# Model Checking Timed UML State Machines and Collaborations

Alexander Knapp[1], Stephan Merz[1], and Christopher Rauh[2]

http://www.pst.informatik.uni-muenchen.de/projekte/hugo/

[1] Institut für Informatik, Ludwig-Maximilians-Universität München
{knapp, merz}@informatik.uni-muenchen.de
[2] Institut für Informatik, Technische Universität München
rauh@in.tum.de

**Abstract.** We describe a prototype tool, HUGO/RT, that is designed to automatically verify whether the timed state machines in a UML model interact according to scenarios specified by time-annotated UML collaborations. Timed state machines are compiled into timed automata that exchange signals and operations via a network automaton. A collaboration with time constraints is translated into an observer timed automaton. The model checker UPPAAL is called upon to verify the timed automata representing the model against the observer timed automaton.

## 1 Introduction

Object-oriented methods are widely accepted for software development in the business application domain and have also been advertised for the design of embedded and real-time systems [20]. The standard object-oriented software modelling language UML (Unified Modeling Language [3]) accounts for the description of real-time systems by including timed versions of the diagrams used to specify dynamic behaviour [7,2]. Moreover, specialised, UML based real-time modelling tools like Rose RealTime™ or Rhapsody™ provide testing and code generation facilities to support the development process of object-oriented real-time systems, but they lack support for verification.

The UML offers mainly two complementary notations for the specification of dynamic system behaviour: the state/transition-based notion of state machines, and the notion of collaborations that is based on the exchange of messages. In practise, collaborations describe scenarios of desired or unwanted system behaviour and are used in the early phases of software development. The detailed design is later described by state machines. It should therefore be ensured that these two views of a system are indeed consistent by verifying that the state machines may exchange messages according to the test scenarios.

Several tools provide verification support for the state machine view of an untimed UML model via translation into the input languages of model checkers [14,16]. In previous work [19], we have reported on a tool that additionally addresses the problem of consistency between the state machine view and the collaboration view in the untimed fragment of UML.

This paper describes a prototype tool, HUGO/RT, that is designed to automatically verify whether scenarios specified by UML collaborations with time constraints are indeed realised by a set of timed UML state machines. The timed state machines of a UML model are compiled into timed automata, as used by the model checker UPPAAL [13]. A time-annotated collaboration is translated into an observer UPPAAL timed automaton using basically the same techniques as described by Firley et al. [8]. The model checker UPPAAL is called upon to verify the timed automata representing the model against the observer timed automaton. We illustrate the translation and verification procedure by a benchmark case study in real-time systems, the "Generalised Railroad Crossing" (GRC) problem introduced by Heitmeyer et al. [12]. Although our translation is not based on a formal semantics for timed UML state machines, we attempt to be faithful to the informal specification of the UML [18] by following its semantic requirements as closely as possible.

*Related work.* The semantics of timed Harel-Statecharts, the main basis for UML state machines, has been investigated in detail by Damm et al. [5]. In the context of UML, Lavazza, Quaroni, and Venturelli [15] propose a translation of timed state machines into the real-time specification language TRIO, which, however, is not directly model checkable. The translation procedure is demonstrated for the GRC problem, using some extensions of the UML for testing the absence or presence of events, as is possible in Harel-Statecharts. However, compliance of their translation with the UML semantics for state machines, in particular regarding run-to-completion steps, is not obvious. Muthiayen [17] describes the verification of timed UML state machines with the theorem prover PVS. Again, the GRC problem serves as a case study. A translation of UML collaborations with time-constraints into UPPAAL timed automata has been presented by Firley et al. [8], though no implementation seems to be available. Also related is the translation of hierarchical timed automata into flat UPPAAL timed automata by David and Möller [6]. The direct use of this approach for the compilation of UML state machines is limited by the rather different notions of run-to-completion step.

*Overview.* The remainder of this paper is structured as follows: In Sect. 2 we briefly review the notation and semantics of timed automata as used by the model checker UPPAAL. In Sect. 3 we describe a case study, the GRC problem, and provide a UML solution, in passing explaining the UML notation and informal meaning of static structures, timed state machines, and collaborations with time constraints. A more detailed account of the semantics of timed UML state machines and, in particular, of our assumptions on the timing behaviour is given in Sect. 4. The translation of timed UML state machines into UPPAAL timed automata is explained in Sect. 5, the translation of UML collaborations with time constraints into observer UPPAAL timed automata is recapitulated in Sect. 6. We report on the verification results for the GRC case study in Sect. 7. We conclude with a discussion of some loose ends and of future developments.

## 2 Timed Automata

The framework of timed automata was originally defined by Alur and Dill [1]. The model checker UPPAAL that serves as the back end for HUGO/RT expects models to be described as systems of timed automata, extended by primitives for two-way synchronization. We briefly recall the main concepts of UPPAAL timed automata.

A timed automaton is a non-deterministic finite state machine, extended by finitely many real-valued clocks. States may be associated with invariants of the form $x \sim c$ where $x$ is a clock, $c$ is an integer constant, and $\sim \in \{<, \leq\}$. Transitions between states are labelled with triples $(gd, sy, ac)$ where

- $gd$ represents the guard of the transition, expressed as a conjunction of timing constraints $x \sim c$ or $x - y \sim c$ where $x$ and $y$ are clocks, $c$ is an integer constant, and $\sim \in \{<, \leq, =, \geq, >\}$ is a binary relation,
- $sy$ is a (possibly void) synchronization annotation of the form $a!$ or $a?$ that denotes an offer or an acceptance to synchronize over the channel $a$, and
- $ac$ is a set of reset operations $x := c$ on clocks.

Moreover, UPPAAL allows integer variables as well as one-dimensional integer arrays to be declared locally (i.e., for a single automaton) or globally. Transition labels may include constraints on integer variables and array components in their guards, similar to timing constraints, and may specify assignments to variables in their actions, which are executed sequentially. However, clocks and integer variables represent different, incomparable types. In particular, it is not possible to compare the values of a clock and an integer variable, or to assign an integer variable to a clock or vice versa.

The state of a system of timed automata consists of the control state for each automaton, plus a valuation $\nu$ of the clocks and variables. Runs of timed automata are infinite sequences of system states that satisfy the invariants, separated by actions that represent either the passage of time or the (instantaneous) execution of transitions. A transition can be taken only if its guard evaluates to true in the current system state. If the transition carries a synchronization annotation of the form $a?$ or $a!$ then some corresponding transition (labelled by $a!$ or $a?$) of some other timed automaton has to be taken simultaneously. Finally, the resulting system state is obtained by updating the control states of the timed automata involved in the transition, and by updating the valuation $\nu$ according to the action part of the transition label(s), i.e. by resetting clocks and assigning values to variables. Note that the transition is not allowed to occur if the resulting system state would violate the invariant associated with the target location(s). In the case of a synchronization action, the assignments specified by the "sending" automaton (whose transition is labelled by $a!$) are performed before those of the "receiving" automaton. This allows synchronous binary communication to be modelled with the help of global variables. Time passage actions do not affect the control state or the values of the variables, but increase the valuation of all clocks by the same amount, reflecting the assumption of perfect clocks. Again, the resulting system state is required to satisfy

all relevant state invariants. In particular, time-outs can be modelled by state invariants that disallow passage of time beyond the specified deadline.

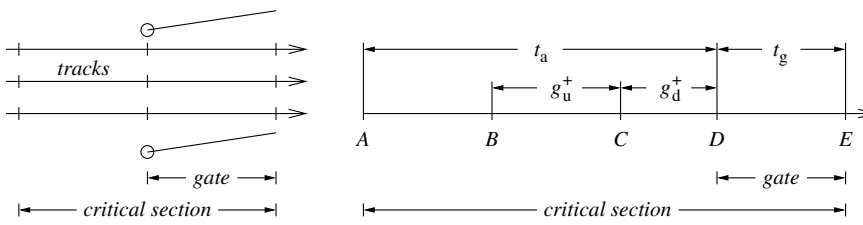
As an additional modelling element, automata locations may be classified as either *committed* or *urgent*. Both of these annotations disallow the passage of time while the location is active. Additionally, committed locations require the next system action to involve a transition whose source state is the committed location. In this way, atomic transactions that involve more than a single transition can be modelled by labelling the intermediate locations as being committed. In particular, multiway synchronization can be modelled with the help of an atomic sequence of binary synchronizations. A channel can be declared *urgent* to disallow the passage of time as soon as synchronization over the channel is enabled.

## 3 GRC Case Study

We illustrate our translation of UML state machines and collaborations into UPPAAL timed automata by means of a UML model for the "Generalised Railroad Crossing" (GRC) case study [12].

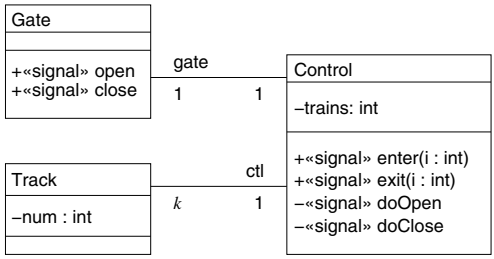
The GRC problem asks for a system operating a gate at a railroad crossing: A gate for several railroad tracks lies in a critical section of the tracks, see Fig. 1(a). All trains pass the critical section in the same direction. The critical section is guarded by two sensors for each track indicating whether a train is entering or exiting the critical section. For every track at most one train passes the critical section, but trains on different tracks may pass at different speeds and overtake each other. Whenever the gate is occupied, i.e., some train is passing the gate, the gate must be closed (safety property). A utility property specifies that within certain tolerance intervals, prior and past being occupied the gate must be open; moreover, when the gate initiates opening, it must become fully open and must stay open for a certain period.

*Timing annotations.* We more concretely assume, see Fig. 1(b), that the minimal resp. maximal time a train may take to pass the distance between entering the critical section at $A$ (the position of the entry sensor) and arriving at the gate at

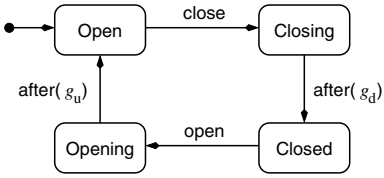



(a) Railroad crossing (b) Timing annotations

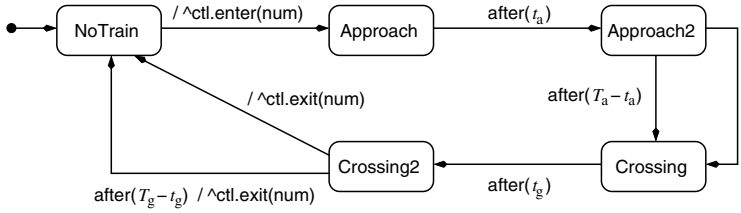
**Fig. 1.** "Generalised Railroad Crossing" problem



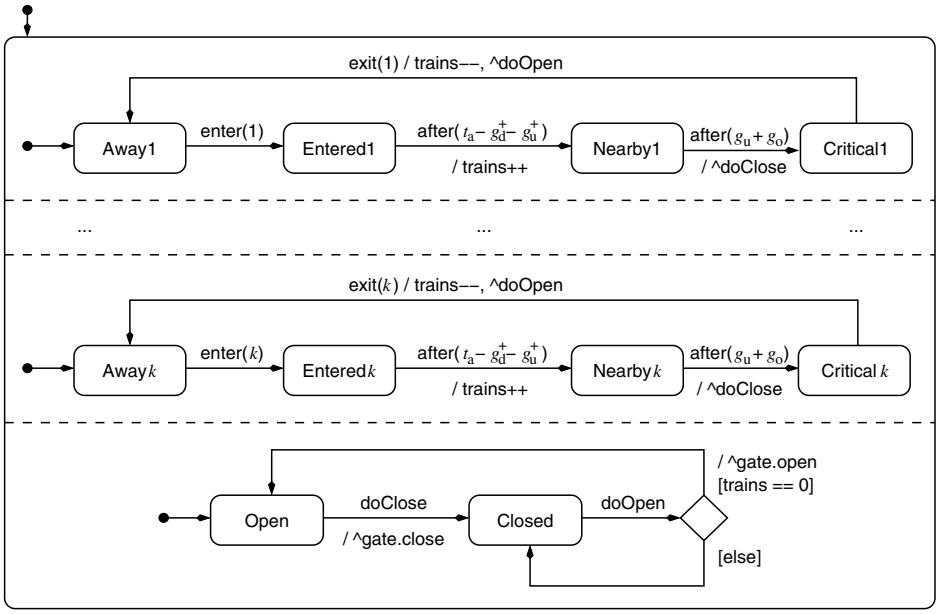

(a) Class diagram




(b) Gate state machine




(c) Track state machine



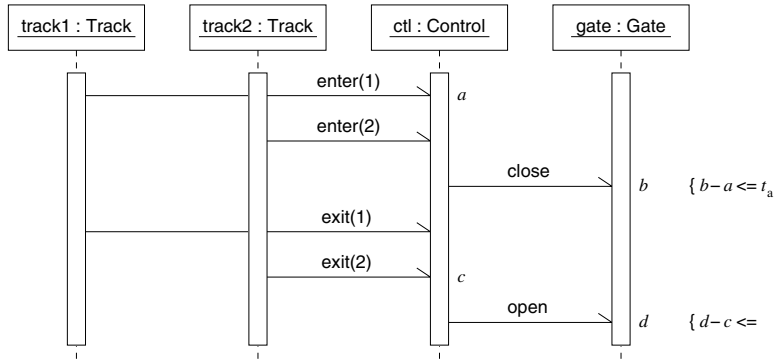

(d) Control state machine

**Fig. 2.** UML model of the Generalized Railroad Crossing.

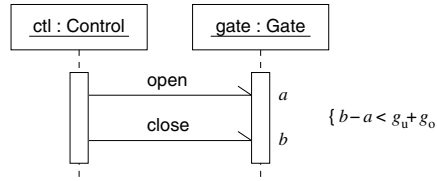
$D$ is $t_\mathrm{a}$ resp. $T_\mathrm{a}$; and that the minimal resp. maximal time a train may take to pass the gate from $D$ to $E$ is $t_\mathrm{g}$ resp. $T_\mathrm{g}$. The gate bars take the time $g_\mathrm{u}$ to go up from fully closed to fully open, and take the time $g_\mathrm{d}$ to go from fully open to fully closed. The minimal period the gate has to stay open is denoted by $g_\mathrm{o}$. Thus, taking into account possible delays in communication, it is at $g_\mathrm{d}^+ = g_\mathrm{d} + \Delta$ before the fastest train may reach the gate after entering the critical section that the gate must initiate closing, that is, at location $C$ determined by $t_\mathrm{a} - g_\mathrm{d}^+$. Moreover, in order to avoid the gate opening partly and then closing immediately again, the gate may only initiate opening when at least $g_\mathrm{u}^+ = g_\mathrm{u} + g_\mathrm{o} + \Delta$ time units remain before the next closing is scheduled, that is, when the train that is closest to the gate is guaranteed to be before location $B$ determined by $t_\mathrm{a} - g_\mathrm{d}^+ - g_\mathrm{u}^+$. We assume that the thereby determined distance between $A$ and $B$ must at least allow for the communication delay, i.e. $t_\mathrm{a} - g_\mathrm{d}^+ - g_\mathrm{u}^+ > \Delta$, in order to prevent premature openings.

*UML model.* Our UML model for the GRC problem is presented in Fig. 2. The *static structure* is set out in the class diagram in Fig. 2(a). Every (instance of) *class* Control refers to a single gate and, vice versa, every Gate is controlled by a single control. Moreover, every Control is connected to $k$ instances of Track, each track knowing its control via ctl and holding its number in num. A Control records the number of trains currently in the critical region in the attribute trains. A Gate reacts to two public signals open and close by initiating opening and closing of the gate. A Control reacts to the public signals enter($i$) and exit($i$) for an integer $i$ reporting a train entering or exiting the critical section on track $i$, as well as to the private signals doOpen and doClose that represent internal requests to open or close the gate.

The dynamics is described by *state machines* for the classes Gate, Track, and Control, see Fig. 2(b)–(d). Each instance of these classes is governed by a separate instantiation of the respective state machine. The state machine for Gate in Fig. 2(b) shows an *initial state* and four *simple states* Open, Closing, Closed, and Opening. The *transition* from *source state* Open to *target state* Closing can only be *fired* when a *signal event* for close is present as its *trigger*. The transition from Closing to Closed will be fired when a *time event* occurs, which is raised after $g_\mathrm{d}$ time units have elapsed since Closing has been *activated*. Thus, in particular, closing and opening a gate takes the required amount of time. Analogously, the state machine for Track in Fig. 2(c) sojourns, once having activated Approach, in this state for the time $t_\mathrm{a}$ before firing the transition to Approach2. However, the state machine may dwell in Approach2 for up to $T_\mathrm{a} - t_\mathrm{a}$ time units or leave this state prematurely via the alternative transition, triggered by a *completion event* which occurs when all activities of a state, of which there are none in this case, have been finished. In the same vein, the transition from NoTrain to Approach may be fired immediately after NoTrain has been activated, as there are again no activities; but this transition also shows an *effect*, viz., that signal enter(num) is raised for the instance of Control referred to as ctl. Hence, the state machines for the tracks simulate the entering and exiting of trains in the critical section; the minimal time a train may take for this distance is $t_\mathrm{a} + t_\mathrm{g}$,




(a) Sequence diagram for the safety property




(b) Sequence diagram for the utility property

**Fig. 3.** UML model of the Generalized Railroad Crossing (cont'd.).

the maximal time is $T_a + T_g$. Finally, the state machine for Control in Fig. 2(b), shows a *concurrent composite state* consisting of several *orthogonal regions* which are again *composite*, though sequential, *states*. The upper $k$ orthogonal regions, the $i$th region handling the entering and exiting of a train on the $i$th track, all provide the same behaviour, ensuring that when a train on track $i$ has entered the critical region, an internal signal doClose requesting the closing of the gate is raised after $t_a - g_d^+ - \Delta$, and that when a train on track $i$ leaves the critical region an internal signal doOpen requesting the opening of the gate is raised. The last orthogonal region actually handles closing and opening of the gate: When in Open and receiving a signal event for doClose, a signal event for close to the instance in gate is sent. However, when in Closed only a signal event for doOpen is reacted to—should a signal event for doClose arrive in this state it is discarded by the whole state machine as there is no other transition taking such an event as its trigger. The transition fired by a signal event for doOpen has two possible target states, linked by a *junction pseudo-state*: If the *guard* trains==0 is true on firing the transition indeed a signal event for open on gate is raised and Open is activated; otherwise the other branch is taken: Closed is first *deactivated* but becomes activated immediately again.

Finally, two test cases for the system behaviour are depicted in the *collaborations*, shown as sequence diagrams, in Fig. 3 that partially describe the safety and utility properties for the UML model.

The sequence diagram in Fig. 3(a) describes a safe behaviour. The diagram specifies that, given two tracks track1 and track2 for a control ctl surveying the gate gate, when a *stimulus* for the first *message*, carrying an enter(1) signal, is received by ctl from track1 at time point $a$ it is possible that the gate receives a stimulus for the third message, carrying signal close, at time point $b$ within $t_{\mathrm{a}}$, independently of when a signal enter(2) is sent from track2 to ctl in-between. Analogously, after the last train exited the gate, as indicated by ctl receiving a signal exit(2) from track2 at time point $c$, the gate may receive a signal open from ctl at time point $d$ such that at most the communication delay $\Delta$ has elapsed after $c$.

The sequence diagram in Fig. 3(b) describes a behaviour that is not allowed to occur: it must be impossible that after the gate received an open signal, a close signal arrives before at least time $g_{\mathrm{u}} + g_{\mathrm{o}}$ has elapsed, as this would contradict the second part of the utility property.

## 4 Model of Computation

The UML specification of the semantics of state machines [18, Ch. 2.12] can be summarized as follows: The actual state of a state machine is given by its *active state configuration* and by the contents of its *event queue*. The active state configuration is the tree of active states; in particular, for every concurrent composite state each of its orthogonal regions is active. The event queue holds the events that have not yet been handled by the machine. The *event dispatcher* dequeues the first event from the queue; the event is then processed in a *run-to-completion* (RTC) step. First, a maximal consistent set of enabled transitions is chosen: a transition is *enabled* if all of its source states are contained in the active state configuration, if its trigger is matched by the current event, and if its guard is true; two enabled transitions are *consistent* if they do not share a source state. For each transition in the set, its *least common ancestor* (LCA) is determined, i.e. the lowest composite state that contains all the transition's source and target states; the transition's main source state, that is the direct substate of the LCA containing the source states, is deactivated, the transition's actions are executed, and its target states are activated.

The UML semantics deliberately does not prescribe the timing assumptions that underly the computation of timed state machines. For example, it is left unspecified whether state transitions are instantaneous or durative, although the zero-time assumption adopted for Harel-Statecharts [10,11] is explicitly mentioned as a possible model [18, p. 2-161]. Similarly, arbitrary queueing delays are allowed to occur between the time an event is received by a state machine and the time it is dispatched for processing.

Because we are interested in a precise analysis of timed UML state machines, we have to assume a specific computational model. Following ideas from timed automata and synchronous languages, our basic assumption is that noticeable delays are only due to communication between objects whereas local computation is (infinitely) fast. Formally, we make the following assumptions:

1. The run-to-completion (RTC) step performed locally by a state machine takes no time. Similarly, the event queue is eager to dispatch events it has received.
2. The delay between the sending of an event and its reception at the target object is bounded by a constant $\Delta$.
3. A state machine may delay arbitrarily before generating completion events, unless that delay is restricted by an explicit constraint.

Our basic tenet is that the specifier should have complete control over the behavior of the model. For example, the event queue is an implicit part of every state machine and is outside the control of the specifier. If we allowed the event queue to introduce arbitrary delays it would obviously be impossible to guarantee any lower bounds on the response time. We have therefore chosen to impose a zero-time assumption on the behavior of the queue. Should the specifier wish to allow for delays, these can always be modelled explicitly.

Assuming zero-time behavior of the event queue and the RTC step implies that usually, the times of reception, dispatch, and consumption of an event are the same. However, after sending a synchronous call event to another object, a state machine will be blocked until the notification about the dispatch of the event at the receiver machine has arrived at the sender. During this period, events may be received by the event queue, but they will be dispatched only after the synchronous call event has been handled.

Our second assumption is similar in spirit because, again, the mechanism for inter-object communication is an implicit part of the model and cannot be controlled by the specifier. Although we model inter-object communication as being time-consuming, we introduce a user-definable constant $\Delta$ to represent the maximum network latency. We assume, however, that messages that represent internal signals or operations are not sent over the network, and therefore do not incur any communication delay. Obviously, this model of communication could be refined, for example by imposing a minimum communication delay or by distinguishing several degrees of "remoteness" (e.g., faster communication within a single package etc.).

Finally, we do not restrict the delay before raising completion events because these concern a part of the model that is under the control of the modeller, using either time events or clock constraints. For example, the transition from NoTrain to Approach in the track state machine of Fig. 2(c) can be delayed arbitrarily. On the other hand, once the completion event has been raised, it will be inserted into the event queue without any further delay.

## 5 Representation of UML State Machines in uppaal

We now describe in detail our approach to compiling a system of UML state machines into a set of UPPAAL timed automata. For simplicity, the current version of HUGO/RT assumes that there is only a single instance of any class declared in the UML model, that the names of operations and signals are distinct, and that

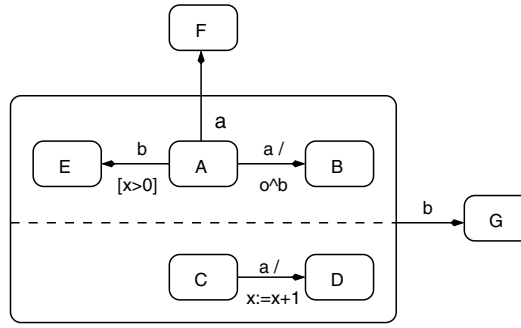
events do not carry parameters. These assumptions could be easily removed at the expense of a slightly more elaborate naming scheme.

### 5.1 State Configurations and Transitions

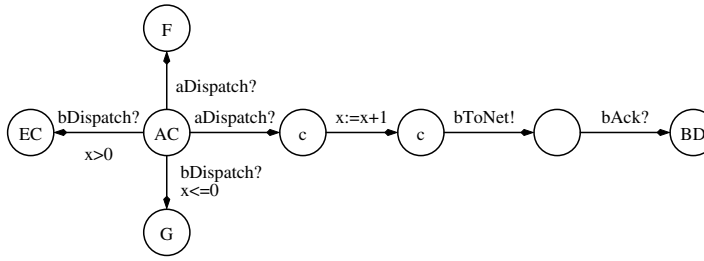
Although both UML state machines and timed automata describe state transition systems, the hierarchical state configurations of UML state machines have to be encoded as states of "flat" timed automata. The original implementation of HUGO described in [19] was based on a translation of untimed UML state machines to PROMELA, the input language of the SPIN LTL model checker, that represented each substate of a UML state machine by a separate process. The main benefit of that strategy was a high degree of modularity and flexibility, at the cost of an inefficient state representation, which was largely offset by the state-space compression techniques available in SPIN. Because UPPAAL offers neither compression nor structured data types beyond one-dimensional integer arrays, we decided to flatten the state configurations and to compile each UML state machine into only two timed automata that represent, respectively, the transitions of the state machine and its associated event queue. A state configuration of the UML state machine, described as a tree of states, is encoded by a single location of the first timed automaton, and any attributes that are declared in the UML class diagram are translated into local variables of the UPPAAL timed automaton.

Possible transitions of the UML state machine are represented in the UPPAAL model as sequences of transitions. Given a location of a timed automaton that represents a configuration tree of the UML state machine, we first determine the set of events (including time and completion events) that may trigger a transition, starting at the leaves of the configuration tree. Whenever a higher-level state may react to the same event as a lower-level state, the negation of the guard associated with the lower-level transition is added to the guard of the higher-level transition. This ensures that transitions originating from inner states take priority over outermost transitions, as prescribed by the UML semantics. For each event $e$, we compute the sets of transitions that may be triggered by $e$ and successor configurations, by calculating the transition's main source and target states. In particular, we must consider the case that the same event is consumed by states in several orthogonal regions. All pertinent guards are copied to the first transition of the UPPAAL model, as is a synchronization that indicates consumption of the event.

We then consider the effects of the UML transition. First, we model the deactivation of states, beginning at the leaf states of the source configuration tree, and working upwards towards the main source state. Updates to instance variables are again copied verbatim as assignments to the corresponding local variables of the UPPAAL model. For every signal sent in the UML model, we generate an intermediate committed state to let the UPPAAL model synchronize over an appropriate channel (observe that UPPAAL transitions may carry at most one synchronization annotation). Next, we translate the effects that are explicitly given by the UML transition in a similar way. If the transition generates a




(a) UML state machine




(b) UPPAAL timed automaton (fragment)

**Fig. 4.** Translation of UML transitions.

synchronous call the UPPAAL model enters a non-committed intermediate state, awaiting the notification of dispatch to arrive from the receiving machine. Our current prototype does not allow sequences of call actions to occur in a transition. Finally, target states are being activated, and any entry actions are translated to corresponding actions in the UPPAAL model. This part of the translation is symmetric to deactivation, but starting at the main target state and working downwards towards the leaves. The current version of HUGO/RT, unlike the untimed version, does not support "do" activities associated with states; this is partly offset by the possibility to delay, abstracting from changes of attribute values.

Finally, we add loops for all events that occur as input events of the state machine but do not trigger a transition from the current state configuration. These transitions correspond to situations where the current state configuration does not react to the current event; the UML semantics prescribes that the event should then be discarded. HUGO/RT does not currently handle deferred events.

Figures 4(a) and 4(b) show an excerpt of the translation of a hypothetical UML state machine into the corresponding UPPAAL timed automaton that exhibits some of the difficulties that can arise. In particular, event a may trigger two different transitions in the state configuration that consists of the states A

and C that lead, respectively, to the target configurations BD and F. The first of these transitions is represented by a sequence of transitions in the UPPAAL model that involve consumption of the trigger event, updates of local variables, and network communication. For this example, we have assumed that b is a call event that should be sent to the object o. After sending the event over the network (cf. Sect. 5.3), the timed automaton waits in a non-committed intermediate state for the corresponding acknowledgement to arrive.

The same state configuration AC may react in two different ways to event b. The innermost transition (resulting in configuration EC) is prioritized; therefore, the outermost transition can be taken only if the guard x>0 is false.

*Completion events.* Transitions of a UML state machine for which no trigger event is shown are triggered by an implicit completion event. Simple states raise a completion event after all internal activity has terminated. Composite states raise a completion event when all orthogonal regions have reached a designated final state.

Since HUGO/RT does not consider "do" activities or non-atomic entry actions, we generate a completion event for any simple state with outgoing completion transitions via a transition to an intermediate "completion" state[1]. Similarly, a completion event for a composite state with outgoing completion transitions is generated whenever all its orthogonal regions have reached final states; this can be readily seen from the configuration. Consumption of completion events is similar to the consumption of regular events discussed above, except of course that the same completion event cannot be consumed by different states. Completion events are discarded if the state to which it corresponds is no longer active; this may occur if another (completion) event has caused some containing composite state to be deactivated. As explained in Sect. 5.2, the implementation of event queues ensures that completion events take priority over signal and call events.

*Compound transitions.* The UML allows several transitions to be connected by pseudo-states; in particular, the resulting compound transition may have several source or target states. The translation of fork and junction transitions poses no particular problems. Join transitions are required by the UML to be triggered by completion transitions from all of its source states. Our UPPAAL translation uses a similar technique as for the completion of composite states: for each source state $s$ of a join transition, we add an auxiliary state that indicates that $s$ has completed; completion of the last source state, which is again evident from the active state configuration, fires the join transition.

*Time events.* Transitions of timed UML state machines may be triggered by the elapse of time, indicated by an annotation of the form after($d$) where $d$ is

[1] One may be tempted to suppress the completion event altogether in such a simplified model, but note that a completion event should nevertheless be handled in a separate RTC step.

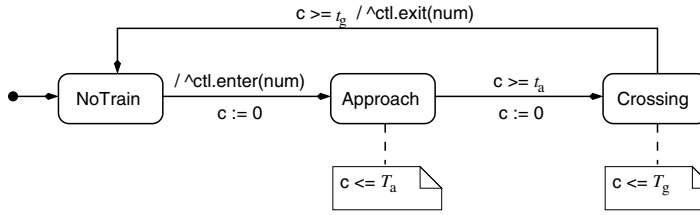



**Fig. 5.** Track state machine with clock.

a non-negative integer constant. HUGO/RT defines a clock $c_s$ for every state $s$ with an outgoing time transition. Every location of the timed automaton that corresponds to a state configuration containing $s$ is required to satisfy the invariant $c_s \leq d$; incoming transitions to any such locations reset $c_s$ to 0. A time event is raised during a transition from $s$ to an intermediate state, guarded by the condition $c_s = d$. Consumption of time events is analogous to that of completion events.

The support UML offers for time annotations in state machines is rather limited, and it can be cumbersome to model real-time systems using only basic time events of the form after($d$), which represent precise deadlines. For example, the track state machine shown in Fig. 2(c) combines time transitions and completion transitions to specify upper bounds on the occurrence of transitions, and introduces auxiliary states to specify lower bounds. We therefore extend the UML notation by allowing clocks to be declared explicitly in a UML class diagram. These clocks can be tested for in transition guards, and can be reset as the effect of transitions, in the same way as this is possible in UPPAAL timed automata. Similarly, clock invariants may be associated with the states of a UML state machine to model timeouts. For example, Fig. 5 shows an alternative presentation of the track state machine using an explicit clock c. This modest addition makes the notation more expressive because strict comparisons such as $x < c$ can not be expressed using triggers of the form after($d$). Besides, time constraints become more localized, which should make the models easier to understand.

### 5.2 Representing the Event Queue

The second timed automaton that is generated for every state machine of the UML model represents its associated event queue. A local array variable of user-definable capacity holds the current contents of the queue such that the first array element represents the head of the queue (event types are encoded as integer constants). A local integer variable indicates the number of events in the queue. The queue reacts to incoming events by synchronizing on its input channel and appending the transmitted event to the current contents of queue. A possible overflow is indicated by moving to a distinguished overflow (sink) state. Completion events are enqueued similarly, but are prepended to the contents of the queue, shifting all array elements by one position towards the end. This

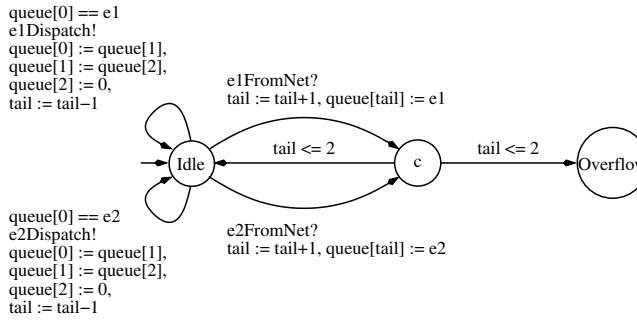



**Fig. 6.** Event queue with capacity two for two events.

ensures that completion events take precedence over regular events, as prescribed by the UML semantics. Observe that the event queue is always ready to accept a new event unless an overflow has occurred.

For every input event of the state machine, the UPPAAL model declares an urgent channel shared by the automata representing the event queue and the proper state machine. The queue communicates the event at the head of the queue to the associated state machine by offering a synchronization on the associated channel. The event is then dequeued by shifting all array elements one position towards the head of the queue, decrementing the length of the queue, and assigning a null value to the last array position. Because the channels representing input events are declared urgent and the state machine accepts synchronization on all these channels in every location where an RTC step can be started, no time is allowed to elapse whenever an RTC step is enabled. Figure 6 shows a sample event queue automaton with capacity two that can hold two types of events.

Our style of implementation of the event queue, which would be inefficient in a conventional programming language, ensures that identical configurations of a state machine are mapped to a unique system state of the UPPAAL model, and that state repetitions can therefore be reliably identified. If we used a more conventional implementation based on two index positions corresponding to the head and the tail of the queue, identical state configurations could be mapped to UPPAAL states that differ in the particular layout of the queue array.

### 5.3 Representing the Network

The final addition to complete the UPPAAL model associated with a system of state machines is a timed automaton that represents messages in transit between different state machines; recall from Sect. 4 that we consider remote communication to be time-consuming. The network automaton essentially consists of a user-defined number of buckets that may hold messages. For every event $e$ it listens on a global channel $e$ToNet for $e$ being sent by the state machines and then places $e$ in the lowest-numbered unused bucket. With every bucket we as-

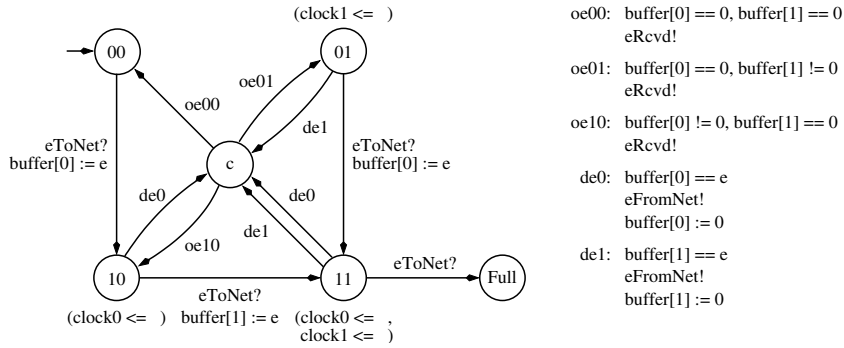



**Fig. 7.** Network with capacity two (transition annotations to the right).

sociate a clock that is reset when the bucket is filled, and times out after $\Delta$ time units have elapsed. If the bucket is full it may offer synchronization on the corresponding input channel of the receiving object's event queue, communicating the event to the receiving state machine. Notifications of dispatch of call events are handled slightly differently, as they are communicated directly to the state machine, bypassing the event queue.

For UML models that also specify a collaboration to be checked, the network also informs the observer timed automaton generated from the collaboration (cf. Sect. 6 below) whenever it communicates an event to the receiving state machine so that the observer automaton may react appropriately. Figure 7 presents a network with capacity two.

The network described above only imposes an upper bound on the communication delay, allowing messages to be reordered. We have also experimented with a more efficient, though perhaps less realistic, network model where every message is delayed by a fixed amount of time. Because both implementations offer the same external interface, they can be exchanged for each other easily.

## 6 Representation of UML Collaborations in UPPAAL

We briefly describe the translation of UML collaborations with time constraints into observer UPPAAL timed automata. The translation is similar to the construction by Firley et al. [8], but includes stuttering states, i.e., states that allow arbitrary stimuli to occur in-between the stimuli that are required by the messages of the UML collaboration. Following the UML, we thus view collaborations as incomplete specifications of possible system runs where arbitrary message exchange may occur between the explicitly specified messages.

We assume a total order on the messages in an interaction. Each message $m$ is represented by an UPPAAL channel $m$Rcvd over which the observer automaton learns of a stimulus for message $m$ being received by an instance. A constraint of the form $v - u \sim c$ with $\sim \in \{<, \leq, \geq, >\}$ is associated with a clock $x_{vu}$.

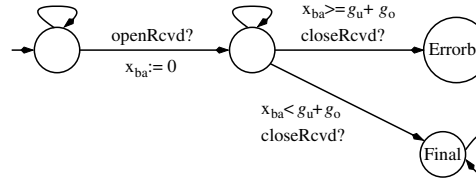



**Fig. 8.** Observer UPPAAL timed automaton for collaboration in Fig. 3(b)

Each state of the observer UPPAAL timed automaton checks either the occurrence of a reception of a stimulus according to the order of the collaboration or the violation of a timing constraint. The automaton registers the reception of a stimulus complying to message $m$ by a transition accepting communication on channel $m$Rcvd. Furthermore, if the reception of message $m$ is annotated by $u$ for a timing constraint $v-u \sim c$ the clock $x_{vu}$ is initialised when a stimulus for $m$ is successfully registered. Conversely, if the reception of message $m$ is annotated by $v$ for a timing constraint $v - u \sim c$, there are two transitions accepting communication on channel $m$Rcvd: A transition guarded by $\neg(x_{vu} \sim c)$ leads to an error state, indicating that the timing constraint is violated; another transition guarded by $x_{vu} \sim c$ enables the remaining messages. Each state allows for an arbitrary stuttering of stimuli. Reaching the final state indicates the successful performance of the collaboration.

Figure 8 illustrates this construction for the GRC test case collaboration in Fig. 3(b) where each looping transition abbreviates arbitrary stuttering, i.e., offering synchronisation on every channel of the form $m$Rcvd?.

The registering procedure for stimuli that we employ in our translation uses the assumption that the sender and receiver of a stimulus are uniquely determined by the stimulus. A more elaborate scheme checking the sender and receiver stored in additional global variables is easily devised [8]. Note, however, that Firley et al. [8] check the originator of a stimulus before communication is accepted which may become problematic when different instances send stimuli for the same signal to the same instance.

## 7 Verification

HUGO/RT analyzes a model by calling on UPPAAL to verify that the final state of the observer timed automaton is reachable in the model that consists of the timed automata representing the UML state machines and the observer automaton generated from the collaboration. For our case study, the feasibility of the collaboration shown in Fig. 3(a) is confirmed in a fraction of a second (all timings were taken on a Pentium III with 768MB of memory running UPPAAL 3.2.2 on Linux 2.2.16).

Some correctness properties of systems can be expressed by requiring certain executions to be impossible. For example, we can check the second part of the utility property that requires the gate never to open for less than $g_o$ time units

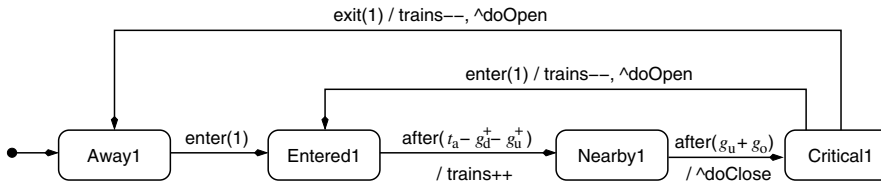



**Fig. 9.** Corrected control state machine for the GRC problem (representative region).

by verifying that the collaboration shown in Fig. 3(b) is not feasible. The verification of this property requires an exhaustive search of the model and takes approximately 11 seconds.

However, not all interesting properties are expressible using UML collaborations with time constraints. In particular, UML collaborations do not allow to check for the *absence* of signals as can be done, for example, using live sequence charts [4]. We have therefore resorted to model checking invariants to verify the safety and utility properties stated for the GRC case study. The basic safety property requires the gate to be down whenever a train is crossing, expressed as the formula

$$\forall\Box\big((\mathsf{Track1.Crossing} \vee \mathsf{Track2.Crossing}) \Rightarrow \mathsf{Gate.Closed}\big)$$

Our first attempt to verify this formula resulted in a counter-example where a train was crossing but the gate was closing, with precisely $g_\mathrm{d}$ time units elapsed since the Closing state was entered. Rather than adding some safety margins, we deemed this behavior acceptable and checked for a weaker invariant that also allowed for this boundary case.

Much to our surprise, UPPAAL again produced a counter-example: very briefly after a train had left the crossing, a second train entered on the same track (this is allowed by our model since a completion event for the NoTrain state may be generated immediately after the state has been activated). Because of variable network delays, the corresponding enter signal actually arrived at the controller before the preceding exit signal, that is, when the corresponding region of ctl was still in its Critical state. Because that state does not define a transition for the enter signal, it was discarded, and the gate was opened when the exit signal arrived, leaving the gate open while the second train was approaching and crossing.

This counter-example represents an actual error in our gate controller, which none of us had realized, and which would have been difficult to find by simulation alone. The error can be fixed by adding a transition from the Critical to the Entered states of the controller state machine of Fig. 2(d) that is triggered by an enter signal, decrements the attribute trains, and raises a doOpen signal, cf. Fig. 9. (Observe that the following exit signal will be discarded at the Entered state.) Another solution, which is not currently possible to analyze using HUGO/RT, would be to mark the enter signal as deferred in state Critical.

Rerunning the verification over the modified model establishes that the invariant is now satisfied; the analysis takes ca. 2.5 seconds.

We would also like to verify the first part of the utility property of the GRC problem that asserts that the gate is closed only if some train is crossing or close to the gate. A first idea would be to express this property by the formula

$$\forall\Box\big(\mathsf{Gate.Closed} \implies \bigvee_{i=1,2} \mathsf{Track}i.\mathsf{Crossing} \vee (\mathsf{Track}i.\mathsf{Approach} \wedge \mathsf{Track}i.\mathsf{c} \geq t_a)\big)$$

in terms of the track state machine shown in Fig. 5. Unfortunately, this property cannot hold because the gate will still be closed for a short while after the trains have left. A possible solution is to enhance the model by an additional clock measuring the time since a train has left the gate.

Finally, we want to establish the absence of deadlocks in our solution of the GRC problem. UPPAAL provides a designated state formula deadlock to identify deadlock states. Our translation, however, may introduce extra deadlocks: a full network cannot accept any further stimuli, the event queues may overflow, and the observer automaton may run into an error state. In all of these cases designated deadlock states (Full, Overflow, Error$vu$) are entered. In fact, UPPAAL reports that these are the only deadlocks of the translated model, confirming the absence of deadlocks in the UML model.

## 8 Conclusions

We have described procedures for translating timed UML state machines and UML collaborations with time constraints into UPPAAL timed automata. The translation has been implemented in a prototype tool called HUGO/RT. The results of model checking the translation of scenarios given by UML collaborations with time constraints against timed UML state machines have been illustrated for the "Generalised Railroad Crossing" case study.

The current prototype shows several limitations: HUGO/RT by now only handles a subset of the possibilites of UML state machines. Most prominently, history pseudo-states and deferred events still need to be implemented, events cannot have parameters, and there can only be a single instance of any given class. Some optimizations for more efficient analysis should also be considered. For example, the number of clocks could be minimized by reusing clocks in different states. More importantly, the expressiveness of UML collaborations to describe correctness properties is rather limited. We therefore plan to integrate live sequence charts as a specification formalism into HUGO/RT. For specifying state-based properties on the level of state machines it would be useful to integrate constraints in UML's textual annotation language OCL.

Even so, HUGO/RT is a first step towards the application of model checking techniques "behind the scenes" to real-time object-oriented designs, even spanning several phases of software development. Both the design model and the properties of a system are described in the unifying framework of the UML. For seamless integration with existing tools, HUGO/RT imports state machines and

collaborations from the XMI (XML metadata interchange) output produced by standard UML editors. The time overhead for compiling into UPPAAL models is tolerable and the time for verification is encouraging.

## References


1. Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theo. Comp. Sci.*, 126:183–235, 1994.
2. Rodolphe Arthaud, Udo Brockmeyer, Werner Damm, Bruce P. Douglass, Francois Terrier, and Wang Yi, editors. *Proc. Wsh. Formal Design Techniques for Real-Time UML*, York, 2000. http://wooddes.intranet.gr/workshop.htm.
3. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison–Wesley, Reading, Mass., &c., 1998.
4. Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
5. Werner Damm, Bernhard Josko, Hardi Hungar, and Amir Pnueli. A Compositional Real-Time Semantics of STATEMATE Designs. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Proc. Int. Symp. Compositionality (Revised Lectures)*, volume 1536 of *Lect. Notes Comp. Sci.*, pages 186–238. Springer, Berlin, 1998.
6. Alexandre David and M. Oliver Möller. From HUPPAAL to UPPAAL — A Translation from Hierarchical Timed Automata to Flat Timed Automata. Technical Report BRICS RS-01-11, Department of Computer Science, Aarhus Universitet, 2001.
7. Bruce P. Douglass. *Real-Time UML*. Addison-Wesley, Reading, Mass., &c., 1998.
8. Thomas Firley, Michaela Huhn, Karsten Diethers, Thomas Gehrke, and Ursula Goltz. Timed Sequence Diagrams and Tool-Based Analysis — A Case Study. In France and Rumpe [9], pages 645–660.
9. Robert B. France and Bernhard Rumpe, editors. *Proc. 2nd Int. Conf. UML*, volume 1723 of *Lect. Notes Comp. Sci.* Springer, Berlin, 1999.
10. David Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comp. Program.*, 8(3):231–274, 1987.
11. David Harel and Eran Grey. Executable Object Modeling with Statecharts. *Computer*, July:31–42, 1997.
12. Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Comparing Different Approaches for Specifying and Verifying Real-Time Systems. In *Proc. 10th IEEE Wsh. Real-Time Operating Systems and Software*, pages 122–129, New York, 1993.
13. Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. J. Softw. Tools for Techn. Transfer*, 1(1–2):134–152, 1997.
14. Diego Latella, Istvan Majzik, and Mieke Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-Checker. *Formal Aspects Comp.*, 11(6):637–664, 1999.
15. Luigi Lavazza, Gabriele Quaroni, and Matteo Venturelli. Combining UML and Formal Notations for Modelling Real-Time Systems. In *8th Europ. Conf. Software Engineering*, Wien, 2001.
16. Johan Lilius and Iván Porres Paltor. Formalising UML State Machines for Model Checking. In France and Rumpe [9], pages 430–445.

17. Darmalingum Muthiayen. *Real-Time Reactive System Development — A Formal Approach Based on UML and PVS*. PhD thesis, Concordia University, Montreal, Canada, 2000.
18. Object Management Group. Unified Modeling Language Specification, Version 1.4. Specification, OMG, 2001. http://cgi.omg.org/cgi-bin/doc?formal/01-09-67.
19. Timm Schäfer, Alexander Knapp, and Stephan Merz. Model Checking UML State Machines and Collaborations. In Scott Stoller and Willem Visser, editors, *Proc. Wsh. Software Model Checking*, volume 55(3) of *Elect. Notes Theo. Comp. Sci.*, Paris, 2001. 13 pages.
20. Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, New York, 1994.