Specification and Refinement Checking of Dynamic Systems*

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

Albert Zündorf
Software Engineering Research Group
University of Kassel
Wilhelmshöher Allee 73
D-34121 Kassel, Germany
zuendorf@uni-kassel.de

ABSTRACT

Software is increasingly used in systems which have to support self* properties like self-adaptation, -management or -optimization. The key enabler for a consistent model-based development approach is refinement. Refinement facilitates to preserve properties of abstract models in more concrete models. Note, that abstract models are of paramount importance to formal verification in complex safety critical systems. 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-based development approach. We present a modeling approach, called Timed Story Charts, which supports a flexible specification of properties like self-adaptation, and furthermore, we will present an integrated refinement check.

1. INTRODUCTION

Advanced software systems, like mechatronic 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. 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-based development approach. There are some approaches for modeling the structural aspects of reconfiguration or the behavioral aspects but none of them take into account both aspects [BCDW04].

Based on our approach Mechatronic UML [GHH+08], we present a modeling approach, called Timed Story Driven Modeling, an extension of Story Driven Modeling [Zün01], supporting a flexible specification of reconfiguration. Further on, we will present an integrated refinement check.

Mechatronic UML is based on a methodical decomposition of the embedded software and its constituent components. This supports compositional verification. Because of the dynamics in the behavior as well as the dynamics in the number of participating components, there is a need for modeling support for dynamic structures. This is also an existing problem of UML-components and -parts as e. g. creation and deletion of a part and its (delegated) port is not supported in UML. Similarly, UML does not address the question whether an embedded component (part) is a correct refinement of the protocol behavior of the surrounding component.

The presented Timed Story Driven Modeling approach supports the specification of complex dependencies of evolving behavior and it supports the specification of timing constraints for hard real-time systems like timed automata [AD94]. So, it combines the power of story driven modeling as well as timed automata, the quasi standard for the specification of real-time behavior. Hence, a consistent formalism is defined which is required by the analysis of timed behavior with complex dynamic changes at runtime. In contrast to former work [HHG08], we have a well defined consistent formalism and did not have to reason about different formalisms (which is difficult and could be error prone). Therefore, we only have to reason about analysis techniques for one formalism and not for different ones (e. g. statecharts and graph transformation systems). In this paper, we present only an overview of the Timed Story formalism and refinement checking. For more details, we refer to [Hei09].

In the following section, we present our Timed Story Driven Modeling approach. In Section 3, we present a refinement check for the Timed Story Driven approach. Related work is discussed in Section 4. We conclude with a summary and future work in Section 5.

2. TIMED STORY DRIVEN MODELING

To specify a system being able to dynamically change the structure (and therefore also the behavior) at runtime, we specify the structure of the system with UML 2.0-components and a many to many association between components (using so-called multi ports) and / or an to many association to its embedded components (called multi parts, see Section 2.1). The reconfiguration of the architectural parts like components, parts or ports are specified by (Timed) Story Diagrams and (Timed) Story Pattern (see Section 2.2 and Section 2.3). It seems, that in some cases we can specify the timing constraints of a reconfiguration by the behavior which triggers the reconfiguration (in form of timing constraints for the trigger). Hence, the timed ver-

*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.

sion of Story Pattern and Story Diagrams are in some cases optional. The behavior of the system, which also triggers the reconfiguration, is specified by Timed Story Charts (see Section 2.4).

2.1 System architecture

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 1). An example class diagram is shown in Figure 2.

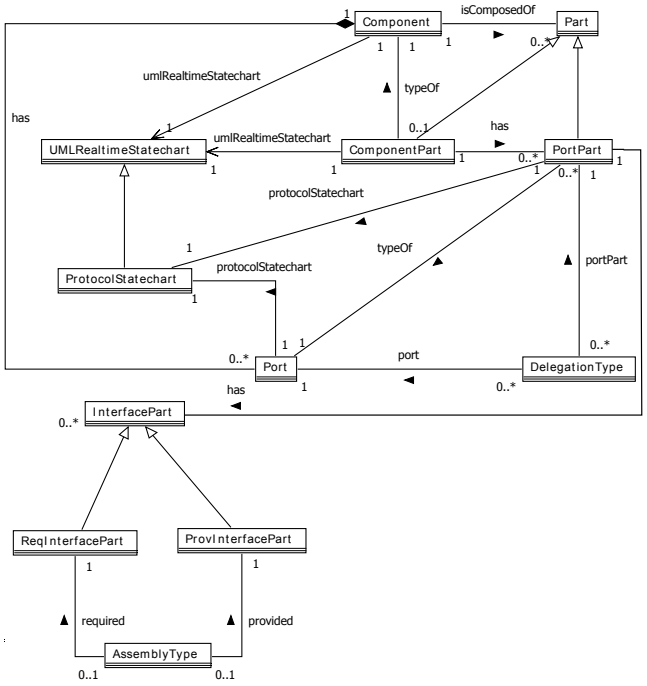

Figure 1: Component and parts meta model

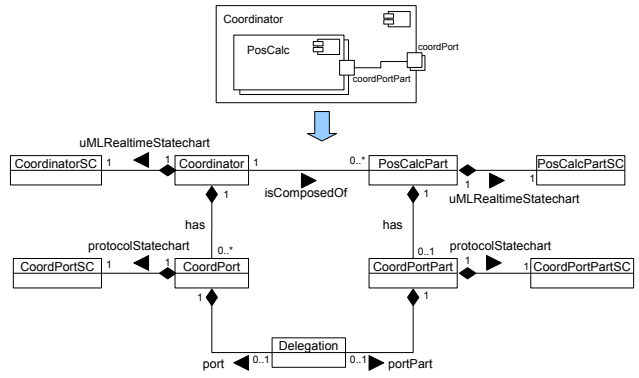

Figure 2: Example class diagram

2.2 Timed Story Pattern

In this section, we extend Story Pattern by time. Similar to the approach described in [Hir08], which extends the groove syntax and semantics by time, the timing concept of Timed Story Pattern is based on the semantics of Timed Automata [AD94]. We therefore support the specification of clocks, clock resets, time guards, and invariants.

Clocks are described by *clock instances* (see Figure 3). That means, clocks are represented by objects. Clocks are defined by a clock instance and its links to the objects of a graph. A definition of a clock instance for an edge is indirectly specified by the related objects of the edge.

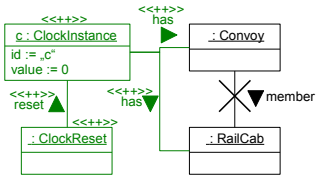

Figure 3: Defining a clock c

Clock resets are also modeled by objects which have a link reset to a clock instance. A clock reset object is instantiated at the same time as the associated clock instance. A *time guard* is implemented by a "standard" Story Pattern condition by referring to clock values. A precondition for specifying a time guard is, that the referred clock instance has to be bound in the Story Pattern. *Invariants* are specified by a special Story Pattern having no right hand side and the referred clock instance has to be bounded.

2.3 Timed Story Diagrams

The difference between a Timed Story Diagram and a Story Diagram is that a pattern of a Timed Story Diagram is a Timed Story Pattern. The semantics is the same as for Story Diagrams.

2.4 Timed Story Charts

The Timed Story Chart formalism supports abstract states, time constraints, and integrates dynamic adaptation by triggering a reconfiguration specified by (Timed) Story Pattern or (Timed) Story Diagrams.

Figure 4 shows the meta model of Timed Story Charts to support the specification of states and timing by objects. Transitions are implicitly implemented by rules[1]. In contrast to [Zün01], we did not use a framework for the execution semantics of the Story Chart as a reachability analysis would be difficult due to the single method executing the transitions. Hence, a transition would not be (easily) identifiable. Instead, we specify a story diagram which describes the execution semantics. In more detail, we explain in the following an overview of the syntax of some elements of Timed Story Charts and its execution semantics. We focus on the implementation of some specific statechart constructs. Clocks, guards, time invariants, time guards, clocks resets are implemented in Timed Story Pattern (see Section 2.2). Deadlines are implemented by the use of invariants and time guards and therefore are not discussed anymore.

States are represented by an extra object of type State. The name of the state is implemented by an attribute. *AND*

[1] An explicit transition object would lead to extra computations by the analysis

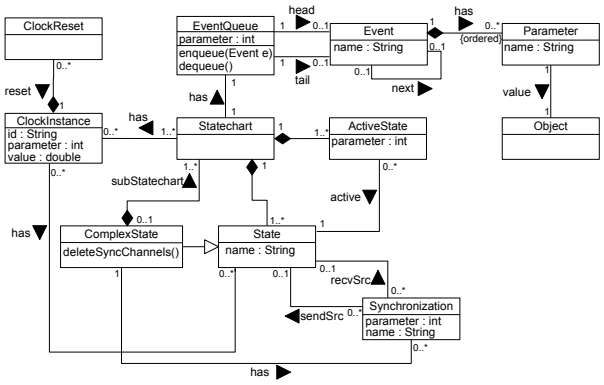

Figure 4: Meta model for the mapping of real-time statecharts to story diagrams

States are implemented by the class ComplexState. ComplexState can embed a set of statecharts. The *Active State* is implemented by an ActiveState object having an association to the active state. To differ between different instances of a Statechart (e.g. instances of a parameterized Statechart), we add an attribute parameter to the ActiveState object. If a statechart is instantiated more than once, an ActiveState object for each statechart is instantiated with a specific parameter. With this technique it is possible to manage the statechart instances of multi-parts and multi-port (see Figure 5).

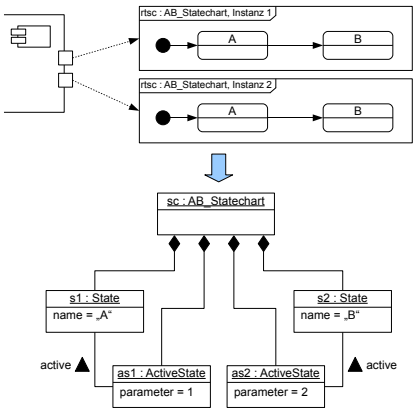

Figure 5: Example of a mapping of a statechart to a graph

An event is implemented by the Event type. The name of the event is implemented by an attribute of Event. Parameterized events are implemented by an ordered set of parameters by the type Parameter. The value of the parameter is specified by an association to a corresponding object. An event-queue is associated to each statechart (instance). An example of a trigger event a is shown in Figure 6. 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. Further more, Figure 6 shows the implementation of transition by a Story Diagram. The syntax of a story is that of a Timed Story Pattern (see Section 2.2).

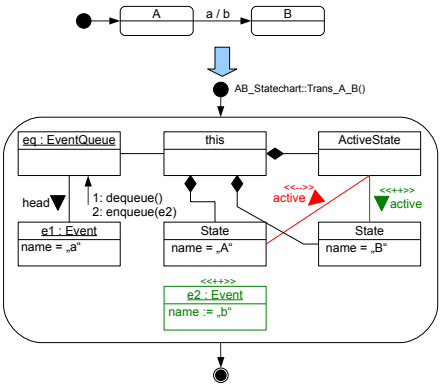

Figure 6: Events

Synchronization is implemented by an extra Synchonisation object. By a name attribute the name of the synchronization is implemented and the parameter attribute refers to a specific statechart instance. As the transitions which have to be synchronized have to fire simultaneous a joint story diagram is specified including both transitions. An example is shown in Figure 7.

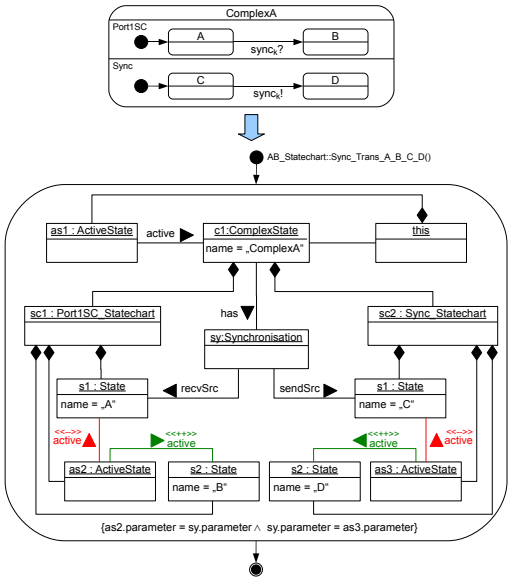

Figure 7: Synchronization

The syntactical mappings of (Real-Time) Statechart constructs defined above are now combined to a sequence of transformations (see Figure 8) defining the execution semantics of Timed Story Charts. Note, we discuss only a view relevant constructs in this paper. A complete definition of Timed Story Charts is presented in [Hei09]. A transition of a Real-Time Statechart is mapped to a modular Timed Story Chart with a set of stories. Hence, the exchange of the semantics is easy. As an example, we take into account the semantics of a transition without deadline. Figure 8 shows a schematic Timed Story Chart. The activities are stories. The first story analyzes whether the precondition is fulfilled. This includes binding the source state, the event trigger, and synchronization channels. Furthermore, all (time) guards

are considered. If all bindings and (time) guards are fulfilled, the transition can fire (1. story). The 2. story takes the relevant events from the queue. The 3. story eliminates all synchronization objects from the source state. In story 4. the exitAction is executed. The side effect is executed in story 5. and the trigger events are eliminated. Next, in story 7. , raised events are created and clocks are reseted. Finally, in story 8. the synchronization channels of the target state are instantiated and, in story 9., the target state is entered.

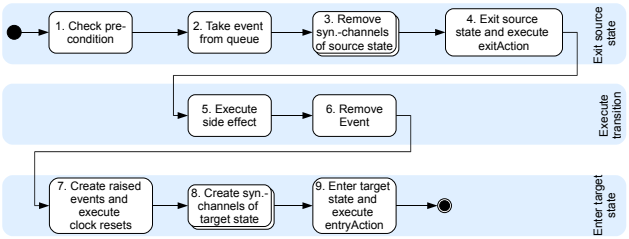

Figure 8: Execution semantics

3. REFINEMENT

Basically, we have two requirements for the refinement: 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 (cf. Figure 9).

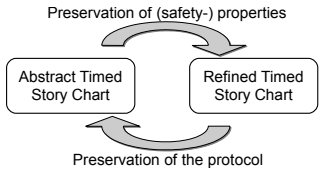

Figure 9: Requirements to the refinement

For supporting requirement 1), we require that each external visible trace consisting of events (in- and out-going) and its timing, is implemented by the refined behavior. That means each event supported by the abstract behavior has to be supported by the refined behavior[2] as well and the refined behavior has to react in the same time interval as the abstract one. In contrast to other definitions, we allow the refined behavior to have a more relaxed receive interval. For requirement 2), we require that each trace of the refinement is related to (simulated by) a trace of the abstract behavior. That means the abstract behavior simulates the refined and therefore compositional verification results are preserved [CGP00]. In [Hei09] a detailed definition is presented.

In the following we present the refinement check by first computing a reachability graph of the abstract and the refined Timed Story Chart (see Section 3.1). In principle the reachable graph could be infinite. As we take into account hard real-time systems, the reachable graph is finite as dependent behavior (statecharts) has an upper hard timing constraints or the behavior are independent and therefore only one possible instance of the behavior has to be considered. In [Hei09] we discuss an alternative approach which could also take into account an infinite system. Based on the reachable graphs, we illustrate an algorithm for checking the refinement in Section 3.2.

[2]It is allowed that the refined behavior supports more events than the abstract behavior

3.1 Reachability Analysis

The timed reachability analysis is based on the computation of reachability graphs as introduced in [Zün09]. 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 [Zün09] 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 a library provide isomorphism check [Zün09], 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 a dedicated clock zone ([CGP00, p. 280]), cf. [Hei09]. 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.

Note, in general the computation of a reachability graph may not terminate. In our case, we model the execution of finite timed story charts and the timing constraints result in another restriction concerning the length of execution pathes.

3.2 Refinement Check

The refinement check is realized by a depth-first search (see algorithm 1). The algorithm checks for each event of the refined behavior if a corresponding event in the abstract behavior exist. First the algorithm starts with a (timed) reachability analysis as described in the last section. Then it is checked if for the starting states a structural refinement exist. If nodes exist which are not expanded a successor is checked (row 8) and expanded (row 9). For each successor it is checked if the successor is already known. If a successor is not known it is checked if an event is triggered or raised and, if so, it is checked if a corresponding trace exist. If a node is already known it could be the case that a circle is closed or two traces are joined. In both cases, if an event is triggered or raised it is checked if a corresponding trace exist. In cases of a circle, we check if the circle is well-formed. That means, we check if the circle has a corresponding trace.

4. RELATED WORK

In [Gie07] 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. [HT04] 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. [GRPS02] examine refinement for graph transformation systems based on an algebra but they did not take into account time.

Algorithm 1 Refinement check

```
 1: function checkCorrectRefinement(TimedStoryChart abs, TimedStoryChart ref)
 2:     absReach = startReachabilityAnalysis(abs)
 3:     refReach = startReachabilityAnalysis(ref)
 4:     success := checkStructureRefinement (absReach.initial, refReach.initial)
 5:     OPEN.push(refReach.initial)
 6:     while OPEN ≠ ∅ ∧ success do
 7:         n := OPEN.pop( )
 8:         success := refReach.hasSuccessor(n)
 9:         for all n' ∈ refReach.expand(n) do
10:             if n' is not known then ▷ Case 1: new node
11:                 OPEN.push(n')
12:                 if (n, n') has event e then
13:                     success := checkPath((n, n'))
14:                 end if
15:             else ▷ Case 2: node is known
16:                 if (n, n') closed cycle then ▷ a) Edge closes an circle
17:                     if (n, n') has event e then
18:                         success := checkPath((n, n'))
19:                     end if
20:                     success := isWellFormedCycle(n')
21:                 else ▷ b) Joining of two traces
22:                     if (n, n') has event e then ▷ The same as case 1
23:                         success := checkPath((n, n'))
24:                     end if
25:                 end if
26:             end if
27:         end for
28:     end while
29:     if success then
30:         success := checkCoverage(absReach)
31:     end if
32:     return success
33: end function
```

5. CONCLUSION AND FUTURE WORK

We presented in this paper the Timed Story Driven approach and a refinement check for Timed Story Charts, the behavioral specification language of the Timed Story Driven approach. We gave in more detail an overview of the syntax and semantics of Timed Story Charts. The presented refinement check, which preserves compositional verification results and the external visible real-time behavior, is based on a reachability analysis.

The expand of the reachability analysis is a manual task. Especially for timed systems, this could be error prone. Hence, a future task is to develop an automatic expand.

6. REFERENCES

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[BCDW04] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel 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.

[CGP00] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. January 2000.

[GHH+08] Holger Giese, Stefan Henkler, Martin Hirsch, Vladimir Roubin, and Matthias Tichy. Modeling techniques for software-intensive systems. In Dr. Pierre F. Tiako, editor, *Designing Software-Intensive Systems: Methods and Principles*, pages 21–58. Langston University, OK, 2008.

[Gie07] Holger 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.

[GRPS02] Martin Große-Rhode, Francesco Parisi Presicce, and Marta Simeoni. Formal software specification with refinements and modules of typed graph transformation systems. *J. Comput. Syst. Sci.*, 64(2):171–218, 2002.

[Hei09] Christian Heinzemann. Verifikation von Protokollverfeinerungen. Master Thesis, Software Engineering Group, University of Paderborn, Nov 2009. to appear.

[HHG08] Martin Hirsch, Stefan Henkler, and Holger 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.

[Hir08] Martin Hirsch. *Modell-basierte Verifikation von vernetzten mechatronischen Systemen*. PhD thesis, University of Paderborn, Paderborn, Germany, September 2008.

[HT04] Reiko Heckel and Sebastian 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.

[Zün01] Albert Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001.

[Zün09] Albert Zündorf. Model Checking the Leader Election Protocol with Fujaba. In *5th International Workshop on Graph-Based Tools (GraBaTs)*, July 2009.