

Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite*

Sven Burmester**, Holger Giese, Martin Hirsch, and Daniela Schilling**

Software Engineering Group, University of Paderborn,
Warburger Str. 100, D-33098 Paderborn, Germany
[burmi | hg | mahirsch | das]@upb.de

Abstract. Model checking of complex time extended UML (UML/RT) models is limited today due to two main obstacles: (1) The state explosion problem restricts the size of the UML/RT models which can be addressed and (2) standard model checking approaches cannot be smoothly integrated into the usually incremental and iterative design process. The presented solution for incremental design and verification with UML/RT within the FUJABA Real-Time Tool Suite overcomes these two obstacles by applying a compositional reasoning approach [1] that is based on a restricted notion of UML patterns and components. A mapping of a subset of the UML/RT component model and additional real-time extensions for UML state diagrams to HUppaal is presented which enables the automatic, compositional formal verification of partial models such as patterns and components by means of a model checking PlugIn. The developed tool support makes an incremental and iterative design and verification process possible where only the patterns and components which have been modified have to be rechecked rather than the whole UML/RT model.

1 Introduction

Current distributed embedded real-time systems become increasingly complex, and a large fraction of development cost and time is consumed by the development of the included control software. As these systems are often used in a safety-critical environment, this software has to meet strict quality criteria. The presented approach and the related FUJABA Real-Time Tool Suite¹ therefore support a rigorous system design and formal verification of UML/RT models to enable the incremental design of software for safety-critical systems.

To address the complexity of the system design at the architectural level, we propose to use a subset of the current UML 2.0 proposal [2] which includes the basic concepts and notations of UML/RT [3]. The employed subset is described in more detail in [4]. Our approach requires that *all* collaborations are described via a connector and multiple ports in form of reusable patterns [1]. These patterns are further used to derive the

* This work was developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

** Supported by the International Graduate School of Dynamic Intelligent Systems.

¹ www.fujaba.de

required component behavior in a process that integrates these design activities with verification.

To resolve the state explosion problem for large UML/RT models, we employ a formal semantics definition for the employed subset of the UML/RT component model and real-time extension of UML state diagrams which enables the automatic, compositional formal verification of the models by means of components and patterns [1]: At first, complex software systems are composed from domain-specific patterns which model a particular part of the system coordination in a well-defined context. The correctness of each pattern can be verified individually. As the complete communication of a pattern is described within its roles and the number of roles is fixed the verification task is usually feasible.

The composition of these patterns to describe the complete component behavior and the overall system behavior is prescribed by a rigorous syntactic definition which guarantees that the verification of the components and the system behavior does not invalidate the results of the verification of the individual patterns.

The presented approach further supports the logical modeling of real-time behavior by means of a real-time extension of UML state diagrams named Real-Time Statecharts (RTSC) [5]. Automatic code generation ensures a proper mapping of the high level timing attributes to the physical model level. Therefore, verification results obtained for the UML models can be safely transferred to the implementation due to the automatic code generation. In addition, a formal verification at the level of the abstract logical UML models is usually feasible while this might not be the case for the physical level models (cf. *UML Profile for Schedulability, Performance, and Time* [6]).

Another obstacle for the applicability of formal verification in the context of model-based development is the more or less batch-oriented integration of model checking into the tools which is at odds with the incremental character of human design activities. We address this problem by integrating the constraints directly into the UML model and explicitly maintain the status of each constraint within the model consistency management of the FUJABA CASE tool [7]. The designer is thus always aware whether a constraint holds, is violated, or requires formal verification. As the compositional verification often permits to check required constraints within seconds, the tool further permits to check local constraints in the background rather than waiting for the user to initiate the model checking explicitly.

The paper is organized as follows: After reviewing the proposed UML/RT-based model-driven development of the software in Section 2, we describe the mapping of the logical model to the HUppaal environment to enable model checking (see Section 3). Then, the incremental design and verification of constraints and the integration between the constraints and consistency management are outlined in Section 4. We finally review relevant related work in Section 5 and present our final conclusions and an outlook on future work.

2 Model Based Development with UML/RT

As a concrete example for a complex mechatronic product, we use a simplified version of the software for the RailCab research project². The vision of the RailCab project is a

² <http://www-nbp.upb.de/en/index.html>

mechatronic rail system where autonomous shuttles apply the linear drive technology, used in the Transrapid, but travel on the existing passive track system of the standard railway. One particular problem, which has been previously described in [1], is to reduce the energy consumption due to air resistance by coordinating the autonomously operating shuttles in such a way that they form convoys whenever possible. Such convoys are created on-demand and require small distances between the shuttles in order to achieve significant economies. Coordination between the speed control units of the shuttles becomes a safety-critical aspect and results in a number of hard real-time constraints, which have to be addressed when building the control software of the shuttles.

2.1 Patterns and Components

Within our modeling and verification approach for the software of complex real-time systems [1], modeling is divided into modeling the interaction between components of the system by reusable *coordination patterns* and modeling the detailed behavior of the components by relating to the behavior of the applied patterns.

A pattern describes communication and therefore consists of multiple communication partners, called *roles*. Roles interact through ports which are linked by a connector. The communication behavior of a role is specified by a RTSC and is restricted by an invariant. The behavior of the connector is described by another RTSC that is used to model channel delay and reliability, which are of crucial importance for real-time systems. The overall behavior of a pattern is restricted by a pattern constraint, whereas the behavior of a role can be restricted by a role invariant.

Within the shuttle example, distance coordination between two shuttles is modeled as a pattern. This DistanceCoordination pattern consists of two roles, the frontRole and the rearRole and one connector that models the wireless radio link between the two shuttles. The pattern specifies the behavior needed to coordinate two successive shuttles. The main requirement of the pattern is to ensure that no rear-end collision happens when the first shuttle has to brake suddenly, e.g. in case of an emergency. If the shuttle is the head of a convoy, it may brake only with reduced force, because another shuttle drives behind it with a reduced, minimal distance and which reacts with delay. We thus require that the front shuttle must not brake with full power if it is in *convoy* mode. For the rear shuttle, we require that it does brake with full power. These two requirements are called role invariants. On the other hand, the overall pattern constraint forbids the rear role to be in mode *convoy* while frontRole is in mode *noConvoy*.

The pattern constraint and role invariants can be annotated to the pattern resp. its roles using ATCTL³ formulas. The pattern with its annotated constraint and invariants is depicted in Figure 1. The behaviors are presented in the remainder of this section.

After the patterns have been specified, the concrete software components can be built. Components are designed by coordinating and refining each role RTSC of the involved patterns. The refinement has to respect the role RTSC (i.e. not add additional behavior or block guaranteed behavior) and additionally has to respect the guaranteed

³ ATCTL is the subset of timed computation tree logic which only contains **always** path operators.

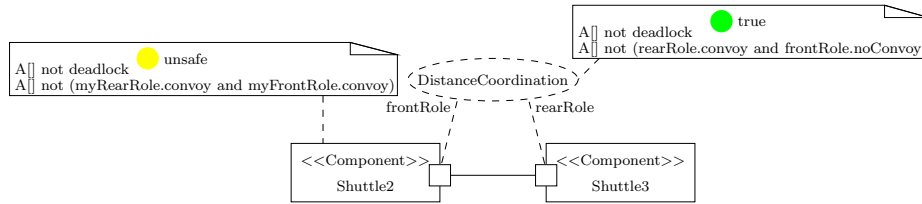


Fig. 1. The DistanceCoordination pattern

behavior of the roles in form of their invariants. An additional internal RTSC for coordination is used to describe the required coordination of the refined roles. We further refer to the refined roles as component ports or ports in short.

In our example, the shuttle component must conform to the DistanceCoordination pattern and has to operate as both a rearRole and a frontRole as it may be followed by another shuttle as well as itself can follow another shuttle. The synchronization RTSC on the other hand is responsible for the decision whether to build or break a convoy and to synchronize the two ports. Figure 2 shows several shuttle components with the frontRole as well as the rearRole within their ports.

The complete system is built by a number of components and patterns which overlap at their ports resp. their roles. Thus, we can build an arbitrarily complex combination of shuttle components connected by the DistanceCoordination pattern using multiple instances that are accordingly adjusted to permit their composition. Figure 2 shows a system with four shuttle instances and three instances of the distance coordination pattern. Within the figure, the squares at the component borders depict the pattern roles as well as the component ports, which indicates the overlapping of patterns and components w.r.t. the ports/roles.

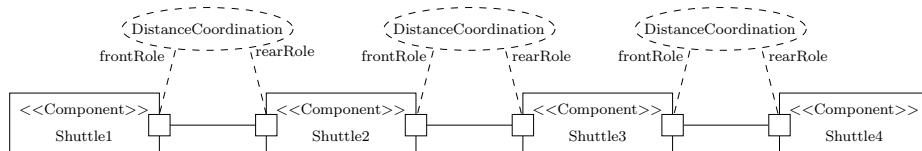


Fig. 2. System with four shuttle instances and three instances of the distance coordination pattern

2.2 Real-Time Statecharts

As described above, the role behaviors of a pattern are specified by RTSC. We apply RTSC [8] instead of pure UML state machines because the patterns have to fulfill real-time requirements.

In RTSC, time-dependent behavior is modeled by using clocks which are reset when firing transitions or entering resp. leaving a state. The idea of introducing clocks stems

from the concept of Timed Automata [9] and permits to emulate the simpler UML timing constructs such as after and when.

Clocks are more expressive than the ordinary UML timing constructs. Using different clocks enables us to refer to different points in time. A single clock can be referenced in multiple states and therefore permits timing constraints which are not bound to a single state like the after-construct.

In addition to UML state machines, RTSC offer transition priorities, asynchronous communication, and the integration of complex data models.

Similar to Timed Automata, transitions are extended by so called *time guards* and states are associated with *time invariants*. Thus, a transition is triggered when the specified event is available, the guard is true and the time guard evaluates to true as well. The automaton may only reside in states whose invariants are true. Once the invariant becomes false, it has to leave the state.

In contrast to Timed Automata, firing a transition in a RTSC consumes time. Therefore, transitions are associated with *deadline intervals* $d = [d_{min}, d_{max}]$, specifying the minimum and maximum point in time when the firing of the transition has to be finished, relative to a clock or relative to the point of activation. It is to be noted that together with worst-case execution times (wcet), which are included in the model, this more realistic assumption is a critical prerequisite to enable an automatic code generation which can equally guarantee the specified timing properties of the implemented model (cf.[10]).

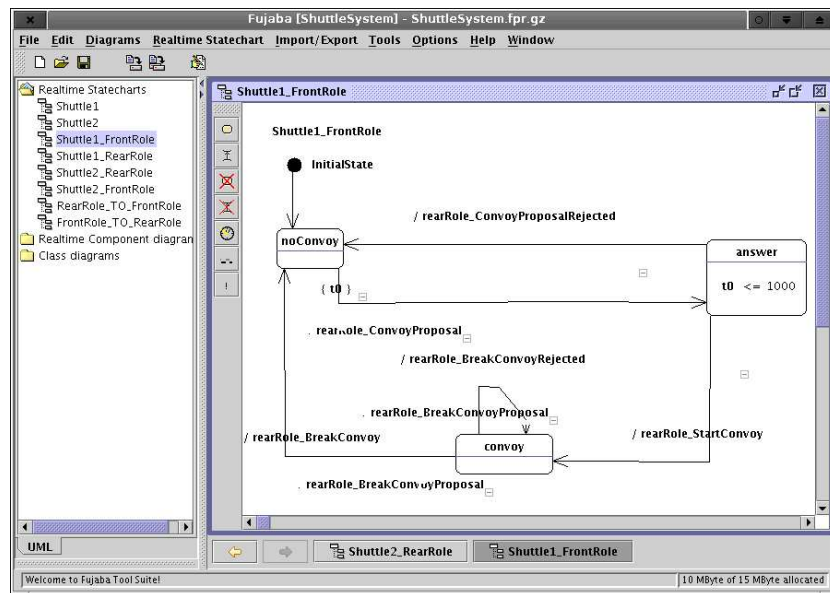


Fig. 3. The frontRole

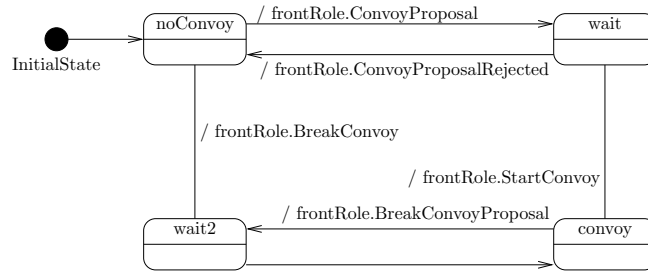


Fig. 4. The rearRole

Figures 3 and 4 show the frontRole and the rearRole behaviors:⁴ When the frontRole resides in state noConvoy, it may non-deterministically choose to send a convoy proposal to the rearRole, which answers with a ConvoyProposalRejected or StartConvoy message. When sending ConvoyProposal the clock t0 is set to zero. If the frontRole does not receive an answer it switches back to state noConvoy after 1000 msec. When sending resp. receiving StartConvoy, both communication partners change their state to convoy. Breaking the convoy is done by a similar handshake protocol.

When specifying the concrete behavior of the shuttle component, it must realize the (refined) behavior of its roles. Therefore, the RTSC from Figure 5, which shows the Shuttle behavior, consists of 3 orthogonal states: The upper orthogonal state myFrontRole refines the the front role behavior. The orthogonal state myRearRole refines the rearRole behavior. The third orthogonal state is responsible for synchronizing the different roles realized by the component.

3 Mapping UML/RT Models to HUppaal

The proposed UML/RT subset only supports components and patterns as structural modeling concepts. All behavioral aspects are modeled only with RTSC. Therefore, the addressed mapping of the UML/RT model to the HUppaal tool [11, 12] at first requires a mapping for RTSC (see Section 3.1) and a subsequent mapping of the structural modeling concepts to the parallel composition of Hierarchical Timed Automata (see Section 3.2).

3.1 Mapping of Real-Time Statecharts to HUppaal

The semantics of RTSC has been defined by a mapping to an Extended Hierarchical Timed Automata (ExHTA) model [5], which extends Hierarchical Timed Automata (HTA) as established in [11] for the HUppaal approach. The HUppaal approach provides an automatic mapping from HTA to flat timed automata by the use of the Vanilla

⁴ In our notion, $r_e!$ denotes an synchronous send signal and $r_e?$ an synchronous receive signal which is send from r resp. received from r . We use r_e for receiving an asynchronous event from r and $/r_e$ to denote an asynchronous event e which is send to r .

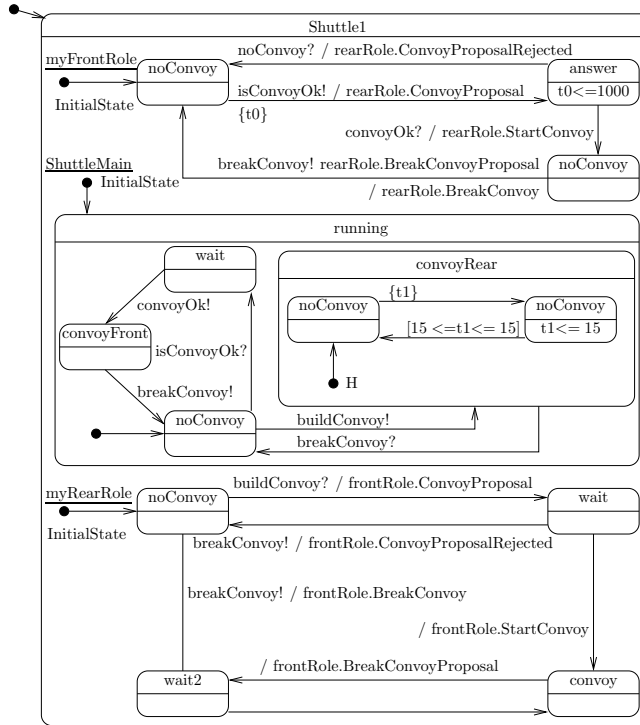


Fig. 5. The complete Shuttle component

tool⁵ [11] such that the Uppaal [13] model checker can be applied. Therefore we introduce mapping rules from ExHTA to HTA.

In Figure 6, an overview of the basic constructs available in RTSC, ExHTA, and HTA is given. In contrast to RTSC, no time consuming transitions and *do*, *enter*, or *exit actions* are present in ExHTA, as described in Appendix A.2. The mapping from RTSC to ExHTA essentially maps *do actions* and the time consuming transitions of the RTSC to a view conform with Timed Automata where time only elapses within states and not within transitions. In a second step, ExHTA are transformed to HTA (see Appendix A.1) by a mapping of the additional ExHTA concepts, such as priorities, asynchronous communication, and history, to HTA so that the HUppaal approach is applicable. Note that the constructs for parallelism, hierarchy, integer variables, and clocks can be adopted without changes, because they are provided by ExHTA as well as by the final target model HTA.

Transition To handle the time consuming transitions, the `exit()`-operation of the source state, the data assignment and the target state's `entry()`-method have to be addressed first. They are mapped to the ExHTA model by putting their sequential execution into the *do action* of an additional state. This leads to two different kinds of

⁵ <http://www.brics.dk/omoeller/hta/vanilla-1/>

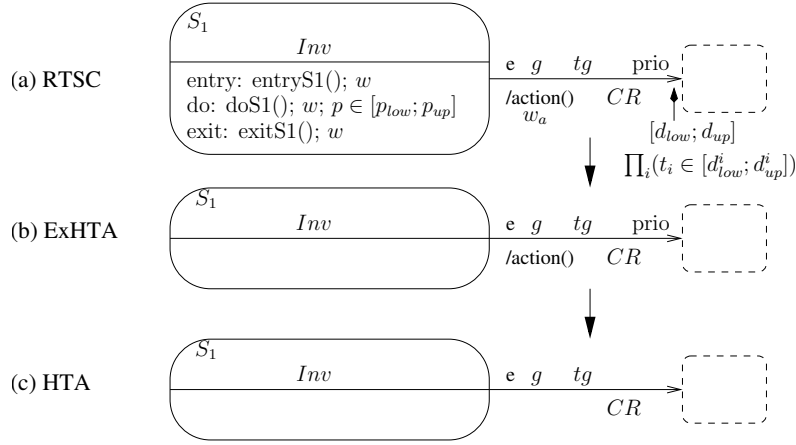


Fig. 6. Overview about the basic elements of RTSC, ExHTA and HTA

locations: *State locations*, representing states, and *action locations*, in which operations are executed.

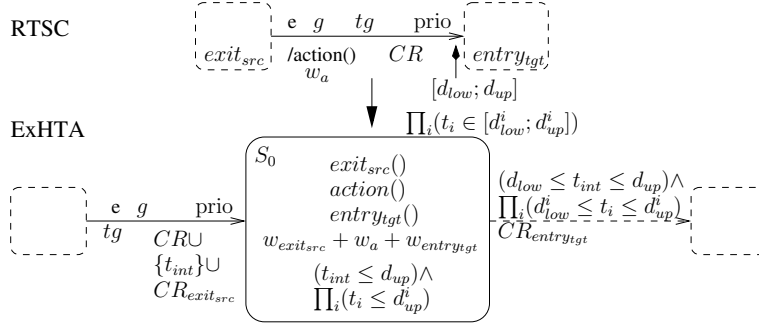


Fig. 7. Mapping of transitions

Figure 7 shows how a transition is mapped to an additional action location and two transitions of an ExHTA as defined within the Appendix in Definition 2. Solid arrows denote urgent, dashed arrows denote non-urgent transitions. If a transition is activated for a specific time interval, an urgent one fires immediately, a non-urgent one fires at any point of time of the interval. The event, guard, time guard and priority from the RTSC are recovered in the urgent transition as these attributes shall trigger the actions.

The deadline is reused in multiple places. On the one hand, it is used to extend the invariant of the action location. On the other hand, it is used to prevent premature leaving of the action location before the point of time, specified by the lower bound of the deadline. As the last point of time when the action location may be left is specified

by the upper bound of the deadline, it is used as the upper bound of the time guard of the non-urgent transition.

Do Action Figure 8 shows how a non-hierarchical state S_1 is mapped to a location in the ExHTA model as defined in Definition 2 within Appendix A.2: The name and the invariant are kept. To model the periodic execution of the $\text{do}()$ -operation, a new state S'_1 is introduced. The non-urgent transitions $S_1 \rightarrow S'_1$ and $S'_1 \rightarrow S_1$ are created. The $\text{do}()$ -operation is assigned to the first one. The latter one resets the new introduced clock t and is only triggered at $t \geq p_{low}$. To ensure that the $\text{do}()$ -operation is executed not later than p_{up} , the invariant $t \leq p_{up}$ is assigned to S'_1 . All transitions leading to S_1 are kept, while all leaving transitions are doubled so that one of them has its origin in S_1 and one in S'_1 . The assumption that the *do action* cannot interfere with other transitions is justified by the fact that the maximal time consumed by firing each transition is checked w.r.t. this effect in our tool.

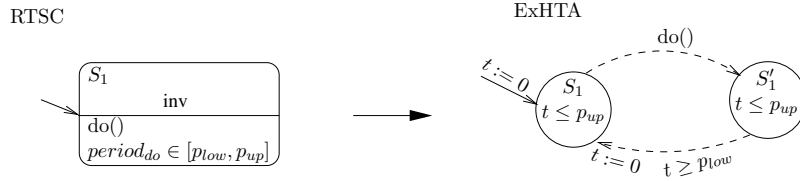


Fig. 8. The transformation of ordinary states with $\text{do}()$ -methods

Priorities As mentioned in Definition 2, a transition is also marked with a priority flag. If a state has more than one outgoing transition and more than one is enabled at once the transition with the highest priority fires. As there is no support for priorities in the HUppaal model, we have to provide a mapping for this concept. We have to distinguish between two cases: (1) a transition is only triggered by a guard and (2) a transition is triggered by a guard and an event. In the first case, we use the fact that boolean guards can be inverted. In the second case, the problem we have to deal with is that events cannot be inverted like boolean guards and that two participants running in parallel are synchronized when a transition fires.

In Figure 9, the mapping scheme is depicted. A state S of an ExHTA with transitions $S \rightarrow S_1 (t_1), \dots, S \rightarrow S_n (t_n)$ which synchronize via events $e_i \square$ with $\square \in \{?, !\}$ and which are additionally marked with a guard $guard_i$ and a priority value $prio_i$ is mapped to a HTA. An additional state S' associated with a committed flag is present in the HTA. Due to the fact that S' is marked as committed, the state must be left at once again after it was entered. In addition, for every event $e_i \square$, a variable $flag_{e_i}$ is added. The flag characterizes whether the related event is enabled on both the sender and receiver side ($flag_{e_i} == 1$), is enabled only for one of the participants ($flag_{e_i} == 0$), or is not enabled for neither of the participants ($flag_{e_i} == -1$). The flag $flag_{e_i}$ is tested and

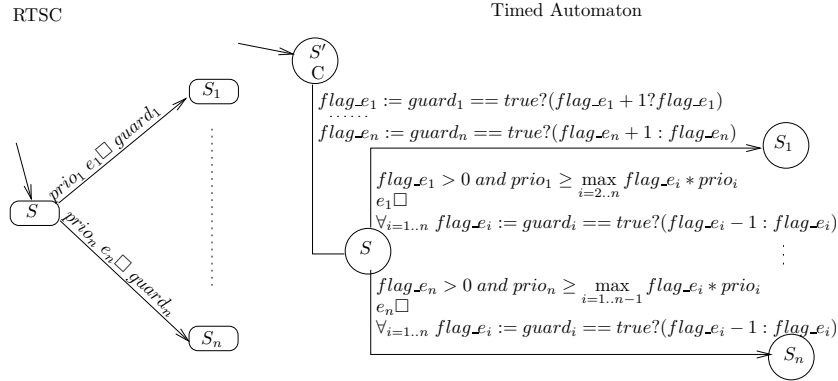


Fig. 9. Transformation of Priority

updated as follows:⁶

$$flag_e_i := guard_i == true?(flag_e_i + 1 : flag_e_i)$$

If the guard $guard_i$ evaluates to true, $flag_i$ is incremented. Otherwise, $flag_i$ remains the same.

$$flag_e_i > 0 \text{ and } prio_i \geq \max_{j=1..n, j \neq i} flag_j * prio_j$$

From all outgoing transitions of the state S the guards are formed as follows: $flag_e_i > 0$ is added to ensure that the original guard $guard_i$ is true and both participants are ready to synchronize. To ensure that only the transition with the highest priority can fire, we additionally check that there is no other enabled transition with a higher priority than the current one.

$$\forall_{i=1..n} flag_e_i := guard_i == true?(flag_e_i - 1 : flag_e_i)$$

When any transition fires, the value of all flags $flag_i$ has to be refreshed. If any $flag_i$ has been incremented before, it is thus accordingly decremented.

Asynchronous communication RTSC interact via synchronous and asynchronous events. External events from other statecharts are always asynchronous. Synchronous events are just used for internal communication. HUppaal supports the synchronization mechanism provided by the Timed Automata model. For asynchronous communication, this concept has to be extended. A new Timed Automaton realizing a queue for each event is added to the system.

Figure 10 shows the scheme for the Timed Automata which is used to queue a given set of asynchronous events e_1, \dots, e_n . The size of the queue is specified separately for each RTSC in the UML model [14].

The automaton consists of three states, S_1 , S_2 and *Error*. When the event $send_e1$ is sent by an automaton, the event is added to the queue. This happens at transition

⁶ $a?b : c$ is according to "if a then b else c "

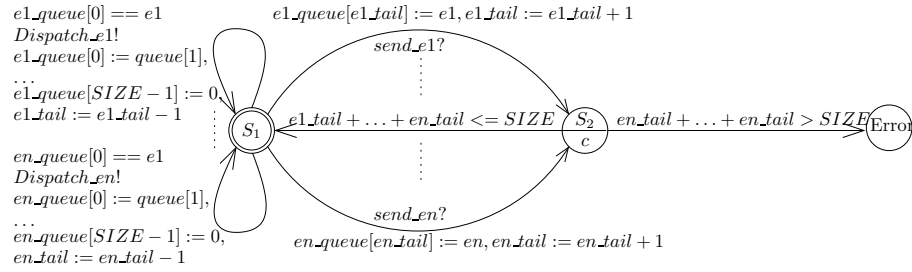


Fig. 10. Scheme for queuing asynchronous events

$S_1 \rightarrow S_2$, where the event is added to the array `e1_queue`. If the entire event queue is not full, the automaton will turn back to the initial state. Otherwise, the automaton will switch to the *Error* state. If the automaton is in state S_1 consumed events can be dispatched (self-transitions $S_1 \rightarrow S_1$).

History Within hierarchical states of RTSC, it is possible to define a history flag. In the syntax and semantics of the HTA, the history is defined, but there is no tool support when HTA are flattened to ordinary Timed Automata. We thus also introduce a mapping for history.

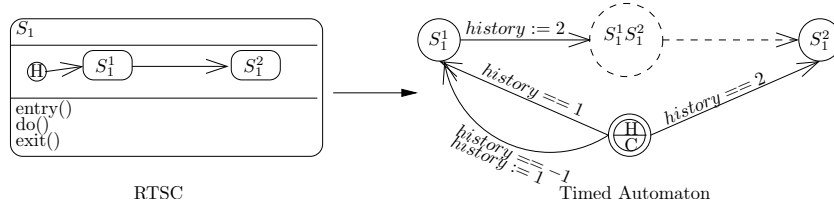


Fig. 11. Transformation of history

In Figure 11 (left side), the hierarchical state S_1 of a RTSC contains the substates S_1^1 , S_1^2 and the history flag (H). The effect of the latter one is that the state which last was active will be occupied again when S_1 is entered. To realise this behavior in the Timed Automata model, a history location H and a variable history are added for each history flag. On each transition, the variable history is set to a unique value, representing a location, e.g. `history:=1`. When the state S_1 is entered for the first time, history is set to `-1`. From the history location H, a transition to every other location is added. Each individual outgoing transition from H is marked with a guard `history==i`, where `i` is a unique id of the target location. In Figure 11(right side), the transformed Timed Automaton is depicted. For flat history, as in the example, the construction is only done for the highest level of the RTSC. In case of deep history, the construction also has to be done for all other levels.

Data Elements We also have to transform the data elements as defined in syntax and semantics of ExHTA to HTA. In RTSC, it is also possible to define method calls as guards, e.g. `convoyUseful()` in addition to ordinary guards, e.g. `convoy==true` (cf. Figure 12). Note that we assume that integer variables are only updated in specific integer updates, while method calls in guards or updates do not affect them.

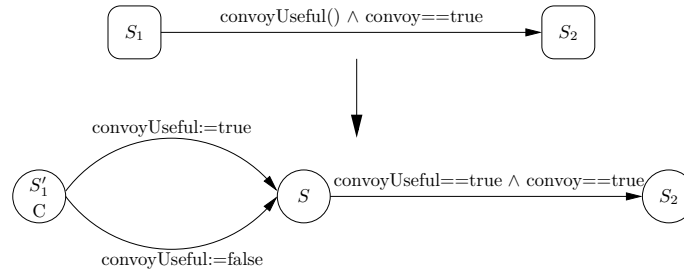


Fig. 12. Transformation of guards as a combination of variable and method call

For the mapping, we decided to ignore the data updates in side effects and simply simulate the two possibilities of the result of the method call in the guard using an additional flag. If a method call `convoyUseful()` is present in the guards of a state S , we add an additional flag `convoyUseful` and state S'_1 with a committed flag to the HTA. Two transitions connecting S'_1 with S_1 are added and marked with `convoyUseful:=true` resp. `convoyUseful:=false`.

External Events Due to the employed compositional approach, the mapping can result in a closed or in an open system depending on whether events without counterpart exist in the model. These external events, which can only show up in the port RTSC, are simply omitted and the transitions are transformed into non-urgent ones. This mapping ensures that an erratic non-deterministic behavior of the system environment is simulated while additional deadlocks are excluded.

3.2 Mapping Components and Patterns to HUPpaal

Finally, we have to construct the formal models for the patterns, components, and systems based on the transformation of RTSC to HTA as outlined in the last Section.

Patterns When checking a single pattern, the parallel composition of the HTA for the role behavior as well as the connector results in a closed HTA model which can be fed into HUPpaal.

Components The situation is slightly more complex for components. The orthogonal states in the RTSC of the component result in a HTA which includes the refined role behaviors as well as a synchronization part which contains unconnected external events

and is thus open. The mapping for unresolved events outlined in the last Section still permits to model check the resulting HTA model with HUPpaal assuming an erratic environment operating non-deterministically.

Systems When considering the whole system or an arbitrary fraction of the system, the included patterns and components have to be unified to a single HTA model. While for the components the HTA mapping which reflects the orthogonal -states in the RTSC of the component can be reused for an adjusted set of unconnected external events, only the HTA resulting from the mapping of the connectors have to be included for the patterns. Again, the resulting HTA model can be checked with the HUPpaal approach. If the complete system is considered, we usually have a closed system, while for arbitrary fractions of the system the mapping results in an erratic non-deterministic environment.

4 Incremental Design and Verification

Due to the outlined mapping of RTSC to HUPpaal and the specific treatment of unresolved external signals, the partial models which relate to each pattern or component can be employed to model check the required local properties.

4.1 Compositional Model Checking

In Section 2, we outlined the proposed design approach. When verification is also integrated, the development process consists of the following five steps which were first introduced in [1]: (1) design the patterns and their roles, (2) verify each pattern individually, (3) design the components by refining the roles associated to each port, (4) verify each component, and (5) compose the overall system by using the verified patterns and components. Note that steps 1 and 2 have to be repeated for every required pattern. When steps 3 and 4 have already been performed with incomplete sets of patterns, additional parallel statecharts that refine the additional roles have to be added incrementally. Step 5 finally ensures correct semantical composition by correct syntactical composition.

Within design step 1, an overall pattern constraint, given as an ATCTL formula, is assigned to each of the patterns. In step 2, model checking is used to verify whether the pattern is deadlock free and the constraint is met by the pattern or not. It is sufficient to check the pattern in isolation; other patterns and components need not be considered.

To check the components' correctness, step 4, the role invariants are used. A component is correct if it fulfills all invariants associated with the roles it refines and if it is additionally deadlock free.

It has been shown in [1] that an additional sixth step to verify the correctness of the overall system is not necessary. Due to the compositional nature of the approach, each system that is composed of verified patterns and components in a syntactically correct manner is also semantically correct. Following this approach, when we have to verify a system, consisting of n instances of the distance coordination pattern and $n + 1$ shuttle components, it is sufficient to only check the pattern and the component behavior once.

The verification of the pattern `DistanceCoordination` using the model checker `Uppaal` takes 1.12 sec for deadlock freedom (`A[] not deadlock`). To check the property `A[] not (rearRole.convoy and frontRole.noConvoy)` the model checker needs 0.41 sec. Both properties are fulfilled and have been checked within 1.53 sec.

For the verification of one component the following properties are checked. First, the deadlock freedom `A[] not deadlock` is checked for the component, which takes about 1.23 sec. To ensure the role invariants, we have to prove that the `frontRole` and `rearRole` are not in the state `convoy` at the same time. Thus, the condition `A[] not (rearRole.convoy and frontRole.convoy)` is checked which takes 0.31 sec. Again, both properties are evaluated to true and the check took only 1.54 sec.

It is to be noted that due to the compositional approach these checks are sufficient to prove the correctness of syntactically correct composed systems with an arbitrary number of shuttles, while checking such models directly is only feasible for a moderate number of shuttles (cf. [15]).

Another important advantage is that the verification is already performed during the design phase, i.e. it is not required to have a complete model of the system before the first verification steps are performed.

4.2 Model Checking and Consistency Management

As outlined in Section 2, the required properties have been integrated into the UML model to ease their handling and comprehension. The compositional verification approach further enables us to check each required property using only the small relevant fraction of the UML behavioral and structural model they are attached with as it does not require a complete model. This results in a large number of small verification tasks, instead of a single one, which have to be managed somehow.

Therefore, a tight integration with the consistency management mechanism of the `FUJABA Tool Suite` has been realized [16, 7], which maintains a continuously up-to-date property status. This status is either true when the property holds, false when the property does not hold, or unsafe when the model elements and the last check are possibly not consistent any more (see Figure 1).

To maintain consistency between the properties and structural and behavioral model elements, the following rules are realized by a consistency mechanism: (1) If a pattern or any of its role behaviors in form of a RTSC has been changed, the associated properties are marked as unsafe and all properties of involved components are marked as unsafe, too. The latter marks are necessary because the behavior of the components have to be a refinement of the port roles of the patterns. (2) If a component or its RTSC have been modified, the related properties have to be marked as unsafe.

The small models involved in the compositional model checking of properties usually result in rather small verification times (about 2 sec in our example). Therefore, the model checking `PlugIn` of the `FUJABA Real-Time Tool Suite` permits to automatically handle these verification tasks in a background process: If a pattern or component has been changed, the appropriate component resp. pattern is added to a `checkList` and a background process automatically handles the verification task [17]. It is also possible to deactivate this background model checking.

Consider the following example: An error is detected by the model checker in the shuttle component. As the properties are directly associated with the components, it is no problem to find the erroneous component and the related RTSC. After correcting the RTSC (e.g. by inserting a missing clock reset) a new verification process is started as soon as the user changes the view from the RTSC to another diagram. During verification, the property is marked as unsafe. When the verification process has finished, the result (true,false) is automatically shown in the property, and the developer is thus informed. The advantage is that the developer does not have to keep in mind all the changed RTSC because the background consistency management manages them in a list. If the model is large, this becomes very important.

The outlined scenario shows that, within an iterative design process, the formal verification can be smoothly integrated and fully automated. In the outlined approach, the formal correctness becomes a question of permanently maintaining model consistency rather than time consuming additional verification activities. However, if a property does not hold, the developer is still in charge of resolving the problem by analyzing and adjusting the UML model accordingly.

5 Related Work

As described in the previous sections our RTSC are mapped to HUPpaal [12, 11]. In HUPpaal Hierarchical Timed Automata are introduced which extend Timed Automata by hierarchy. The difference to Timed Automata is that a HTA consists of locations which are either simple states as in Timed Automata or composite states consisting of several substates. Composite states can be either XOR-states, which means if this state is active exactly one of its substates is active too, or AND-states, i.e. if this state is active, all of its substates are active too. To verify HTA, they have to be flattened, i.e. transformed to Timed Automata which can be done by the tool Vanilla[11]. Although HTA are more suitable for our approach than flat Timed Automata, they still lack of some required high level concepts such as priorities, asynchronous communication, or history which have been proven to be useful for modeling real-time systems. Therefore, we decided to introduce UML/RT which offers additional modeling concepts and can be mapped to HTA.

Knapp et al. present in [18] a tool called HUGO/RT. Within this tool, models are described by UML state machines. The properties to be checked are given as scenarios written as sequence diagrams extended by time annotations. For verification, HUGO/RT transforms the Statecharts into Timed Automata and the sequence diagrams into Observer Timed Automata. Afterwards, the model checker Uppaal is started, which checks whether the Observer Timed Automata describe a reachable behavior of the system. The approach of Diethers and Huhn [19] is similar to this. Their tool Voodoo translates the UML state machines and sequence diagrams of a commercial CASE tool (Poseidon) into Timed Automata and Observer Timed Automaton which are used as the input for Uppaal. After Uppaal has performed the model checking process, the error trace generated by Uppaal in case of an incorrect specification is transformed back to the UML tool. The modeling of real-time behavior is restricted within HUGO/RT and Voodoo due to the fact that models are described by UML state machines, which support only

after and *when* constructs. In addition, state machines do not contain clocks or priorities which are useful when modeling real-time systems.

Another project that aims at modeling and verifying real-time and embedded systems with UML is the OMEGA IST project.⁷ The project does not support the complete UML language. Instead, a subset of the UML which is essential for the modeling of industrial real-time applications [20] is defined. In addition in [21] a subset of the UML is extended by some timing constructs which are necessary when modeling real-time systems. The project plans to provide an environment that integrates different existing tools for modeling and verification. In [22], models specified with a UML subset are mapped to Communication Extended Timed Automata (CETA) which serve as the input language of the validation tools. Properties to be checked are given as Observer Automata in the static case and as UML Observers in the dynamic case. The integrated validation tools support simulation, verification of the properties and automatic test generation. The state space explosion is tackled by techniques based on data flow analysis, slicing methods and simple forms of abstraction. In contrast to the presented approach, compositional reasoning is only supported for the interactive theorem prover PVS and thus is only semi-automatic.

All related approaches only provide a loose integration of non-compositional model checking, while the presented approach offers a tight integration by using the consistency management subsystem of the FUJABA CASE tool for managing the required compositional verification steps.

6 Conclusion and Future Work

The presented approach and its tool support realized within the FUJABA Real-Time Tool Suite enables a smooth integration of design and formal verification during system development. The employed compositional model checking approach enables an incremental handling of changes or additions during the development, as only the involved patterns or components have to be rechecked. Therefore, the impact of a single modification or addition remains bounded to the small number of model elements which directly refer to the altered element.

The realized model checking PlugIn only supports the HUppaal tool. The realized architecture, however, permits to attach different model checker backends to our tool. A first additional backend interface which is currently under development will support the discrete time symbolic model checker RAVEN [23].

Other planned extensions are round-trip support which maps the output of the model checker back to the UML model in form of UML sequence diagrams with time annotations as well as support for the modeling of hybrid behavior [24].

Acknowledgement

The authors thank Florian Klein and Matthias Tichy for comments on earlier versions of this paper.

⁷ <http://www-omega.imag.fr/>

References

1. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the Compositional Verification of Real-Time UML Designs. In: Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland, ACM Press (2003) 38–47
2. Object Management Group: UML 2.0 Superstructure Specification. (2003) Document ptc/03-08-02.
3. Selic, B., Rumbaugh, J.: Using UML for Modeling Complex Real-Time Systems. Techreport, ObjectTime Limited (1998)
4. Burmester, S., Tichy, M., Giese, H.: Modeling Reconfigurable Mechatronic Systems with Mechatronic UML. In: Proc. of Model Driven Architecture: Foundations and Applications (MDAFA 2004), Linköping, Sweden. (2004)
5. Giese, H., Burmester, S.: Real-Time Statechart Semantics. Technical Report tr-ri-03-239, University of Paderborn, Paderborn, Germany (2003)
6. Object Management Group: UML Profile for Schedulability, Performance, and Time Specification. OMG Document ptc/02-03-02 (2002) URL: <http://cgi.omg.org/docs/ptc/02-03-02.pdf>.
7. Burmester, S., Giese, H., Niere, J., Tichy, M., Wadsack, J., Wagner, R., Wendehals, L., Zündorf, A.: Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. International Journal on Software Tools for Technology Transfer (STTT) (2004) (accepted).
8. Burmester, S., Giese, H.: The Fujaba Real-Time Statechart Plugin. In: Proc. of the Fujaba Days 2003, Kassel, Germany. (2003)
9. Alur, R., Courcoubetis, C., Dill, D.: Model-checking for Real-Time Systems. In: Proc. of Logic in Computer Science, IEEE Computer Press (1990) 414–425
10. Burmester, S., Giese, H., Schäfer, W.: Code Generation for Hard Real-time Systems from Real-time Statecharts. Technical Report tr-ri-03-244, University of Paderborn, Paderborn, Germany (2003)
11. David, A., Möller, M.O.: From HUPPAAL to UPPAAL: A Translation from Hierarchical Timed Automata to Flat Timed Automata. Unpublished BRICS RS-01-11, Department of Computer Science, University of Aarhus (2001)
12. David, A., Möller, O., Yi, W.: Formal Verification of UML Statecharts with Real-time Extensions. In Kutsche, R.D., Weber, H., eds.: Proceedings of FASE 2002. Number 2306 in Lecture Notes in Computer Science, Springer Verlag (2002) 218–232
13. Larsen, K., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. Springer International Journal of Software Tools for Technology **1** (1997)
14. Leue, S., Mayr, R., Wei, W.: A scalable incomplete test for the boundedness of UML RT models. In Kurt Jensen, A.P., ed.: Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS2004. Volume 2988 of Lecture Notes in Computer Science., Springer-Verlag (2004) 327–341
15. Giese, H., Schilling, D., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the Compositional Verification of Real-Time UML Designs. Technical Report tr-ri-03-241, University of Paderborn, Paderborn, Germany (2003)
16. Hirsch, M., Giese, H.: Towards the Incremental Model Checking of Complex RealTime UML Models. In: Proc. of the Fujaba Days 2003, Kassel, Germany. (2003)
17. Hirsch, M.: Effizientes Model Checking von UML-RT Modellen und Realtime Statecharts mit UPPAAL. Master's thesis, University of Paderborn (2004)
18. Knapp, A., Merz, S., Rauh, C.: Model Checking timed UML State Machines and Collaborations. 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002), Oldenburg, September 2002, Lecture Notes in Computer Science volume 2469 pages 395-414. Springer-Verlag (2002)

19. Diethers, K., Huhn, M.: Voodoo: verification of Object-Oriented Designs Using UPPAAL. In Jensen, K., Podelski, A., eds.: Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS2004. Volume 2988 of Lecture Notes in Computer Science., Springer Verlag (2004) 139–143
20. Damm, W., Josko, B., Votintseva, A., Pnueli, A.: A Formal Semantics for a UML Kernel Language. Technical report (2003)
21. Graf, S., Ober, I., Ober, I.: Timed Annotations with UML. In: Workshop on Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS2003), a satellite event of UML 2003, San Francisco, October 2003. (2003)
22. Ober, I., Graf, S., Ober, I.: Validation of UML models via a mapping to communicating extended timed automata. In: Model Checking Software: Proceedings of the 11th International SPIN Workshop. Volume 2989 of Lecture Notes in Computer Science., Barcelona, Spain, Springer Verlag (2004)
23. Ruf, J.: RAVEN: Real-Time Analyzing and Verification Environment. *j-jucs* **7** (2001) 89–104
24. Giese, H., Burmester, S., Schäfer, W., Oberschelp, O.: Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In: Proc. of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004), Newport Beach, USA, ACM (2004)

Appendix

A.1 Hierarchical Timed Automata

In the following we present the syntactical definition of Hierarchical Timed Automata as introduced in [12].

Definition 1 A Hierarchical Timed Automaton is a tuple $\langle S, S_0, \delta, \sigma, V, C, Inv, Ch, T \rangle$ where

- S is a finite set of locations. $root \in S$ is the root.
- $S_0 \in S$ is a set of initial locations.
- $\delta : S \rightarrow 2^S$ is a function, mapping a location l to all possible substates of l and generating a tree structure with root as the root. Applying δ on sets, delivers the intuitively expected results.
- $\sigma : S \rightarrow \{AND, XOR, BASIC, ENTRY, EXIT, HISTORY\}$ associates a type to every location
- V, C, Ch are sets of variables, clocks, and channels and are used for the specification of Guard, Reset, Sync, and Invariant as described in the following.
- $Inv : S \rightarrow Invariant$ assigns an invariant to every location.
- $T \subseteq S \times (Guard \times Sync \times Reset \times \{true, false\}) \times S$ is the set of transitions. A transition is composed of a source (l) and a target location (l'), a guard g , an assignment r (including clock resets), and an urgency flag u . The notation $l \xrightarrow{s, g, r, u} l'$ is used for a transition $t = (l, s, g, r, u, l')$. We omit s, g, r, u when they are necessarily absent (or false, in the case of u).

Guards, synchronizations, resets, and assignments use Guard, Reset and Sync, as described in the following:

- V is the finite set of integer variables. $V(l) \subseteq V$ is the set of integer variables local to a superstate l .

- Let C be the finite set of clock variables. $C(l) \subseteq C$ contains the clocks local to a superstate l . If l is a history state, $C(l)$ contains only clocks, that are declared as forgetful. The other clocks of l (local clocks) belong to $C(\text{root})$.
- Let Ch be the finite set of synchronization channels. $Ch(l) \subseteq Ch$ contains the channels that are local to a superstate l , i.e., it is not possible to synchronize along a channel $c \in Ch(l)$ between one transition inside l and one outside l .
- Ch leads to the finite set of channel synchronizations(Sync). It holds: $c \in Ch \Rightarrow c?, c! \in \text{Sync}$. If $s \in \text{Sync}$, then \bar{s} denotes the corresponding complementary(e.g. $\bar{c!} = c?$ and $\bar{c?} = c!$).
- Data constraints are boolean expressions of the form $A \square A$ with A is an arithmetic combination of elements from V and $\square \in \{<, >, =, \leq, \geq\}$. Clock constraints are boolean expressions of the form $x \square n$ or $x - y \square n$, with $x, y \in C$, $n \in \mathbb{N}$ and $\square \in \{<, >, =, \leq, \geq\}$. A clock constraint is called downward closed, if $\square \in \{<, =, \leq\}$. A guard is a finite conjunction over data and clock constraints. An invariant is a finite conjunction over downward closed clock constraints. The set of guards and invariants, joint with $\{\text{true}, \text{false}\}$ are called Guard respectively Invariant.
- Clock resets are of the form $x := 0$, with $x \in C$. Data assignments are of the form $v := A$, with $v \in V$ and A is an arithmetic combination of elements from V . Reset is the set of clock resets and data assignments.

A.2 Extended Hierarchical Timed Automata

The semantics of RTSC [5] employs an Extended Hierarchical Timed Automaton model which extends the Hierarchical Timed Automata model [12] by means of a general data model, asynchronous event handling, and priorities. These extensions result in the following definition for an Extended Hierarchical Timed Automata.

Definition 2 *The syntax of an Extended Hierarchical Timed Automaton is defined by a tuple $\langle S, S_0, \delta, \sigma, D, V, C, Inv, Ch, T \rangle$ where*

- S is a finite set of locations. $\text{root} \in S$ is the root.
- $S_0 \subseteq S$ is a set of initial locations.
- $\delta : S \rightarrow 2^S$. δ maps l to all possible substates of l . δ is required to give rise to a tree structure with root root . We readily extend δ to operate on sets of locations in the obvious way.
- $\sigma : S \rightarrow \{\text{AND}, \text{XOR}, \text{BASIC}, \text{ENTRY}, \text{EXIT}, \text{HISTORY}\}$ is a type function on locations.
- D, V, C, Ch are a data model, sets of variables, clocks, and channels. They give rise to Guard, Reset, Sync, and Invariant as defined in the following.
- $Inv : S \rightarrow \text{Invariant}$ maps every location l to an invariant.
- $T \subseteq S \times (\text{Guard} \times \text{Sync} \times \text{Reset} \times \mathbb{N}) \times S$ is the set of transitions. A transition connects two locations l and l' , has a guard g , a synchronization s , an assignment r (including clock resets), and an priority p . We use the notation $l \xrightarrow{g,s,r,p} l'$ for this and omit g, s, r, p when they are necessarily absent (or 0, in the case of p).

The data components in guards, synchronizations, resets, and assignment expressions are defined:

- D is a possibly infinite data model. A set of possible queries $[D \rightarrow \{\text{true}, \text{false}\}]$ and transformations $[D \rightarrow D]$ are assumed. To further take into account that data model updates or queries cannot be considered to be timeless and atomic their interference has to be controlled in an appropriate manner. Therefore, we assume a symmetric function conflict

that for an update up and an update or query up' determines whether their concurrent execution can result in such interference ($conflict(up, up') = true$). A conflict can be excluded between a guard and an update only, when the update cannot effect the outcome of evaluating the guard. For two updates in contrast, basic concurrency techniques such as monitors or semaphores may be employed to ensure interference free execution which is modelled by different forms of interleaving.

- V is the finite set of integer variables. $V(l) \subseteq V$ is the set of integer variables local to a superstate l .
- Let C be a finite set of clock variables. The set $C(l) \subseteq C$ denotes the clocks local to a superstate l . If l has a history entry, $C(l)$ contains only clocks, that are explicitly declared as forgetful. Other locally declared clocks of l belong to $C(root)$.
- Let Ch be a finite set of synchronization channels with subsets Ch_s and Ch_a for synchronous and asynchronous behavior. $Ch(l) \subseteq Ch$ is the set of channels that are local to a superstate l , i.e., there cannot be synchronization along a channel $c \in Ch(l)$ between one transition inside l and one outside l . We further restrict the asynchronous communication channels to be only global ones.
- Ch gives rise to a finite set of channel synchronizations, called $Sync$. For synchronous channels $c \in Ch_s, c?, c!, \tau \in Sync_s$. For $s \in Sync_s - \{\tau\}$, \bar{s} denotes the matching complementary, i.e., $\bar{c!} = c?$ and $\bar{c?} = c!$. For events $c \in Ch_a$ we further can send and receive simultaneously on all asynchronous channels described by $Sync_a : [Ch_a \rightarrow \mathbb{N}] \times [Ch_a \rightarrow \mathbb{N}]$. We use the notation $(c_1?, c_2?, \dots)$ respectively $(c_1!, c_2!, \dots)$ to describe which events are received respectively send how often. A synchronization is then, for example, $(c?, (c_1?, c_2?, \dots), (c_1!, c_2!, \dots))$ and the overall set of synchronizations is build by $Sync := Sync_s \times Sync_a$ which permits multiple asynchronous send and receives.
- The data model constraint is any boolean query $q_D \in [D \rightarrow \{true, false\}]$. A variable constraint is a boolean expressions of the form $A \square A$, where A is an arithmetic expression over V and $\square \in \{<, >, =, \leq, \geq\}$. A clock constraint is an expression of the form $x \square n$ or $x - y \square n$, where $x, y \in C$ and $n \in \mathbb{N}$ with $\square \in \{<, >, =, \leq, \geq\}$. A clock constraint is downward closed, if $\square \in \{<, =, \leq\}$. A guard is a finite conjunction over data model constraints, variable constraints and clock constraints. An invariant is a finite conjunction over downward closed clock constraints. Guard is the set of guards, Invariant is the set of invariants. Both contain additionally the constants *true* and *false* and therefore when no additional guard is required simple the constant *true* will be used.
- A clock reset is of the form $x := 0$, where $x \in C$. A data assignment is of the form $v := A$, where $v \in V$ and A an arithmetic expression over V . Any $up \in [D \rightarrow D]$ is a data model update. Reset is the set of clock resets, data assignments, and data model updates. $Reset^-$ is the set data assignments, and data model updates.

The well-formedness rules and formal semantics of the Extended Hierarchical Timed Automaton model which follows the concepts presented in [12] can be found in [5].