Refinement Checking of Self-Adaptive Embedded Component Architectures[*]

Christian Heinzemann, Stefan Henkler
Software Engineering Group,
Heinz Nixdorf Institute
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
[chris227|shenkler]@uni-paderborn.de

Martin Hirsch
Fraunhofer-Institute for Software- and
Systems-Technique ISST
Emil-Figge-Straï£¡e 91
D-44227 Dortmund, Germany
martin.hirsch@isst.fraunhofer.de

ABSTRACT

Software is increasingly used in embedded systems which have to support self* properties like self-adaptation, -management or -optimization. These systems enhance their functionality and improve their performance by building networks of embedded components which exploit local and global knowledge. Such systems include complex coordination protocols which require execution in real-time and reconfiguration of the software structure at runtime to adjust their behavior to the changing system goals leading to self-adaptation. Due to the complex nature of networked embedded systems and their usually safety-critical and hard real-time operations, e. g. lives may be at risks in case of failure, model-driven development of the software has become the means to construct reliable software. The key enabler for a consistent model-driven development approach is refinement. Refinement facilitates to preserve properties of abstract models in more concrete models. Despite the increased significance of self* properties in the last years, surprisingly there is a lack in support of refinement techniques being integrated in a model-driven development approach. We present a modeling approach, called Timed Story Charts, which defines a common formalism for real-time behavior and reconfigurations of the software structure to address the challenge of modeling the behavior self-adaptive embedded component architectures. Based on this common formalism and a well defined internal component architecture, we introduce a refinement definition and -check which preserves safety and bounded liveness properties as well as self-adaptation in form of runtime reconfigurations.

1. INTRODUCTION

Advanced embedded software systems increasingly exhibit self* properties like self-adaptation, -management or -optimization. That implies software reconfiguration at runtime which increases the complexity of the software additionally. As these systems are often used in a safety critical environment, formal verification on abstract models is required to ensure a proper functioning of the software. Consequently, refinement is of importance as this facilitates to preserve properties of abstract models in more concrete models.

Despite the increased significance of self* properties in the last years, surprisingly there is a lack in support of refinement techniques being integrated in a model-driven development approach. There are some modeling approaches for self-adaptive systems [3]. The approaches either support no refinement definition or time is not supported.

Based on MECHATRONIC UML (e. g. [6]), we present a modeling approach, called Timed Story Charts, which defines a common formalism for real-time behavior and reconfigurations of the software structure. Timed Story Charts are an extension of Story Charts [16]. Story Charts are an implementation of statecharts by an restrictive set of Story Diagrams, which are object oriented graph transformations. Based on this common formalism and a well defined internal component architecture, we introduce a refinement definition and -check which preserves safety and bounded liveness properties as well as self-adaptation in form of runtime reconfigurations of the software structure.

MECHATRONIC UML is based on a methodical decomposition of the embedded software and its constituent components. This supports compositional verification. As oftentimes the collaboration between a flexible number of participants is required, we extend in [11] our MECHATRONIC UML approach such that we can model collaborations between components which include structural adaptation in form of new or removed ports as well as multi-ports. The modeling of complex collaborations is possible by means of rules which describe how to join and leave these collaborations via ports or multi-ports: hierarchical state machines with a dynamic number of sub-machines are introduced to model the behavior of the multi-ports. For the collaborations they are employed to describe the multi-port protocols.

For the components we use the collaborations in this paper to refine the role behavior to model a proper synchronization with other parts of the component behavior for example in form of delegating the role behavior to embedded components (parts). This is enabled by the Timed Story Chart formalism and a well defined internal component architecture. The Timed Story Chart formalism takes the basic formalisms of the collaborations, namely Graph Transformation Systems and Parameterized Real-Time Statecharts, as input to define a common formalism for the behavior and the reconfigurations. The internal component architecture is defined hierarchically by multi-parts, -ports, and -delegations. Each of these elements support a flexible number of participants by the specification of functional behavior and the specification of adaptational behavior as explained above by the example of collaborations.

The proper synchronizations with other parts of the component behavior yields to complex internal reconfigurations triggered by the structural adaptation of the collaborations. This is also an existing problem of UML-components and -parts as for example a creation and a deletion of a part and its (delegated) port is not supported in UML. Similarly, UML does not address the question

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

whether an embedded component (part) is a correct refinement of the protocol behavior of the surrounding component.

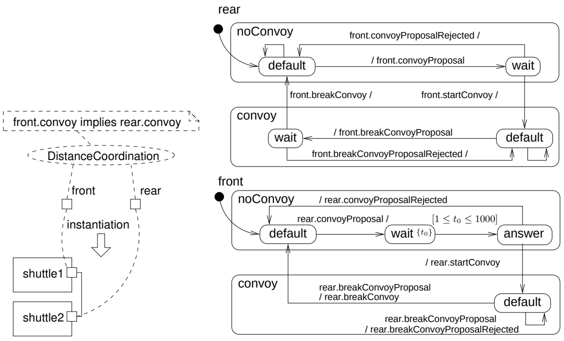
A concrete example for a complex self-adaptive system with the need to coordinate a varying number of components is the RailCab project[1]. The vision of the RailCab project is a mechatronic rail system where autonomous vehicles called shuttles apply the linear drive technology, as used by the Transrapid system, but travel on the existing passive track system of a standard railway system.

One particular problem is the convoy coordination of certain system components, e.g. the shuttles [7]. Shuttles drive in a convoy in order to reduce energy consumption caused by air resistance and to achieve a higher system throughput. Such convoys are established on-demand and require small distances between the shuttles. These required small distances cause the real-time coordination between the speed control units of the shuttles to be safety critical which results in a number of constraints, that have to be addressed when building the shuttles' control software. In addition a complex coordination is required when the convoy consists of more than two shuttles. Since shuttles can join or leave a convoy during runtime a flexible structure for the specification of the coordination is needed.

In the following section, we provide further information on MECHATRONIC UML. Thereafter, we present the Timed Story Charts in Section 3 and a refinement definition and -check for Timed Story Charts in Section 4 along with some evaluation results. Related work is discussed in Section 5. We conclude with a summary and future work in Section 6.

2. MECHATRONIC UML

In our approach, the architecture is given by components, their ports, and the connections between those ports. This model is formally described by an adaptation of the UML 2.0 component model. Among other things, our model especially covers the definition of restrictions on how ports have to be connected such that communication via different ports of the same component is guaranteed to be side-effect-free.



(a) Structure (b) Role behavior

Figure 1: Real-Time Coordination Pattern for a Shuttle Convoy

Communication between autonomous components has been defined by so-called *coordination patterns* [7]. A coordination pattern, as depicted in Figure 1(a), describes the communication between two components and consists of multiple communication partners, called *roles*. Roles are linked by a connector. The communication behavior of a role is specified by a real-time statechart.

Real-Time Statecharts are an extension of UML state machines which support more powerful concepts for the specification of real-time behavior. They are semantically based on the timed automata formalism such that a formal analysis is possible using the model checker UPPAAL[2].

The behavior of the connector is described by another real-time statechart that, in addition to the transport of the messages, models the possible delay and the reliability of the channel, which are of crucial importance for many systems.

Safety constraints which have to hold (and are model checked) for these patterns are either a so-called pattern constraint or a role invariant which concerns a property of a single role only. A role invariant specifies a property that has to be satisfied by the communication partner. A pattern constraint specifies a property that has to be satisfied by all communication partners and connectors. Both constraint types are defined in TCTL[3].

The role behavior is refined by ports that build the interfaces of our components, i.e. the ports implement the external behavior as specified by the role behavior. The refinement has to respect the role behavior (do not add possible behavior or block guaranteed behavior) and additionally has to respect the guaranteed behavior of the roles given by its invariants [7].

An additional statechart for synchronization is used to describe required dependencies between role behaviors (as the component behavior is not necessarily only a parallel composition of the different role behaviors). This allows for the strict separation of communication behavior and internal component behavior.

In our application example as shown in Figure 1(a), the coordination between two shuttles is modeled by the DistanceCoordination pattern. It consists of two roles, the front role and the rear role and one connector that models the link between the two shuttles. The pattern specifies the protocols to coordinate two successive shuttles.

Initially, both roles are in state noConvoy::default, which means that they are not in a convoy. The rear role decides whether to propose building a convoy or not. After the decision to propose a convoy, a message is sent to the other shuttle resp. its front role. The front role decides to reject or to accept the proposal after max. 1000 msec. In the first case, both statecharts revert to the noConvoy::default state. In the second case, both roles switch to the convoy::default state. Eventually, the rear shuttle decides to propose a break of the convoy and sends this proposal to the front shuttle. The front shuttle decides to reject or accept that proposal. In the first case, both shuttles remain in convoy-mode. In the second case, the front shuttle replies by an approval message, and both roles switch into their respective noConvoy::default states.

A safety requirement of the pattern is that no collision happens. The pattern constraint enforces the shuttle role to be in state Convoy while the coordinator role is also in state Convoy (shuttle.convoy implies coordinator.convoy).

3. TIMED STORY CHARTS

Timed Story Charts are a common formalism for the specification of state based behavior and reconfigurations of self-adaptive systems. They are specified as an restriction of Timed Story Diagrams which are Story Diagrams using Timed Story Pattern as described in Section 3.3 instead of Story Patterns ([16]).

Timed Story Charts as introduced in this section support the semantics of Parameterized Real-Time Statecharts ([11]) and enable reconfigurations of one to many associations using so called multi ports and multi parts.

The example shown in Figure 2 extends the shuttle application

[1] http://www-nbp.upb.de

[2] www.uppaal.com

[3] TCTL: Timed Computation Tree Logic

presented in the last Section by a ConvoyCoordination pattern. The extended example reflects the real physically running prototype build in 1:2.5 build on the campus of the University of Paderborn. To build a stable and safe convoy, a convoy leader is required which controls the convoy by calculating for example for each convoy member the position (:PosCalc). For each member a :PosCalc component and a coordinator port is required leading to a :PosCalc multi-part and coordinator multi-port. The use of hierarchical component structures arise the need for a proper delegation of the behavior to embedded parts as well as a refinement of such behavior which is not supported by our previous approach.

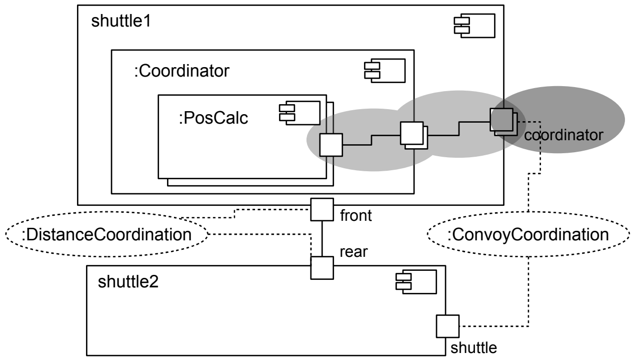


Figure 2: ConvoyCoordination pattern with multi-ports / -parts

In the remainder of this section, we first introduce the foundations of the Timed Story Chart formalism. These are Timed Graph Transformation Systems (Timed GTS) and Parameterized Real-Time Statecharts allowing to define behavior for multi-ports, -parts, and -delegations. Afterwards, we define the architecture of components and Timed Story Pattern based on Timed GTS. Finally, we introduce the syntax of Timed Story Charts and show the mapping of the semantics of Parameterized Real-Time Statecharts to Timed Story Charts.

3.1 Foundations

In the following, we define Timed Graph Transformation Systems and Parameterized Real-Time Statecharts being the basic formalisms for Timed Story Charts.

3.1.1 Timed Graph Transformation Systems

The reconfiguration of an embedded or mechatronic system architecture defined by graph transformation rules, is often time critical. Especially the creation or deletion of a port object or component might involve some complex operations. Verification of time constraints of reconfiguration operations should consequently be verifiable on the model level as well.

Timed graph transformation systems extend graph transformation systems ([13]) by the notion of time as known from timed automata ([1]). The basis for our graph transformations are UML object diagrams which are typed over a class diagram (cf. [12, 16]). These can be mapped to directed, attributed, labelled graphs and corresponding transformations. The definition of a common formalism for structure and behavior as described in this section can be achieved by object-oriented graph transformation (cf. [16]). Therefore, we define timed graph transformations based on object diagrams such that we can use them for the definition of Timed Story Charts in Section 3.4. First, we outline the idea of how time can be integrated into graph transformations and second, we define the semantics of timed graph transformation systems as they are used in our approach.

Syntax.

The extension of graph transformation systems with time requires adding clocks to the graphs as they are known from timed automata. A clock is valid for a certain subgraph of a graph. In contrast to timed automata, a subgraph may occur more than once in a graph and all occurences might be created at different times. Therefore, the clock has to be added several times to the same graph (once for each subgraph it applies to). As all these instances of the clock can have different values, we use the term *clock instance* to refer to the clocks added to the graphs. The *clock* serves as a type for the clock instances.

A timed graph is an UML object diagram extended by a set of clock instances and their possible valuations. The type graph defines a (possibly infinite) set of graphs. Furthermore, we assume a set C of clocks over which the clock instances to be added to the graph are typed.

DEFINITION 1 (TIMED GRAPH). *A timed graph G_t is a tripel $(G, CI, \mathcal{Z})$ where G is an attributed graph over a type graph, CI is a set of clock instances over the clocks in C, and $\mathcal{Z}$ is a set of clock zones ([1]) representing the possible valuations for the clock instances in CI.*

The clock instances are contained as nodes in the graph and have references to the elements of the graph to which they apply. That allows an easy binding of clock instances using the typed graph matching. It is possible that multiple clock instances apply to one object. The possible valuations for the clock instances are stored in clock zones as in timed automata (cf. [1, 2]). We proceed with the definition of the rules for our graph transformation system. The rules are applied to a timed graph which we call the *host graph*. First, we define timed transformation rules.

DEFINITION 2 (TIMED GRAPH TRANSFORMATION RULE). *A timed graph transformation rule $tr := (P_l, P_r, T, R)$ consists of a left hand side P_l and a right hand side P_r as defined for typed graph transformations. The set T contains a set of time guards over the clocks in C and $R \subseteq C$ denotes the set of clocks to be reset to 0. Let h be an isomorphic matching of P_l to the host graph. Then, the constraints and resets are applied to all clock instances in h.*

Time guards have the form $c_i - c_j \sim n$ with $c_i, c_j \in C$, $\sim \in \{<, \leq, =, \geq, >\}$ and $n \in \mathbb{N}$ (cf. [1, 2]). A reference clock c_0 having the value 0 at all times can be used to express lower and upper bounds on a single clock. Note that constraints and resets are expressed on clocks instead of clock instances. This is due to the fact that the clock instances are not known in advance and cannot be used in the constraints for that reason. This implies that the constraints can be used for all clock instances created for the respective clocks. These constraints are also used for the invariant rules defined next.

DEFINITION 3 (INVARIANT RULE). *An invariant rule $ir := (P_l, T)$ consists of a left hand side P_l and a time constraint T over the set of clock instances C. Let h be be an isomorphic matching of P_l to the host graph. The time condition T must be fulfilled for all clock instances c contained in h.*

For an invariant rule, we restrict the set of feasible clock constraints to $c_i - c_0 \sim n$, where c_i, c_0, and n are defined as before and $\sim \in \{<, \leq\}$ to be consistent to timed automata. Finally, clock instance rules adding clock instances to the graph can be defined as follows.

DEFINITION 4 (CLOCK INSTANCE RULE). *A clock instance rule $cr := (P_l, c)$ consists of a left hand side P_l and a clock $c \in C$. Let h be an isomorphic matching of P_l to the host graph. If h is empty, the set CI of clock instances remains unchanged. If h is not empty and there exists no clock instance $ci \in CI$ over clock c that references exactly the elements in h, then a new clock instance over c is added to CI which references all elementes in h. The clock instance is initialized with value 0.*

Having defined the timed graph and all necessary kinds of rules, we can now define the timed graph transformation system itself.

DEFINITION 5 (TIMED GRAPH TRANSFORMATION SYSTEM). *A timed graph transformation system $\mathcal{G}_t$ is a tuple (TG, G_0, C, TR, IR, CR), where TG is a type graph inducing the set of feasible timed graphs, G_0 is a timed start graph over TG. C denotes the set of clocks serving as types for the clock instance, TR is a set of timed transformation rules, IR is a set of invariant rules, and CR is a set of clock instance rules.*

The operational semantics of timed graph tranformation systems is similar to the one of timed automata and is defined in the following.

Semantics.

The semantics of timed story diagrams is defined by a Timed Graph Transition System (TTS). The TTS represents the complete reachable behavior and is defined as follows.

DEFINITION 6 (TIMED GRAPH TRANSITION SYSTEM (TTS)). *Let $\mathcal{G}$ be the set of all possible timed graphs, $\mathcal{R}$ a set of transformation rules, $\mathcal{I}$ a set of invariant rules. The Timed Graph Transition System (TTS) is a triple $\mathcal{T} = (S, s_0, T)$ where S represents the states of the TTS, $s_0 \in S$ is the initial state and T represents the transitions. A state $s \in S$ is a tuple $s = (g, z)$ with $g \in \mathcal{G}$ and z a non-empty clock zone over the clock instances contained in g. In s_0, all clock instances are 0.*

There exists a transition t from s_1 to s_2, $s_1 \xrightarrow{t} s_2$, iff there exists a transformation rule $r \in \mathcal{R}$ such that s_2 is a successor state of s_1.

The states are tuples consisting of a timed graph and the current clock interpretations represented by a clock zone ([1]). The clock zone contains intervals for all clock instances representing the possible values as well as the differences between those values. The definition of the TTS is analogous to the definition of zone graphs ([1, 2]), the only difference is that the states contain a timed graph instead of an automaton location. Before each successor computation, all clock instance rules have to be executed to add all necessary clock instances to the graph.

DEFINITION 7 (SUCCESSOR STATE). *Let $s_1 = (g_1, z_1), s_2 = (g_2, z_2)$ be states of a TTS. s_2 is a successor state of s_1 iff*

- *there exists a transformation rule $r \in \mathcal{R}$ such that r transforms g_1 into a graph isomorphic to g_2 and*
- *$z_2 = (((z_1 \wedge I(s_1)) \Uparrow) \wedge I(s_1) \wedge guard(r))[reset(r)]$ and z_2 non-empty.*

The definition of a successor state is analogous to the definition of successor states in timed automata ([1]). The only difference is that a location switch is represented by a transition in timed automata whereas the changes in the TTS result from applications of transformation rules. The computation of the successor clock zone remains the same. First, the clock zone is intersected against all constraints of invariant rules applicable to g_1 denoted by $I(g_1)$. Then, time passes ($\Uparrow$) which is implemented by removing the upper bounds of all clocks (cf. [2]) and then, the intersection against the invariants is repeated. After that, the resulting clock zone is intersected with the time guards of the applied transformation rule and the clock resets specified by this rule are executed. Please note that there might exist more than one possible successor state for the same transformation rule as a rule might be applicable to several subgraphs of the graph.

3.1.2 Parameterized Real-Time Statecharts

The behavior of components is specified by Parameterized Real-Time Statecharts [11]. Parameterized Real-Time Statecharts enable the specification of multi-elements (multi-ports, -parts, and -delegations).

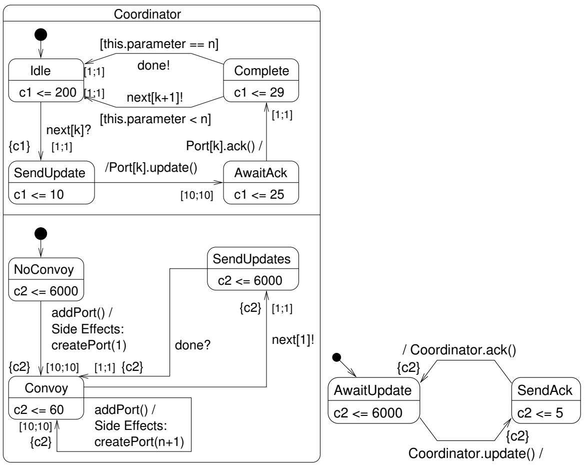
A multi-element is specified by functional (sub-element) - and adaptational behavior in form of a hierarchical statechart (see Figure 6). The adaptational behavior controls the instantiation of subelements. Both behavioral parts are specified by Parameterized Real-Time Statecharts (see Figure 3(a)). Parameterized Real-Time Statecharts are semantically based on a parameterized timed automaton, a timed automaton extended by parameterized signals. In the following, we present the relevant formalization of parameterized timed automata.

DEFINITION 8. *A* parameterized timed-automaton *A is a 7-tuple $A := (\Sigma, \mathcal{S}, \mathcal{S}^0, X, I, Sig(l), T)$ with Σ a finite set of words, $\mathcal{S}$ a finite set of locations, $\mathcal{S}^0 \subseteq \mathcal{S}$ a finite set of start-locations, $X := (x_1, .., x_n)$ a finite set of clock variables with $x_i \in \mathbb{R}^+$, I is a function $I \rightarrow \mathcal{C}(X)$, which maps a set of inequalities to the locations, the so called invariants, $Sig(l)$ is finite set of signals, parameterized with l. T is the set of transitions. $\mathcal{C}(X)$ is the set of conditions over clock-variables X. $\mathcal{C}(X)$ consists of a set of inequalities $x_i \prec c \vee c \prec x_i$ with $\prec$ $<$ or $\leq$ and $c \in \mathbb{N}^+$. T is defined as $T \subseteq \mathcal{S} \times \Sigma \times \mathcal{C}(X) \times 2^X \times Sig(l) \times \mathcal{S}$. A single transition from location s to s' is described by a 6-tuple $(s, a, \varphi, \lambda, sig, s')$. $a \in \Sigma$ denotes the labels of the transition, φ a condition, which has to be true to enable the transition, and $\lambda \subseteq X$ a number of clock variables, which a set to 0 when firing the transition. $sig \subseteq Sig(l)$ is a signal parameterized by l.*

3.1.3 Extended Convoy Example

The parameterized role coordinator is depicted in Figure 3(a). The upper statechart shows the functional behavior for each role instance and the lower one shows adaptational behavior which triggers the creation (the deletion is not shown) of new ports. If a new port is triggered by receiving a addPort message the side-effect createPort(n) is called (where n is the current number of convoy members; initially set to 1). The side-effect is specified by Timed GTS. After adding a port, the role instance behavior is triggered to send an update message for example the current maximum velocity of the convoy to each convoy member. The update message is send every 200 time units and the maximum time for updating the complete convoy is 6000 time units. This characterizes also the considered domain. Each action requires a hard timing constraint as otherwise the environment of the system could be endangered. Besides time invariants and -guards, we specify the deadline and the worst case of execution (WCET) of a transition, too. For example the deadline and WCET of the transition from NoConvoy to Convoy is [10;10] (ten for each).

The real-time statechart of the shuttle role consists of states AwaitUpdate and SendAck (see Figure 3(b)). The role awaits at least every 6000 time units an update message and sends after receiving the update message an acknowledgment.



(a) Coordinator behavior (b) Shuttle behavior

Figure 3: Real-Time Coordination Pattern for a Shuttle Convoy

An example of the creation of a new port by the Timed GTS formalism is omitted. In concrete syntax, as presented in [11], the GTS would simply add a new port role instance and link it to the last created port to enable an update from the first shuttle to the last one in the convoy in a structured way.

3.2 System architecture

As presented in Section 3.1, the system architecture is specified by UML 2.0-components and -parts. For each component diagram, a class diagram is automatically synthesized. The class diagram includes classes for each component and its ports, for each embedded part, and for all delegations and assemblies. The structure of the class diagram is based on the meta model of the component diagram (see Figure 4). An example class diagram of the Coordinator component (see Figure 2) is shown in Figure 5.

The use of multi ports and multi parts introduces the need for an adaptation because we have to control the creation and removal of the instances. This adaptation is performed by an additional adaptation layer as shown in Figure 6. As we have multiple instances of the port and the part, we also need multiple instances of the delegation. Thus, we need an adaptation for the delegation as well. The delegation behavior itself simply forwards all events or might implement some conversion function.

3.3 Timed Story Pattern

Based on Timed GTS introduced in Section 3.1, we define Timed Story Pattern as an extension of Story Pattern by time. Timed Story Pattern are, like Story Pattern, a short-hand notation for graph transformations with time. Thus, we use the syntax introduced in [16] enriched with the timing constructs of Timed GTS.

The changes in the timing constructs from Timed GTS to Timed Story Pattern are syntactically, only. Clock instances are repre-

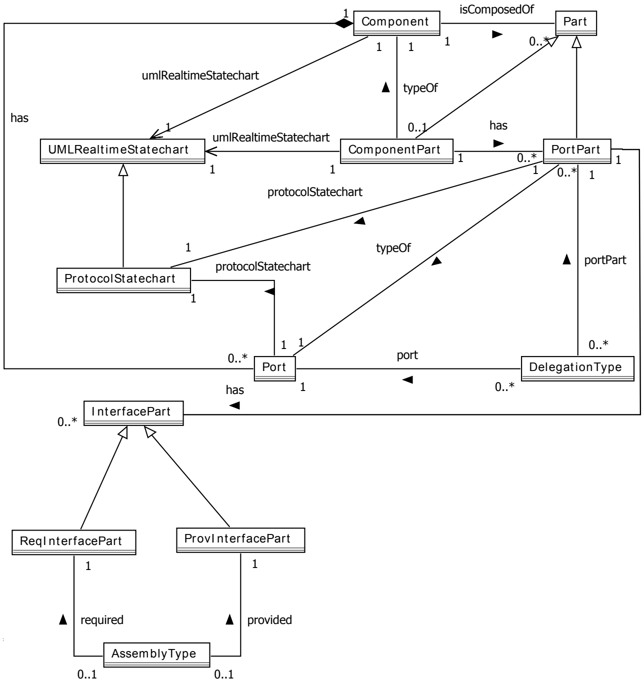


Figure 4: Component and parts meta model

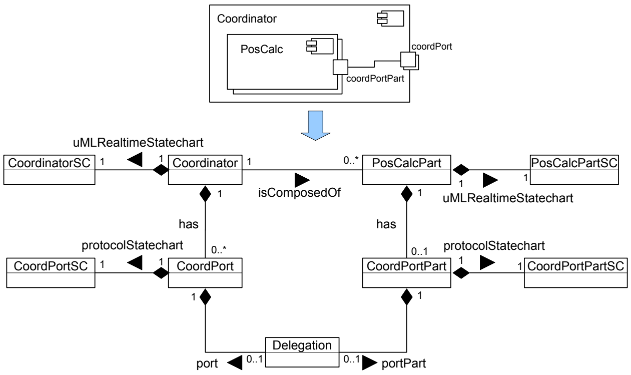


Figure 5: Example class diagram

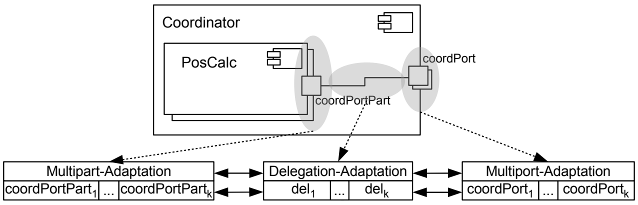


Figure 6: Multi-part, -port, and -delegation.

sented as objects and are created like objects by «++» annotations. Clock instances always hold for a certain subgraph of the host graph. The objects for which the clock instance is valid are annotated by links from the clock instance object to the respective objects of the host graph as shown in Figure 7.

The clock instance rule in Figure 7 is used to add a clock instance of type c2 to a subgraph consisting of a RailCab, its CoordinationSC, and the ActiveState object of that statechart. The first story binds the subgraph the clock instance applies to. Then, the second story checks whether the clock instance already exists. If the clock instance exists, another possible subgraph is seached. If the clock instance does not exist, it is created in the third story at the bottom.

Figure 8 shows an example for an invariant rule. The invariant specifies that the state Convoy of the CoordinationSC as depicted in Figure 3(a) must only be active for clock values up to 60. The invariant rule binds the subgraph along with the clock instance and contains the invariant condition in curly braces. As in Timed GTS, the invariant rule has a left hand side, only.

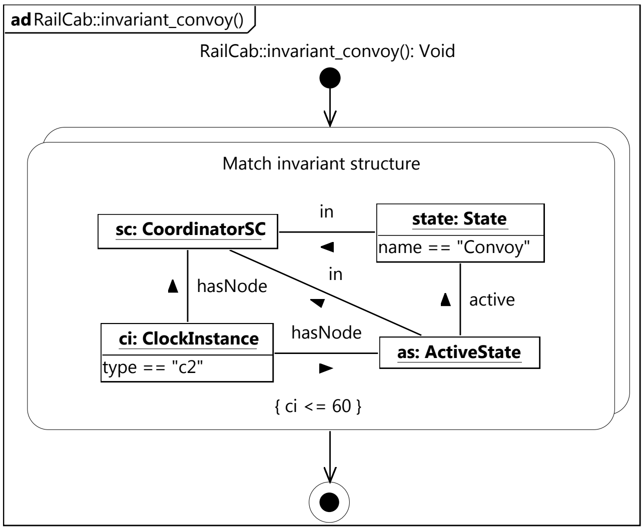


Figure 8: Invariant Rule.

3.4 Syntax of Timed Story Charts

The Timed Story Chart formalism supports abstract states, time constraints, and integrates dynamic adaptation by triggering reconfigurations specified by (Timed) Story Diagrams. In this section, the syntax of some of the key features of Timed Story Charts will be introduced. The examples used in this section are kept on an abstract level to underline the generality of the formalism.

The basis of the Timed Story Charts is the meta model shown in Figure 9. For each statechart that should be translated into a Timed Story Chart, there exists one subclass of class Statechart having the same name as the statechart it represents. Each state of the statechart is mapped to an object of type State having the name of the state as an attribute. A complex AND-state containing several substatecharts is modeled as an object of class ComplexState which has references to all embedded statecharts. Transitions between the states are implicitly mapped to rules of the Timed Story Chart as described below [4]. In contrast to [16], we did not use a dedicated framework for the execution of the Story Chart as a reachability analysis would be difficult due to the single method executing all transitions. Hence, a transition would not be (easily) identifiable.

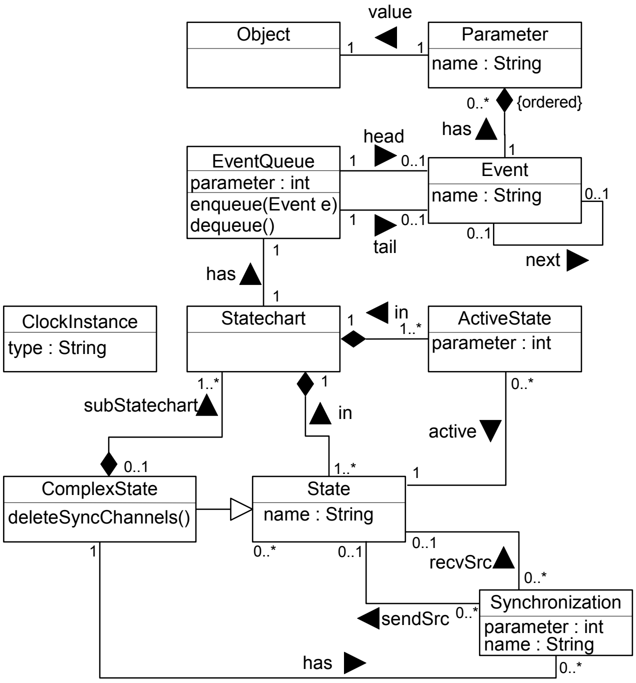


Figure 9: Meta model for the mapping of Real-Time Statecharts to Story Diagrams

Dealing with multi ports, we have defined Parameterized Real-Time Statecharts as described in Section 3.1.2. Here, all instances have the same set of states. Thus, it would be inefficient to create dedicated objects for the states of each statechart instance. Instead, we exploit this fact by introducing a parameterized ActiveState object for each instance where the parameter of the current instance is stored as an attribute in the corresponding object. It is necessary to provide unique parameter IDs for the statecharts at their creation which is also necessary for the synchronization among the instances using parameterized signals called synchronization channels. An example for this mapping is shown in Figure 10 where we have two instances of the same statechart only differentiated by their parameters in the Timed Story Chart. The ActiveState object has at any time exactly one reference to the currently active state of the statechart, i.e. the state in which the protocol currently is.

The execution of transitions is performed by graph transformations changing the link from the ActiveState object to another state object. Figure 11 shows an example for a complex transition incluing a synchronization. Here, for the embedded statecharts CoordRoleSC and AdaptationSC the states are changed from Idle to sendUpdate and from Convoy to sendUpdates respectively. The example omits the deadlines specified at the transitions in order to retrieve a single rule for the transitions.

Figure 11 also shows the synchronization of two statecharts using a synchronous synchronization channel. Such channels are modeled by objects of type Synchronization carrying the name of the channel as well as the parameter for synchronization of parameterized statecharts as attributes. We provided a joint Story Diagram including both synchronized transitions because these transitions have to fire simultaneously. The synchronization object has links to the source states of both transitions requesting the synchroniza-

[4] An explicit transition object would lead to extra computations by the analysis and provides no further information.

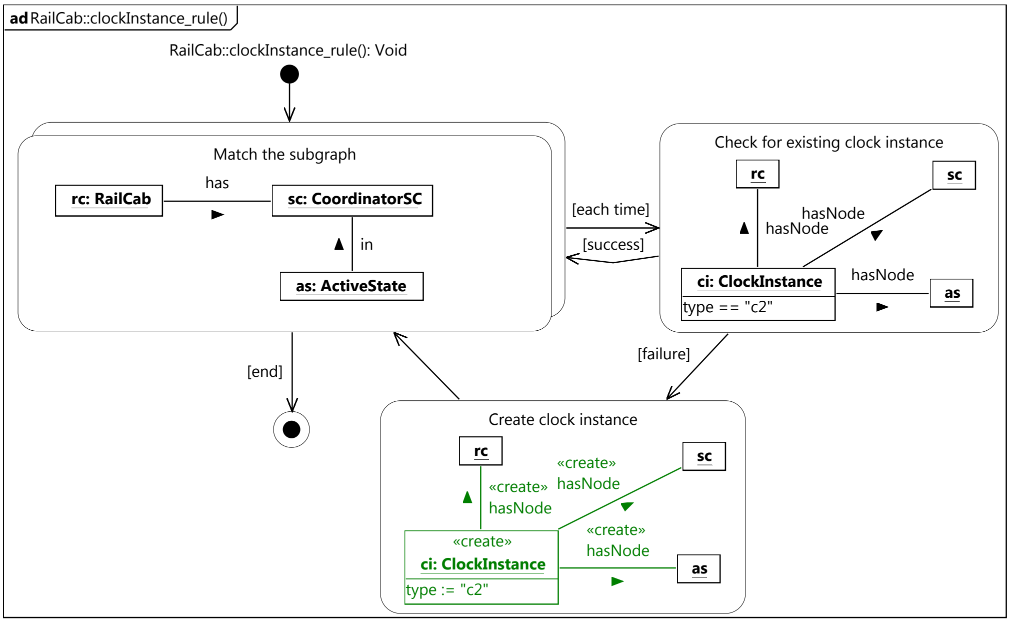


Figure 7: Adding a Clock Instance c2 to the Graph

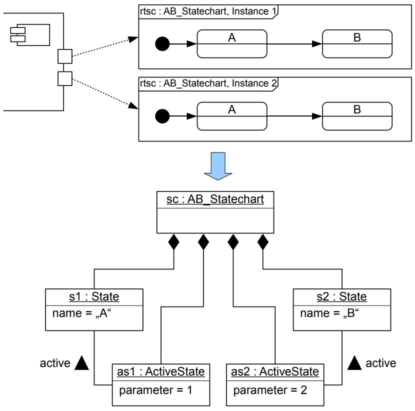


Figure 10: Example of a mapping of a statechart to a graph

tion, which are modeled by the recvSrc and sendSrc associations in the meta model.

A core feature of Timed Story Charts is the integration of reconfiguration with the aforedescribed state based concepts. Reconfigurations of the system are modeled as (Timed) Story Diagrams and can be called using so-called *Collaboration Messages* (cf. [16]). They can be used to directly call another Story Diagram. All actions such as entry or exit actions of states are mapped to Collaboration Messages as well. An example is given by the call to deleteSyncChannels which invokes a method on the cs object.

The time related aspects of Real-Time Statecharts such as clocks, time guards, clock resets, and deadlines are mapped to the corresponding elements of Timed Story Pattern. Invariants are mapped to special invariant rules having no right hand side as described in Section 3.3. Clocks instances are mapped to objects of type ClockInstance. We use the name clock instance because we have to instantiate all clocks of a parameterized statechart for each instance of the statechart. Thus, we have multiple instances of the same clock with potentially different values due to different active states in the different instances. For sake of clarification, we consider the clock to be on type level and the clock instance to be the current instance associated with an instance of a statechart. The clock instance has references to all objects of the subgraph it applies to. By referencing the ActiveState object, a clock instance refers to a specific statechart instance. An example for a clock instance is shown in Figure 12.

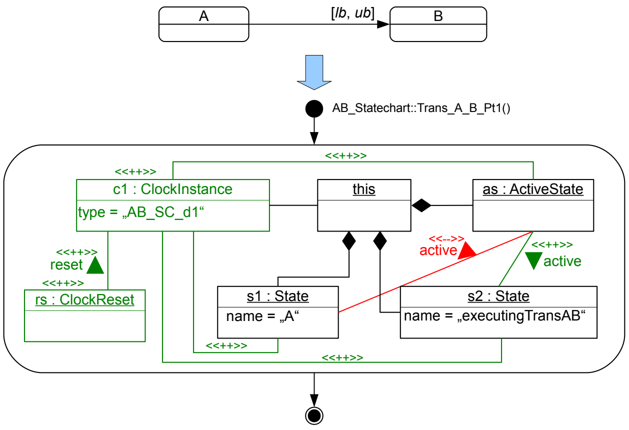


Figure 12: A transition with deadline, pt. 1

Along with a clock instance, a ClockReset object is created and linked to the clock instance. It does not need to carry the parameter itself as it is linked to exactly one clock instance already having the parameter. Each time this ClockReset object is bound in a Story Pattern along with the clock instance, the clock instance is reset to 0.

Figures 12 and 13 together represent the mapping of a deadline and thus, in our real-time statechart semantics the elapse of time. Since no time can elapse during the execution of a rule of a Timed Story Chart as in Timed Automata [1], we split the transition in two transitions with an intermediate state. The semantics of the

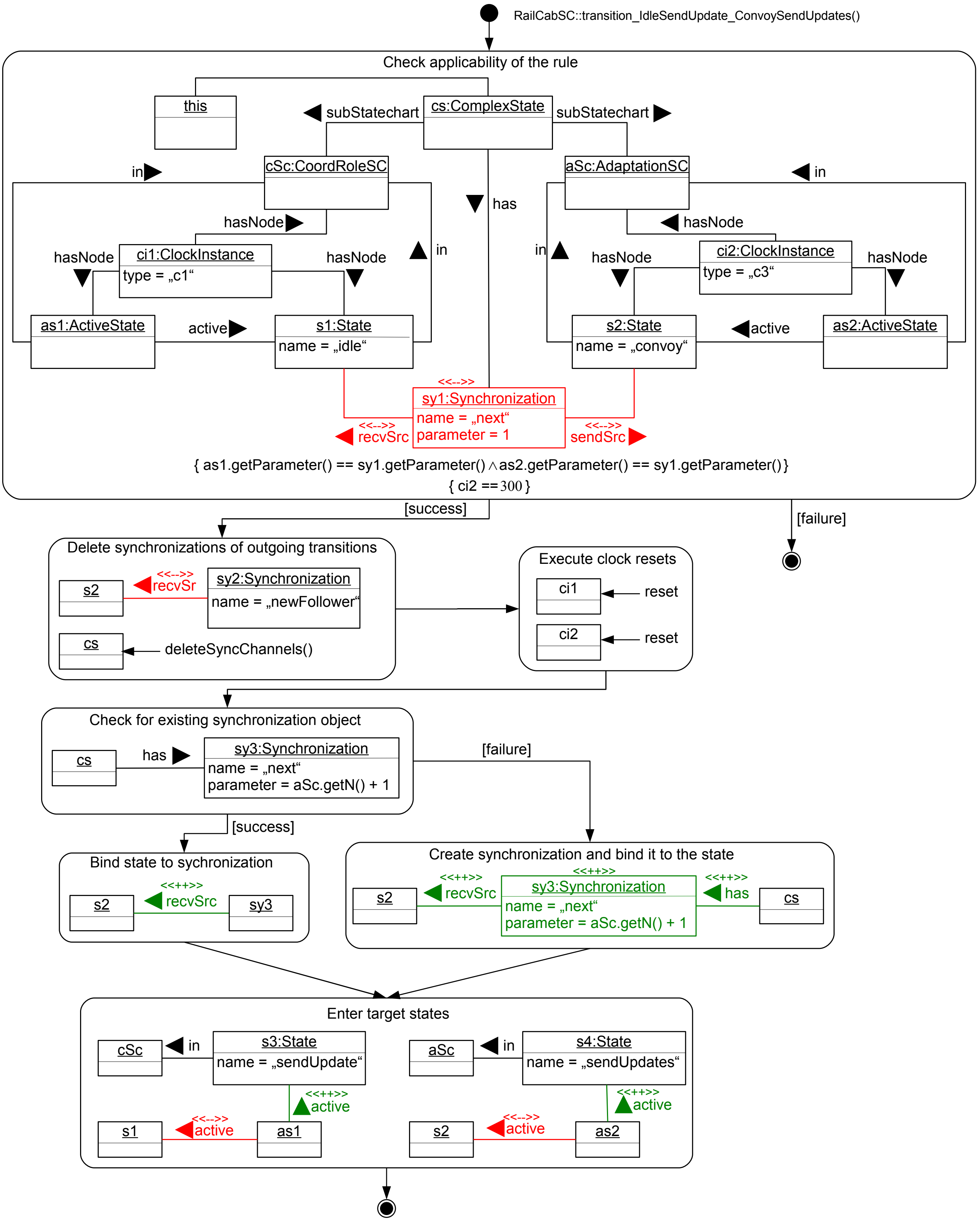


Figure 11: Complex Timed Story Chart Transformation Rule Including a Synchronization

transition will be explained in Section 3.5.

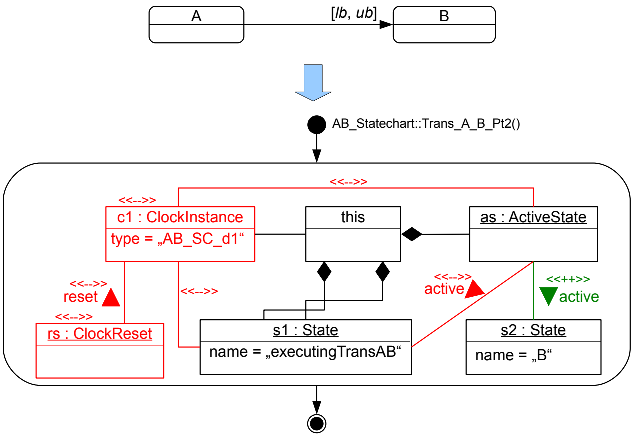


Figure 13: A transition with deadline, pt. 2

Figure 13 shows the syntax of a time guard. They are modeled as Boolean conditions for the binding of the Story Pattern (cf. [16]), i.e. a match will only be accepted if the condition can be evaluated to true. The time guard is shown in curly braces at the bottom of the pattern and contains an inequality of the form $c \prec x$ where c is the variable name of a clock instance bound in the Story Pattern, x is an integer, and $\prec$ is a comparison operator out of $\{<, \leq, =, \geq, >\}$. They can be conjuncted with normal transition guards as shown in Figure 13.

Finally, we want to introduce briefly the event concept of the Timed Story Charts. Events are considered to be asynchronous and can carry parameters. We have to distinguish between trigger events, that activate a transition, and raised events, that are created when the transition fires. Both are modeled as instances of the class Event in the meta model and have an ordered set of parameters. The event as well as the parameter has a name and the parameter additionally has a link to a value of type Object. This can be any basic type like integer or double as well as a complex type such as a string or an arbitrary object. Each statechart instance has an EventQueue serving as an inbox for incoming events. The EventQueue is organized as a simple FIFO queue in our example, but different semantics can be introduced by changing the function that inserts new events. A trigger event is specified by binding the event object as head of the event queue as it is shown in Figure 14 for event a. A raised event is implemented by an event object with modifier «++» . Raised events have to be added to the event queue of the receiving statechart (instance) which is not shown in the figure.

3.5 Semantics of Timed Story Charts

The syntactical mappings of (Real-Time) Statechart constructs defined above are now combined to a sequence of stories defining the execution semantics of Timed Story Charts. The execution semantic meets the semantic of the Real-Time Statecharts [7]. Figure 15 shows a transition with all relevant features including a deadline. The transition specifies a guard requiring the shuttles' speed to be less than 10 and a time guard allowing the transition to be activated only for clock values between 10 and 20 for clock c1. Additionally, the transition has a trigger event a, a raised event b, and synchronizes with some other transition using the channel sync. Finally, the transition executes a side effect and therefore has a deadline $[2; 5]$ meaning that the firing will terminate at least 2 unit of time after it started but at most 5 after it. Before entering the target state B, a clock reset on c1 is performed. A schema of the resulting Timed Story Chart rules is shown in Figure 16. The boxes represent the different stories of the Timed Story Diagram created for the transition.

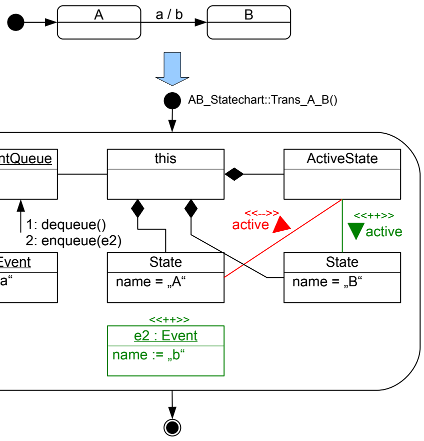


Figure 14: Events

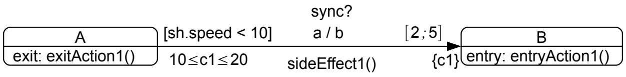


Figure 15: Transition of a real-time statechart

For the transition to fire, the precondition must be fulfilled. The precondition requires, of course, that the statechart is in the source state A which is represented by the link active pointing to the corresponding object in the Timed Story Chart. Additionally, all guards and time guards must be evaluated to true making them part of the first story. Finally, the trigger event must be first in the event queue and all synchronizations must be available, i.e. the synchronization partner is also ready to execute the transition (cf. Section 3.4). If this precondition holds, the event is removed from the queue as stated in the second story. Afterwards, all synchronization channels belonging to other transitions than the one currently taken, have to be removed as they are no longer available. This is performed by the third story, possibly in multiple steps as indicated by the double border line of the story. Then, the source state is left and an intermediate state is entered which only allows to proceed into the target state. When leaving the source state, the exit action of the source state has to be executed. This behavior is captured by the forth story. The example transition has a deadline which states that it takes at least 2 units of time for it to fire but at most 5. This time has to be measured by a clock (instance) which is created and initialized to 0 in the fifth story. As the main time consumption of the transition is caused by the execution of the side effect, the side effect is executed after creating the deadline clock. As the event is processed by the side effect and therefore after its execution no longer needed, it is deleted afterwards. This behavior is elaborated by the stories 6 and 7. At this point, time has to pass until the expiration of the deadline. Therefore, the first Story Diagram terminates and the execution of the transition continues in the second Story Diagram after the deadline expired.

To force the execution of the second rule, an additional invariant rule is used forbidding to stay in the intermediate state longer than the 5 units of time forming the upper bound in our example. The precondition of the second Story Diagram requires the system to be in the intermediate state and contains a time guard requiring the deadline clock to have a value greater or equal to the lower bound of the deadline. As the deadline clock is no longer needed for this transition, it can be deleted to improve the efficiency of the computations. Before entering the target state, the raised events have to be created and the clock resets have to be performed as done in the ninth story. Entering the target state enables all synchronization channels of its outgoing transitions. That causes them to be created in story 10. Since there may be more than one outgoing transition, the step might require several substeps. At last, the target state is entered and the entry action of the target state has to be executed afterwards. As all Collaboration Messages are executed after the graph transformation, the entry action is indeed executed after entering the target state.

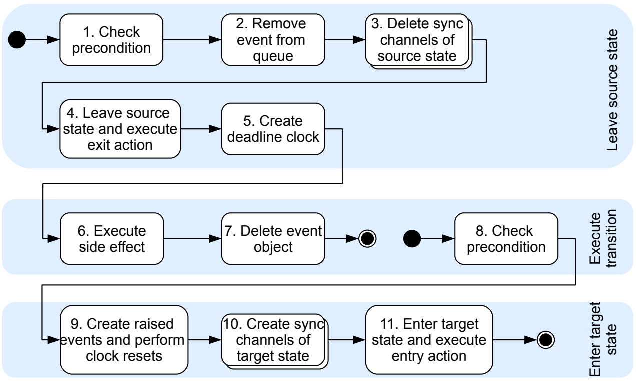

1. Check precondition
2. Remove event from queue
3. Delete sync channels of source state
4. Leave source state and execute exit action
5. Create deadline clock
Leave source state
6. Execute side effect
7. Delete event object
8. Check precondition
Execute transition
9. Create raised events and perform clock resets
10. Create sync channels of target state
11. Enter target state and execute entry action
Enter target state

Figure 16: Execution semantics

4. REFINEMENT

In this section, we will first define the refinement along with the requirements it must meet. Then, we introduce the idea of an refinement check based on the definition. Finally, we show the decidability of the check and present some evaluation results.

4.1 Definition

A correctly refined protocol should be usable as an implementation of an abstract protocol such that it does not affect the communication partner, i.e. a communication partner will not encounter any difference to the abstract protocol. This arises two requirements: 1) the external visible real-time behavior has to be fulfilled by the refined behavior and 2) the (formal) compositional verification results of the abstract behavior have to be preserved by the refinement. The fulfillment of these requirements ensures that a communication partner will not experience any difference between the abstract protocol and the refined protocol.

The refinement definition we propose is based on timed traces as defined in [15]. Timed traces capture the externally visible behavior of the protocol executed by the port. Additionally to the behavior, we also have to monitor the reconfigurations of the system as they are part of the protocol. Throughout this paper, we will use the distinction between action transitions and delay transitions made in [2], because it enables us to precisely obtain the points in time when a state is entered and how long it is possible to stay in the particular state. The definition presented in [4] contains these information only implicitely. Definition 9 captures the information necessary for the refinement of the traces.

DEFINITION 9 (TIMED TRACE). *Let $\mathcal{G}_t = \langle TG, G^0, TR, IR\rangle$ be a timed graph transformation system with externally visible events $A = A_i \cup A_o$ of a timed story chart with A_i received events and A_o sent events. Let $\mathcal{Z}$ be a set of clock zones on clock instances C. A timed trace $\xi = (S_\xi, R_\xi)$ is a sequence of rule applications of TR with states S_ξ and transitions R_ξ for which hold*

$$\xi = \langle g_0, z_0\rangle \Rightarrow_{\delta_0} \langle g_0, z_0^\uparrow\rangle \Rightarrow_{a_0} \langle g_1, z_1\rangle \Rightarrow_{\delta_1} \langle g_1, z_1^\uparrow\rangle ...$$

with

- $z^\uparrow = \{z + d | z \in \mathcal{Z}, d \in \mathbb{R}_+\}$
- $\langle g_0, z_0\rangle \Rightarrow_{\delta_0} \langle g_0, z_0^\uparrow\rangle$ *corresponds to* $\langle g_0, z_0\rangle \Rightarrow \langle g_0, z_0^\uparrow \wedge I(g_0)\rangle$ *with* $I(g_0)$ *invaraint applicable to* g_0
- $\langle g_0, z_0\rangle \Rightarrow_{a_0} \langle g_1, z_1\rangle$ *corresponds to* $\langle g_0, z_0\rangle \Rightarrow \langle g_1, ((z_0 \wedge g)[\lambda := 0]) \wedge I(g_1)\rangle$ *with* $I(g_1)$ *invariant applicable to* g_1*,* g *time guard of the transition and* λ *a set of clock instances,* $\lambda \subseteq C$*, being reset to 0.*

Each state of the trace consists of a timed graph as defined in Section 3.1.1 having a set of associated clocks and a clock zone [1] describing the possible clock values. For each pair of clock instances there is one inequality of the form $x_1 - x_2 \prec d$ where x_1 and x_2 are clock instances, d is an integer, and $\prec$ is one of the operators $\{<, \leq\}$. Additionally, the clock zone contains a reference clock x_0 which is always set to 0 to specify comparisons which integers.

Transitions in a trace can occur due to an event, which is either sent or received, or due to a delay representing the elapse of time. Event transitions are executed by a rule of the Timed Story Chart as introduced in Section 3 and may be the concatenation of multiple internal operations also executed by rules of a Timed Story Chart with one external visible event being sent or received. During such an event transition, no time can elapse, but the clock zone of the source state has to be intersected with the guard and, after applying the resets, with the invariant of the target graph. A delay transition models the elapse of time by removing the upper bounds from all clocks and intersecting the resulting clock zone with the invariants of the current graph. The resulting clock zone defines how long the situation described by the given graph is valid.

The aforedescribed traces can then be refined as described by Definition 10.

DEFINITION 10 (REFINED TRACE). *Let $\xi_a = \langle S_{\xi,a}, R_{\xi,a}\rangle$, $\xi_k = \langle S_{\xi,k}, R_{\xi,k}\rangle$ be timed traces of timed graph transformation systems $\mathcal{G}_t^a = \langle TG_a, G_a^0, TR_a, IR_a\rangle$, $\mathcal{G}_t^k = \langle TG_k, G_k^0, TR_k, IR_k\rangle$. Let abs : $\mathcal{G}_t^k \rightarrow \mathcal{G}_t^a$ be an abstraction function associating objects of $\mathcal{G}_t^k$ with objects of $\mathcal{G}_t^a$. Furthermore, let $D_{reset}(s, c)$ be a relation returning for a given clock zone s and a clock c all clock zones containing a reset on c since the last event before zone s. Let $c_{diff}(c, s_1, s_2)$ be a function computing the difference of the upper bound of clock c in the clock zones s_1, s_2, i.e. $ubound(s_1.z.c) - ubound(s_2.z.c)$, with $s_1 \in S_{\xi,a}$, $s_2 \in S_{\xi,k}$. ξ_k is a refined trace of ξ_a, $\xi_k \leq \xi_a$, if*

1. *$G_a^0 \subseteq abs(G_k^0)$ and all clocks are 0 in $s_{a,0}.z$, $s_{k,0}.z$*
2. *For each transition $t_i \in R_{\xi,a}$ with $s_a \Rightarrow_{a_o} s'_a$ having event $a_o \in A_o$ exists a transition $t_j \in R_{\xi,k}$ with $s_k \Rightarrow_{a_o} s'_k$, for which holds*
   - $s'_a.g \subseteq abs(s'_k.g)$
   - *For all clocks c in $s'_a.z$:* $\sum_{\{z|z\in D(s'_a,c)\}} ubound(z.c) + ubound(s'_a.z.c) = \sum_{\{z|z\in D(s'_k,c)\}} ubound(z.c) + ubound(s'_k.z.c) + c_{diff}(c, s_a, s_k)$

3. *For each transition $t_i \in R_{\xi,a}$ with $s_a \Rightarrow_{a_i} s'_a$ mit $a_i \in A_i$ exists a transition $t_j \in R_{\xi,k}$ with $s_k \Rightarrow_{a_i} s'_k$, for which holds*
   - $s'_a.g \subseteq abs(s'_k.g)$
   - *For all clocks c in $s'_a.z$:* $\sum_{\{z|z \in D(s'_a,c)\}} ubound(z) + ubound(s'_a.z) \leq \sum_{\{z|z \in D(s'_k,c)\}} ubound(z) + ubound(s'_k.z) + c_{diff}(c, s_a, s_k)$
4. *All external events are defined over the same namespace in ξ_a and ξ_k.*

We want to introduce the term *corresponding states* for pairs of states in both traces having the same prior event sequences as well as a compatible amount of elapsed time. By definition, the initial states of the traces are corresponding because there have been no events before them and because all clocks are 0 (Condition 1). The Conditions 2 and 3 define the timing conditions that must be fulfilled for sent events a_o and received events a_i respectively. For a sent event, the deadline for sending the event must be met exactly indicated by the equal operator. The requirement is necessary as we consider hard real-time protocols which don't allow the deadline to be extended. As we can't make assumptions on the communication partner, reducing the deadline is also impossible as we can't be sure that the communication partner can receive the event earlier.

Having a received event, we consider the currently checked side of the protocol. Under the assumption that we have a buffer for incoming messages which stores messages until they are taken out of it, we can relax the deadlines for such events. Again, we can't make the deadline smaller as the communication partner can sent the event at the end of the original deadline. But it is possible to extend it as the buffer is able to take the message at any point in time. The exact point in time at which the message is removed from the buffer and actually processed by the protocol automaton doesn't affect the communication partner. So, relaxing the upper bound is possible as long as it doesn't violate internal properties of the component executing the protocol. This can be ensured by performing a model checking of the basic components that don't delegate their port behaviors to embedded parts.

The term $s'_a.g \subseteq abs(s'_k.g)$ defines a structural refinement ([9]), i.e. both graphs, the abstract and the refined one, must have the same kinds of objects. Since we have different level of abstraction, the actual classes over which the objects are typed are likely to differ as shown in Figure 5. Therefore, we use the abstraction function according to [9] to map the object types. The structural refinement is in our case relaxed to the condition that the abstract and the refined system must have the same set of ports which are connected in the same way by delegations.

The forth condition for refined traces is that the events are defined over the same namespace. However, if the namespaces differ, both namespaces must be mappable onto each other by a bijective conversion function.

Using Definition 10 we are able to refine the refinement relation between two protocols fulfilling the two requirements.

DEFINITION 11 (REFINEMENT). *Let $\mathcal{G}_t^a, \mathcal{G}_t^k$ be timed graph transformation systems with external behaviour $Traces(\mathcal{G}_t^a)$ and $Traces(\mathcal{G}_t^k)$ respectively. $\mathcal{G}_t^k$ is a refinement of $\mathcal{G}_t^a$, $\mathcal{G}_t^k \leq \mathcal{G}_t^a$, if*

1. *for each trace $\xi_k \in Trace(\mathcal{G}_t^k)$ exists a trace $\xi_a \in Trace(\mathcal{G}_t^a)$ with $\xi_k \leq \xi_a$ and*
2. *$\neg\exists \xi_k \in Trace(\mathcal{G}_t^k) : succ(s) = \emptyset$ for a state $s \in \xi_k$ and*
3. *each trace in $Trace(\mathcal{G}_t^a)$ has been covered.*

The Conditions 1 and 2 define a weak timed simulation relationship ([14]) with the abstract protocol simulating the refined one. Condition 1 states that for each refined path there must be a path in the abstract protocol fulfilling Definition 10. Condition 2 states that the refinement is free of deadlocks. The simulation relationship ensures the preservation of ATCTL formulas which is the relevant class of verified properties in our compositional verification approach. Condition 3 of the refinement ensures the complete fulfillment of the protocol. There must be a path in the refined protocol for each path of the abstract protocol. Considering only the simulation relationship, it would be possible that the abstract protocol has more paths than the refined one. A path being covered means that there is a state marked as corresponding between every two events. If this is the case, all paths have been considered by the simulation and thus the abstraction doesn't contain more path than the refinement.

4.2 Refinement Check

The refinement check consists of two steps. First, a timed reachability analysis is performed for both systems. Then, the refinement is checked based on the reached transition systems.

The timed reachability analysis is based on the computation of reachability graphs as introduced in [17]. Basically, the Timed Story Patterns used to execute the Timed Story Charts are transformed such that the matching and the rewrite step are separated into two operations. Then, the matching operation is embedded into a for each construct, searching for all possible matches of a given Timed Story Pattern in the current graph. For each match, we use a library operation introduced in [17] in order to create a copy of the current graph and then, the rewrite operation is applied to that graph copy. We do this for all Story Patterns that are enabled for the current graph. Thus, for a given start graph the expansion step described above computes the set of all possible successor graphs reachable with the available Story Patterns. We apply this expansion step to all reachable graphs as long as possible. During the expansion, we use an isomorphism check provided by a library [17], to compare each new graph with all other derived graphs. Thereby, we identify and merge graphs that may be reached by different sequences of Story Pattern applications. During the application of Timed Story Patterns, the timing constraints are maintained using clock zones [4]. Accordingly, handling of the clock zones is incorporated in the graph copy operation and especially in the graph isomorphism check. Thus, two timed graphs are considered isomorphic, if the graph structure is isomorphic and if the clock zones are equivalent.

The refinement check algorithm then traverses the reached transition system for the refined system using a depth first approach. The algorithm investigates all possible traces and identifies corresponding traces in the abstract system. The algorithm then compares both traces according to Definition 10. Corresponding states are added between every two events for states that fulfill the conditions stated in the definition. It is possible that there exists more than one possible corresponding trace in the abstract system for a given trace of the refined system. In this case, all possible abstract traces are investigated. If no trace can be found such that the conditions of Definition 10 hold, the check fails and thus the refinement is not correct. After all traces of the refined system have been checked, the abstract system is traversed in order to check the coverage criterion. If this check is successful, the refinement is correct.

4.3 Decidability

In general, the verification of correct refinements based on trace inclusion is undecidable for timed systems [1] due to the possibility that the reachability computation may not terminate and paths may be infinite. We restrict our input to hard real-time models, as required by the considered application domain, which lead to a finite number of reachable graphs.

The specification of a hard real-time system requires each transition of the state behavior to carry a deadline which ensures the elapse of time along transitions of the input Real-Time Statecharts. A further property of a hard real-time systems is that each state has to carry an invariant to force transitions to happen. Thus, time constantly increases. These properties lead to a finite reachable graph as dependent behavior (e. g. , a dependency between sub-elements as shown in Section 3.1.3) has an upper hard timing constraint or the behaviors are independent and therefore only one possible instance of the behavior has to be considered. These restrictions lead to a finite set of traces in which every infinite trace will end in a loop. Thus, the total number of reached trace states is finite. As finite problems are always decidable, the verification of correct refinements is decidable for our class of problems.

4.4 Evaluation

We made an evaluation of the reachability analysis and the refinement check based on the convoy coordination example with different numbers of ports. As we expected, the size of the graphs grew slowly with the number of ports. The maximum size of a reachable graph rose by 6 for each participant including 2 clock instances, 2 reset objects, 1 ActiveState object, and 1 object for the port itself. The resulting runtimes of the reachability analysis with and without consideration of time are shown in Figure 17.

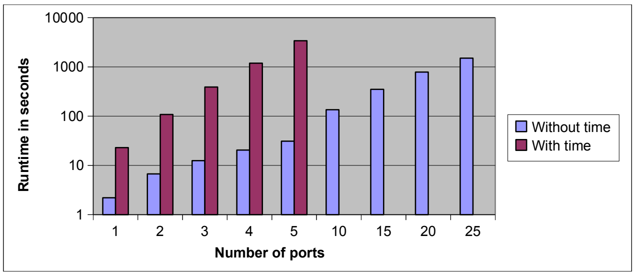


Figure 17: Runtime of the reachability analysis

For the evaluation, the abstract protocol was checked against itself with positive verification result. As the scalar on the Y-axis is logarithmic, the runtime of the reachability analysis with time is exponential.

5. RELATED WORK

In [3] an overview of modeling approaches for self-adaptive systems is presented. The approaches either support no refinement definition or time is not supported. In [5] a refinement is defined for hybrid graph transformation systems which preserves verification results of the abstract behavior. The focus is not, as in our case, to define a more relaxed refinement which enables a more flexible integration of possible refined behavior and it is not required that the external visible real-time behavior is still preserved by the refined behavior. [9] considers graph transformation systems for the specification of service oriented architectures. The presented refinement should preserve the external visible services. The approach did not take into account time and the ability to preserve verification results. [8] examine refinement for graph transformation systems based on an algebra but they did not take into account time.

6. CONCLUSION AND FUTURE WORK

In this paper, we introduced Timed Story Charts which provide a common formalism for modeling real-time behavior and self-adaptation in form of runtime reconfigurations of the software structure based on our previous work [11, 10]. Timed Story Charts combine state-based real-time behavior with the ability to reconfigure the system architecture by graph transformations based on Story Diagrams. Based on this formalism and a well defined internal component architecture, we introduced a refinement definition and a refinement check that preserves safety and bounded liveness properties as well as the externally visible real-time behavior. Our refinement check is based on a reachability analysis on Timed Graph Transformation Systems. In our future work, we want to provide more automatisms as the transformation of statecharts to Timed Story Charts is a manual task by now. Furthermore, we plan to introduce a refinement check based on static rule analysis (e. g. inductive invariants) as this technique might allow us to extend our refinement definition to applications with infinite state systems and improve the runtime of the refinement check.

7. REFERENCES

[1] R. Alur. Timed automata. *Theoretical Computer Science*, 126:183–235, 1999.

[2] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.

[3] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 28–33, New York, NY, USA, 2004. ACM.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[5] H. Giese. Modeling and verification of cooperative self-adaptive mechatronic systems. In *Reliable Systems on Unreliable Networked Platforms*, volume 4322 of *Lecture Notes in Computer Science*, pages 258–280. Springer Berlin / Heidelberg, 2007.

[6] H. Giese, S. Henkler, M. Hirsch, V. Roubin, and M. Tichy. Modeling techniques for software-intensive systems. In D. P. F. Tiako, editor, *Designing Software-Intensive Systems: Methods and Principles*, pages 21–58. Langston University, OK, 2008.

[7] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the compositional verification of real-time uml designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, pages 38–47. ACM Press, September 2003.

[8] M. Große-Rhode, F. P. Presicce, and M. Simeoni. Formal software specification with refinements and modules of typed graph transformation systems. *J. Comput. Syst. Sci.*, 64(2):171–218, 2002.

[9] R. Heckel and S. Thöne. Behavioral refinement of graph transformation-based models. In *Proc. of the ICGT 2004 Workshop on Software Evolution through Transformations (SETra 04)*, pages 139–151. Electronic Notes in Theoretical Computer Science, 2004.

[10] C. Heinzemann, S. Henkler, and A. Zündorf. Specification and refinement checking of dynamic systems. In P. V. Gorp, editor, *Proceedings of the 7th International Fujaba Days*, pages 6–10, Eindhoven University of Technology, The Netherlands, November 2009.

[11] M. Hirsch, S. Henkler, and H. Giese. Modeling Collaborations with Dynamic Structural Adaptation in Mechatronic UML. In *Proc. of the ICSE 2008 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'08),Leipzig, Germany*, pages 33–40. ACM Press, May 2008.

[12] Object Management Group. *UML 2.2 Superstructure Specification*, 2009. Document – formal/09-02-02.

[13] G. Rozenberg. *HANDBOOK of GRAPH GRAMMARS and COMPUTING by GRAPH TRANSFORMATION, Volume 1: Foundations*. World Scientific, 1997.

[14] C. Weise and D. Lenzkes. Efficient scaling-invariant checking of timed bisimulation. In *Proc. of STACS'97, LNCS 1200:pages 177–188*, pages 177–188. Springer-Verlag, 1997.

[15] W. Yi and B. Jonsson. Decidability of timed language-inclusion for networks of real-time communicating sequential processes. In *Foundation of Software Technology and Theoretical Computer Science*, volume 880 of *Lecture Notes in Computer Science*, pages 243–255, 1994.

[16] A. Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001.

[17] A. Zündorf. Model Checking the Leader Election Protocol with Fujaba. In *5th International Workshop on Graph-Based Tools (GraBaTs)*, July 2009.