable.

Our refined region equivalence relation is based on the relation given in [3] and extended by the region equivalence relation for timed automata with diagonal constraints presented by Bengtsson and Yi [5], and Fersman et al. [8].

Definition 8. *(Refined region equivalence $\approx$ [8, 3, 5]) For a clock $x \in X$, let C_x be a natural number. For a positive real number t, let $\{t\}$ denote the fractional part of t, and*

$\lfloor t \rfloor$ its integer part. Let $u, v \in \mathcal{V}$ be two regions, $\mathcal{G}$ a finite set of diagonal constraints in the form $x - y \bowtie \mathbb{Z}_{\geq 0}$ where $\mathbb{Z}_{\geq 0}$ is the set of non-negative integers, and $\bowtie \in \{<, \leq, =, \geq, >\}$.

We define $u \approx v$, i.e. u and v are refined-region-equivalent iff

1. *for each clock x, either $\lfloor u(x) \rfloor = \lfloor v(x) \rfloor$ or $u(x) > C_x$ and $v(x) > C_x$,*
2. *for each clock x, if $u(x) \leq C_x$, then $\{u(x)\} = 0$ iff $\{v(x)\} = 0$,*
3. *for all clocks x, y, if $u(x) \leq C_x$ and $u(y) \leq C_x$ then $\{u(x)\} \leq \{u(y)\}$ iff $\{v(x)\} \leq \{v(y)\}$, and*
4. *$u \models g$ iff $v \models g$ for all $g \in \mathcal{G}$.* □

Given Definition 8 of refined region equivalence, we can postulate that operations over regions will not disrupt the refined region equivalence relationship on TAU.

Lemma 1. *Given a timed automaton with updates, let $\mathcal{G}$ denote the set of diagonal constraints in the automaton and C_x be the maximum of M_x (the ceiling of x) and all constants appearing in the guards and invariants of the automaton involving clock x. Let $u, v \in \mathcal{V}$ and $t, t' \in R_{\geq 0}$. Then $u \approx v$ implies*

1. *$u + t \approx v + t'$ for some real number t' such that $\lfloor t \rfloor = \lfloor t' \rfloor$,*
2. *$u[x \mapsto 0] \approx v[x \mapsto 0]$ for a clock x, and*
3. *$u[x \mapsto y] \approx v[x \mapsto y]$ for all pairs of clocks x and y.*

Proof. We will start from two regions for which refined region equivalence relation holds. Next, we will split the problem into several cases and prove that, for each case, the implication of Lemma 1 holds.

Given two regions $u, v \in \mathcal{V}$, let us assume that that they are refined region equivalent $u \approx v$. This implies that they are also region equivalent $u \sim v$ [8]. Also, let $\mathcal{G}$ be a finite set of diagonal constraints in the form $x - y \bowtie \mathbb{Z}_{\geq 0}$ where $\mathbb{Z}_{\geq 0}$ is the set of non-negative integers, $\bowtie \in \{<, \leq, =\geq, >\}$, and C_x the largest integer among all constraints.

From the work by Larsen and Wang [14], we know that $u \sim v$ implies $u + t \sim v + t'$ for positive real numbers t and t' such that $\lfloor t \rfloor = \lfloor t' \rfloor$ and $u[x \mapsto 0] \sim v[x \mapsto 0]$ for a clock x. To extend these region equivalences $\sim$ to refined region equivalences $\approx$, we need to take a look at whether the diagonal constraints have stayed consistent between regions after the operations. Formally, $u + t \models g \Leftrightarrow v + t' \models g$ and $u[x \mapsto 0] \models g \Leftrightarrow v[x \mapsto 0] \models g$ for all $g \in \mathcal{G}$, which was proven true by Fersman et al. [8].

Next, we need to prove that the refined region equivalence of u and v holds after a clock is assigned the value of another clock $u[x \mapsto y] \approx v[x \mapsto y]$.

Let us consider several special cases of the expression $u[x \mapsto y] \approx v[x \mapsto y]$ while assuming $u \approx v$ and considering Definition 8.

Only one clock assignment. In the cases which involve only one clock assignment, the first criterion of Definition 8 for the modified clock x, can be expressed as

$\lfloor u[x \mapsto y](x) \rfloor = \lfloor v[x \mapsto y](x) \rfloor \vee u[x \mapsto y](x) > C_x \wedge v[x \mapsto y](x) > C_x$

This expression can be directly implied from the assumption $u \approx v$ for the clock y, i.e. $\lfloor u(y) \rfloor = \lfloor v(y) \rfloor \vee u(y) > C_y \wedge v(y) > C_y$. Since all other clocks are unaffected, the rest of the criteria can be trivially proven in a similar manner. For example, for the fourth criteria, any diagonal constraint that contained clock x will be evaluated as the already existing diagonal constraint for y.

Multiple clock assignments. Second case covers the situations when there are multiple clock assignments on a single edge. We will generalize it by assigning the value of the clock z to w, while examining a third clock x which might or might not have been altered. Assuming $u' \equiv u[z \mapsto w]$ and $v' \equiv v[z \mapsto w]$, from Definition 8, we can write

1. for each clock x, either $\lfloor u'(x) \rfloor = \lfloor v'(x) \rfloor$ or $u'(x) > C_x \wedge v'(x) > C_x$;
2. for each clock x, such that $u'(x) \leq C_x$, $\{u'(x)\} = 0$ iff $\{v'(x)\} = 0$;
3. for all clocks $x, y \in X$ if $u'(x) \leq C_x$ and $u'(y) \leq C_y$, then $\{u'(x)\} \leq \{u'(y)\}$ iff $\{v'(x)\} \leq \{v'(y)\}$;

4. $u' \models g$ iff $v' \models g$ for all $g \in \mathcal{G}$.

Since they deal with single clocks, the first and second criteria reduce to the previous case.

For the third criterium, we can observe that all clocks have the same relative ordering established in both valuations if they are $\leq C_x$. These orderings are inherited by clocks that are being assigned new values which implies that, between valuations, the orderings will still be equivalent.

Let us observe a specific guard from $\mathcal{G}$ from the fourth criterium.

$$u[z \mapsto w] \models x - y < C \text{ iff } v[z \mapsto w] \models x - y < C$$

If z is not identical to x or y, the assignment does not affect the guard. The interesting case is when x or y or both x and y are modified. Since these cases are symmetrical, let us observe $z \equiv x$ case.

$$u[x \mapsto w] \models x - y < C \text{ iff } v[x \mapsto w] \models x - y < C$$

If $w \equiv y$, the value of the guard for u and v valuation becomes exactly 0 and thus satisfies the requirement. To analyze the case when $w \not\equiv y$, we will split the clocks x, y, and w into their fractional and integer parts.

$$\begin{array}{ll} u[x \mapsto w] \models \{x\} + \lfloor x \rfloor - (\{y\} + \lfloor y \rfloor) < C & \text{iff} \\ v[x \mapsto w] \models \{x\} + \lfloor x \rfloor - (\{y\} + \lfloor y \rfloor) < C & \end{array}$$

We have previously shown that the integer parts of clocks are equal between valuations in u'. So let us group them together.

$$\begin{array}{ll} u[x \mapsto w] \models \{x\} - \{y\} < C - \lfloor x \rfloor + \lfloor y \rfloor & \text{iff} \\ v[x \mapsto w] \models \{x\} - \{y\} < C - \lfloor x \rfloor + \lfloor y \rfloor & \end{array}$$

From the Definition 8, we can infer that $\{x\}, \{y\} \in [0, 1)$, thus their difference satisfies $\{x\} - \{y\} \in (-1, 1)$. Because of this, if the right side of the inequality evaluates to 1 or greater, or less than -1, the statement will not depend on values of fractional parts and integer parts are identical between valuations. It follows that the only case when the fractional parts could affect the equivalence is when the integer side evaluates to 0.

$$u[x \mapsto w] \models \{x\} - \{y\} < 0 \text{ iff } v[x \mapsto w] \models \{x\} - \{y\} < 0$$

Definition 8 states that the relative ordering of clock valuations' fractional parts has to be identical between the valuations u and v. We will observe the case when $\{x\} \leq \{y\} \leq \{w\}$. While the guard is true before the assignment, after the assignment, the guard becomes false. But since the order has to be identical in both valuations, the equivalence stays true.

All other cases can be trivially solved by repeated application of a similar procedure. □

Lemma 2. *(Bisimulation of TAU) Let us assume a timed automaton with updates, a location l and clock assignments u and v. Then $u \approx v$ implies that:*

1. *when $(l, u) \to (l', u')$ then $(l, v) \to (l', v')$ for some v' such that $u' \approx v'$, and*
2. *when $(l, v) \to (l', v')$ then $(l, u) \to (l', u')$ for some u' such that $u' \approx v'$.*

Proof Outline. The proof follows from Lemma 1. Assume a location l and clock assignments u, and v, such that $u \approx v$. The refined region equivalence relation $\approx$ defines that the guards will evaluate in both u and v to the same truth values. Therefore, the set of enabled transitions is equal in both valuations. □

Lemma 3. *(Location Reachability) The location reachability problem for timed automata with updates and invariants is decidable if the bound M_x for each clock x is known.*

Proof. Lemma 1 shows that for each location l of the automaton, there is a finite number of equivalence classes derived from the bisimulation relation $\approx$. Since the number of locations is finite, the entire state space of an automaton can be partitioned into a finite number of equivalence classes and these equivalence classes can be effectively generated and searched. □

4.2 Model Bisimulation

Once we have encoded the entire ATA_{EDF} system as a network of TAU, we need to show that there exists a bisimulation between the original model and the encoding.

Our main result is described by Lemma 4 below, for which we outline the proof. In Definition 9, we first introduce the concept of schedulability as reachability.

Definition 9. *(Schedulability) The adaptive task automaton A with initial state (l_0, u_0, q_0) and scheduling strategy Sch is not schedulable iff there exists a trace $(l_0, u_0, q_0)(\longrightarrow_{\mathsf{Sch}})^*(l', u', q')$ such that in the state (l', u', q') there is a task t_i with more than zero computation time left, $c_i > 0$, and no more time to execute, that is $d_i \leq 0$. The state (l', u', q') is marked as (l', u', Error).* □

Lemma 4. *Let A be an adaptive task automaton and Sch the EDF scheduling strategy presented in Definition 6 . Assume that (l_0, u_0, q_0) and $(\langle l_0, \mathsf{Idle}\rangle, u_0 \cup v_0)$ are the initial states of A, and the product automaton $E(A)||E(\mathsf{Sch})$, 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:*

For all l and u: $(l_0, u_0, q_0) \rightarrow^* (l, u, \mathsf{Error})$ *implies* $(\langle l_0, \mathsf{Idle}\rangle, u_0 \cup v_0) \rightarrow^* (\langle l, \mathsf{Error}\rangle, u \cup v)$ *for some v.*

For all l, u, and v: $(\langle l_0, \mathsf{Idle}\rangle, u_0 \cup v_0) \rightarrow^* (\langle l, \mathsf{Error}\rangle, u \cup v)$ *implies* $(l_0, u_0, q_0) \rightarrow^* (l, u, \mathsf{Error})$.

Proof. In this proof, we will construct three sets of states that correspond to one or more states in the automata A and $E(A)||E(\mathsf{Sch})$. These new sets of states can be characterized as *no running tasks*, *a task is running*, and *a task has exceeded its deadline*. Next, we will establish a mapping between the new states and the states of the original two automata. Then, by observing the transitions from each of the states, we will establish that the mapping defines a bisimulation relation between the automata A and $E(A)||E(\mathsf{Sch})$.

The states of A and $E(A)||E(\mathsf{Sch})$ can be correlated and a bisimulation mapping between these states can be shown. Let us begin by defining three tuples S_1, S_2, and S_3 that capture the states of both automata.

There are no tasks in the queue:
$S_1 = \{ (l, u, q), (\langle l, \mathsf{Idle}\rangle, (u \cup v)) \} | q = \emptyset$

There is at least one task in the queue:
$S_2 = \{ (l, u, q), (\langle l, \mathsf{Run}\rangle, (u \cup v)) \} | Cnd_1 \wedge Cnd_2 \wedge Cnd_3 \wedge Cnd_4$

The system has been determined as unschedulable:
$S_3 = \{ (l, u, \mathsf{Error}), (\langle l, \mathsf{Error}\rangle, (u \cup v)) \} | Cnd_1 \wedge Cnd_2 \wedge Cnd_3 \wedge Cnd_5$

Where (l, u, q), and (l, u, Error) are states of A, an adaptive task automaton. $(\langle l, *\rangle, (u \cup v))$ are states of $E(A)||E(\mathsf{Sch})$, the product automaton. And $Cnd_1, ...$ are predicates. The predicates of the states S_2 and S_3 are:

- $Cnd_1 : \forall t_i \in q : d_i = D_i - \mathsf{d}_i$,
  where d_i is the incrementing clock of the task's deadline for the automaton $E(A)||E(\mathsf{Sch})$ and d_i is the decrementing value of the time left until the deadline in the automaton A,

- $Cnd_2 : \forall t_i \in q, t^{\mathsf{EDF}}_{next}(t_i) \neq \emptyset : c_i = \mathsf{r}_i - \mathsf{c}_i - (\mathsf{r}_{t^{\mathsf{EDF}}_{next}(i)} - \mathsf{c}_{t^{\mathsf{EDF}}_{next}(i)})$,
  where by $t_i \in q, t^{\mathsf{EDF}}_{next}(t_i) \neq \emptyset$ we define tasks in the queue that are not currently executing,
- $Cnd_3 : \forall t_i \in q, t^{\mathsf{EDF}}_{next}(t_i) = \emptyset : c_i = \mathsf{r}_i - \mathsf{c}_i$,
  in our encoding, the currently executing task is the only task whose computation time left c_i is not dependant on any other task since it is not being preempted,
- $Cnd_4 : \forall t_i \in q : d_i \geq 0 \wedge c_i \geq 0$,
  specifies that any task currently in the queue has a non-negative time left untile the deadline is reached as well as leftover computation time,
- $Cnd_5 : \exists t_i \in q : d_i \leq 0 \wedge c_i > 0$,
  there exists a task in the queue for which the amount of computation time left is greater than zero and the deadline has passed.

The selector $t^{\mathsf{EDF}}_{next}(t_i) =_{def} (\epsilon t_j \in \mathcal{T} | \, Tnxt_1 \vee Tnxt_2)$, according to Definition 7, selects a task t_j such that it has the next higher priority in the queue, compared to the task t_i. According to the current state of the queue, the selected task will finish execution just before the task t_i. The selector is encoded as a disjunction of two predicates, $Tnxt_1$, $Tnxt_2$:

$\forall t_i, t_j \in \mathcal{T} : t^{\mathsf{EDF}}_{next}(t_i) = t_j \iff Tnxt_1 \vee Tnxt_2$
$Tnxt_1 =_{def} t_i \in q \wedge t_j \in q \wedge$
$\quad (\forall t_k \in q : (d_k < d_i \vee d_k = d_i \wedge k > i) \Rightarrow (d_k > d_j \vee d_k = d_j \wedge k \geq j))$
$Tnxt_2 =_{def} t_i \notin q \wedge t_j \in q \wedge$
$\quad (\forall t_k \in q : (d_k < D_i \vee d_k = D_i \wedge k > i) \Rightarrow (d_k > D_j \vee d_k = D_j \wedge k \geq j))$

The selector uses the task's index as a pre-established, unique, and strictly ordered label assigned to each task instance.

Next, we establish that $S = S_1 \cup S_2 \cup S_3$ is a bisimulation. In order to do this, we will show that each of the components is a bisimulation.

(S_1) Let us assume that the initial state is in S_1: $((l, u, q), (\langle l, \mathsf{Idle}\rangle, (u \cup v))) \in S_1$ and that we reach another state $(l, u, q) \xrightarrow{a} (l', u', q')$ by an action transition $l \overset{g,p,a,r}{\longrightarrow} l'$. Then the clock valuation u satisfies the set of clock guards $u \models g$, and the queue satisfies the set of adaptivity predicates $q \models p$. Let the set of tasks released in the state l' contain one task t_i, $M(l') = t_i$.

The product automaton can make the following transition.

$$(\langle l, \mathsf{Idle}\rangle, (u \cup v)) \xrightarrow{a} (\langle l', \mathsf{Busy}\rangle, (u' \cup v'))$$

The task t_i is added to the queue and the corresponding variables are initialized: $c_i = C_i$, $d_i = D_i$, $\mathsf{c}_i = 0$, $\mathsf{d}_i = 0$, and $\mathsf{r}_i = C_i$. The variables in u are updated according to the edge First task release of Figure Figure 4. At this point, t_i is the currently running task since it is the only task in the queue.

The conditions Cnd_1, Cnd_3, and Cnd_4 are obviously satisfied, while Cnd_2 is not applicable, and Cnd_5 is not satisfied. Therefore the new state is in the set S_2.

$$((l', u', \mathsf{Sch}(M(l') :: q)), (\langle l', \mathsf{Busy}\rangle, (u' \cup v'))) \in S_2$$

For the delay transition of (l, u, q), assume $(l, u, q) \xrightarrow{t} (l, u + t, \mathsf{Run}(q, t))$, where the invariants are still satisfied after the delay, $(u + t) \models I(l)$. Then, the product automaton can make the following delay transition.

$$(\langle l, \mathsf{Idle}\rangle, (u \cup v)) \xrightarrow{t} (\langle l, \mathsf{Idle}\rangle, (u + t \cup v + t))$$

Since the state of the queue continues to be empty, the state stays in the set S_1.

$$((l, u + t, \mathsf{Run}(q, t)), (\langle l, \mathsf{Idle}\rangle, (u + t \cup v + t))) \in S_1$$

(S_2) Let us assume that the state of the product automaton is in S_2 and that we are observing an action transition a from l to l' that adds tasks from $M(l')$ to the task queue. Also assume that by adding this task into the queue we will not make the system unschedulable.

$$((l,u,q),(\langle l,\mathsf{Busy}\rangle,(u\cup v)))\in S_2$$
$$(l,u,q)\xrightarrow{a}(l',u',\mathsf{Sch}(M(l')::q))$$

Also assume that this is caused by the transition $l \xrightarrow{g,p,a,r} l'$ which implies that the guards and ATA predicates are satisfied, $u \models g$, and $q \models p$.

Let $M(l') = t_i$ and the starting queue consist of two tasks $q = [t_m, t_n]$ while $t^{\mathsf{EDF}}_{next}(t_m) = \emptyset$ (i.e. $t_{\boldsymbol{run}} = t_m$). The product automaton can make the transition:

$$(\langle l,\mathsf{Busy}\rangle,(u\cup v))\xrightarrow{a}(\langle l',\mathsf{Busy}\rangle,(u'\cup v'))$$

ending up in one of the following queue states:

$$q=[t_i,t_m,t_n],\ q=[t_m,t_i,t_n],\text{ or }q=[t_m,t_n,t_i],$$

depending on the value of the selector $t^{\mathsf{EDF}}_{next}(t_i)$. Note that the queue always maintains the relative ordering of the previously released tasks due to the characteristics of the EDF scheduling policy.

In the first case, resulting in $q = [t_i, t_m, t_n]$, $t^{\mathsf{EDF}}_{next}(t_i) = \emptyset$ holds. The scheduler automaton arrives into this state via High priority task release transition, and corresponding updates are executed. The currently executing task $t_{\boldsymbol{run}}$ becomes t_i and all other response times are increased by C_i. Since t_i has just been released, Cnd_3 holds. Cnd_1 is unchanged since there has been no delay. For Cnd_2, the following holds:

$$\begin{aligned} c'_m &= \mathsf{r}'_m - \mathsf{c}'_m - (\mathsf{r}'_{t^{\mathsf{EDF}}_{next}(m)} - \mathsf{c}'_{t^{\mathsf{EDF}}_{next}(m)}) \\ &= \mathsf{r}'_m - \mathsf{c}'_m - (\mathsf{r}'_i - \mathsf{c}'_i) \\ &= \mathsf{r}_m + C_i - \mathsf{c}_m - (C_i - 0) \\ &= \mathsf{r}_m - \mathsf{c}_m \end{aligned}$$

which corresponds to the value of c_m in the previous state in S_2. A similar observation can be applied to c'_n. Cnd_4 also holds since there was no delay and the newly released task does not make the system unschedulable.

In the second case, the queue becomes $q = [t_m, t_i, t_n]$, when $t^{\mathsf{EDF}}_{next}(t_i) = t_m$ holds. Here, the edge Low priority task release is taken in the scheduler automaton. The task t_m is not modified and t_n is affected in the similar manner as in the first case. The condition Cnd_1 obviously holds since the task is just released, Cnd_3 is unchanged, and in Cnd_2, we have:

$$\begin{aligned} c_i &= \mathsf{r}'_i - \mathsf{c}'_i - (\mathsf{r}'_m - \mathsf{c}'_m) \\ &= \mathsf{r}_m + C_i - \mathsf{c}_m - (\mathsf{r}_m - \mathsf{c}_m) \\ &= C_i \end{aligned}$$

which is true for newly released tasks.

The third case can be observed as a variation of the second case.

Thus, in all three cases, the new state is in the set S_2.

$$((l',u',\mathsf{Sch}(M(l')::q)),(\langle l',\mathsf{Busy}\rangle,(u'\cup v')))\in S_2$$

Let us observe the delay transition in S_2. Assume that $(l,u,q)\xrightarrow{\delta}(l,u+\delta,\mathsf{Run}(q,\delta))$. Let there be a task t_i for which $\delta \geq d_i$ (i.e. $\delta \geq D_i - \mathsf{d}_i$) and $c_i > d_i$ hold. The scheduler automaton can make any number of transitions in δ time corresponding to execution of the tasks in the queue and task switching. Eventually, the scheduler automaton will make

a delay transition δ_1 that will coincide with the remaining part of time until the deadline of t_i. After this delay, due to the invariant on the Busy state, no more delay transitions can be taken. Eventually, the only available transition will be the transition leading to the Error state. Thus the system will result in $((l', u', \mathsf{Error}), (\langle l', \mathsf{Error}\rangle, (u' \cup v'))) \in S_3$.

As long as all of the tasks in the queue have time left until the deadline ($d_i > 0$), the following delay transitions can occur in the scheduler automaton:

(i) $\delta < c_{run}$ - If the delay δ is shorter than the remaining computation time of the currently executing task, then the scheduler automaton will take no actions and $((l, u + t, \mathsf{Run}(q, t)), (\langle l, \mathsf{Busy}\rangle, (u + t \cup v + t))) \in S_2$. The condition Cnd_1 is satisfied by assumption. In Cnd_2, both c-s are increased by δ, thus not changing the remaining computation times of currently non-executing tasks. The assumption $\delta < c_{run}$ implies that Cnd_3 and Cnd_4 will hold.

(ii) $t = c_{run} \wedge \mathsf{Run}(q, t) \neq \emptyset$ - In case that the delay is equal to the computation time of the currently executing task $t_{\boldsymbol{run}}$, and $\mathsf{Run}(q, t)$ does not result in an empty set, the system will still satisfy conditions S_2 in a similar manner to the previous case. Also, the transition Task *run* done and $q \neq \emptyset$ will be enabled at the end of the delay action. This transition will remove the completed task from the queue and replace $t_{\boldsymbol{run}}$ with the next highest priority task.

After taking the action transition, the condition Cnd_1 is satisfied by the assumption about deadlines. Cnd_2 is still satisfied since one task is removed from the condition, and for the rest, the delay of the clocks cancels itself out in a manner similar to the previous case. For the Cnd_3, assume that $t^{\mathsf{EDF}}_{next}(t_i) = t_{run}$, that is task t_i is the task with second highest priority in the queue. Then, just after the delay transition, Cnd_2 for task t_i is $c'_i = \mathsf{r}'_i - \mathsf{c}'_i - (\mathsf{r}'_{run} - \mathsf{c}'_{run})$, where $\mathsf{r}'_{run} - \mathsf{c}'_{run} = 0$ according to the assumption on the delay transition. This is equal to the expected value for the Cnd_3, so the product automaton is still in S_2.

(iii) $t = c_{run} \wedge \mathsf{Run}(q, t) = \emptyset$ - After the delay transition in this case, the product automaton will have completed the last task in the current ready queue. At this point the state is still S_2, which can be proven in a similar manner to the previous case. Unlike the previous case, the transition Task *run* done and $q \neq \emptyset$ is not enabled since there is no task in the queue that is not the currently executing task. On the other hand, the transition Task *run* done and $q = \emptyset$ is enabled and will move the product automaton back to the Idle state. Since the only task in the queue is t_{run}, removing it will result in an empty queue which is the condition for the state S_1.

(S_3) Once this state is reached, the system is considered unschedulable.

In the previous few paragraphs, we have sketched out one direction of the bisimulation between A and $E(A)||E(\mathsf{Sch})$. The other direction can be shown in a similar manner. Thus we conclude our proof of Lemma 4. □

Since we have proven that the reachability problem is decidable for TAU, stated by Lemma 3, also that every $\mathrm{ATA}_{\mathrm{EDF}}$ can be translated into a bisimilar TAU, we can conclude that the problem of checking schedulability of $\mathrm{ATA}_{\mathrm{EDF}}$ is decidable as well.

5 Related Work

Our work tries to unify schedulability analysis with modeling and analysis of adaptive embedded systems. At the same time, a number of works address problems in those two separate fields, as well as non-modeling methods for analysis of schedulability in adaptive contexts. While this is by no means an exhaustive list of the works in these areas, we will try to list those that are closest to ours.

In the following works, verification of adaptive embedded systems is done on a more coarse scale than in our approach. Most of these approaches could be used in synergy with ours to provide system level verification, while ours provides task level granularity. Adler et al. [1] use Kripke structures as the underlying presentation of the system and specify the system's properties using LTL. Schneider et al. [17] have proposed a method to describe and analyze adaptation behavior in embedded systems in which the data flow is augmented with quality descriptions used by configuration rules to determine potential adaptations. Goldsby et al. [9] provide the AMOEBA-RT model focused on run-time verification and monitoring.

In the area of adaptive scheduling, most work [12, 15] was done to achieve a lower energy consumption by exploiting dynamic voltage scaling features of modern CPUs. While such approaches can be used to analyze schedulability in some adaptive contexts, our approach makes it possible to model and analyze more precisely task release patterns of non-periodic tasks.

Finally, other works have approached verification of schedulability by means of timed automata for uniprocessors [7, 16], and multiprocessors [18] without explicit inclusion of adaptive functionality.

6 Conclusion

In this work, we have shown that the verification of adaptive task automata with earliest-deadline-first scheduling policy is decidable. To support our claim, we have encoded our adaptive task automata model as timed automata with updates and presented that the model and its encoding are bisimilar, as well as given a proof that reachability in our variant of timed automata with updates is decidable.

Our main result is the proof of decidability of our ATA extensions. Using ATA, it is possible to model the environment of an embedded system as well as behavior of functional and extra-functional properties in response to internal or environmental changes. Thus we verify the behavior of specified properties throughout the execution of the system.

In this work, we have implemented the EDF scheduling policy. However, by replacing the selector $t_{next}^{\mathsf{EDF}}()$, we can implement any other policy that is deterministic and does not change relative task priorities after their release into the queue. A non-deterministic selector would invalidate the schedulability testing predicates ($sched()$) since the response times predicted when testing a task would not necessarily correspond to the actual response times after the task is released.

During the encoding, we have faced a number of challenges. To support dynamic scheduling policies and schedulability predicates, we have required dynamic construction of task response times, which, in turn, have required a clock copying mechanism that had to be added as an extension of timed automata.

As future work, we plan to further explore removal of the assumptions, specifically extend the framework to support modeling of multi-core systems, smart handling of tasks with variable execution time, shared resources, as well as create a set of templates that correctly model the most commonly utilized task release patterns.

Acknowledgments

This research has been supported by the Swedish Research Council, which is gratefully acknowledged.

References

[1] Rasmus Adler, Ina Schaefer, Tobias Schuele, and Eric Vecchié. From model-based design to formal verification of adaptive embedded systems. In Michael Butler, MichaelG. Hinchey, and MaríaM. Larrondo-Petrie, editors, *Formal Methods and Software Engineering*, volume 4789 of *Lecture Notes in Computer Science*, pages 76–95. Springer Berlin Heidelberg, 2007.

[2] Rajeev Alur. Timed automata. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22. Springer Berlin Heidelberg, 1999.

[3] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, April 1994.

[4] Johan Bengtsson, W.O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Automated verification of an audio-control protocol using uppaal. *The Journal of Logic and Algebraic Programming*, 52 – 53(0):163 – 181, 2002.

[5] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer Berlin Heidelberg, 2004.

[6] Patricia Bouyer, Catherine Dufourd, Emmanuel Fleury, and Antoine Petit. Updatable timed automata. *Theoretical Computer Science*, 321(23):291 – 345, 2004.

[7] Alexandre David, Jacob Illum, Kim Larsen, and Arne Skou. *Model-Based Framework for Schedulability Analysis Using UPPAAL 4.1*, pages 93–119. CRC Press, 2011/12/27 2009.

[8] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149 – 1172, 2007.

[9] Heather J. Goldsby, Betty H.C. Cheng, and Ji Zhang. Amoeba-rt: Run-time verification of adaptive software. In Holger Giese, editor, *Models in Software Engineering*, volume 5002 of *Lecture Notes in Computer Science*, pages 212–224. Springer Berlin Heidelberg, 2008.

[10] J. Goossens and R. Devillers. Feasibility intervals for the deadline driven scheduler with arbitrary deadlines. In *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, pages 54 –61, 1999.

[11] Leo Hatvani, Paul Pettersson, and Cristina Seceleanu. Adaptive task automata: A framework for verifying adaptive embedded systems. In Juan Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, volume 7212 of *Lecture Notes in Computer Science*, pages 115–129. Springer Berlin Heidelberg, 2012.

[12] Ravindra Jejurikar, Cristiano Pereira, and Rajesh Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, pages 275–280, New York, NY, USA, 2004. ACM.

[13] Kim Larsen, Gerd Behrmann, Ed Brinksma, Ansgar Fehnker, Thomas Hune, Paul Pettersson, and Judi Romijn. As cheap as possible: Effcient cost-optimal reachability for priced timed automata. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 493–505. Springer Berlin Heidelberg, 2001.

[14] Kim G. Larsen and Yi Wang. Time-abstracted bisimulation: Implicit specifications and decidability. *Information and Computation*, 134(2):75 – 101, 1997.

[15] Y.-H. Lee, K.P. Reddy, and C.M. Krishna. Scheduling techniques for reducing leakage power in hard real-time systems. In *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, pages 105–112, July 2003.

[16] Marius Mikučionis, Kim Larsen, Jacob Rasmussen, Brian Nielsen, Arne Skou, Steen Palm, Jan Pedersen, and Poul Hougaard. Schedulability analysis using uppaal: Herschel-planck case study. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6416 of *Lecture Notes in Computer Science*, pages 175–190. Springer Berlin / Heidelberg, 2010.

[17] Klaus Schneider, Tobias Schuele, and Mario Trapp. Verifying the adaptation behavior of embedded systems. In *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, SEAMS '06, pages 16–22, New York, NY, USA, 2006. ACM.

[18] Fei Yu, Guoqiang Li, and Naixue Xiong. Schedulability analysis of multi-processor real-time systems using uppaal. In *Information Science and Engineering (ICISE), 2010 2nd International Conference on*, pages 1 –6, dec. 2010.