Electronic Communications of the EASST
Volume 32 (2010)

Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)

Reachability Analysis on Timed Graph Transformation Systems

Christian Heinzemann, Julian Suck, Tobias Eckardt

12 pages

Guest Editors: Juan de Lara, Daniel Varro
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Reachability Analysis on Timed Graph Transformation Systems

Christian Heinzemann, Julian Suck, Tobias Eckardt *

Software Engineering Group
Heinz Nixdorf Institute
University of Paderborn
Warburger Strasse 100
D-33098 Paderborn, Germany
c.heinzemann|jsuck|tobie@uni-paderborn.de

Abstract: In recent years, software increasingly exhibits self-* properties like self-optimization or self-healing. Such properties require reconfiguration at runtime in order to react to changing environments or detected defects. A reconfiguration might add or delete components as well as it might change the communication topology of the system. Considering communication protocols between an arbitrary number of participants, reconfiguration and state-based protocol behavior are no longer independent from each other and need to be verified based on a common formalism. Additionally, such protocols often contain timing constraints to model real-time properties. These are of integral importance for the safety of the modeled system and thus need to be considered during the verification of the protocol. In current approaches either reconfigurations or timing constraints are not considered. Existing approaches for the verification of timed graph transformation systems lack important constructs needed for the verification of state-based real-time protocol behaviors. As a first step towards a solution to this problem, we introduced Timed Story Driven Modeling [HHH10] as a common formalism integrating state-based real-time protocol behaviors and system reconfigurations based on graph transformations.

In this paper, we introduce a framework allowing to perform reachability analysis based on Timed Story Driven Modeling. The framework allows to compute the reachable timed graph transition system based on an initial graph and a set of timed transformation and invariant rules.

Keywords: Verification, Real-time Systems, Graph Transformation Systems, Reachability Analysis

* 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.
This work was developed in the project "ENTIME: Entwurfstechnik Intelligente Mechatronik" (Design Methods for Intelligent Systems). The project ENTIME is funded by the state of North Rhine-Westphalia (NRW), Germany and the EUROPEAN UNION, European Regional Development Fund, "Investing in your future".

1 Introduction

In recent years, software increasingly exhibits self-* properties like self-optimization or self-healing. Such properties require reconfiguration at runtime in order to react to changing environments or detected defects. This causes a significant increase in the complexity of the software as also the reconfiguration process has to be controlled by the software. As software often operates in safety critical environments, it has to meet highest quality standards. Formal verification of safety and liveness constraints as well as verification of joint structural and behavioral constraints [KG07] address these requirements.

For embedded or real-time systems timing constraints for the software have to be taken into account during verification. Model Checkers like Uppaal[1] address these issues as they allow to check timed temporal logic formulas based on timed automata [Alu99]. Standard model checkers for real-time systems, however, are not able to consider changing system topologies resulting from system reconfigurations (cf. Section 6). Graph based model checkers like Groove [Ren08] support dynamic topologies, but are not capable of verifying timing constraints. Unfolding the state-space described by the graph transformation rules and using model checkers to verify the result does not work for constraints referencing both, structural and behavioral parts of the system. Existing approaches combining graph transformations and real-time constructs (cf. Section 6) come with restrictions that do not allow to model timed behavior to the extent that is needed for our systems. As a solution, we realize state-based real-time behavior using graph transformation systems extended with timing constraints derived from timed automata. This enables us to integrate dynamic reconfiguration of the communication structure and to reuse the existing implementation for time computations from Uppaal.

In this paper, we introduce a framework for timed reachability analysis based on Timed Story Driven Modeling [HHH10]. In [HHPS10], reconfiguration and state-based behavior are analyzed independently. There, only pairs of automata were checked and an induction over the reconfigurations of a regular architecture was used to show that no forbidden communication structures can arise. This ensures, that only such communication pairs can arise, that have been verified before. In this paper, we use Timed Story Charts [HHH10] as an explicit common formalism for the verification. This enables us to specify and verify arbitrary constraints that affect both, state-based protocol behavior and structural system state at the same time, e.g. that a reconfiguration may only take place if a certain communication protocol is in a specific state.

Example Scenario Our example scenario stems from the RailCab[2] project. The RailCab system consists of autonomous RailCabs that are fully controlled by software. RailCabs can form contactless convoys to reduce energy consumption. A convoy of RailCabs always needs one coordinating RailCab in order to prevent oscillation of the shuttle distances. A component instance model of a convoy with three RailCabs is shown in Figure 1 where RailCab rc1 coordinates the convoy.

[1] http://www.uppaal.com/
[2] http://www.railcab.de

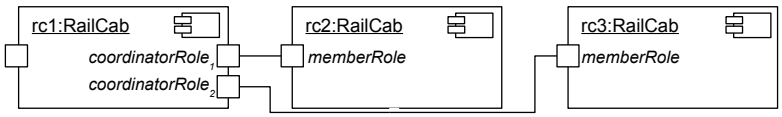

Figure 1: Component instances for the RailCab example.

Outline The remainder of this paper is structured as follows. First, we give a brief overview of Mechatronic UML. In Section 3, we explain how a reachability analysis is performed using Timed Story Driven Modeling while Section 4 provides a description of the framework. We present the results of our evaluation in Section 5 and related work in Section 6. Section 7 concludes the paper.

2 Mechatronic UML

In this section, we briefly introduce MechatronicUML, an adaptation of the UML for modeling mechatronic systems. It provides extensions for modeling and verifying real-time systems and hybrid systems integrating continuous control components. Section 2.1 describes the system architecture of the system and Section 2.2 gives a short overview of Timed Story Driven Modeling with respect to MechatronicUML.

2.1 System architecture

We use a component-based system architecture based on MechatronicUML components. The communication between components is modeled with parameterized coordination patterns as introduced in [HHPS10]. Parameterized coordination patterns are used to specify 1 to n communication protocols between communication partners, called *roles* for cardinality 1 or *multi roles* for cardinality n, respectively. Roles are instantiated at the ports of a component in order to provide the corresponding protocol.

The behavior of roles is specified by real-time statecharts [GB03]. In case of a multi role, a parameterized real-time statechart [HHPS10] is used to define the behavior of all sub-roles. Figure 2 shows an example of a parameterized pattern including the real-time statecharts defining the role behaviors.

The coordinator role sends an update event to the member role which answers with an acknowledgement. The internal synchronization channel next is used to synchronize the different sub-roles of the coordinatorRole as they are not independent in this scenario. The channel is parameterized with a parameter k referencing to the k^{th} instance of the statechart. Thus, one sub-role triggers the next one in the example. Additionally, a property $AG\neg deadlock$ is specified for the pattern. Such properties may be verified for the pattern using the reachability analysis introduced in Section 3.

Following [HHPS10], we use an additional adaptation statechart to synchronize the roles and to manage creation and deletion of roles. An excerpt of an adaptation statechart for the example is shown in Figure 3 on the left. The adaptation statechart is initially in state noConvoy. The decision which RailCab coordinates the convoy is made in another statechart which is omitted

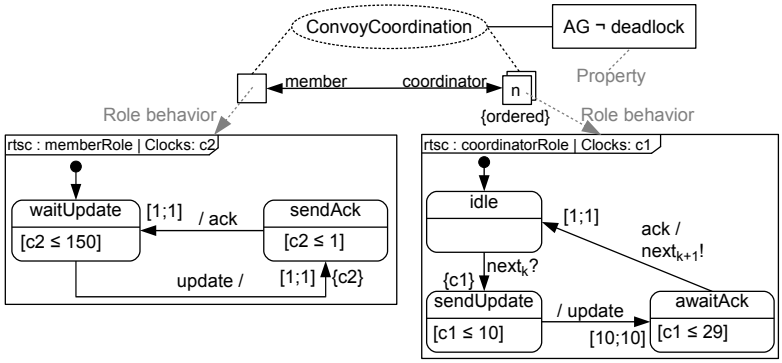

Figure 2: Definition of a Parameterized Coordination Pattern

here. If the RailCab is chosen to coordinate, it changes its state to addMember, thereby performing the side effect createPort(1) of the transition. The side effect is a method of the component being specified by the story diagram [FNTZ00] on the right in Figure 3. The side effect in this example simply creates a new port. The deadline $[10;10]$ of the transition denotes that this reconfiguration takes at least 10 time units and at most 10 time units. After reaching the state convoy, the invariant forces the statechart to switch to state sendUpdates every 150 time units, thereby triggering the first coordinatorRole to send the update.

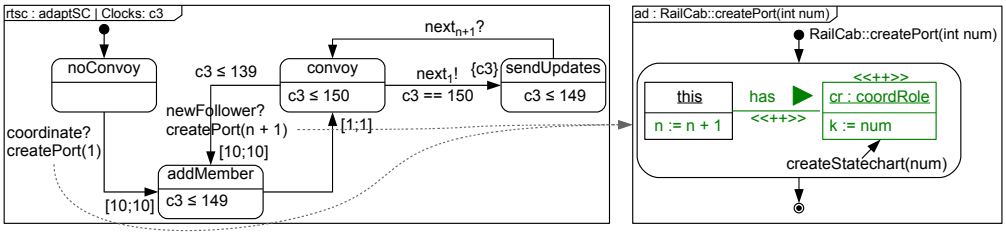

Figure 3: Adaptation Statechart for a Multi Port

2.2 Timed Story Driven Modeling

The Timed Story Driven Modeling [HHH10] approach is based on Timed Story Patterns and Timed Story Diagrams. Timed Story Patterns are a short-hand notation for timed graph transformations [HHPS10, HHH10] that depict the left hand side and the right hand side in one graph. Elements being created (or deleted) by the transformation are labeled with <<++>> (or <<−−>>).

Timed graph transformations operate on timed graphs which contain clocks like timed automata [Alu99], each being associated with a subgraph of the graph. The same clock can occur multiple times, once for each occurrence of the associated subgraph. Therefore, we use the term *clock instance* to denote the instances of a clock. The number of clock instances to be added to the graph, however, has to be finite, but it may vary during run-time. The representation of clock values is realized using clock zones [Alu99, BY03], as in timed automata.

In Timed Story Patterns, three kinds of rules are used: transformation rules, invariant rules,

and clock instance rules. A clock instance rule adds clock instances to the graph which are used by the transformation rules and the invariant rules to specify timed behavior. Transformation rules change the graph while invariant rules put a condition on the values of the clock instances of the timed graph. Due to space limitations, we only show how these rules are implemented in our framework in Section 4.2.

Story diagrams [FNTZ00] extend UML Activity Diagrams by embedding graph transformations specified by story pattern into the activities. We obtain Timed Story Diagrams by embedding Timed Story Patterns into the activities. To allow the execution of real-time statecharts using Timed Story Diagrams while preserving their semantics, we define Timed Story Charts [HHH10] on the basis of Timed Story Diagrams. The core idea is to represent the statecharts and their states as nodes of a graph and to provide graph transformation rules (Timed Story Diagrams) specifying the state changes resulting from transitions. The currently active state of the statechart is represented by an ActiveState-node. The transformation of real-time statecharts to Timed Story Charts has been partially automated [HSJZ10].

3 Reachability Analysis

The reachability analysis computes the Timed Graph Transition System (TTS) based on the given transformation rules and invariants. It represents the complete reachable behavior. In general, the TTS may be infinite. Thus, it cannot be guaranteed that the algorithm will eventually terminate for a given set of rules. This is a general problem when computing graph transition systems. The TTS can be defined as follows:

Definition 1 (Timed Graph Transition System (TTS)) Let $\mathscr{G}$ be the set of all possible timed graphs, $\mathscr{R}$ a set of transformation rules, $\mathscr{I}$ a set of invariant rules. The Timed Graph Transition System (TTS) is a triple (S, s_0, T) where S represents the set of 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 $g \in \mathscr{G}$ 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 $r \in \mathscr{R}$ 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 [Alu99]. 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 [Alu99, BY03], the only difference is that the states contain a timed graph instead of an automaton location.

The computation starts with the initial graph and all clocks being 0. Then, possible successors are computed (see Definition 2). The TTS contains transitions from a state to all its successors.

Definition 2 (Successor State) Let $s_1 = (g_1, z_1), s_2 = (g_2, z_2)$ states of a TTS. s_2 is a successor state of s_1 iff

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

The definition of a successor state is analogous to the one of timed automata [Alu99]. The only difference is that the change in the TTS results from the application of a transformation rule instead of an automaton transition. 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 clock instances (cf. [BY03]). Afterwards, the intersection against the invariants is repeated. Then, the resulting clock zone is intersected with the time guards of the applied transformation rule and the rules' clock resets are executed. We restrict ourselves to guards of the form $c \sim n$ where c is a clock instance, $\sim \in \{<, \leq, =, \geq, >\}$, and $n \in \mathbb{N}$, i.e., comparing the value of a clock instance with an integer. Invariants are further restricted to comparisons $<$ and $\leq$. Please note that there may exist more than one possible successor state for the same transformation rule as multiple matchings can be found.

In order to obtain a finite TTS, isomorphic states of the TTS are merged into one state. Two states are isomorphic, iff their graphs are isomorphic to each other and their clock zones are identical.

4 Verification Framework

We implemented the reachability analysis introduced in Section 3 into our framework. In the following subsections, we introduce the general architecture of our framework at first. Second, we give an introduction how rules can be modeled, third, we explain how the TTS is computed, and finally, we give a brief idea of properties that can be checked using the framework. Part of the framework, not including the timing capabilities, has been shown in [HSJZ10].

4.1 Architecture

An implementation of a reachability analysis as specified in Section 3 requires additional rules which compute the TTS. Specifying concrete rules for the TTS generation for each example by hand is a tedious, error-prone task. Therefore, we provide a framework which requires the user to specify an initial graph and a set of rules, only. The remaining tasks, e.g. the application of rules, are integrated within the framework. Figure 4 shows the class diagram of the framework.

The abstract class ReachabilityComputation encapsulates the whole functionality for computing the timed graph transition system. It contains two abstract methods createInitialGraph() and createRules(). Both have to be implemented by the user, whereas the former defines the initial graph and the latter specifies which graph transformation rules are to be used for the reachability computation and which time constraints these rules have.

Graphs are represented by objects of the class StepGraph. They contain objects of the class Node. To represent different types of nodes in a graph, subclasses of Node can be created. Edges between nodes are represented by associations.

As already mentioned in Section 2.2, there exist three different kinds of rules in a timed graph transformation system, namely *timed graph transformation rules*, *invariant rules* and *clock instance rules*. The first two are represented by the classes TransformationRule and InvariantRule, respectively, whereas the third is represented by the method addClockInstances(). The next sec-

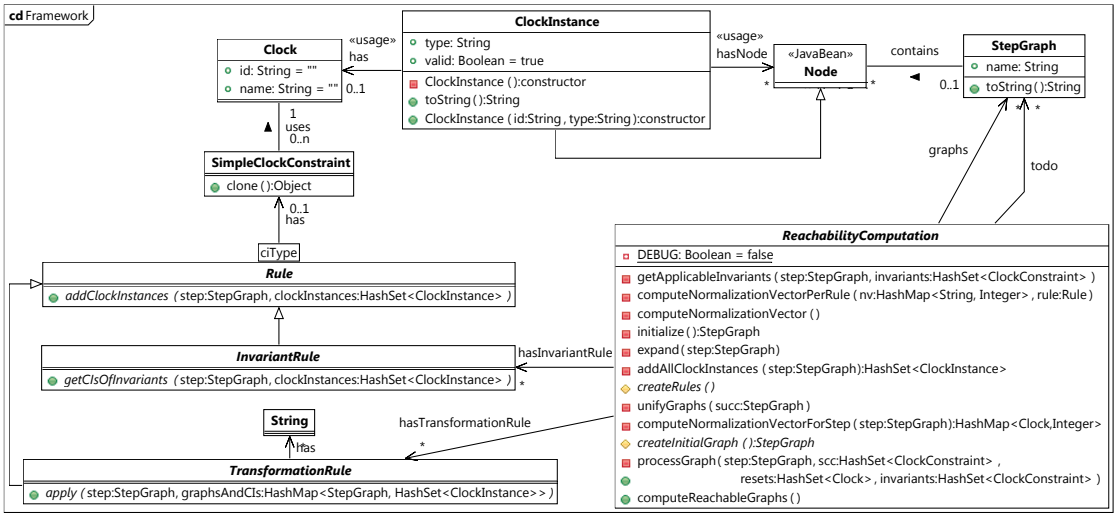

Figure 4: Class Diagram of the Verification Framework

tion contains more detailed information on how to implement transformation and invariant rules.

4.2 Modeling Rules

Timed graph transformation rules are represented by the abstract class TransformationRule. It contains an abstract method apply() which has to be implemented by subclasses to specify a concrete timed graph transformation rule. As parameters, this method receives a graph on which it is to be applied and a reference to a mapping. After the termination of the method, the mapping contains all reached successor graphs together with their respective clock instances used for the application of the rule.

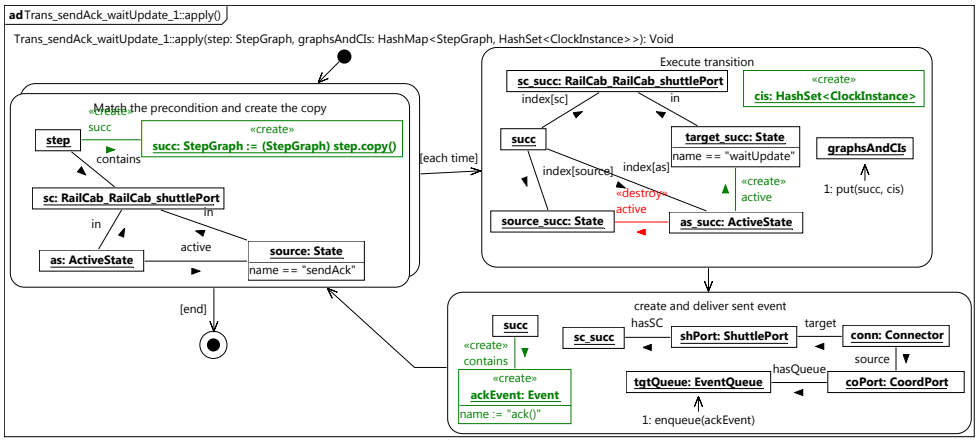

Figure 5: Story Diagram of the Transformation Rule, Modeling the Transition from *sendAck* to *waitUpdate*

Figure 5 shows an implementation of the method apply() modeling the transition from the state sendAck to the state waitUpdate of the member role statechart (cf. Figure 2). The story diagram consists of three activities. The first activity checks whether the rule is applicable, which is the case if the sendAck-state is the active state in the given statechart. If the rule is applicable, the second activity sets the active state from sendAck to waitUpdate. Finally, the third activity enqueues the ack()-Event into the event queue of the statechart of the coordinator port.

Invariant rules are specified by subclasses of InvariantRule. Subclasses implement the method getClsOfInvariant() which receives a graph on which the rule is to be applied and a set as parameters. After the termination of the method, the set contains all clock instances of the graph for which the time invariant specified by the rule is applicable.

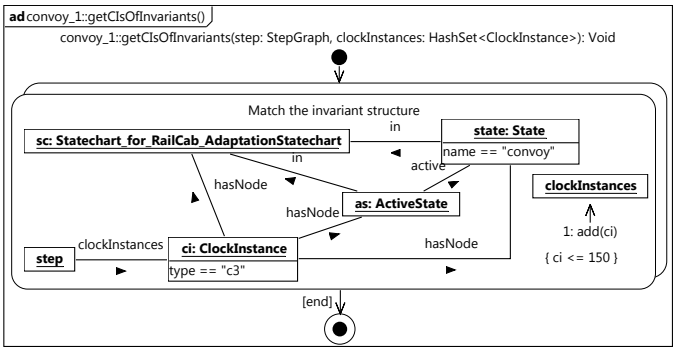

Figure 6: Story Diagram of the Invariant Rule, Modeling the Invariant of State *convoy*

Figure 6 shows a concrete example of a story diagram implementing the method getClsOfInvariant(). The rule represents the invariant $c3 \leq 150$ of the state convoy of the adaptation statechart (cf. Figure 2). The invariant has to hold whenever the adaptation statechart's active state is the convoy-state. The activity matches to all structures which model exactly this situation. Whenever a matching structure is found, the corresponding clock instance is inserted into the set.

4.3 Computing the TTS

The verification of properties requires to compute the TTS according to Definition 1 by applying Algorithm 1 of our framework.

The algorithm maintains the TTS and a TODO list storing all states whose successors have not yet been computed. For each of these states, first all clock instance rules are applied (Line 6). Then, all currently applicable invariants are collected in Line 7. The loop starting in Line 8 applies all transformation rules to the current graph by calling the apply function. For each of the resulting successors, the successor clock zone is computed according to Definition 2 by the function processGraph (Line 11). Finally, unifyGraphs in Line 12 checks for isomorphisms and adds the state to the TTS and the TODO list.

All time computations, i.e. operations on clock zones, are performed using the Uppaal DBM (UDBM) library[3]. The C/C++-library, as originally implemented for the Model Checker Uppaal,

[3] http://www.cs.aau.dk/~adavid/UDBM/

Algorithm 1 Computation of the Timed Graph Transition System

```
1:  function COMPUTETTS(Graph start, TransformationRules rules, InvariantRule inv)
2:      TTS.add(start)
3:      TODO.push(start)
4:      while TODO ≠ ∅ do
5:          curState := TODO.pop()
6:          addAllClockInstances(curState.g)
7:          appInv := getApplicableInvariants(curState.g, inv)
8:          for all r ∈ rules do                          ▷ compute successor graphs
9:              successors := r.apply(curState.g)
10:             for all s ∈ successors do
11:                 processGraph(s, r)
12:                 unifyGraphs(s)
13:             end for
14:         end for
15:     end while
16:     return TTS
17: end function
```

efficiently implements all necessary operations on clock zones (up or delay ($\Uparrow$), intersection ($\wedge$), clock resets ($[reset(r)]$)). Technically, we access the UDBM library using the provided Ruby binding in connection with a (local) client/server-communication between our Java implementation and the Ruby implementation. In Java, clocks, clock zones and clock constraints are represented as classes providing the corresponding operations on those instances as methods. This makes the binding to the UDBM library completely transparent for the developer and allows insertion and removal of clock instances which is not directly supported by the UDBM. In Java, clock instances can be created like normal objects. During runtime, a given clock zone is then transformed into ruby code as well as the desired operation is transformed. This ruby code is sent to the ruby server, which executes it and sends back the resulting clock zone as an encoded string. This string is finally transformed back into a clock zone object representing the result of the operation.

4.4 Verification of Properties

Currently, our framework only supports the verification of CTL (Computation Tree Logic) formulae having the form $EF\varphi$ or $AG\neg\varphi$ where φ is a graph invariant. Thus, it is possible to check whether a specific subgraph eventually occurs in the graph or to check whether a subgraph never occurs. Such properties are modeled as invariant rules containing the respective graph φ. A formula $EF\varphi$ is fulfilled if the rule can be matched eventually to the graph. A formula $AG\neg\varphi$ is not satisfied when a state exists in which the graph cannot be matched. In the TTS, a path from the initial state to the state in which the property does not hold serves as a counter-example.

Additionally, deadlock freedom, denoted by $AG\neg deadlock$ in Figure 2, may be verified. A deadlock corresponds to a state in the TTS with no outgoing transitions. Again, a path in the TTS from the initial state to the deadlock state serves as a counter-example.

Please note that the verification of properties is only decidable if the TTS is finite.

5 Evaluation

We implemented the convoy coordination example shown in Figures 2 and 3 using the pattern and the statecharts. This resulted in 15 transformation rules and 13 invariant rules. In this case, we only needed three clock instance rules, one for each statechart, because we can create a statechart instance as a whole, along with all its clock instances.

The number of RailCabs in a convoy was restricted to a maximum number in order to obtain a TTS for different maximum convoy sizes using our framework described in Section 4. The results are summarized in Table 1.

Table 1: Evaluation results for different convoy sizes

max convoy size	# graphs in TTS	run-time (s)	run-time (s) optimized	max. graph size
2	17	2	1	37
3	52	21	2	52
4	112	125	6	67
5	203	545	16	82
6	329	1856	39	97

A convoy size of 2 corresponds to one leading RailCab and one convoy member RailCab, i.e., the leading RailCab has one *coordinatorRole* statechart instance. For each additional RailCab in the convoy, an additional instance is added. We computed the TTS for our example with a maximum convoy size of 6 RailCabs because the timing constraints in our example do not support larger convoys. We recently found a major performance problem in our implementation that yielded a significant improvement of our run-time as shown in Table 1. The results indicate that the run-time grows exponentially in the number of reached graphs while the maximum graph size grows constantly as expected. The growth in run-time results from the high number of clock instances and the expensive timing computations which consume about 66% of the runtime. Additionally, our isomorphism check on graphs turned out to be inefficient [HSJZ10]. The timing computations along with the definition of isomorphic states cause a certain blowup in the number of reached states, as isomorphic graphs had to be expanded more than once because of differences in the clock zones.

6 Related Work

In the field of timed graph transformation systems, there exist several other approaches. The MOMENT2 framework [BÖ10] provides model transformations based on MOF meta models. The approach supports one unresetable clock per object, timers, that trigger actions, and timed values which can be increased or decreased at a certain, fixed rate. The real-time graph rewrite model checker Real-Time Maude [ÖM07] provides object oriented graph transformations in a textual syntax, but it does not support invariant rules requiring subgraph changes. The approach by Rivera et al. [RDV09] provides only one global clock and durations for the execution of rules but no guarding of rules by time constraints. In [RDV10] a mapping to Real-Time Maude is defined which allows to simulate and model check the specification. De Lara et al. [LV10] map their graph transformation rules to timed Petri nets. Time is not actually part of the model,

but annotated as an interval in which the transformation can be executed after a match has been found. In [THRB10], stochastic graph transformations are introduced. The simulation of these transformations incorporates a scheduling that is based on continuous time and executes a rule at a randomly generated point in time. A drawback of all approaches is their lack of support for flexible clock creation with resets and the specification of time guards at the same time. This, however, is needed for the system models we employ.

There exist some approaches for checking graph transformations without the possibility to consider timing constraints. Groove supports a reachability analysis on labeled graphs and checking graph based CTL formulas on the graph transition system [Ren08]. König et. al. [KK08] use an approximation technique that maps a possibly infinite graph transition system to finite Petri graphs and verifies the specified formula on this Petri graph structure. The inductive invariants introduced in [GS04] support infinite state spaces and only require a static analysis on the set of rules showing that a forbidden graph cannot be produced. It is not possible, however, to verify properties that cannot be depicted as a graph, like deadlock freedom, for example.

Bauer et. al. provide a verification approach for dynamic communication protocols [BSTW06] using over- and underapproximation of the system in order to verify LTL (Linear-time Temporal Logic) formulas with first-order quantification on objects. The approach supports infinite numbers of communicating objects and finite message queues.

The timed model checker Uppaal provides the ability to check timed systems, but not the evolution of the system in terms of adding new statechart behaviors at run-time. The BIP framework [BS10] also provides real-time components and connectors with extensive analysis approaches, but does not support reconfigurations, either.

7 Conclusions

In this paper, we have shown a technique to perform a reachability analysis on Timed Story Diagrams, a dialect of graph transformation systems extended by the notion of time. We use Timed Story Charts as a common formalism to perform a reachability analysis for dynamic real-time communication protocols whose structural evolution is specified by graph transformations.

As future work, we plan to extend our framework by the possibility to verify more complex timing constraints, as introduced in [KG07]. We will also investigate the possibility of applying existing abstraction and approximation techniques to the timed graph transformation systems in order to be able to handle larger state spaces and to obtain a more efficient verification procedure. Finally, we will try to fully automate the Timed Story Chart generation by adding generation of synchronizations and message recipients.

Bibliography

[Alu99] R. Alur. Timed Automata. In Halbwachs and Peled (eds.), *Proc. of the 11th Intern. Conf. on Computer Aided Verification (CAV '99), Trento, Italy*. LNCS 1633, pp. 8–22. Springer, 1999.

[BÖ10] A. Boronat, P. C. Ölveczky. Formal Real-Time Model Transformations in MOMENT2. In *Proc. of the 13th Intern. Conf. on Fundamental Approaches to Software Engineering, FASE 2010*. Pp. 29–43. 2010.

[BS10] S. Bliudze, J. Sifakis. Causal semantics for the algebra of connectors. In *Formal Methods in System Design*. Volume 36(2), pp. 167–194. Springer, 2010.

[BSTW06] J. Bauer, I. Schaefer, T. Toben, B. Westphal. Specification and Verification of Dynamic Communication Systems. In *6th Intern. Conf. on Application of Concurrency to System Design, 2006. ACSD 2006.* IEEE Computer Society Press, 2006.

[BY03] J. Bengtsson, W. Yi. Timed Automata: Semantics, Algorithms and Tools. In Desel et al. (eds.), *Lectures on Concurrency and Petri Nets*. LNCS 3098, pp. 87–124. Springer, 2003.

[FNTZ00] T. Fischer, J. Niere, L. Torunski, A. Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In Ehrig et al. (eds.), *TAGT'98: Selected papers*. LNCS 1764, pp. 296–309. Springer, 2000.

[GB03] H. Giese, S. Burmester. Real-Time Statechart Semantics. Technical report tr-ri-03-239, Software Engineering Group, University of Paderborn, Germany, June 2003.

[GS04] H. Giese, D. Schilling. Towards the Automatic Verification of Inductive Invariants for Infinite State UML Models. Technical report tr-ri-04-252, Software Engineering Group, University of Paderborn, Germany, December 2004.

[HHH10] C. Heinzemann, S. Henkler, M. Hirsch. Refinement Checking of Self-Adaptive Embedded Component Architectures. Technical report tr-ri-10-313, Software Engineering Group, University of Paderborn, Mar. 2010.

[HHPS10] S. Henkler, M. Hirsch, C. Priesterjahn, W. Schäfer. Modeling and Verifying Dynamic Communication Structures based on Graph Transformations. In *Proc. of the Software Engineering 2010 Conf., Paderborn, Germany*. 2010.

[HSJZ10] C. Heinzemann, J. Suck, R. Jubeh, A. Zündorf. Topology Analysis of Car Platoons Merge with FujabaRT & TimedStoryCharts - a Case Study. In Gorp et al. (eds.), *Transformation Tool Contest*. Malaga, 2010.

[KG07] F. Klein, H. Giese. Joint Structural and Temporal Property Specification Using Timed Story Scenario Diagrams. In *Formal Approaches to Software Engineering*. LNCS 4422, pp. 185–199. Springer, 2007.

[KK08] B. König, V. Kozioura. Towards the Verification of Attributed Graph Transformation Systems. In *ICGT '08: Proc. of the 4th Intern. Confe. on Graph Transformations*. Pp. 305–320. Springer, Berlin, Heidelberg, 2008.

[LV10] J. de Lara, H. Vangheluwe. Automating the transformation-based analysis of visual languages. In *Formal Aspects of Computing*. Volume 22(3), pp. 297–326. Springer, 2010.

[ÖM07] P. C. Ölveczky, J. Meseguer. Semantics and Pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation* 20(1-2):161–196, 2007.

[RDV09] J. E. Rivera, F. Duran, A. Vallecillo. A graphical approach for modeling time-dependent behavior of DSLs. *Visual Languages - Human Centric Computing* 0:51–55, 2009.

[RDV10] J. Rivera, F. Duran, A. Vallecillo. On the Behavioral Semantics of Real-Time Domain Specific Visual Languages. In lveczky (ed.), *Rewriting Logic and Its Applications*. LNCS 6381, pp. 174–190. Springer, 2010.

[Ren08] A. Rensink. Explicit State Model Checking for Graph Grammars. In *Concurrency, Graphs and Models*. LNCS 5065, pp. 114–132. Springer, 2008.

[THRB10] P. Torrini, R. Heckel, I. Ráth, G. Bergmann. Stochastic Graph Transformation with Regions. In *GM-VMT'10*. Electronic Communications of the EASST 29. 2010.