Schedulability and Compatibility of Real Time Asynchronous Objects

Mohammad Mahdi Jaghoori, Delphine Longuet and Frank S. de Boer
CWI, Amsterdam, The Netherlands
{jaghouri,longuet,f.s.de.boer}@cwi.nl

Abstract—We apply automata theory to specifying behavioral interfaces of objects and show how to check schedulability and compatibility of real time asynchronous objects. The behavioral interfaces of real time objects specify (the order and timings of) the messages an object may send and receive. Each object is checked against its behavioral interface; first, to guarantee its correct output behavior, and second to make sure that every message it may receive is processed within the designated deadline (schedulability analysis). Next, we propose a new technique for testing whether every object is used as expected (i.e., according to its behavioral interface) when combined with other objects (compatibility check). Compatibility additionally implies schedulability in the context of the actual system. The analyses are automated using the UPPAAL model checker. Our method makes it possible to put a finite bound on the message queue and still obtain schedulability results that are correct for any queue length.

1 INTRODUCTION

The aim of object orientation is to divide a big system to smaller manageable units. To do so, an abstract high level view of objects is first specified in an interface. Each object should have the structure and behavior specified in its structural and behavioral interfaces. The correctness of the interactions between objects is then checked using their interfaces. In a real time system, the behavioral interface can include the expected timings and deadlines for incoming calls.

We apply automata theory to specifying behavioral interfaces of objects and show how to check schedulability and compatibility of objects. In the completely asynchronous setting (cf. the Actor model [1], [2]), objects can send only asynchronous messages and have queues for receiving them. Each message is processed in a method, which is a sequential code that may in turn send messages. Receiving a message schedules the corresponding method, i.e., puts it in the queue to be executed. A message an object sends to itself is called a self call.

In our framework, the specification of an object consists of its methods and behavioral interface, modeled with timed automata [3]. These automata mainly specify what messages are sent and when. A deadline is assigned to each message specifying the time before which the intended job should be accomplished. We allow each object to define its own scheduling policy (rather than, for instance, assuming "First Come First Served (FCFS)" by default) with the condition that a new message cannot preempt the currently running method. A scheduling policy determines the order in which the (methods corresponding to) incoming messages should be executed.

A behavioral interface is a *deterministic* timed automaton modeling the timings (and deadlines) of the messages the implementing objects may send and receive (to/from other objects). For instance, an object providing mutual exclusion between two entities must send a 'permit' after the first 'request', but the second 'request' should wait until the resource is 'released'. A model of such an object is analyzed as a case study in the paper.

In this paper, we show how we can apply the schedulability analysis of individual objects, as described in our previous paper [4], to the actual system. This application involves a new method for testing whether the entire system of objects behaves according to the behavioral interfaces. Furthermore, the behavioral interfaces in this paper include both incoming and outgoing messages, whereas in [4] only a driver automaton was used including the patterns of incoming messages. This amounts to further correctness check during schedulability analysis, i.e., output behavior is verified with respect to the input behavior.

For each object schedulability can be analyzed separately. Schedulability analysis, i.e., checking whether received messages can be processed within their deadlines, is reduced to reachability in timed automata and can be performed in UPPAAL. Although an object is allowed to have an unbounded queue, we can statically find an upper bound on the length of schedulable queues; hence, the behavior of a schedulable object is finite. By this analysis, on one hand, one can guarantee processing messages within their deadlines, assuming that messages arrive as specified in the behavioral interface. On

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

the other hand, one can analyze the object with regard to different scheduling strategies, and find the best strategy.

Once an object is proved schedulable, it can be used as an off-the-shelf component. One can then use the objects that are individually schedulable for making bigger components or systems. This results in a distributed system with multiple processors. However, we need to check whether the real usage of each object follows the expected usage (the behavioral interface), called the compatibility check.

To *prove* compatibility one needs to construct the complete behavior of all objects together. In order to avoid the general state-space explosion we introduce a method for testing compatibility. The product of the automata for behavioral interfaces (call it B) captures all allowed interactions between the objects. Intuitively, compatibility amounts to proving refinement between the system and B. The product B is a *deterministic* timed automaton which abstracts from objects implementation and task queues. We formally define compatibility as checking the inclusion of the traces of the system in the traces of B. While testing, we try to find counter-examples to compatibility, i.e., a trace in the system not included in B.

1.1 Related Work

Schedulability analysis in general has been investigated for different settings, from rate monotonic analysis [5], to non-uniformly recurring tasks modeled as task automata [6]. The former covers a smaller range of real time systems, and the analysis is performed on the whole system. The latter is only for one processor, and tasks only consist of internal computation, i.e., cannot generate other tasks during their execution.

By taking outputs out of behavioral interfaces, they will look 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. The internal tasks are not captured in the behavioral interface. Such an internal task may inherit the (remaining) deadline of the task generating it. Furthermore, a task automaton analyzes only one processor, while we allow combining multiple objects (each having one processor) and test their compatibility.

The problem of compatibility has been addressed for interface automata and timed interfaces by Alfaro et al in [7]. The main difference is that Alfaro et al consider two timed interfaces compatible if there is a way to use them together such that the timing expectations are met. However, in an object oriented setting, we require the objects to follow the timing expectations in the interfaces of each other along *all* executions. Furthermore, since behavioral interfaces provide an over-approximation of the object behavior, it is not enough to check compatibility by considering only the interfaces. We test compatibility to avoid state explosion.

1.2 Paper Structure

In Section 2, we provide the grounds for the approach by explaining the basic model of communication and timed automata. The timed object model is explained in Section 3. The schedulability of one object individually is investigated in Section 4. Compatibility and schedulability in the context of a system is discussed in Section 5. Section 6 demonstrates the modular approach by means of a case study. Section 7 concludes the paper.

2 Preliminaries

2.1 Asynchronous Concurrent Objects

To present our approach, we take an asynchronous subset of Creol [8] that fits the Actor model [2], [1]. Asynchronous objects 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. Objects have local variables, but no shared variables. An object has a dedicated processor. Our approach can be easily adapted to any modeling platform with the above-mentioned characteristics for concurrent objects. To have a concrete method, however, we need to consider further details of the Creol language mentioned next.

In Creol, each object (object) is instantiated from a class and typed by an interface and has one dedicated processor. A class defines a method for each message it can handle. A method is a piece of sequential code, which, among other statements, may send messages. We assume that there is at least a method '$initial$' in each class, which is responsible for initialization. Every class can have known objects, which serve as place holders for the objects that can communicate with instances of that class. We do not consider other Creol features like synchronous communication.

2.2 Timed Automata

We give a formal definition of timed automata and timed traces in this section.

Syntax. Let *Act* be a finite set of *actions*. Let C be a finite set of real-valued variables called *clocks*. We define $\mathcal{B}(C)$ the set of clock constraints as the set of boolean formulas built over elementary constraints $x \sim n$ and $x - y \sim n$ where $x, y \in C$, $n \in \mathbb{N}$, and $\sim \in \{<, \leq, =\}$, with boolean operators $\vee$, $\wedge$ and $\neg$.

A *timed automaton A over Act and C* is a tuple (L, l_0, E, I) where L is a finite set of *locations*; $l_0 \in L$ is the initial location; $E \subseteq L \times \mathcal{B}(C) \times Act \times 2^C \times L$ is a finite set of *edges*; $I : L \to \mathcal{B}(C)$ assigns an *invariant* to each location. Location invariants are restricted to conjunctions of constraints of the form $x < n$ or $x \leq n$ for $x \in C$ and $n \in \mathbb{N}$. We write $l \xrightarrow{g,a,r} l'$ for an edge from location l to location l' guarded by clock constraint g, labeled with action a and resetting the subset r of C.

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 $x <= 0$ to the location. Intuitively, this means that the automaton cannot spend any time in that location [9].

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 $s_0 = (l_0, \mathbf{0})$. There are two types of transitions: action transitions $(l, u) \xrightarrow{a} (l', u')$ where $a \in Act$, if there exists $l \xrightarrow{g,a,r} l'$ such that u satisfies the guard g, u' is obtained by resetting to zero all clocks in r and leaving the others unchanged and u' satisfies the invariant of location l'; delay transitions $(l, u) \xrightarrow{d} (l, u')$ 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 TA. A timed automaton is called *deterministic* if and only if for each $a \in Act$, if there are two edges from l labeled by the same action $l \xrightarrow{g,a,r} l'$ and $l \xrightarrow{g',a,r'} l''$ then the guards g and g' are disjoint (i.e. $g \wedge g'$ doesn't hold) .

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 $(l, u, v) \xrightarrow{a} (l', u', v')$ additionally requires v and v' to be considered in the corresponding guard and update.

Networks of timed automata. In the following, we assume that the set of actions *Act* is partitioned into two disjoints sets: a set Act_I of *input actions* $a?$ and a set Act_O of *output actions* $a!$. A *non-observable internal actions* τ is also assumed. Let $Act_\tau = Act \cup \{\tau\}$.

A system may be described as a collection of timed automata A_i $(1 \leq i \leq n)$ over sets of actions Act^i_τ 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).

Timed traces. A *timed action* is a pair $(a, d) \in Act_\tau \times \mathbb{R}_+$. A *timed sequence* σ is a possibly infinite sequence of timed actions: $\sigma \in (Act_\tau \times \mathbb{R}_+)^*$. Given a timed sequence σ, $\pi_{obs}(\sigma)$ denotes the projection of σ on *Act*, intuitively deleting τ transitions. The sequence $\pi_{obs}(\sigma)$ is called the *observable timed sequence* associated to σ.

A *run* of a timed automaton A from initial state $(l_0, \mathbf{0})$ over a timed sequence $\sigma = (a_1, d_1)(a_2, d_2) \ldots (a_n, d_n)$ is a sequence of transitions

$$(l_0, \mathbf{0}) \xrightarrow{d_1} \xrightarrow{a_1} (l_1, u_1) \to \cdots \xrightarrow{d_n} \xrightarrow{a_n} (l_n, u_n)$$

The set $Traces(A)$ of timed traces of A is the set of timed sequences σ for which there exists a run of A over σ. The set $Traces_{obs}(A)$ of observable timed traces of A is the set $\{\pi_{obs}(\sigma) \mid \sigma \in Traces(A)\}$. In the following we only consider maximal traces, namely those traces for which the corresponding run does not end in a location with an invariant. Intuitively, time may not stop along maximal traces, which is the case in real systems, too.

3 THE TIMED OBJECT MODEL

In this section, we present our formal model of objects. The abstract behavior of an object is specified in its behavioral interface. This interface only consists of the messages the object may receive and send. An implementation of the interface, namely a class, is a set of methods for processing the incoming messages. A method may in turn send messages. Each of these methods is represented by a timed automaton. A scheduler automaton takes care of buffering the incoming messages and running the corresponding methods. An object is then defined as an instance of a class with a specific scheduler.

We assume a finite global set of method names $\mathcal{M}$.

Definition 1 (Behavioral interface)**.** *A behavioral interface B providing a set of method names $M_B \in \mathcal{M}$ is a deterministic timed automaton over alphabet Act^B such that:*

- *Act^B is partitioned into two sets*
  - *output actions: $Act^B_O = \{m? | m \in \mathcal{M} \wedge m \notin M_B\}$*
  - *input actions: $Act^B_I = \{m(d)! | m \in M_B \wedge d \in \mathbb{N}\}$*
- *the edges labeled with output actions have true as guard*

Notice the counter-intuitive notation for inputs and outputs in this definition, i.e., using $m?$ for outputs and $m(d)!$ for inputs. It is explained in Section 4 how this simplifies checking if a specific implementation adheres to the specification in the behavioral interface. The methods M_B (corresponding to the input actions) must exist in the classes implementing the interface B. Other methods are sent by the object and should be handled by the environment.

A behavioral interface can be seen both as the (acceptable) observable behavior of the object, or as an abstraction of the environments in which the object can be used. An action $m(d)!$ represents a message m sent to the object by the environment. A correct implementation of the object should be able to finish method m before d time units.

A behavioral interface abstracts from specific method implementations, the queue in the object and the scheduling strategy. One can define a class as a set of methods implementing a specific behavioral interface.

Definition 2 (Class)**.** *A class R implementing the behavioral interface B is 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 $M_B \subseteq M_R$; and,*
- *for all i, $1 \leq i \leq n$, A_i is a timed automaton representing method m_i with the alphabet $Act_i = \{m! | m \in M_R\} \cup \{m(d)! \mid m \in \mathcal{M} \wedge d \in \mathbb{N}\}$*

Notice that methods have only output actions. Receiving messages (and buffering them) is handled by scheduler automata defined next. Sending a message $m \in M_R$ is called a self call. Self calls may or may not be assigned an explicit deadline. The self calls that are not statically assigned a deadline are called delegation. Delegation implies that the internal task (triggered by the self call) is in fact the continuation of the parent task; therefore, the delegated task inherits the (remaining) deadline of the task that triggers it. An example of delegation is given in the case study in Section 6.

The condition $M_B \subseteq M_R$ requires that a class should provide a method for handling all messages it may receive according to the interface it implements (but may add more methods). Checking whether it will produce correct output behavior, and if it can finish all methods in designated deadlines is explained in Section 4.

3.1 Scheduler Automata

A scheduler automaton for a class R must have the following characteristics:

1) implements a queue as in Definition 3.
2) is strongly input enabled, i.e., can receive any message in M_R at any time, puts it in the queue, and assigns a clock to it to keep track of the time since it is received.
3) whenever a method is finished, selects another message from the queue (based on its scheduling strategy) and starts the corresponding method (called context-switch).
4) has an Error location with no outgoing transitions. This location is reachable whenever queue overflow occurs or the deadline of a task in the queue expires (cf. Definition 3).

We show in Section 4 that we may put a finite bound on the queue and still derive schedulability results that hold for any queue length. 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.

Definition 3 (Queue)**.** *A queue 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 $m(d, x)$ where m is a method name, $d \in \mathbb{N}$ is its deadline and $x \in C_q$ keeps track of how long the task has been in the queue. The deadline of m expires when $x > d$.*

When a task is inserted in the queue, a free clock is assigned to it and is reset. In case of delegation, however, the clock of the currently running task is reused. Examples of scheduler automata are given in Section 6.

4 Schedulability Analysis

An object is an instance of a class together with a specific scheduler automaton. An object cannot be analyzed on its own, because there are an infinite number of ways in which the methods could be called. Therefore, we only consider the method calls specified in its behavioral interface (i.e., the input actions). The analysis in this section is based on the schedulability analysis in [4], which is enhanced to include a local consistency check as well.

Receiving a message from another object (i.e., an input action in the behavioral interface) creates a new task (for handling that message) and adds it to the queue. The behavioral interface doesn't capture (internal tasks triggered by) self calls. In order to analyze the schedulability of an object, one needs to consider both the internal tasks and the tasks triggered by the (behavioral interface, which abstractly models the acceptable) environment.

Definition 4 (Schedulable)**.** *An object 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 in 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 class and its behavioral interface, we can statically determine an upper bound on the queue length of schedulable systems.

Lemma 5. *If an object 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 (method automata or the input actions of the behavioral interface) and b_{min} is the shortest termination time of any of the method automata.*

Proof: Assume that the queue length reaches $\lceil d_{max}/b_{min} \rceil + 1$. 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 must have a deadline less than or equal to d_{max} and all tasks take more than or equal to b_{min} to accomplish. Let Q be the set of the tasks in the queue at this moment. To execute all tasks in Q it takes at least $T = (\lceil d_{max}/b_{min} \rceil + 1) \times b_{min}$ (new tasks may be inserted and executed in the meanwhile which may only add to the time until all tasks in Q finish). It is easy to see that $T > d_{max}$ and so there is at least one task in the set Q that misses its deadline. □

We can calculate the best case runtime for timed automata as shown by Courcoubetis and Yannakakis [10]. The longest deadline can be found by a simple static search of all the transitions.

Theorem 6. *An object is schedulable if, and only if, the scheduler cannot reach the Error location with a queue length of $\lceil d_{max}/b_{min} \rceil$.*

The proof is straightforward considering the definition of scheduler automata and lemma 5.

As a result of this theorem, we can check the schedu-

lability of an object by checking reachability of the Error location using the UPPAAL model checker. In addition, we explain how to ensure that the methods in a class together with a specific scheduler provide the behavior specified in its behavioral interface (local consistency). This includes checking that the object produces expected output given the inputs specified in the interface.

We can generate the possible behaviors of an object by making a network of timed automata consisting of its method automata, behavioral interface automaton B and a concrete scheduler automaton. The inputs of B written as $m!$ will match with inputs in the scheduler written as $m?$ and the outputs of B written as $m?$ will match outputs of method automata written as $m!$. If this network of automata does not deadlock we know that the object cannot produce a behavior disallowed by the interface. Note that this also implies that the Error location is not reachable. One can allow termination by labeling some locations in the behavioral interface as final. Then we should check for deadlock in non-final states. Therefore, absence of deadlock implies schedulability as well as local consistency with the behavioral interface.

5 COMPATIBILITY CHECKING

The behavioral interface of an object is a high level abstract view of the object behavior in terms of the messages it may send and receive. Intuitively, the composition of the behavioral interfaces of some communicating objects should also provide an abstract view on the system behavior in terms of the messages communicated.

In other words, we need to check that the behavior of the system (composition of the objects) is a *refinement* of the parallel composition of the behavioral interfaces, when the system behavior is restricted to the communications between different objects. This ensures that each object is used correctly, i.e., receives messages as specified in its behavioral interface. This is called compatibility check.

In this section, first we don't consider deadlines, i.e., the actions $a(d)!$ and $a(d)?$ are not distinguishable from $a!$ and $a?$, respectively. In the following, B represents the synchronous product of the behavioral interfaces, and S represents the system of communicating objects. Each object consists of the method automata and a scheduler automaton.

Definition 7 (Compatibility)**.** *Consider two timed automata S and B. S is* compatible *with B, denoted by S compat B, if and only if $Traces_{obs}(S) \subseteq Traces(B)$.*

Inclusion of timed languages being in general undecidable, we propose a method for testing it. In particular, we want to be able to exhibit a counter-example if some incompatibility is found.

A test case will be built according to a diagnostic trace given by UPPAAL on the behavioral interfaces. Completing this trace with allowed and forbidden divergences will give us the means to detect possible incompatibilities. A step going out of the trace is forbidden if it is not expected by the behavioral interfaces, allowed otherwise. The test fails if, along the trace, the system performs a forbidden action, that is trying to send a message not expected by the behavioral interfaces. If the test fails, we are then able to give a counter-example to compatibility.

5.1 Test cases

Trace inclusion is usually used for testing correctness (or conformance) between a system and its specification in formal testing frameworks [11]. We use the standard notions and methods from these frameworks. However, as we are checking compatibility, the purpose is different, namely, our main goal is to find a *counter-example* in the case of incompatibility.

We build a test case given a trace from the product of the behavioral interfaces. Such a trace represents abstractly a desired system behavior in terms of the messages communicated between objects. A test case formally is a deterministic timed automaton in the shape of a tree whose location are labeled with verdicts.

Definition 8 (Test case)**.** *A test case is a deterministic acyclic timed automaton $TC = (L, l_0, E, I)$ over Act whose set of locations is divided into three disjoint sets* **Pass**, **Fail** *and* **Inconc**. *For every non-leaf location l and action $a \in Act$,*

- *there exists a transition $l \xrightarrow{g,a,r} l'$; and,*
- *for all transitions $l \xrightarrow{g_i,a,r_i} l'_i$ guards are complementary: $\bigvee_i g_i$ holds.*

A verdict labeling a location allows us to evaluate an execution of the test case terminating on this location. Locations labeled by **Fail** are those which are reachable with forbidden behaviors of the system (a non-specified action or an action happening outside its time constraints in the product of behavioral interfaces, for example). An execution of a test case ending on a **Pass** location means that the system fulfilled the test case requirements. When an inconclusive location **Inconc** is reached, it means that the system behaved correctly but not according to the behavior aimed by the test case. To find a counter-example, we need to search for locations marked **Fail**.

5.2 Generating a test case

In this subsection, we explain how to generate a test case given a trace obtained from the product of behavior automata. In the following, we call the product of the behavior automata B. Assume that this trace consists of the locations l_i and the transitions $l_{i-1} \xrightarrow{g_i,m_i,r_i} l_i$ such that $0 < i \leq n$. Such a trace shows (the timings of) some messages communicated between the objects. We turn this trace into a test case in three steps.

- **Pass**: Label l_n with the verdict **Pass**. Reaching this location implies that the system can perform the desired behavior denoted by this trace.
- **Fail:** Add a location f labeled with the verdict **Fail**. For every location l_i, add a transition $l_i \xrightarrow{\neg g,a,r} f$ if there exists a transition $l_i \xrightarrow{g,a,r} l'$ in B. For every

other action b (i.e., those not allowed at location l_i in B), add a transition $l_i \xrightarrow{true,b,\emptyset} f$. Furthermore, if the location corresponding to l_i in B has an invariant h, add a transition $l_i \xrightarrow{\neg h,\tau,\emptyset} f$ and replace g_{i+1} with $h \wedge g_{i+1}$ in the transition leading to l_{i+1}.

- **Inconc:** Complete the locations l_i, namely, add a transition $l_i \xrightarrow{g,a,r} c$ per every transition $l_i \xrightarrow{g,a,r} l'$ in the product of the behavioral interfaces (except for the one that already exists in the trace), where c is a new location labeled **Inconc**. Finally label every location l_i as **Inconc**.

Intuitively, the test case forces the system to send messages in the specified order. If that is not possible, the result will be inconclusive or failure. We need to show that if the test case fails, the system is not compatible.

Soundness. Assume that running the system in parallel with the test case leads to a **Fail** state, shown as a sequence of locations and timed actions below. This sequence can be projected onto its 'system' and 'test-case' components. Furthermore, the locations in the test case can be mapped to their correspondents in the product of behavioral interfaces (B). This latter step is possible in exactly one way because B is a *deterministic* timed automaton. Below we show this decomposition. The superscripts B and S distinguish between the locations of B and the system. The step from the test case to B is factored out in favor of simpler representation.

$$\begin{array}{ccccc}
(l_0^S, \mathbf{0}^S) & \xrightarrow{d_1,a_1} \cdots \rightarrow & (l_i^S, u_i^S) & \xrightarrow{d_{i+1},a_{i+1}} & (l_f^S, u_f^S) \\
\Uparrow & & \Uparrow & & \Uparrow \\
(l_0, \mathbf{0}) & \xrightarrow{d_1,a_1} \cdots \rightarrow & (l_i, u_i) & \xrightarrow{d_{i+1},a_{i+1}} & (l_f, u_f) \\
\Downarrow & & \Downarrow & & \not\Downarrow \\
(l_0^B, \mathbf{0}^B) & \xrightarrow{d_1,a_1} \cdots \rightarrow & (l_i^B, u_i^B) & \not\xrightarrow{d_{i+1},a_{i+1}} &
\end{array}$$

As shown here, the final step in the trace has no correspondence in B. This is due to the fact that the timed action (d_{i+1}, a_{i+1}) is not possible at location l_i^B, which in turn results from the way the test case is constructed (see above). A complete proof should consider different possibilities of a which is omitted for brevity here.

Non-laxness. Soundness alone is a loose requirement on a test case, because for instance a test case with no failure state is trivially sound. The test cases generated as mentioned above are non-lax, i.e., a test case must reject any system which can be shown incompatible during the execution of the test. Suppose S and T show the system and the test case automata, respectively. Formally T is *non-lax* if "S passes $T \Rightarrow S \,\|\, T$ compat D". In other words, any incompatible behavior must be detected: if the test case contains a wrong behavior with respect to compatibility, then it must lead to a **Fail** state.

To prove this, we need to show that any trace in $S \,\|\, T$ not leading to the **Fail** state also exists in B. We can decompose such a trace into its S and T components. We need to show that at every location l_i^T where T can do (d_i, a_i), B can also take the timed action (d_i, a_i). This can be deduced from the construction of the test case. The proof details are omitted for brevity.

The starting point for test case generation is a trace in the product of behavioral interfaces B. One can use UPPAAL to generate such a trace in two ways. First, the simulation feature can be employed to generate specific hand-made traces. The second way is to use the UPPAAL model checker. A first property to check is possibility of deadlock in B. A trace leading to deadlock in B is a candidate of incompatibility which is a good starting point for testing. Alternatively, one can use certain reachability properties that cover 'interesting' paths in B. Techniques and tricks to get good traces (based on coverage, etc.) are studied in literature (e.g., [12]) and not addressed here.

5.3 Executing the test case

A characteristic of our framework is that we test a model rather than an actual system. The submission of the test case to the model can then be computed as a synchronized product, we do not have to consider such issues as arbitrary short delays between two actions that may be taken into account when testing a real system.

Our model of a system consists of a set of communicating objects. Each object, in turn, consists of methods and a scheduler automata (the scheduler contains a queue). Therefore, a system is a network of timed automata (representing methods and schedulers of all objects) which can run in UPPAAL.

All actions except communication between objects are considered internal. Submitting a test case is then the synchronized product of the system automata and the test case automaton, in which every observable action in the system must synchronize with the test case.

Notice that a test case is a deterministic automaton and intuitively represents a specific order of exchanging messages between the objects. Therefore, requiring the system to synchronize with the test case resolves part of the nondeterminism in the system. However, the internal actions of different objects are nondeterministically interleaved (more precisely, those internal actions that can happen at the same time). Thus, we would need to execute the test case several times to get a verdict. In UPPAAL we can simulate the model and see for instance where the system may diverge from the test case (**Inconc** results) or fail (we will explain how to use UPPAAL for submitting a test case).

Since the system behavior is controlled by the test case, using a model checker would also be practical to check the reachability of a **Fail** or a **Pass** state; whilst model checking the system alone may not be feasible. The system under test *passes* the test, denoted by S passes T, if the execution of the test case on the system does not lead to a **Fail** state. The UPPAAL model checker can provide a diagnostic trace in case of failure showing exactly how and when the system is not compatible.

5.3.1 Deadlines in Compatibility

So far in the compatibility check, we didn't distinguish between different deadline values on actions. Remember that every input action in a behavioral interface is assigned a deadline, unlike the output actions. The schedulability check of individual objects ensures that the methods corresponding to the inputs are scheduled and finished before the specified deadline.

When constructing the product of behavioral interfaces, for every synchronizing pair of actions, there is one deadline (from the input action). Therefore, every timed action in the traces of the product (denoted B) can be augmented with an integer d showing this deadline.

Similarly, in the system (more precisely in method automata) every observable output action (i.e., excluding self calls) has a deadline. The input actions (which appear in the schedulers) have no specific deadlines. Therefore, in the observable traces of the system, every timed action can also be augmented with an integer d' for the deadline.

To include deadlines in the formal definition of compatibility, we need to add the condition $d \leq d'$ for every matching action where d is the deadline in B and d' is the deadline in the system. When submitting a test case, the usual parallel composition is extended to allow synchronization only on actions with compatible deadlines: an action $a(d')$ in the system will be able to synchronize with an action $a(d)$ in the behavioral interfaces if and only if $d \leq d'$.

5.3.2 Using UPPAAL

When submitting a test case, we require that any communication between two objects should synchronize with the test case, as well. Practically, this means that the sender object (in one of its methods), the receiver object (in its scheduler) and the test case should synchronize. UPPAAL does not support three-way synchronization.

Since we do not want to change the specification of the model under test, we solve the problem of three-way synchronization by splitting every action in the test case into two steps. At the first step, the sender object synchronizes with the test case, and *immediately* afterwards, the test case synchronizes with the receiver object. The urgency between these two steps is modeled by using a 'committed' location in the test case between these two steps. Section 6 shows a small typical test case in the context of the case study provided, which also shows how to include deadlines practically in testing.

5.4 Schedulability

Objects can be proved individually schedulable with respect to their behavioral interfaces. Using such objects in a system, compatibility implies schedulability of the whole system. Intuitively, this means that every message in the system will be finished within the designated deadline.

To prove this we can assume that the system is compatible but not schedulable. This means that there is a run of the system which drives the scheduler of one of the objects, say o_i, to the Error state. Since the system is compatible, this run also exists in the product of behavior automata, which can be projected onto the behavior automaton of o_i alone. The projected run in o_i would lead to Error which is in contradiction with the assumption that o_i is schedulable.

6 CASE STUDY

In this section, we describe how we can use the techniques explained throughout the paper for modeling schedulable objects and using them in the context of complete systems. We demonstrate the approach by modeling a 'mutual exclusion' handler and proving its schedulability. We then show how to check compatibility in the context of dining philosophers and bridge controller examples. We use UPPAAL for modeling the timed automata for behavioral interfaces, methods and scheduler automata.

Communication. We use the 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]! in the behavioral interface of MutEx. By setting both sender and receiver as self (in method automata), one can invoke a self call and assign an explicit deadline to it. The delegate channel is used for delegation. The self call made using the delegate channel inherits the deadline of the currently running task (it 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).

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. Notice that the value of the deadline does not affect synchronization, i.e., only message name is important. The receiver, however, cannot use this deadline value in its guard, as guards are evaluated before updates.

6.1 Modeling the scheduler

Figure 1 shows the general structure of a scheduler automaton. The only thing not specified in this general picture is the scheduling strategy. This automaton should have the characteristics mentioned in Section 3.1.

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 x[ca[i]] and d[ca[i]] keep track of the remaining deadline of q[i]. 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. In this general scheduler automaton, there is an edge (left down in the picture) that allows

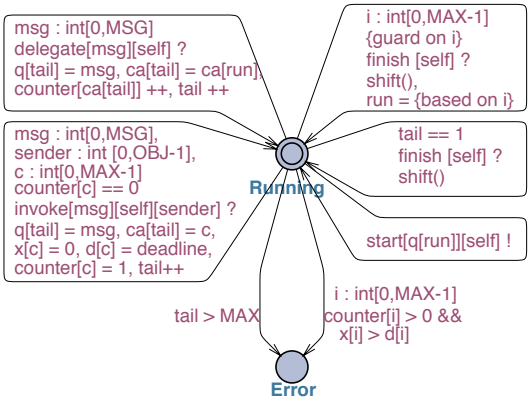

Fig. 1. A general scheduler automaton

receiving (at any time) 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). (d[c] = deadline).

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).

Context-switch 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 assigning run = (i < run) ? i : i−1 (because i is selected before shifting). The guard $x[a] - x[m] >= d[a] - d[m]$ makes sure that the remaining deadline of a, i.e., $x[a] - d[a]$, is bigger than or equal to the remaining deadline of m. 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).

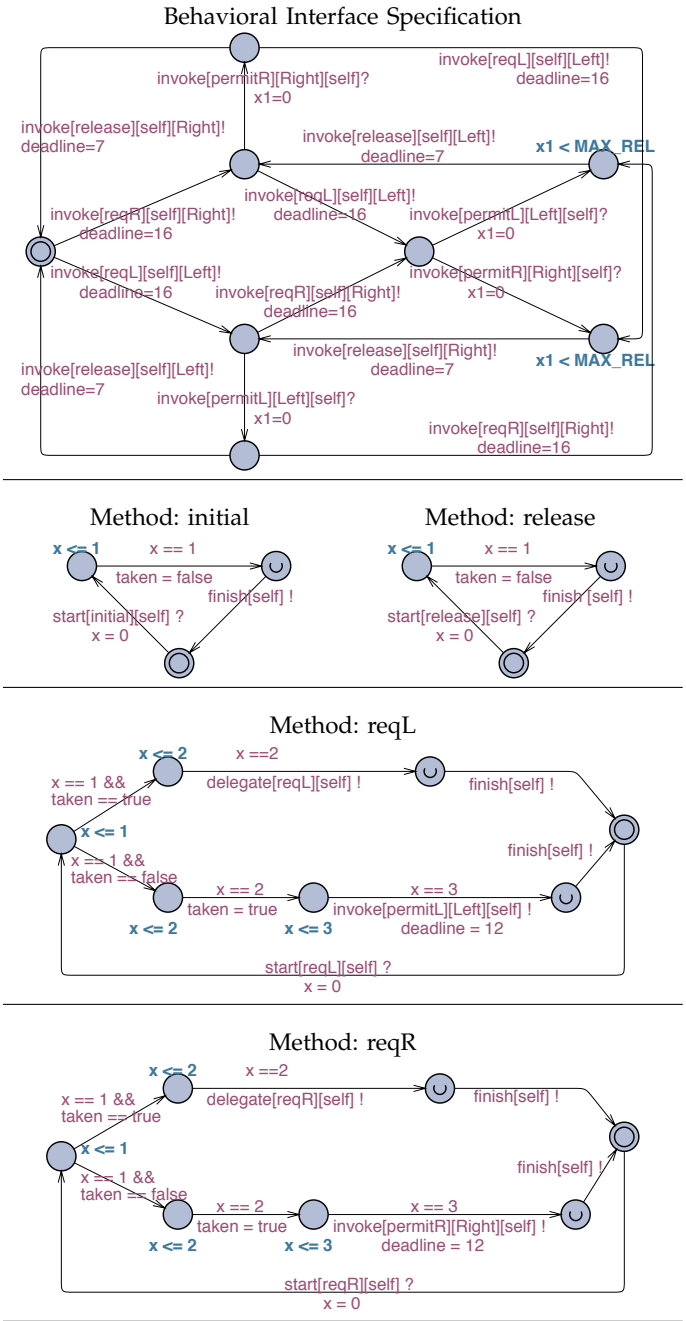

Fig. 2. The timed specification of the MutEx class

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 Mutual Exclusion Object

We model the mutual exclusion handler as a class called MutEx. A MutEx object models a resource shared between two (sets of) entities, referred to as Left and Right. For the sake of simplicity, the request messages from Left and Right are distinguished as reqL and reqR. Respectively, the

MutEx may reply by permitL or permitR, if it is not already taken; otherwise, it puts the request back into the queue (by a self call) until a release message is received.

We specify a class using timed automata templates (as supported by UPPAAL [9]). We may simply say automata instead of automata templates. The automata representing the methods and the behavioral interface of a class are parameterized in an identifier (written self), and the identifiers of the objects communicating with it (e.g., Left and Right for MutEx). In the context of a complete system, every object will be assigned a unique identifier. The MutEx behavioral interface specifies formally what we explained in the previous paragraph. The automata representing a MutEx are given in Figure 2.

A method is executed only when selected by the scheduler. Therefore, the first transition in method automata is a synchronization on start. When a method terminates, the scheduler should select another method. Therefore, the last transition in method automata is a synchronization on finish. The location before finish is urgent so that time won't pass during context switch.

A MutEx is initially, also after being released, set to be not taken. When MutEx is not taken, an incoming request is granted by sending back a permit. If the MutEx is already taken, a self call is made to remember the pending request. This must be modeled as a delegation so that to keep the original deadline for the request. Thus the schedulability of MutEx implies every request is followed by a timely permit. Notice that the behavioral interface requires a release message to arrive before MAX_REL if there is a pending request. This is necessary to ensure the pending request will be granted in time (i.e., for schedulability of MutEx). Intuitively, to guarantee timely response to requests, a resource holder is not allowed to keep it more than MAX_REL.

6.3 Schedulability Analysis

After the methods and the behavioral interface are specified and a scheduling strategy is selected, one can check the schedulability of the object by checking the reachability of the Error state of the scheduler automaton. As explained in Section 4, checking for deadlock includes schedulability analysis plus a local consistency check. By iterating the schedulability analysis, one can refine the constraints in the behavioral interface so that to indicate the minimum requirements, e.g., the smallest deadlines possible and the loosest location invariants.

With such an automatic analysis process, it is easy to study the effect of different scheduling strategies on schedulability. Figure 3 shows a possible scenario in which 'First Come First Served (FCFS)' strategy for a MutEx may cause starvation, and as a result makes MutEx non-schedulable. The figure depicts the time line of a MutEx 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).

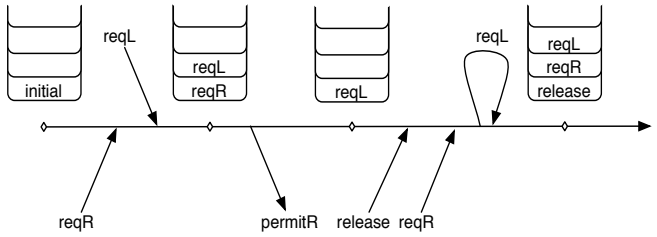

Fig. 3. Postponing reqL infinitely when using FCFS

At the end of this scenario, executing release and reqR would result in a permitR for a second time, ignoring reqL. This can continue infinitely (and is allowed in the behavioral interface). New instances of reqL (modeled as delegation) inherit the remaining deadline of the original reqL, 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'. With such a scheduler MutEx is schedulable.

6.4 Using the Schedulable MutEx

Individually schedulable objects can be used in making different actual systems. However, it is necessary to make sure that each object is used correctly, i.e., according to its behavioral interface. This can be tested by the compatibility check. Once compatibility is ensured, we can immediately deduce the schedulability of all objects in the system.

In this section, we use the schedulable MutEx in the context of dining philosophers and bridge controller systems. The schedulability of these systems (which is deduced from compatibility) implies starvation freedom (because a schedulable MutEx guarantees granting requests in time).

6.4.1 Naive Philosophers

In this case, every philosopher tries to take the right fork and then the left fork. We use MutEx as fork. The model of a philosopher is given in appendix. We test compatibility for a system composed of four philosophers and four forks. It is well known that this naive binding results in deadlock and starvation. We can see that this naive binding is in fact incompatible with respect to the behavioral interface of MutEx.

We choose the deadlock property as the starting point for compatibility check. We get a trace and see that the test case fails.

6.4.2 A Reverse Philosopher

In this case, we let one of the philosophers take the left fork first (while others pick up the right fork first). This reverse philosopher has a different behavioral interface as it sends reqL before reqR. In this case, the product of

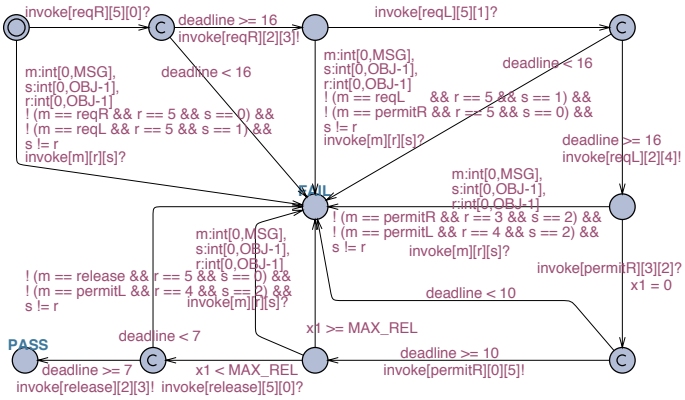

Fig. 4. Test case for bridge controller in UPPAAL

the behavioral interfaces is deadlock free. We select a trace in which every philosopher can perform a cycle of getting both forks until it releases them. Interestingly, this test also fails because it takes too long (i.e., greater than MAX_REL) to release one of the forks (which is somehow due to the asymmetry in the model). By increasing MAX_REL (and accordingly the deadlines) in the behavioral interface of MutEx (such that it is still schedulable), this model of philosophers can become compatible. However, increasing the deadlines implies waiting longer before a request is granted.

6.4.3 Late Philosophers

Another solution to deadlock in dining philosophers is to make half of the philosophers start later than the others. This model passes the compatibility test the previous model failed (without increasing the deadlines). This implies the schedulability of all objects in the context of this system. A consequence is that one can deduce an upper bound since a philosopher starts a round to get two forks until releases them. This can be done because we know that the requests are granted within the specified deadline.

6.4.4 Bridge Controller

There is a bridge which can allow at most one train to pass at a time. Therefore, the trains on the other side must wait until the bridge is free. The trains arriving on the left side of the bridge send reqL and the trains on the right send reqR. The model of a train (arriving on the left side of the bridge) is given in appendix. A train is different from philosophers as every train needs only one MutEx. We use a test case in which a train can take and release the MutEx representing the bridge.

Figure 4 shows the UPPAAL implementation of this test case. Notice that in this figure, the inconclusive location is not explicitly modeled, because we are basically interested in checking the reachability of FAIL or PASS state. If neither is reachable, then the test is inconclusive. To capture the disallowed actions at each location, we use the select feature of UPPAAL to be able to choose any action; in the guard, we exempt the allowed communication actions (by specifying the message, sender and receiver) and self calls (s != r). Notice that in this example, none of the transitions on behavioral interfaces are guarded. The system passes this test case.

7 Conclusions

One of our main contributions is the integration of the abstract formalism of timed automata into a high-level object based modeling language. This integration requires a real-time extension of the object model and the modeling of asynchronous reception of messages as (dynamic) task generation. The high level synthetic view of each real time object is given it its behavioral interface. Furthermore, application-specific scheduling policies are specified at the modeling level.

Schedulability of each class is analyzed individually with respect to its behavioral interface. This is made feasible by putting a finite bound on the task queue such that the schedulability results hold for any queue length. We can then test a system of communicating objects to make sure objects are used as expected. This compatibility further implies the schedulability of the whole system.

We are working on generalizing the analyses to languages with concurrent objects involving more complex synchronization schemes.

References

[1] C. Hewitt, "Procedural embedding of knowledge in planner," in *Proc. the 2nd International Joint Conference on Artificial Intelligence*, 1971, pp. 167–184.

[2] G. Agha, "The structure and semantics of actor languages," in *Proc. the REX Workshop*, 1990, pp. 1–59.

[3] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[4] M. M. Jaghoori, F. de Boer, T. Chothia, and M. Sirjani, "Schedulability of asynchronous real-time concurrent objects," *Logic and Algebraic Programming*, submitted 2008, a preliminary version appeared in NWPT/FLACOS 2007 as an extended abstract.

[5] J. J. G. Garcia, J. C. P. Gutierrez, and M. G. Harbour, "Schedulability analysis of distributed hard real-time systems with multiple-event synchronization," in *Proc. 12th Euromicro Conference on Real-Time Systems*. IEEE, 2000, pp. 15–24.

[6] E. Fersman, P. Krcal, P. Pettersson, and W. Yi, "Task automata: Schedulability, decidability and undecidability," *Information and Computation*, vol. 205, no. 8, pp. 1149–1172, 2007.

[7] L. de Alfaro, T. A. Henzinger, and M. Stoelinga, "Timed interfaces," in *Proc. Embedded Software (EMSOFT)*, ser. LNCS, vol. 2491, 2002, pp. 108–122.

[8] E. B. Johnsen and O. Owe, "An asynchronous communication model for distributed concurrent objects," *Software and Systems Modeling*, vol. 6, no. 1, pp. 35–58, 2007.

[9] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *Proc. Formal Methods for the Design of Computer, Communication, and Software Systems*, ser. LNCS, M. Bernardo and F. Corradini, Eds., vol. 3185, 2004, pp. 200–236.

[10] C. Courcoubetis and M. Yannakakis, "Minimum and maximum delay problems in real-time systems," *Formal Methods in System Design*, vol. 1, no. 4, pp. 385–415, 1992.

[11] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, "Testing real-time systems using uppaal," in *Formal Methods and Testing*, ser. LNCS, vol. 4949, 2008, pp. 77–117.

[12] A. Hessel, K. G. Larsen, B. Nielsen, P. Pettersson, and A. Skou, "Time-optimal real-time test case generation using uppaal," in *Formal Approaches to Software Testing*, ser. LNCS, vol. 2931, 2003, pp. 114–130.

APPENDIX

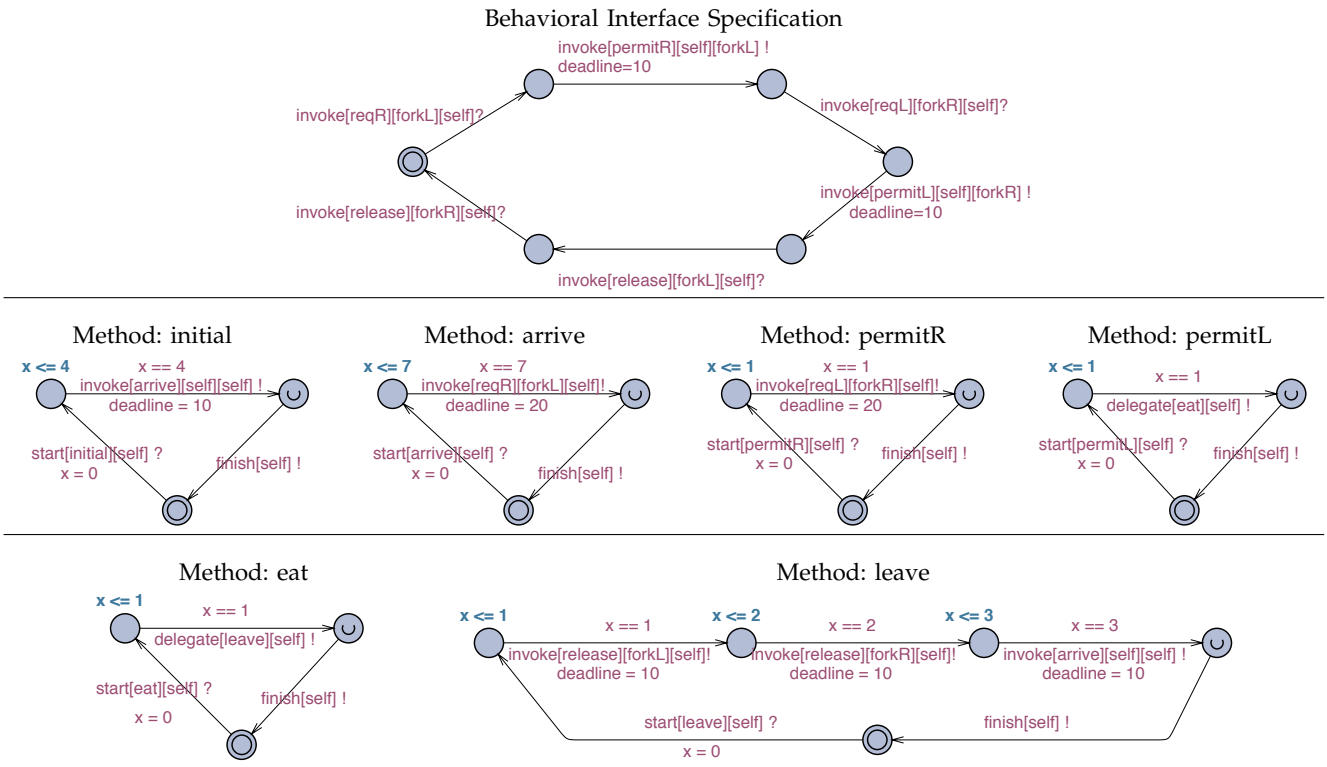

Fig. 5. The timed specification of the Philosopher class

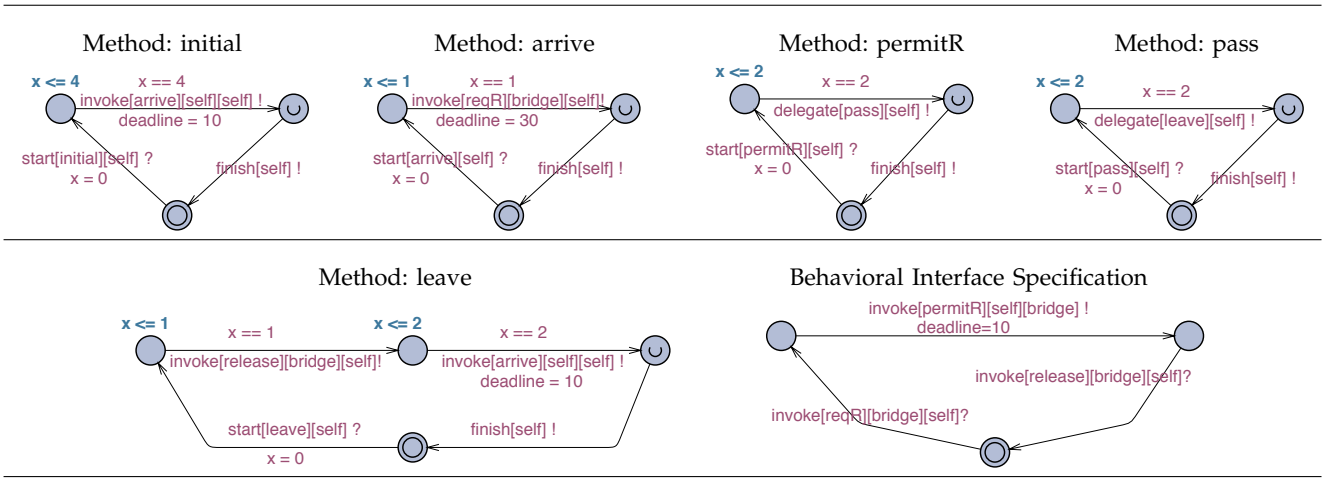

Fig. 6. The timed specification of the Train class