The Journal of Logic and Algebraic Programming 78 (2009) 402–416

Contents lists available at ScienceDirect

The Journal of Logic and Algebraic Programming

journal homepage: www.elsevier.com/locate/jlap

Schedulability of asynchronous real time concurrent objects[☆]

Mohammad Mahdi Jaghoori [a,*], Frank S. de Boer [a], Tom Chothia [a,b], Marjan Sirjani [c,d]

[a] CWI, SEN3, Science Park 123, Amsterdam, The Netherlands
[b] School of Computer Science, University of Birmingham, UK
[c] University of Tehran and IPM, Tehran, Iran
[d] Reykjavík University, Reykjavík, Iceland

ARTICLE INFO

Available online 20 February 2009

Keywords:
Actor model
Concurrent objects
Real time
Scheduling
Timed automata

ABSTRACT

We present a modular method for schedulability analysis of real time distributed systems. We extend the actor model, as the asynchronous model for concurrent objects, with real time using timed automata, and show how actors can be analyzed individually to make sure that no task misses its deadline. We introduce *drivers* to specify how an actor can be safely used. Using these drivers we can verify schedulability, for a given scheduler, by doing a reachability check with the UPPAAL model checker. Our method makes it possible to put a finite bound on the process queue and still obtain schedulability results that hold for any queue length.

1. Introduction

Schedulability analysis for a real time system consists of checking whether all tasks can be finished within their deadlines. We propose a modular method for schedulability analysis by extending the Actor model with real time, wherein actors can be analyzed individually. The Actor model [1,2] is the completely asynchronous setting of concurrent objects. Actors can send only asynchronous messages and have queues for receiving them. An actor has a message server (also called a method) for each message it can handle. We introduce a high-level actor modeling language where message servers are defined using timed automata [3]. Receiving a message schedules the corresponding method, i.e., puts it in the queue to be executed. A method is sequential and may in turn send messages. A message that the actor sends to itself is called a self call.

Actors are suitable for modular modeling, because of the asynchronous nature of communication and the encapsulation of computation (i.e., having no shared variables). This means that actors can be developed independently, and later put in the context of bigger systems. Nevertheless, for an actor to produce a specific service, messages should be sent to it in a correct sequence.

Our extension of the actor model is in line with this modular modeling. The message servers, which correspond to the methods that will be scheduled, are specified by means of timed automata [3]. A deadline is assigned to each message specifying the time before which the intended job should be accomplished. We allow each actor to define its own scheduling policy (rather than, for instance, assuming "First Come First Served (FCFS)" by default) with the condition that inserting a new message in the queue cannot preempt the currently running method. When a message is received, the scheduling policy determines where in the queue the message should be inserted.

☆ This work is partly funded by the European IST-33826 STREP project CREDO on Modeling and Analysis of Evolutionary Structures for Distributed Services.

* Corresponding author. Tel.: +31 20 5924299; fax: +31 20 5924199.
E-mail addresses: jaghouri@cwi.nl (M.M. Jaghoori), f.s.de.boer@cwi.nl (F.S. de Boer), t.chothia@cwi.nl (T. Chothia), msirjani@ut.ac.ir (M. Sirjani).

1567-8326/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jlap.2009.02.009

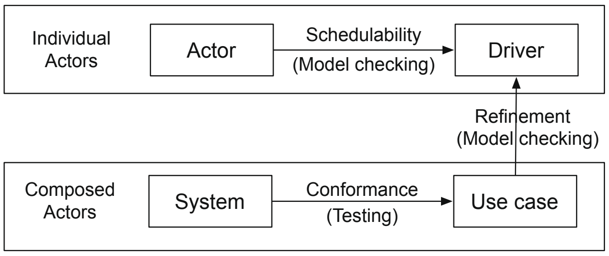

Fig. 1. Modular approach to schedulability analysis.

The modeler gives the possible timings for the arrival of messages to the actor (excluding self-calls) in another timed automaton called the *driver*. A driver is a most general specification of how an actor might be used. This driver also contains the deadlines of the incoming messages. Using drivers makes it possible to perform schedulability analysis for actors individually. From the actor's point of view, the driver describes the pattern of generating tasks in the actor. This makes a driver very much similar to a task automaton [4]; nevertheless, while running a task (i.e., a message server), the actor may in turn send messages to itself and generate other (internal) tasks. It may also be the case that an internal task should inherit the (remaining) deadline of the task generating it (called delegation). The actor is said to be schedulable if all the (external and internal) tasks are finished within their deadlines.

In our approach, an actor (given a specific scheduling strategy) is put together with its driver and analyzed for schedulability. Although an actor is allowed to have an unbounded queue, we can statically find an upper bound on the length of schedulable queues; hence, the behavior of the actor is finite. We show that this analysis is decidable by reducing it to the problem of reachability check for timed automata [5]. This way, one can analyze the actor with regard to different scheduling strategies, and find the best strategy. The analysis can be performed with the tools for timed automata verification, e.g., IF [6], RED [7] or OpenKronos [8]; we have chosen to use Uppaal [9] for our analysis.

One can use actors that are individually schedulable with respect to their drivers for making bigger components or systems. In principle, checking the schedulability of such a multi-processor system is subject to state-space explosion. In line with the aims of the Flacos workshop, we propose to use the 'design by contract' approach [10] based on automata [11]. A driver being a specification of the general conditions under which instances of the model are schedulable (given by the actor developer), a system developer has to provide the specific 'use-case' U of the actor in the given system. U is a timed automaton like a driver but which describes the actual use of an actor in a particular configuration of connections. Compatibility is defined as the inclusion of the timed traces of U in those of D, i.e., this particular usage is in the allowed usages. Since D is deterministic, trace inclusion is decidable [3,12]. If an actor is schedulable with respect to a driver D and D is compatible with a use-case U then the actor is also schedulable in the particular configuration of connections described in U.

In this paper, we focus on the maximal use of model checking techniques for schedulability analysis while avoiding state-space explosion, which is achieved by modular verification (depicted in Fig. 1). This includes analyzing an actor for schedulability with respect to its driver; and, checking whether a given use case of the actor is compatible with the general driver specification. This provides the means for modeling schedulable actors and ensuring their correct usage. The user of a schedulable actor may further need to check, as an extra step, whether the real system implementation conforms to the given use case. This step is not addressed in this paper, however, we envision that testing is appropriate for analyzing conformance of potentially big systems.

1.1. Related work

There has been lots of work on schedulability analysis of real time systems. Here we compare our work with some most relevant ones, focussing on different aspects of the work.

- We address schedulability at a modeling level, as in [4,13–15], whereas [16,17] are applied to programming languages. In the latter case, a given application is augmented with real time requirements (like deadlines) and automata are derived from code. In addition in TAXYS [16], an 'event handler' which is similar to our scheduler automata is synthesized automatically from the annotations of the program. In contrast, we start by using automata for modeling schedulers, actors and their drivers.
- For schedulability analysis, the task generation pattern should be provided. Tasks could be periodic specified with their inter-arrival times [13], while tasks are scheduled based on a resource management policy. Rate monotonic analysis techniques are extended in [14] for fixed priority systems in which tasks synchronize with the arrival of multiple events. Task automata, on the other hand, are used for describing non-uniformly recurring tasks [4]. Intuitively, in a task automaton, transitions may trigger an instance of a task, which will be put into the queue for execution, according to a given scheduling strategy, e.g., FPS (fixed priority scheduling) or EDF (earliest deadline first) [18]. Although [16,17] also use automata, they do not address decidability. Fersman et al. [4] have studied the decidability of the problem of checking schedulability for task automata for different settings. For example, when using a non-preemptive scheduler, or when

tasks have fixed computation times, schedulability for a task automaton can be reduced to checking for reachability in timed automata, and is therefore decidable [5].
The drivers are similar to task automata in the sense that they model the pattern of task generation. However, in our framework tasks are specified as timed automata (rather than just execution times) and can therefore trigger other (internal) tasks during execution. Such an internal task may inherit the (remaining) deadline of the task generating it (called delegation). The internal tasks are not captured in the driver, because their arrival depends on the scheduling of the parent tasks, which in turn depends on the selected scheduling strategy.

- Schedulability has usually been analyzed for a whole system running on a single processor, whether at modeling [4,13] or programming level [16,17]. We address distributed systems where each actor has a dedicated processor and scheduling policy. We propose a modular approach to schedulability analysis similar to the ideas of modular model checking [19] (cf. next bullet). The work in [14] is also applicable to distributed systems but is limited to rate monotonic analysis. A modular actor language addressing schedulability is proposed in [15]. This approach is modular in the sense that the untimed specification of the actors, and the timing constraints (specified separately) can be reused. However, they still analyze a complete system, rather than individual actors. Furthermore, a deadline in their framework includes only the time until an event is received. Hence, their approach cannot address complications like delegation of a task to subtasks.
- In our approach, drivers are key to modularity. A driver models the most general message arrival pattern for an object. The driver can be viewed as a contract as in ‘design by contract’ [10] or as a most general assumption in modular model checking [19] (based on assume-guarantee reasoning); schedulability is guaranteed if the real use of the actor satisfies this assumption. In the literature, a model of the environment is usually the task generation scheme in a specific situation. However, a driver in our analysis covers all allowable usages of the actor, which in turn adds to the modularity of our approach; every use of the actor foreseen in the driver is verified to be schedulable. Comparatively, for instance in TAXYS [16], this model of the environment can also be general enough to cover all uses of the program but it is used to analyze a complete program and is not used modularly.

1.2. Paper structure

In Section 2, we provide the grounds for the approach by explaining the actor language and timed automata. Section 3 informally demonstrates the modular approach to modeling schedulable real time actors. The schedulability analysis method is formally investigated in Section 4. An overview of how to check compatibility is given in Section 5. We explain how to perform the analyses in practice using Uppaal in Section 6. Section 7 concludes the paper.

2. Preliminaries

2.1. Timed automata

In this section, we define timed automata syntax and semantics as it forms the basis of the analyses in the paper.

Definition 1 (*Timed Automata*). Suppose $\mathcal{B}(C)$ is the set of all clock constraints on the set of clocks C. A timed automaton over actions Σ and clocks C is a tuple $\langle L, l_0, \longrightarrow, I\rangle$ representing

- a finite set of locations L (including an initial location l_0);
- the set of edges $\longrightarrow \subseteq L \times \mathcal{B}(C) \times \Sigma \times 2^C \times L$; and,
- a function $I : L \mapsto \mathcal{B}(C)$ assigning an invariant to each location.

A location can be marked *urgent* which is equivalent to resetting a fresh clock x in all of its incoming edges and adding an invariant $x \leq 0$ to the location.

An edge is written as $l \xrightarrow[g\,,\,r]{a} l'$. It means that action ‘$a$’ may change the location l to l' by resetting the clocks in r, if clock constraints in g (as well as the invariant of l') hold. In addition, when a location is marked as *urgent*, intuitively, it means that the automaton cannot spend any time in that location [9].

Definition 2 (*Timed Automata Semantics*). A timed automaton defines an infinite labeled transition system whose states are pairs (l, u) where $l \in L$ and $u : C \to \mathbb{R}_+$ is a *clock assignment*. We denote by $\mathbf{0}$ the assignment mapping every clock in C to 0. The initial state is $(l_0, \mathbf{0})$. There are two types of transitions:

- action transitions $(l, u) \overset{a}{\to} (l', u')$ where $a \in \Sigma$, if there exists $l \xrightarrow[g\,,\,r]{a} l'$ such that u satisfies the guard g, u' is obtained by resetting all clocks in r and leaving the others unchanged and u' satisfies the invariant of l';
- delay transitions $(l, u) \overset{d}{\to} (l, u')$ where $d \in \mathbb{R}_+$, if u' is obtained by delaying every clock for d time units and for each $0 \leq d' \leq d$, u' satisfies the invariant of location l.

Deterministic timed automata. A timed automaton is called *deterministic* if and only if for each $a \in \Sigma$, if there are two edges from l labeled by the same action $l \xrightarrow[g\ ,\ r]{a} l'$ and $l \xrightarrow[g'\ ,\ r']{a} l''$ then the guards g and g' are disjoint (i.e., $g \wedge g'$ is unsatisfiable).

Variables. As accepted in Uppaal, we allow defining variables of type boolean and bounded integers for each automaton. Variables can appear in guards and updates. The semantics of timed automata changes such that each state will include the current values of the variables as well, i.e., (l, u, v) with v a variable assignment. An action transition $(l, u, v) \overset{a}{\to} (l', u', v')$ additionally requires v and v' to be considered in the corresponding guard and update.

Networks of timed automata. A system may be described as a collection of timed automata communicating with each other. In these automata, the action set is partitioned into input, output and internal actions. The behavior of the system is then defined as the parallel composition of those automata $A_1 \parallel \cdots \parallel A_n$. Semantically, the system can delay if all automata can delay and can perform an action if one of the automata can perform an internal action or if two automata can synchronize on complementary actions (inputs and outputs are complementary).

2.2. The modeling language for asynchronous concurrent objects

The Actor model was introduced by Hewitt [1] as an agent-based language; and, later developed by Agha [2] into a concurrent object-based model. Actors are units of distribution and concurrency, and have encapsulated states and behavior. They communicate via asynchronous (non-blocking) message passing, and the arrival of the messages is guaranteed. Due to their intrinsic asynchrony, actor-based languages can be used for modeling classical concurrent and distributed applications, as well as, modern web services. Actors have local variables, but no shared variables. An actor has a dedicated processor. Our approach can be easily adapted to any modeling platform with the above-mentioned characteristics for concurrent objects, e.g., Rebeca [20], Creol [21], etc. To have a concrete method, we need to have a more detailed language mentioned next.

We define reactive classes as templates for actors. A reactive class defines a message server for each message it can handle. A message sever (also called a method) is defined using a timed automaton, which may send messages. There is at least a message server '*initial*' in each reactive class, which is responsible for initialization. A reactive class can have known actors, which serve as place holders for the actors that can communicate with instances of that class. When no ambiguity arises we may use the term actor instead of reactive class.

Each actor has an unbounded queue for storing its incoming messages. The message at the head of the queue determines the method to be executed next. At the initial state, a number of actors are created statically, and an '*initial*' message is implicitly put in their queues. The model continues running as actors send messages to each other and the corresponding methods are executed.

3. Modeling real time actors and drivers

In this section, we informally describe our modular method of modeling real time systems. We demonstrate the approach by modeling a *mutual exclusion handler*, called `MutEx` in the following. Section 4 provides the formal actor model.

For every actor, we model each message server with a timed automaton. These automata are parameterized in the identity of the actor itself, and the identifiers of the actors communicating with it (namely, its known actors). For instance, a `MutEx` may communicate with the two objects on its left and right which try to enter a critical section. Each actor is coupled with another timed automaton, called a *driver*, specifying in the most general terms the context in which the actor can be safely used. Next section describes formally how a driver helps us analyze each actor individually. In Section 6, we will show how to model these automata in Uppaal.

Fig. 2 depicts the automata for the message servers of the `MutEx`, called method automata. These automata are parameterized in `self`, `Left` and `Right`. An action `Left.permitL(15)!` means a message `permitL` is sent to the actor representing the `Left` known actor, and a deadline 15 is assigned to it. We abstract from computation, which is represented only by a time delay. However, some variables (e.g., 'taken' in `MutEx`) may be needed for correct behavior of the actor. This is explained later in this section.

A `MutEx` is initially not taken (the 'taken' variable is set to false in `initial` method). The object on the left (specified as `Left`) may ask for the `MutEx` by sending reqL. In response, the `MutEx` sends a permitL back, if the `MutEx` is not already taken. Similarly, the message reqR may be sent by `Right`. If the `MutEx` is taken, the request is put back in queue by a self call. A release message can be sent by either object on the left or right.

Fig. 3 shows the specification of the driver automaton for `MutEx`. The modeler specifies the correct way of using each actor in its driver, i.e., the driver is the highest level abstraction of all environments in which the actor instances can be used. In other words, a driver characterizes the expected pattern of incoming messages. The `MutEx` driver includes all possibilities of receiving two requests followed by two releases.

The drivers should include the expected timing information for the arrival of messages. Specifically, a time interval between messages is necessary to have a schedulable actor. The given `MutEx` driver keeps track of the time since every request, and expects that the corresponding release arrives no sooner than MIN_REL time units. The driver assumes that

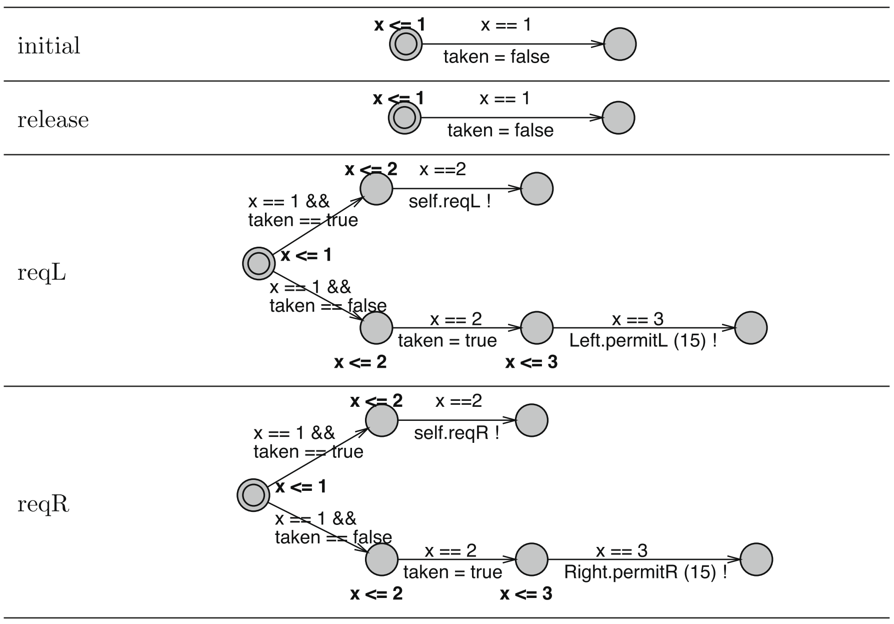

Fig. 2. Specification of a mutual exclusion handler (`MutEx`).

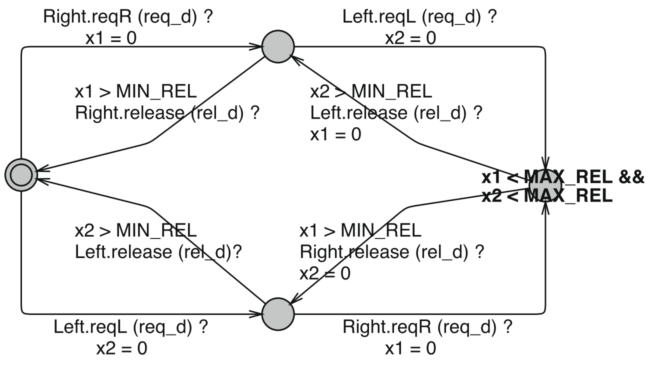

Fig. 3. A driver automaton specification for `MutEx`.

the first request is always granted (because `MutEx` is not taken initially). If the next request arrives before a release, it will be waiting for the `MutEx` to be released. Whenever a request is pending, a release must arrive before MAX_REL time units, otherwise the pending request will miss the deadline. By iterating the schedulability analysis (explained in the next section) this interval can be refined so that to indicate the minimum requirement.

The timing constraints in method automata show the amount of computation needed for each step. The invariants on the locations of method automata together with the guards on edges, require that the send operations succeed; otherwise, time cannot progress. This is necessary in order to model the asynchronous nature of the message passing.

The send and receive actions (in method and driver automata) should be assigned a deadline. Self calls inherit the remaining deadline of the parent task (called delegation), unless an explicit deadline is assigned (called invocation). Delegation may be used when a task is split over two or more methods.

Another common scenario for delegation happens when a task (say handling a 'request' in a `MutEx`) cannot be accomplished immediately (say because the `MutEx` is already 'taken'). Therefore, it needs to make a self call, which must terminate within the original deadline. In other words, the original deadline of a request requires a bound on the time until the request is granted (i.e., a permit is sent back). Variables ('taken' in this example) are used to keep the current state of the actor and thus to bound such delegation loops. In modeling, we cannot abstract from such variables, because it would let the loops in delegation to continue infinitely. This results in nonschedulability, because the (inherited) deadline becomes smaller every time.

4. The timed actor model

In this section, we present our formal model of actors. Each actor has a set of method names, which represent the methods the actor provides. Each of these methods is represented by a timed automaton. An actor also needs a queue for storing incoming messages and a scheduling strategy for determining the order of executing messages. Upon receiving a message, the corresponding automaton, representing the task to be executed, is inserted into the process queue. Once the currently executing automaton reaches a final state, the next task in the queue is started.

A model of an actor consisting of only methods cannot be analyzed for schedulability on its own, because there are an infinite number of ways in which the methods could be called. Therefore, we add a *driver* that defines how the methods could be called by the outside environment. Furthermore, we show in Section 4.3 that we may put a finite bound on the queue and still derive schedulability results that hold for any queue length.

We assume a finite global set of method names $\mathcal{M}$ (corresponding to the messages that can be sent and received).

Definition 3 (*Driver*). A driver D with a set of method names $M_D \subseteq \mathcal{M}$ is a deterministic timed automaton $(L_D, l_D, \rightarrow_D, I_D)$ over the alphabet $\Sigma_D = \{m(d)?| m \in M_D \wedge d \in \mathbb{N}\}$ and clocks C_D.

The driver can be viewed as a high-level specification of how an actor can be used. It specifies the (acceptable) input behavior of the actor, and abstracts from queue and scheduling strategy. An action $m(d)?$ represents a message m sent to the actor by the environment with the deadline d. A correct actor implementation should be able to finish an incoming call $m(d)?$ before d time units.

Definition 4 (*Actor*). An actor R matching the driver D is defined as a set $\{(m_1, A_1), \ldots, (m_n, A_n)\}$ where:

- $M_R = \{m_1, \ldots, m_n\} \subseteq \mathcal{M}$ is a set of method names such that $M_D \subseteq M_R$; and,
- for all i, $1 \leq i \leq n$, $A_i = (L_i, l_{0_i}, \rightarrow_{A_i}, I_i)$ is a timed automaton representing method m_i with the alphabet $\Sigma_i = \{m!| m \in M_R\} \cup \{m(d)! \mid m \in \mathcal{M} \wedge d \in \mathbb{N}\}$ and clocks C_i. We also write $\rightarrow_{m_i}$ for $\rightarrow_{A_i}$. The method A_1 is considered to be the *initial* method and it is given a preassigned deadline d.

Method automata only send messages while computations are abstracted into time delays. Receiving messages (and buffering them) is handled by a scheduler defined next. Sending a message $m \in M_R$ is called a self call. As explained in the previous section, self calls modeling delegation are not statically assigned a deadline; in this case, they inherit the (remaining) deadline of the task that triggers them. Delegation implies that the internal task is in fact the continuation of the parent task. The condition $M_D \subseteq M_R$ requires that a class should at least provide method implementation for handling all messages it may receive according to its driver.

The driver does not capture internal tasks triggered by self calls. In order to analyze the schedulability of an actor, one needs to consider both the internal tasks and the tasks triggered by the environment (the driver). We construct the *behavior automaton* of an actor by executing the automata representing the message servers as controlled by the driver. First, we need to define formally a model of the queue and scheduler.

4.1. Scheduler

An actor needs a queue for storing incoming messages before they are scheduled to be executed. In this section, we give the formal definition of a queue and scheduling strategies. Our scheduler model is inspired by and extends the ideas of task automata [4]. Tasks, being specified using timed automata, may perform self calls, which need to be handled by the scheduler, whereas in task automata tasks are just modeled with their execution time. We also need to support inheriting deadlines for delegation.

For each task, a queue needs to store the method name and its deadline. Furthermore, it needs a clock to keep track of the time since the task is triggered. This enables us to check if a deadline is missed. In theory a queue can be unbounded. We show in Section 4.3 that we may put a finite bound on the queue and still derive schedulability results that hold for any queue length. The first task in the queue is the currently running task.

Definition 5 (*Queue*). A queue q with an upper bound *MAX* is a list of at most *MAX* tasks together with a set C_q of *MAX* clocks. Each task is written as $m(d, c)$ where m is a method name, $d \in \mathbb{N}$ is its deadline and $c \in C_q$ keeps track of how long the task has been in the queue. The deadline of m expires when $c > d$.

To construct the behavior automaton, a scheduling strategy should be defined for the actor, in terms of a scheduler function. A scheduler is used to insert tasks into the queue. Typically the scheduler could dynamically examine the remaining time of each task in the queue. However, during construction of the behavior automata, the values of the queue clocks are not known. Therefore, the scheduler function returns the set of all possibilities for putting the new task in the queue depending on different clock values. The function also assigns an unused queue clock to the new task.

Definition 6 (*Scheduler Function*). Given a queue q with clocks C_q and a method name m with deadline d, a scheduler function '$sched(q, m(d))$' returns a set of triples $\{(G, c, q')\}$, where

- G is a guard on clocks in C_q (possibly based on d);
- $c \in C_q$ is a clock not used (i.e., not assigned to any tasks) in q; and,
- q' is a queue with the clocks C_q and represents the queue q after inserting $m(d, c)$ in a particular position as implied by the guard G.

We define an overloading of the scheduler function as $sched(q, m(d, c))$ that inserts a task into the queue using a given clock c. By reusing the deadline and the clock already assigned to a task in the queue, we can model inheriting the deadline. In the case of delegation, the clock assigned to the currently running task is reused.

A scheduler function is *preemptive* if it can place the new task in the first position. Recall that the first task in the queue is the task that is currently running. In this paper we only consider non-preemptive schedulers. Section 4.4 briefly discusses the decidability of preemptive scheduling.

Example 7. Consider a queue $q = [m_1(d_1, c_1), \ldots, m_k(d_k, c_k)]$ with the set of clocks C_q. A 'first come first served' scheduler would always put the new job at the back of the queue. This scheduler would not impose any constraints:
$sched\_FCFS(q, m(d)) = \{(true, c, [m_1(d_1, c_1), \ldots, m_k(d_k, c_k), m(d, c)])\}$
where $c \in C_q$ is not assigned to any task in q.

An 'earliest deadline first' scheduler would insert tasks into the queue based on the remaining deadlines of the existing queue members:

$$
\begin{aligned}
&sched\_EDF(q, m(d)) = \\
&\quad \{(d < d_2 - c_2, c, [m_1(d_1, c_1), m(d, c), m_2(d_2, c_2), \ldots, m_k(d_k, c_k)]), \\
&\quad (d_2 - c_2 \le d < d_3 - c_3, c, [m_1(d_1, c_1), m_2(d_2, c_2), m(d, c), \ldots, m_k(d_k, c_k)]), \\
&\quad \ldots \\
&\quad (d_{k-1} - c_{k-1} \le d < d_k - c_k, c, [m_1(d_1, c_1), m_2(d_2, c_2), \ldots, m(d, c), m_k(d_k, c_k)]), \\
&\quad (d_k - c_k \le d, c, [m_1(d_1, c_1), m_2(d_2, c_2), \ldots, m_k(d_k, c_k), m(d, c)])\}
\end{aligned}
$$

where $c \in C_q$ is not assigned to any task in q. The scheduler function cannot reorder the queue, so here we assume as an invariant that the tasks already in the queue are in the right order.

In our implementation, we will use a timed automaton to act as both the queue and the scheduler function (cf. Section 6).

4.2. Actor behavior model

Provided a scheduling strategy, an actor can be analyzed in the context of its driver. We will show in the next subsection that we can put a finite bound on the queue length and derive schedulability results that hold for any queue length.

Definition 8 (*Behavior Automaton*). Suppose Q is the domain of all queues with upper bound MAX and using the clocks in C_q. Given a scheduler function $sched$, the behavior automaton of an actor $R = \{m_1 : A_1, \ldots, m_n : A_n\}$ with the driver D is a timed automaton $H = (L_H, l_H, \rightarrow_H, I_H)$ over the alphabet Σ_H and clocks C_H:

- $\Sigma_H = \{m(d)! | m \notin M_R\} \cup \{m(d)? | m \in M_D\} \cup \{m | m \in M_R\}$, where $d \in \mathbb{N}$ denotes a deadline.
- $C_H = C_D \cup \left(\bigcup_{i \in [1..n]} C_i\right) \cup C_q$, where C_i and C_D are the clocks for A_i and D, respectively, and C_q is the set of queue clocks.
- $L_H = \{error\} \cup \left(\left(\bigcup_{i \in [1..n]} L_i\right) \times L_D \times Q\right)$, where L_D and L_i are the sets of locations of D and A_i, respectively.
- The `initial` location l_H is $(l_D, start(A_1), [m_1(d, c)])$, where l_D is the initial location of D, A_1 is the automaton for the '`initial`' method (corresponding to m_1), d is the initial deadline and $c \in C_q$.
- The edges $\rightarrow_H$ are defined with the rules in Fig. 4.
- The location invariants are defined as $I(l, l_d, q) = I(l) \wedge I(l_d)$ and $I(error) = true$. Furthermore, the locations $(l, l_d, [T_1, \ldots, T_k])$ such that $l \in final(T_1)$ and $k \ge 2$ are marked as urgent locations.

In Fig. 4, functions $start(A)$ and $final(A)$ give the initial location of A and the set of locations in A with no outgoing transitions, respectively. Each location of the behavior automaton is written as (l, l_d, q), where q is the queue, l_d is the current location of the driver and l is the current location of the method being executed, i.e., the automata corresponding to the first element of the queue. An edge of an automaton A with action a, guard g and update r is shown in this figure as $\xrightarrow[g\ ;\ r]{a}_A$. Finally, k shows the number of tasks in the queue.

When the driver allows receiving a message, or when a self call is made, the message is scheduled into the queue. In the case of delegation the clock of the currently running task is reused for the new task so that it inherits the remaining deadline. The *internal step* rule captures other transitions in a method.

When a task terminates the next task in the queue is started, unless there is no other task in the queue. In the latter case, context switch is not performed. Instead, this terminated task is exempted from the *missed deadline* rule. As soon as a new task is added to the queue, the context switch rule is enabled. Notice that the context switch rule is immediately executed when enabled, due to the urgency of the source location (cf. last bullet in Definition 8).

$$(l, l_d, [T_1, \ldots, T_k]) \xrightarrow[g \wedge G\ ;\ X, c_i := 0]{m(d)?}_H (l, l'_d, q') \qquad [receive]$$
$$\quad \textit{if } (l_d \xrightarrow[g\ ;\ X]{m(d)?}_D l'_d) \ \ \&\ \ k \le MAX \ \ \&$$
$$\qquad (G, c_i, q') \in sched([T_1, \ldots, T_k], m(d))$$

$$(l, l_d, [T_1, \ldots, T_k]) \xrightarrow[g \wedge G\ ;\ X, c_i := 0]{m}_H (l', l_d, q') \qquad [self\ \ call :$$
$$\quad \textit{if } (l \xrightarrow[g\ ;\ X]{m(d)!}_{T_1} l') \ \ \&\ \ m \in M_R \ \ \&\ \ k \le MAX \ \ \& \qquad invocation]$$
$$\qquad (G, c_i, q') \in sched([T_1, \ldots, T_k], m(d))$$

$$(l, l_d, [m_1(d, c), T_2, \ldots, T_k]) \xrightarrow[g \wedge G\ ;\ X]{m}_H (l', l_d, q') \qquad [self\ \ call :$$
$$\quad \textit{if } (l \xrightarrow[g\ ;\ X]{m!}_{m_1} l') \ \ \&\ \ m \in M_R \ \ \&\ \ k \le MAX \ \ \& \qquad delegation]$$
$$\qquad (G, c, q') \in sched([m_1(d, c), T_2, \ldots, T_k], m(d, c))$$

$$(l, l_d, [T_1, \ldots, T_k]) \xrightarrow[g\ ;\ X]{m(d)!}_H (l', l_d, [T_1, \ldots, T_k]) \qquad [external\ \ call]$$
$$\quad \textit{if } (l \xrightarrow[g\ ;\ X]{m(d)!}_{T_1} l') \ \ \&\ \ m \notin M_R \ \ \&\ \ k \le MAX$$

$$(l, l_d, [T_1, \ldots, T_k]) \xrightarrow[g\ ;\ X]{}_H (l', l_d, [T_1, \ldots, T_k]) \qquad [internal\ \ step]$$
$$\quad \textit{if } (l \xrightarrow[g\ ;\ X]{}_{T_1} l') \ \ \&\ \ k \le MAX$$

$$(l, l_d, [T_1, T_2, \ldots, T_k]) \xrightarrow[C_l := 0]{}_H (l', l_d, [T_2, \ldots, T_k]) \qquad [context\ \ switch]$$
$$\quad \textit{if } l \in final(T_1) \ \ \&\ \ 2 \le k \le MAX \ \ \&$$
$$\qquad C_l = local\_clocks(T_2) \ \ \&\ \ l' = start(T_2)$$

$$(l, l_d, [T_1, \ldots, T_k]) \longrightarrow_H Error \qquad [queue\ \ overflow]$$
$$\quad \textit{if } (k > MAX)$$

$$(l, l_d, [\ldots, m_i(d_i, c_i), \ldots]) \xrightarrow[(c_i > d_i)]{}_H Error \qquad [missed\ \ deadline]$$
$$\quad \text{for every } 1 \le i \le k \text{ unless } k = 1 \text{ and } m_1 \text{ is terminated}$$

Fig. 4. Calculating the edges of the behavior automaton.

4.3. Schedulability analysis

We build the behavior automata for an actor based on a given driver. We can then check if the actor is schedulable for the environments that match the driver.

Definition 9 (*Schedulable*). A behavior automata is schedulable if the deadlines of the tasks in the queue (including the currently executing task) never expire, i.e., there is no reachable state such that a clock of one of the tasks of the queue is greater than the deadline.

Checking schedulability with an unrestricted queue length might require us to check an infinite system. Whereas artificially fixing the queue may lead to false negatives. Luckily, for a given actor, we can determine an upper bound on the queue length of schedulable systems before constructing the behavior automaton.

Lemma 10. *If a behavior automata is schedulable then it does not put more than $\lceil d_{max}/b_{min} \rceil$ tasks into the queue, where d_{max} is the longest deadline for any methods called on any transition of the automata (A_i or D) and b_{min} is the shortest termination time of any of the method automata (the A_i's).*

Proof. Assume that the queue length reaches $\lceil d_{max}/b_{min} \rceil + 1$. We show that in this case, the actor is not schedulable. All methods are called with a deadline, and delegated deadlines are equal to, or less than, the original deadlines. Therefore, all tasks in the queue, including the last task, must have a deadline less than or equal to d_{max}. The scheduler can only insert tasks into the queue, it cannot reorder the queue; therefore, $\lceil d_{max}/b_{min} \rceil + 1$ tasks must be executed before the last task in the queue terminates. Each of these tasks takes at least time b_{min}; therefore, the final task terminates in time greater than $(\lceil d_{max}/b_{min} \rceil + 1) \times b_{min} > d_{max}$ and so misses its deadline. □

We can calculate the best case runtime for timed automata as shown by Courcoubetis and Yannakakis [22]. The longest deadline can be found by a simple static search of all the transitions.

Theorem 11. *A behavior automaton is schedulable if, and only if, it cannot reach the error state with a queue length of $\lceil d_{max}/b_{min} \rceil$.*

Proof. A behavior automaton can only reach the *error* state by using the [*queue overflow*] rule or the [*missed deadline*] rule. The guard of the [*missed deadline*] rule implies that a deadline has been missed and therefore the behavior automaton is not schedulable. Lemma 10 implies that if the [*queue overflow*] rule was used then the behavior automaton is also not schedulable.

In the other direction, if the behavior automata cannot reach the *error* state then the guard of the [*missed deadline*] rule must never hold, so no deadlines are missed. □

As a result of this theorem, we can check the schedulability of behavior automata by checking the reachability of the *error* state using the UPPAAL model checker.

4.4. Discussion on preemptive scheduling

We have focused our work on "non-preemptive" schedulers, i.e., schedulers that finish one task and then pick the next task to run using a given policy. Preemptive schedulers, on the other hand, will interrupt and switch between running tasks. It is exactly this switching that gives the effect of truly concurrent processes on a single CPU machine.

In the most general case, testing the schedulability of a preemptive scheduler is undecidable. Fersman et al. prove this for Task Automata [4] and their proof may be applied to our framework. In short they show that it is possible to implement a 2-counter machine by encoding the counters using clocks in the interval $[0, 1]$. Their system repeatedly loops and the value of the counter is taken to be 2^{1-c} where c is the value of the clock at the end of each loop. Preemption allows the value c to be arbitrarily halved and doubled so that the counter may be incremented and decremented.

The work by Kloukinas and Yovine [17] uses discrete-time automata to handle preemption and proposes a methodology to cope with the state-space explosion due to it. In the dense model of time, however, the most general preemptive scheduler is undecidable because the amount of time between preemptions can be any value in the real domain. If we are willing to restrict this interval and only allow preemption every t seconds then schedulability is decidable for every $t > 0$. This is enough to accurately model real systems; t could, for instance, be set to equal the target machine's clock speed. Schedulability is decidable in this framework because we can break up any of the method automata into a number of smaller automata each of which runs for t time units and then adds the next part of the method to the queue, so giving the scheduler the opportunity to preempt them. Unfortunately this would lead to the size of our system increasing exponentially as t decreases. We hope that a full investigation of preemption will make promising further work.

5. Compatibility checking

Individually schedulable actors can be used as off-the-shelf components in making bigger components and systems. Checking the whole system for schedulability is subject to state space explosion, given the fact that each actor has its own processor and scheduling strategy. Using the analysis method in the previous section, one can develop in a modular manner actors that are schedulable when their actual use is compatible with their drivers. In this section, we briefly discuss one way to check for such a compatibility in the context of the 'design by contract' approach [10] based on automata [11].

In this approach, a developer of an actor model provides a driver as a specification which describes the most general conditions under which instances of the model are schedulable. A system developer (as opposed to the actor developer) instantiates actor models in the context of a particular configuration of connections. In order to check whether each actor

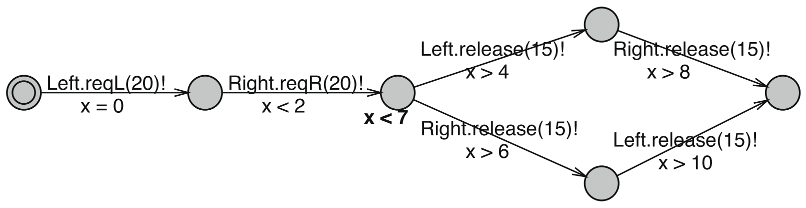

Fig. 5. A use-case for `MutEx`.

instance of such a system is schedulable, a system developer has to provide a 'use-case' which is a timed automaton like a driver but which describes the actual use of an actor in a particular configuration of connections.

Compatibility between a driver D and a use-case U is defined in terms of *trace-inclusion*, that is, the timed traces of U are contained in those of D, denoted $U \sqsubseteq D$. Clearly, if an actor is schedulable with respect to a driver D and D is compatible with a use-case U then the actor is also schedulable in the particular configuration of connections described by U.

We define the drivers to be deterministic in Definition 3. This does not reduce the expressiveness, because what is important for schedulability analysis is the timed traces in the driver. This implies the decidability of the compatibility check (see for example [12] for a way of checking trace inclusion in UPPAAL).

5.1. Example

Recall the `MutEx` actor modeled in Section 3. A classic example where the `MutEx` actor can be used is as a fork in the dining philosophers example. Fig. 5 shows a use case of `MutEx` in such a system. This specifies that the objects on left and right of the `MutEx` ask for entering the critical section exactly once, and in the order specified. Depending on which gets the `MutEx` first, they may release in different orders. It is assumed that the objects release the `MutEx` no earlier than 4 time units. The invariant on the location after reqR shows an upper bound on the time the `MutEx` will be released.

Inclusion of the traces of this use-case in those of the `MutEx` driver can be done in UPPAAL as proposed in [12] when the approach is extended to incorporate the deadlines into account. It is explained in the next section how to consider deadlines in this check. If the values of min and max time the `MutEx` is released does not match MIN_REL and MAX_REL in the driver (see Section 3), the use-case will not be compatible.

6. Using UPPAAL for analysis

In this section, we explain how to use UPPAAL [9] to perform schedulability analysis for a given actor. The details are explained with the help of the `MutEx` example from Section 3. We model drivers, methods implementations and schedulers (which in turn include a queue) with timed automata. The actor behavior model (corresponding to behavior automaton) can be generated by making a network of these timed automata in UPPAAL.

Communication. We use two channels invoke and delegate for sending messages. The channel invoke has three dimensions (parameters), the message name, the sender and the receiver, e.g., invoke[release][self][Left]! replaces Left.release in the driver of `MutEx`. Notice that in drivers, the actions are modeled as outputs (! instead of ?), which is necessary for handling deadlines explained next. In method automata, by setting both sender and receiver as self, one can invoke a self call (when a deadline is to be given). The delegate channel is used for delegation. The self call made using the delegate channel inherits the deadline of the currently running task (it is taken care of by the scheduler automaton). Since a delegation is used only for self calls, no sender is specified (it has only two parameters). Fig. 6 shows the driver and the method reqL of `MutEx` modeled in UPPAAL (cf. Section 3).

Deadlines. We take advantage of the fact that when two edges synchronize, UPPAAL performs the updates on the emitter before the receiver. Hence we can use a global variable deadline. The emitter sets the deadline value into this variable which is read by the receiver. The receiver, however, cannot use this deadline value in its guard, as guards are evaluated before updates.

6.1. Modeling the scheduler

A scheduler function (Definition 6) can be implemented as a scheduler automaton. This automaton also contains a queue as in Definition 5. Fig. 7 shows the general structure of a scheduler automaton. This general picture does not specify any specific scheduling strategy. Unlike the Definition 6, the scheduler automata applies the scheduling strategy at dispatch time instead of insertion time, but the resulting behavior is the same. The reason is to enable using deadlines in the strategy. As explained in the previous subsection, the deadline value cannot be used (in the guard) on the same transition where a message is received.

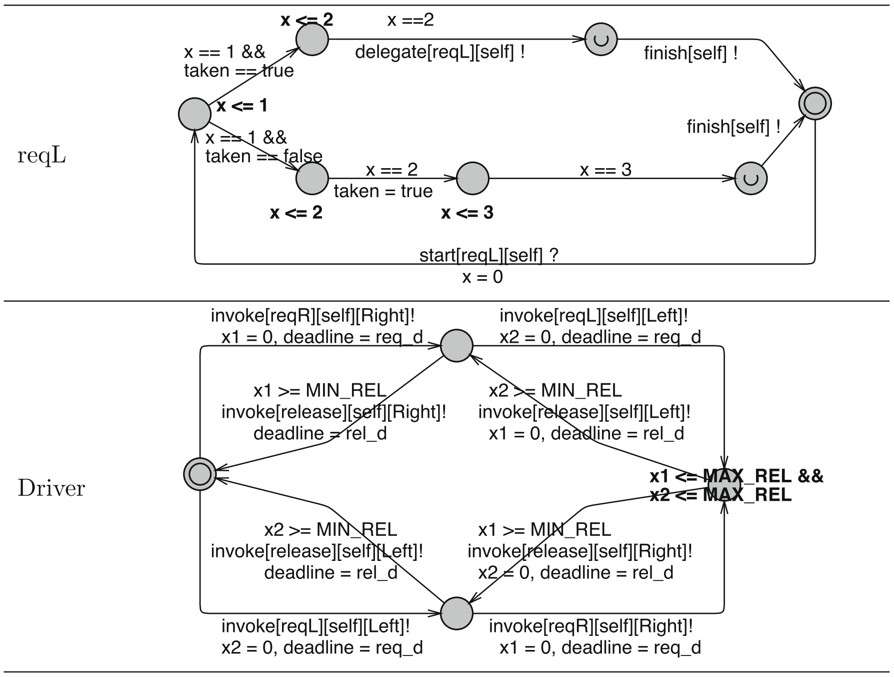

Fig. 6. Modeling method and driver automata in UPPAAL.

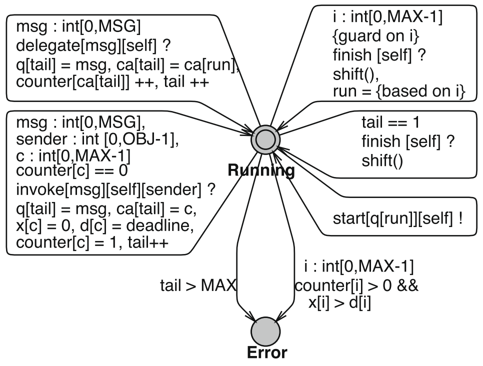

Fig. 7. A general scheduler automaton.

Queue. The triple $m(d, x)$ for each task in the queue is modeled using the arrays q, d and x, respectively. The array ca shows the clock assigned to each message (task), such that ‘d[ca[i]] – x[ca[i]]’ represents the remaining deadline of q[i] at any time. counter[i] holds the number of tasks using clock x[i]. A clock is free if its counter is zero. When delegation is used, the counter becomes greater than one.

Input-enabledness. A scheduler for a class R should allow receiving any message in M_R at any time. In Fig. 7, there is an edge (left down in the picture) that allows receiving a message on the invoke channel (from any sender). To allow any message and sender, ‘select’ expressions are used. The expression msg : int[0,MSG] nondeterministically selects a value between 0 and MSG for msg. This is equivalent to adding a transition for each value of msg. Similarly, any sender (sender : int[0,OBJ−1]) can be selected. This message is put at the tail of the queue (q[tail] = msg), and a free clock (counter[c] == 0) is assigned to it (ca[tail] = c), and the deadline value is recorded (d[c] = deadline). The synchronization between this transition and the driver (resp. method automata) corresponds to the rule *receive* (resp. *self call* : *invocation*) in Fig. 4.

A similar transition accepts messages on the delegate channel. In this case, the clock already assigned to the currently running task (parent task) is assigned to the internal task (ca[tail] = ca[run]). In a delegated task, no sender is specified (it is always self). The variable run shows the index of the currently running task in the queue (which is not necessarily the first task). This handles the rule *self call* : *delegation*.

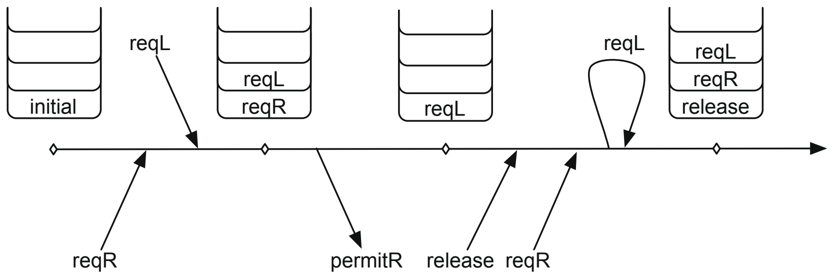

Fig. 8. A possible trace for a fork based on FIFO scheduling policy.

Context-switch. It is performed in two steps (without letting time pass). When a method is finished (synchronizing on finish channel), it is taken out of the queue (by shift()). If it is not the last in the queue, the next method to be executed should be chosen based on a specific scheduling strategy (by assigning the right value to run). For a *concrete* scheduler, the guard and update of run should be well defined. If run is always assigned 0 during context switch, the automaton serves as a First Come First Served (FCFS) scheduler. An Earliest Deadline First (EDF) scheduler can be encoded using a guard like:

```
i < tail && i != run &&
forall (m : int[0,MAX-1])
  (m == run) || (x[ca[i]] - x[ca[m]] >= d[ca[i]] - d[ca[m]])
```

and i will show the task with the smallest remaining deadline. Notice that $x[a] - x[m] \geq d[a] - d[m]$ is equivalent to $d[m] - x[m] \geq d[a] - x[a]$. The rest ensures that an empty queue cell (i < tail) or the currently finished method (run) is not selected.

If the currently running method is the last in the queue, nothing needs to be selected (i.e., if tail == 1 we only need to shift). The second step in context-switch is to start the method selected by run. Having defined start as an urgent channel, the next method is immediately scheduled (if queue is not empty).

Error. The scheduler automaton moves to the Error state if a queue overflow occurs (tail > MAX) or a deadline is missed (x[i] > d[i]). The guard counter[i] > 0 checks whether the corresponding clock is currently in use, i.e., assigned to a message in the queue.

6.2. Schedulability analysis of `MutEx`

With such an automated analysis process, it is easy to study the effect of different scheduling strategies on schedulability. Fig. 8 shows a possible scenario in which 'First Come First Served (FCFS)' strategy for a `MutEx` may cause starvation, i.e., makes `MutEx` non-schedulable. This scenario is obtained by running a `MutEx` as controlled by its driver. The figure depicts the time line of a fork and its queue. The queue contents are shown only at context switch, i.e., when a method is finished and a new method is taken from queue head to start its execution (shown by a diamond on the time line). The same scenario generated by UPPAAL is shown in Fig. 9. This is part of the trace generated when checking for the reachability of the Error location.

At the end of this scenario, executing release and reqR would result in a 'permitR' to the right object for a second time, ignoring the request from the left one. This can continue infinitely. New instances of 'reqL' inherit the deadline associated to their parents, which shrinks continuously. After postponing 'reqL' for enough number of times, its deadline is missed, resulting in nonschedulability of `MutEx`. Using an Earliest Deadline First (EDF) strategy would favor old reqL to new reqR in this scenario. In addition, the EDF scheduler must give a higher priority to 'release' as opposed to 'request'.

The `MutEx` actor with the given driver needs a queue length of at least 5. Considering the driver, the reason is that 2 requests and 2 releases may be in the queue, while a delegated request can be added. Having chosen the proper scheduling strategy and queue length, we can repeat the scheduling analysis (checking for reachability of Error location) to find the best values for MIN_REL and MAX_REL and the deadline values for request and release.

6.3. Compatibility check

As explained in Section 5, compatibility checking amounts to checking trace inclusion between a use case U and the driver D. It is explained in [12] how to reduce trace inclusion problem to reachability check in UPPAAL. We extend this approach to take deadlines into account.

The basic step in this reduction is the construction of an automaton called D^{err} from the driver D. D^{err} is constructed by adding a location *error* to D and transitions '$l \xrightarrow[g]{a} error$' for all locations l and action labels a in such a way that this transition is enabled if no other a-transition is enabled from l. Furthermore, an internal transition from l to *error* with the guard $\neg Inv(l)$ is added and all location invariants are removed.

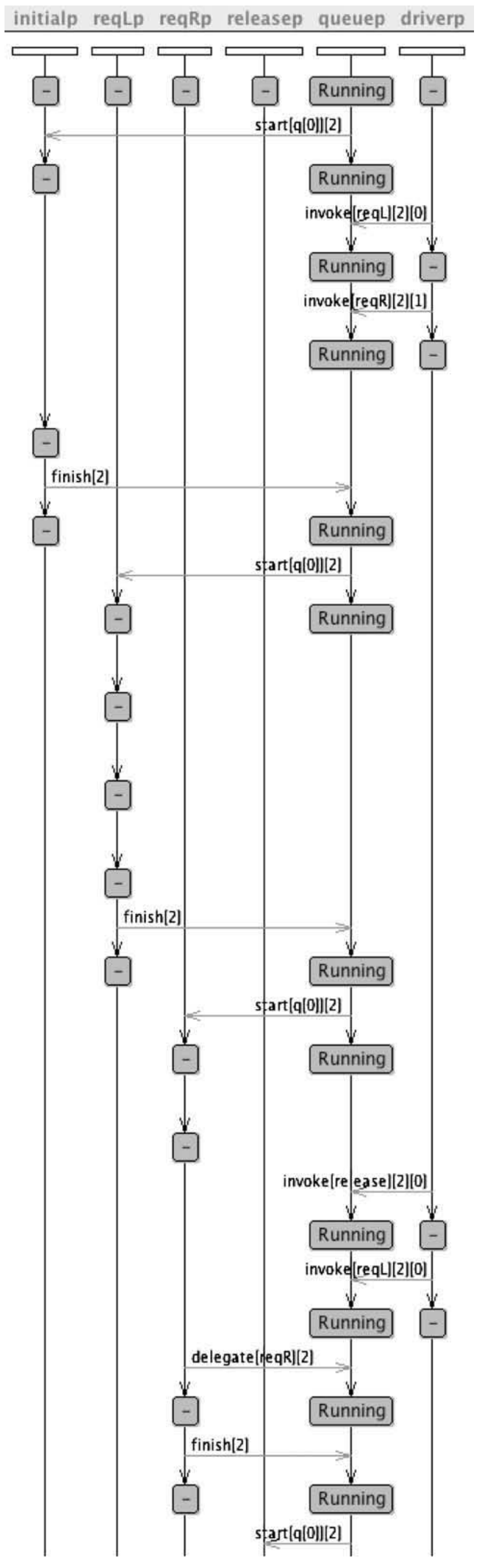

Fig. 9. Starvation scenario for FCFS given by Uppaal. The parameters self, Left and Right are instantiated to 2, 0 and 1, respectively. The arrows labeled invoke and delegate show enqueuing of messages. The arrows labeled start and finish determine the execution period of methods.

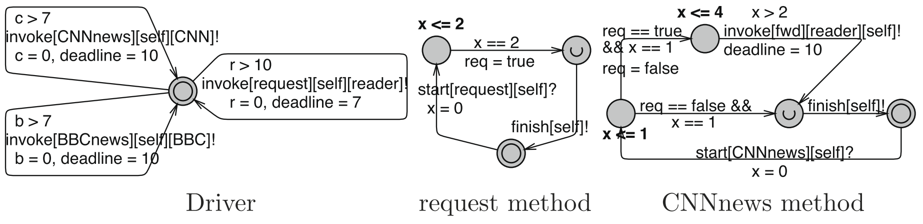

Fig. 10. News reader.

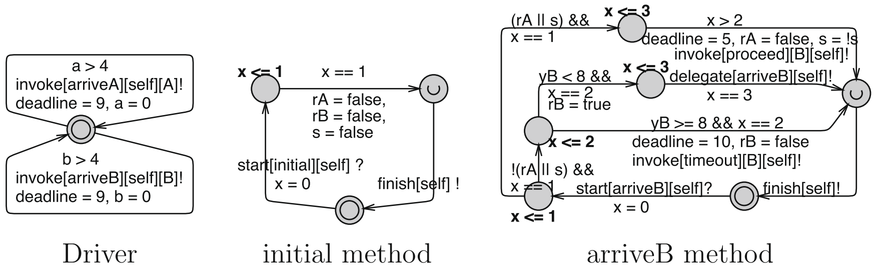

Fig. 11. Barrier Synchronizer.

If a driver of a schedulable actor accepts an action $m(d)$, the actor is guaranteed to accomplish m before d. A corresponding action $m(d')$ in the use case is therefore acceptable only when $d' \geq d$. To check this in Uppaal, an action $m(d')$ in U is modeled as $m!$ along with setting the global variable *deadline* to d'. In constructing D^{err}, every transition $l \xrightarrow[g\;,\;r]{m(d)} l'$ is changed into the transitions $l \xrightarrow[g\;,\;r]{m?} l'', l'' \xrightarrow[deadline \geq d]{} l'$ and $l'' \xrightarrow[deadline < d]{} error$ where l'' is a fresh *committed* location. When $m!$ and $m?$ synchronize, *deadline* will have the value of d'. Since l'' is a committed location, Uppaal has to take an action to leave this location before any other action (without time passing).

By this construction, $U \sqsubseteq D$ if and only if *error* is not reachable in the network of the automata U and D^{err} in Uppaal.

6.4. More examples

Fig. 10 shows the model of a news reader actor. It continually receives news updates from two sources modeled as its known actors BBC and CNN. Whenever this actor gets a request from its `reader` known object, it will forward the next news update from BBC or CNN, whichever is faster. In the case that both BBC and CNN provide an update (almost) at the same time, it only forwards one of the updates. A ‘fixed priority scheduler (FPS)’ could be used to favor one of the news agencies when both are available. The `CNNnews` method (and similarly `BBCnews`) only forwards the news updates if there is a pending request for it (the variable `req` is set to false in the `initial` method). The given driver shows the minimum inter-arrival intervals (measured by `b`, `c` and `r` clocks) and deadline constraints that lead to a schedulable actor.

The model in Fig. 11 shows an actor for providing barrier synchronization between its two known actors represented as A and B. The variables `rA` and `rB` show if A and B are ready. Since the synchronization should happen in real time, if one of A or B arrives but the other one is too late, the actor will send back a `timeout`. The variable `s` indicates that synchronization has been successful. The method `arriveA` can be obtained from `arriveB` by swapping As and Bs. In this method, waiting for the other partner is modeled using the delegation mechanism. The clock `yB` is used to bound the loops of delegations and to provide a timeout when the other partner is not in time. This actor is schedulable with the ‘first come first served (FCFS)’ strategy.

Checking schedulability for strategies like FPS and FCFS is computationally simpler and therefore faster than EDF. An EDF scheduler requires examining the remaining deadlines, which translates to potentially checking clock differences for all queue elements. Considering the existing model checking tools, our experiments confirm that FPS and FCFS can be easily verified while EDF is inevitably more costly to analyze.

7. Conclusions

We presented a modular method for schedulability analysis of real time distributed systems by extending the actor model. Methods of actors are specified with timed automata. The behavior of an actor can be analyzed individually for schedulability by introducing a driver as the most general specification of how the actor can be used. We can calculate a finite bound on the task queue such that the schedulability results hold for any queue length. Thus, the behavior automaton for an actor is finite. Schedulability analysis is reduced to reachability check for timed automata and hence is decidable. We showed how to carry out the schedulability analysis using the UPPAAL toolset.

One of our main contributions is the integration of the abstract formalism of task automata into a high-level Actor based modeling language. This integration requires a real time extension of actors and the modeling of asynchronous reception of messages as (dynamic) task generation. Furthermore, it allows for the specification and analysis of application-specific scheduling policies at the modeling level. We are working on applying this approach for the schedulability analysis of Creol [21], which supports concurrent objects involving more complex synchronization schemes.

References

[1] C. Hewitt, Procedural embedding of knowledge in planner, in: Proceedings of the Second International Joint Conference on Artificial Intelligence, 1971, pp. 167–184.
[2] G. Agha, The structure and semantics of actor languages, in: Proceedings of the REX Workshop, 1990, pp. 1–59.
[3] R. Alur, D.L. Dill, A theory of timed automata, Theoretical Comput. Sci., 126 (2) (1994) 183–235.
[4] E. Fersman, P. Krcal, P. Pettersson, W. Yi, Task automata: schedulability, decidability and undecidability, Inform. and Comput. 205 (8) (2007) 1149–1172.
[5] T.A. Henzinger, X. Nicollin, J. Sifakis, S. Yovine, Symbolic model checking for real-time systems, Inform. and Comput. 111 (2) (1994) 193–244.
[6] M. Bozga, S. Graf, I. Ober, I. Ober, J. Sifakis, The IF toolset, in: M. Bernardo, F. Corradini (Eds.), Formal Methods for the Design of Real-Time Systems, LNCS, vol. 3185, Springer, 2004, pp. 237–267.
[7] F. Wang, Efficient verification of timed automata with BDD-like data structures, STTT 6 (1) (2004) 77–97.
[8] S. Tripakis, S. Yovine, A. Bouajjani, Checking timed Büchi automata emptiness efficiently, Formal Methods Syst. Des. 26 (3) (2005) 267–292.
[9] K.G. Larsen, P. Pettersson, W. Yi, UPPAAL in a nutshell, STTT 1 (1–2) (1997) 134–152.
[10] B. Meyer, Eiffel: The Language, Prentice-Hall, 1992, First printing: 1991.
[11] L. de Alfaro, T.A. Henzinger, Interface automata, in: Proceedings of the ESEC/SIGSOFT FSE, 2001, pp. 109–120.
[12] D.P.L. Simons, M. Stoelinga, Mechanical verification of the IEEE 1394a root contention protocol using uppaal2k, STTT 3 (4) (2001) 469–485.
[13] K. Altisen, G. Gößler, J. Sifakis, Scheduler modeling based on the controller synthesis paradigm, Real-Time Syst. 23 (1–2) (2002) 55–84.
[14] J.J.G. Garcia, J.C.P. Gutierrez, M.G. Harbour, Schedulability analysis of distributed hard real-time systems with multiple-event synchronization, in: Proceedings of the 12th Euromicro Conference on Real-Time Systems, IEEE, 2000, pp. 15–24.
[15] L. Nigro, F. Pupo, Schedulability analysis of real time actor systems using coloured petri nets, in: Proceedings of the Concurrent Object-Oriented Programming and Petri Nets, LNCS, vol. 2001, Springer, 2001, pp. 493–513.
[16] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venter, D. Weil, S. Yovine, TAXYS: a tool for the development and verification of real-time embedded systems, in: G. Berry, H. Comon, A. Finkel (Eds.), Proceedings of the Computer Aided Verification (CAV01), LNCS, vol. 2102, Springer, 2001, pp. 391–395.
[17] C. Kloukinas, S. Yovine, Synthesis of safe, QoS extendible, application specific schedulers for heterogeneous real-time systems, in: Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS 2003), IEEE Computer Society, 2003, pp. 287–294.
[18] C.L. Liu, J.W. Layland, Scheduling algorithms for multiprogramming in a hard real-time environment, J. ACM 20 (1) (1973) 46–61.
[19] O. Kupferman, M.Y. Vardi, P. Wolper, Module checking, Inform. and Comput. 164 (2) (2001) 322–344.
[20] M. Sirjani, A. Movaghar, A. Shali, F.S. de Boer, Modeling and verification of reactive systems using Rebeca, Fund. Inform. 63 (4) (2004) 385–410.
[21] E.B. Johnsen, O. Owe, An asynchronous communication model for distributed concurrent objects, Software Syst. Model. 6 (1) (2007) 35–58.
[22] C. Courcoubetis, M. Yannakakis, Minimum and maximum delay problems in real-time systems, Formal Methods Syst. Des. 1 (4) (1992) 385–415.