Topology Analysis of Car Platoons Merge with FujabaRT & TimedStoryCharts - a Case Study

Christian Heinzemann[1], Julian Suck[1], Ruben Jubeh[2], Albert Zündorf[2]

[1] University of Paderborn, Software Engineering Group,
Warburger Str. 100,
33098 Paderborn
chris227|jsuck@upb.de
[2] Kassel University, Software Engineering Research Group,
Wilhelmshöher Allee 73,
34121 Kassel, Germany
ruben|zuendorf@cs.uni-kassel.de

Abstract. This paper addresses the topology analysis case study for the Transformation Tool Contest 2010. The case study presents a car platoon merge protocol with a dynamic number of participants. The task is to compute a reachability graph for all system configurations generated by the graph rewrite rules. Using the Fujaba Real-Time Tool Suite, we modeled the merge protocol as statechart in concrete syntax as this is more intuitive and use a special generator to derive the corresponding graph rewrite rules ([6, 5]). This has been combined with the hierarchical graphs library presented in [12], to compute the reachable graph transition system.

1 Introduction

This paper addresses the topology analysis case study for the Transformation Tool Contest 2010. The case study description introduces an example of a dynamic communication structure. Dynamic communication structures occur whenever an arbitrary, at design time unknown number of participants have to coordinate each other. The provided example is a group of cars that form a platoon of arbitrary length in which they travel behind a leader car. The example is structurally similar to the dynamic convoy situations (cf. [7]) in our RailCab project[3].

Systems employing dynamic communication like the car platoon often operate in safety critical environments as it is the case for both, the car platooning example and the RailCab convoys. Therefore, verification is of crucial importance for the safety of such systems to guarantee correct functionality. One inherent problem of such systems is that they often have an infinite reachable state space which makes the verification impossible without suitable abstraction or verification approaches.

We used our Mechatronic UML (e.g. [3]) approach to model the provided car platooning case study. Mechatronic UML is an adaptation of the UML allowing to model and verify systems with a dynamic communication structure ([7]). The modeling process defined by Mechatronic UML [7] allows formal verification of safety properties for arbitrary numbers of participants using inductive invariants [4]. However, inductive invariants do not support verification of all kinds of properties that are important for such systems like e.g. deadlock freedom. Since the case study explicitly requested solutions performing a reachability analysis of the protocol specification, we will show a framework for reachability analysis on graph transformation

[3] http://www.railcab.de/

systems. This enables us to verify a larger set of properties, like e.g. deadlock freedom, for a given number of instances.

We modeled the merge protocol of the car platooning case study with a statechart in concrete syntax as this is more intuitive than a direct specification in terms of graphs rewrite rules. Then, we use a partially automated generation of statecharts into our (Timed) Story Chart formalism ([6]) that allows an integrated specification of state-based protocol behavior and dynamic graph transformation. (Timed) Story Charts consist of graph transformation rules employing the behavior of the statechart and of the original graph transformation rules describing the dynamic transformations in the system structure, e.g. adding new cars to the system. We used the computation of reachability graphs as proposed in [12] to obtain the reachable graph transition system. Our framework manages sets of graphs and provides functions for copying graphs, computing hash codes for graphs, and checking isomorphisms of graphs. The resulting rules of our (Timed) Story Charts identify all possible matches for the given rule, then copy the graph and finally apply the transformation to the copy.

The concepts of the Mechatronic UML have been integrated into the Fujaba4Eclipse Real-Time Tool Suite (Fujaba RT, [10])[4] as an extension to the Fujaba4Eclipse CASE tool. The Mechatronic UML supports the specification of real-time statecharts ([2]) that allow to model timing constraints for states and transitions. We can also handle this using our Timed Story Charts and perform a reachability analysis including the timing constraints using timed story diagrams, a dialect of timed graph transformation systems ([7]). In our opinion, such timing constraints would be needed to obtain a realistic specification of the merge protocol as it is needed for the convoy coordination in our RailCab project. As this case study did not contain real-time requirements, we did not utilize the real-time capabilities of our framework.

2 Framework

This section introduces the framework which was used to model and perform the reachability analysis. Section 2.1 describes the generation of Timed Story Charts out of statecharts. Section 2.2 introduces some improvements that were made in contrast to the version described in [12].

2.1 Generating Timed Story Charts

In [6, 5], we introduced the Timed Story Chart formalism which allows to map real-time statecharts ([2]) to story diagrams extended with time. Timed Story Charts preserve the semantics of real-time statecharts while real-time statecharts can be mapped to hierarchical timed automata ([1]) which are a proper input for the Uppaal model checker ([9]) and preserve the semantics of Uppaal timed automata. Thus, Timed Story Charts have a proper semantics defined over timed automata.

The general idea is to model protocol specifications as statecharts in concrete syntax and to transform them automatically into graph rewrite rules for dynamic numbers of participants as in the car platooning case study. For a fixed number of participants, a verification using standard model checking tools is much more efficient. The first step of the transformation is to generate an object diagram for the statechart containing an object for the statechart itself as well as one object for each state of the statechart. For multiple instances of the same statechart, e.g. the statecharts of two car processes, we exploit the fact that all these instances have the same structure by generating the state structure only once. The active state of each instance is marked by an ActiveState object pointing to the object representing that state. Transitions of the statechart

[4] http://www.fujaba.de/projects/real-time.html

are mapped to Story Diagrams such that for each transition there exists one Story Diagram which executes this transition. The execution includes changing the active state, consuming received messages, generating sent messages, and performing side effects like writing local variables of the process, e.g. for storing the leader.

Asynchronous, message based communication between different statechart instances is supported by the EventQueue objects. There exists one event queue for each car process which buffers all incoming messages for the statechart. Thus, for receiving a message, a statechart reads the head message of the queue and dequeues it afterwards. For sending a message, the message is inserted into the queue of the receiving statechart by invoking the enqueue method as shown in Figure 13.

The automated transformation can generate the whole statechart structure as well as the story diagrams executing the transitions. Currently, the generation of model dependent calls like the computation of message recipients is not possible, but we are planning to extend the transformation to these aspects. The transformation itself is implemented as a model-to-model transformation using story diagrams. An example transformation rule is shown in Figure 16. It shows the top-level rule generating a class representing the transformation rule which actually executes the transition (cf. Section 2.2).

2.2 Improvements of the HierarchicalGraphsLib

The solution presented in last year's contest for the leader election protocol [12] provided a generic hierarchical graph library and a problem specific model for the leader election protocol. The graph library used runtime reflection mechanisms to provide generic copy and checkIsomorphism operations. This turned out to be inefficient, as reflective code is several times slower than generated code. We adapted the template based Fujaba code generator [8] to generate application specific copy and isomorphism check methods in the subclasses of *de.fujaba.Node* and *de.fujaba.Graph*. To be included in the isomorphism check and caching hash calculations, all associated classes have to be derived from *de.fujaba.Node* and associations have to be marked with the *usage* stereotype.

In order to ease the use of the graphs lib, we introduced a framework executing the reachability analysis which is shown in Figure 1. The core of the framework is specified in the class ReachabilityComputation. This class maintains a list of StepGraphs that were reached during the analysis and a list of StepGraphs that have to be expanded. StepGraph is a subclass of *de.fujaba.Graph* and denotes the graphs reached during the analysis. Expanding graphs, merging isomorphic graphs, and maintaining the timing computations is implemented independent from the concrete rule set. Rules are defined as subclasses of Rule. The class TransformationRule is the super class for all rules executing transitions. As we have no timing constraints in the example, the additional rules for the timing are omitted here. For specifying a reachability analysis, the user only has to specify the rule set, an initial graph, and has to instantiate the rules for the reachability analysis.

3 Modeling the merge protocol

3.1 At design time: Modeling the protocol

The underlying structure of the case study is defined by the class diagram shown in Figure 2. We introduced the class CarProcess representing the car processes. This class has four associations to itself representing the channels *leader*, *follower*, *aux*, and *bldr*. We used four unidirectional transitions in our model to imitate the behavior described in the case study. Our model would also allow to use bidirectional associations. This would enable us to express the two channels leader and follower by one 1:n association.

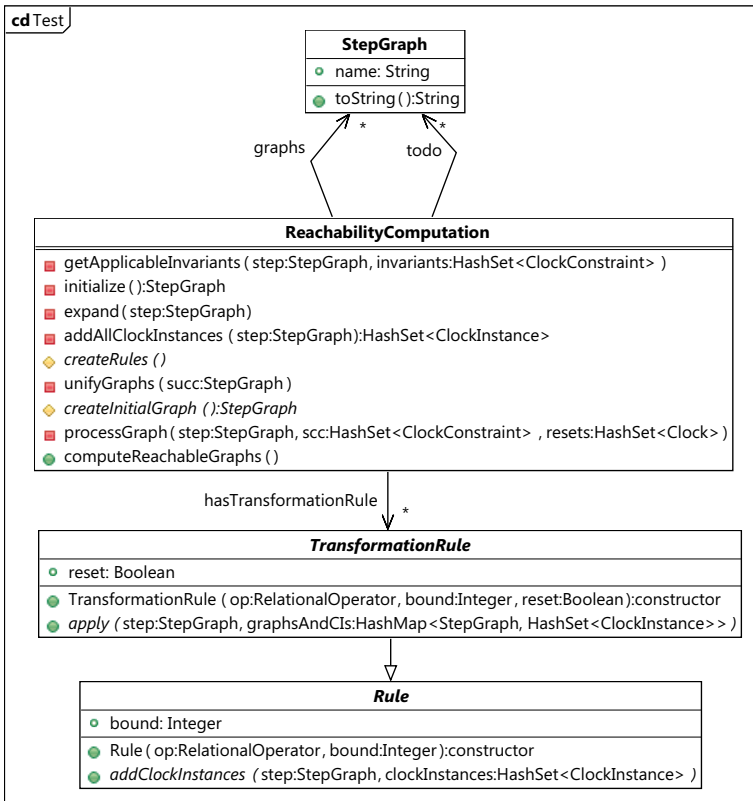

Fig. 1. Framework for the reachability analysis

The class Environment describes the environment in which the cars operate. The environment can generate new cars using the CreateNewCarProcess rule as well as it generates the car ahead (ca) messages for the cars such that they join a platoon or such that two platoons merge. There exist two additional rules for that purpose. The application of the CreateNewCarProcess rule is limited by a constant providing the desired number of cars.

The classes State, ActiveState, Parameter, EventQueue, and Statechart classes from the Timed Story Chart model described in Section 2.1 are used for the mapping of the process statechart. The Carprocess is a special parameter which can be used to attach a car process to an event. This is needed for the *ca* and *req* messages as they contain process identities. The class Car_Car_Port1 represents the statechart of the car process shown in Figure 3.

The statechart itself is directly adopted from the protocol specification contained in the case study description and uses the same states and transitions. The concrete syntax of the transitions is adopted to the syntax of our real-time statecharts. Received messages, called trigger events, are depicted in front of a "/", sent messages, called raised events, are depicted after the "/". Transition guards restrict the execution of the transition, e.g. the transition from hob to hod should only be executed if the former leader does not have any followers left. This is expressed by the guard in square brackets attached to this transition.

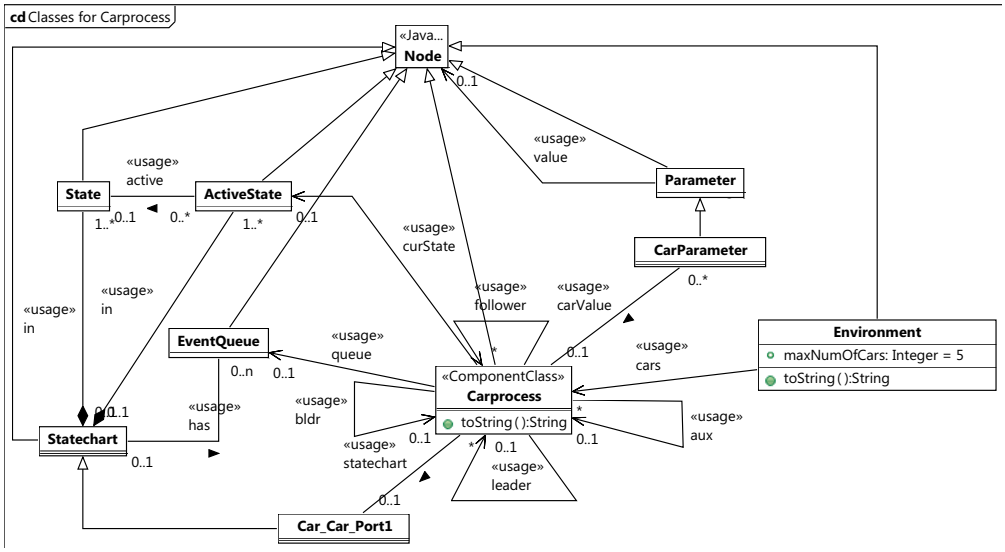

Fig. 2. Class diagram for the merge protocol specification

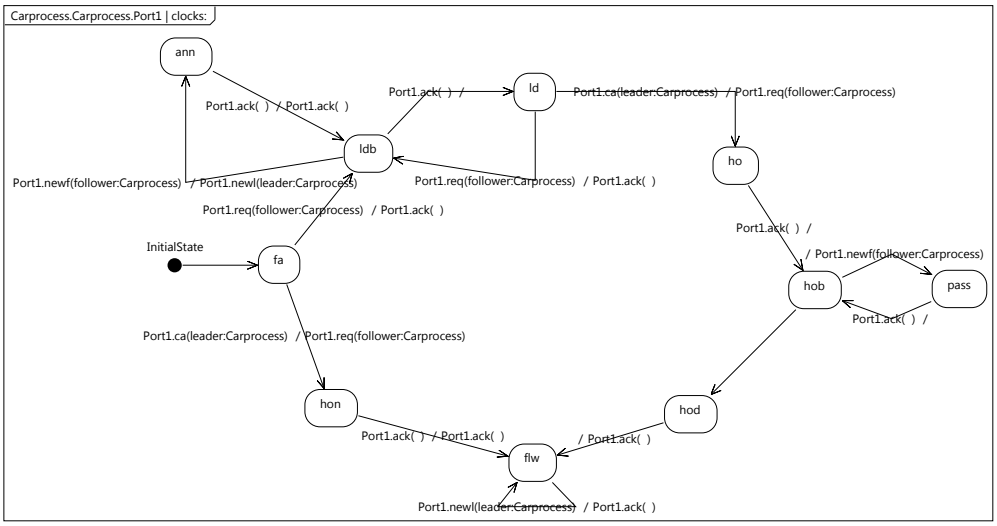

Fig. 3. Statechart specifying the protocol

3.2 At Runtime: computing the Reachability Graph

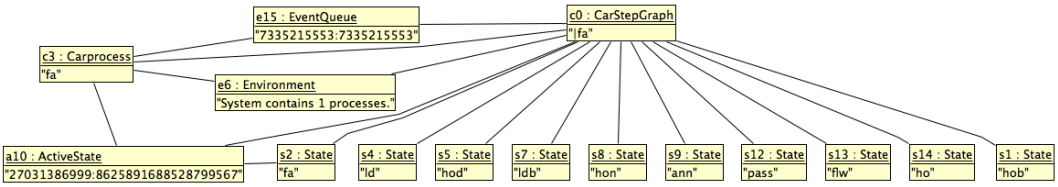

Fig. 4. Initial Graph

The runtime structure of our program is visualized by the eDOBS tool ([11]), a graphical object browser attached to the debug interface. It shows the objects in the heap, their attributes and links to other objects and helps visualizing the runtime state of our program. Figure 4 shows the initial graph we start with. There is a single car process with active state pointing to the state *fa*, an event queue, an environment and objects for each possible state a car process can be in.

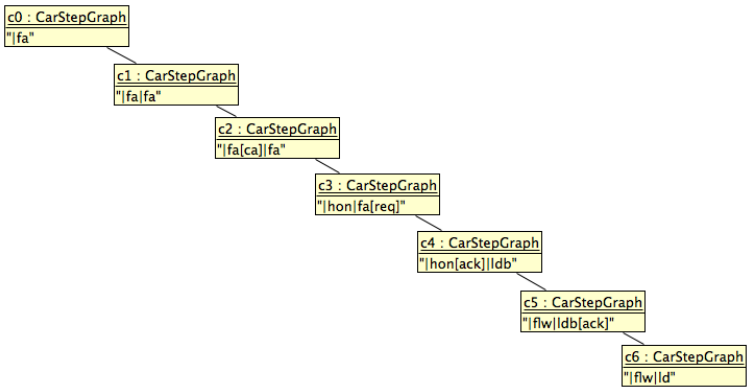

Fig. 5. Simplest reachability graph

The core of our algorithm is the `ReachabilityComputation` class. It starts with the initial graph and iteratively expands the graph by copying the graph and applying rules. After that, `unifyGraphs` (c.f. Figure 6) tries to identify isomorphic `CarStepGraphs`. Each step graph contains a certain state of the whole car process model. Figure 5 shows the structure of a reachability analysis with maximum two car processes. Each analysis state is represented as a `CarStepGraph` instance. These are ordered by an predecessor/successor-relation. In that example, each car step graph has only a single successor, as isomorphic graphs are removed after each expand step. The analysis ends with one process being the follower and the other one being the leader, which is isomorphic to *ld*|*flw*.

Figure 7 shows the full reachability graph for an analysis with max processes = 3. Isomorphic states are already removed, but the unify step also marks step graphs with same predecessor in the reachability graph.

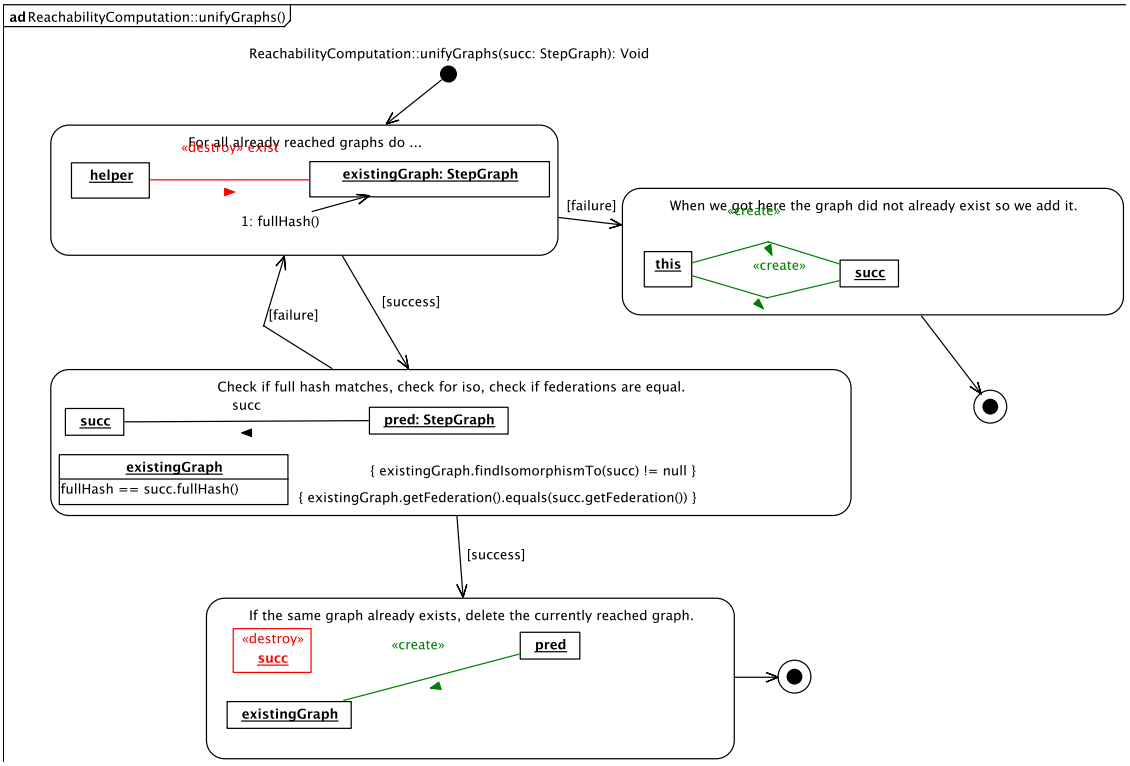

Fig. 6. unifyGraphs() rewrite rule

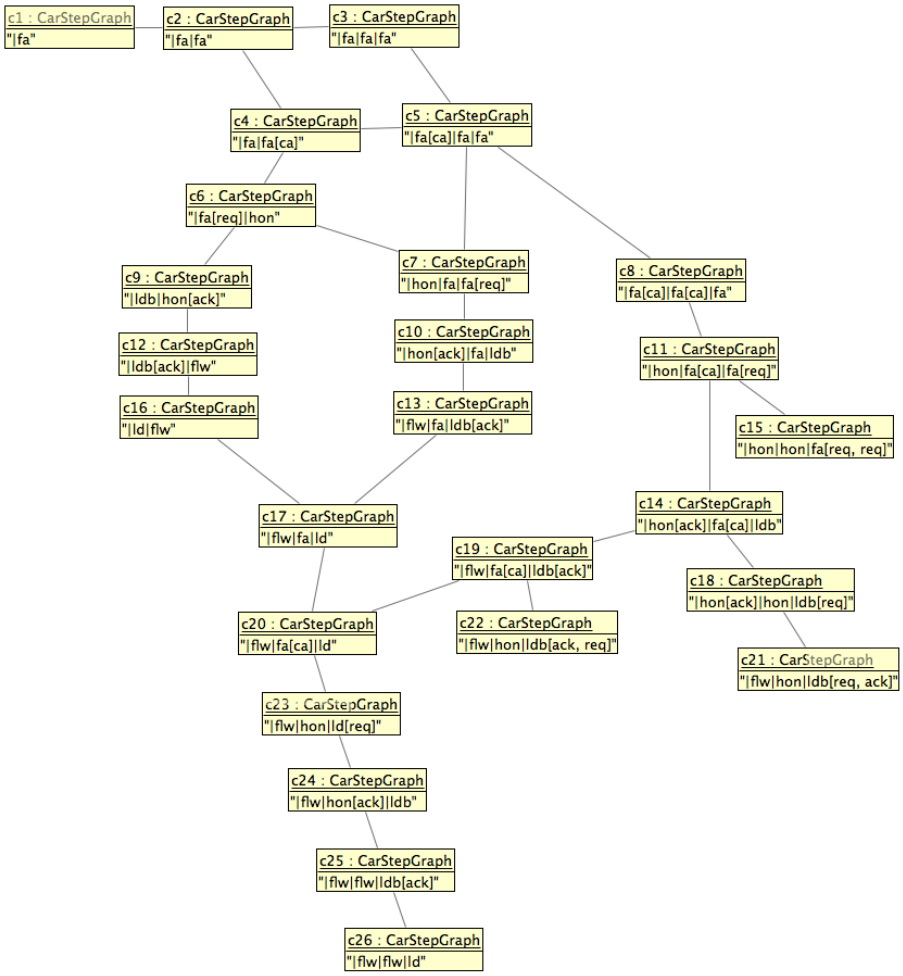

Fig. 7. Reachability graph for max processes = 3

For example, adding a new car process to the graph `c4` results in the same successor step graph as applying the rule `StartPlatoon`, which generates a *ca* environment message to `c3`, that is is the step graph `c5`.

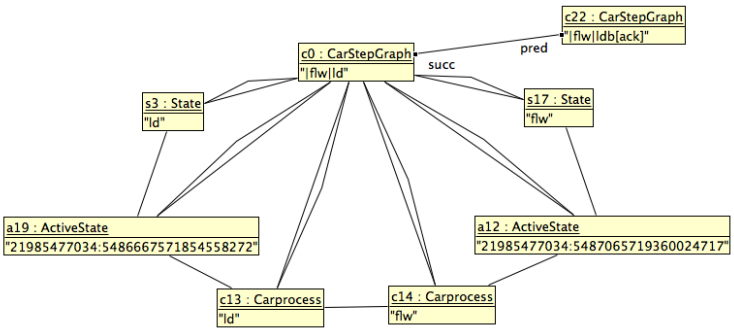

Fig. 8. Final graph for max processes = 2

The eDOBS tool can be used to verify the generated topologies. Back to the simple example with just two car processes, we inspect the final step graph, shown in figure 8. The states *ld* and *flw* associated with the car processes are correct. There is a bidirectional link between `c13` and `c14`, representing the ldr and flws edges as expected. Also, note that the previous step graph, `c22`, is the one with a process in *ldb* and an *ack* message in it's queue.

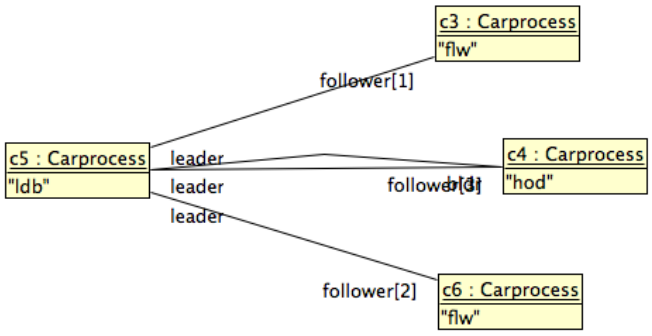

Fig. 9. Platoon Merge

More complex examples can be inspected by setting breakpoints in the corresponding rule and inspecting the step graph in eDOBS. Figure 9 shows the situation just before a platoon merge: one car process is currently in the *hod* state, just the edges between the car processes are shown. By looking at the predecessors of the current step graph, we can see previous states of the whole system.

3.3 Modeling the constraints

The given constraints to be checked during analysis are modeled as rules having a LHS only. They are derived from the `Rule` class, thus they are added to the chain of transformation rules to be checked on each iteration. They simply print out any violation. Figure 10 shows the graph pattern match rule for checking that no two processes in state *flw* are connected to each other with an *ldr* edge. Similar, figure 14 and 15 in the appendix show the two remaining constraints.

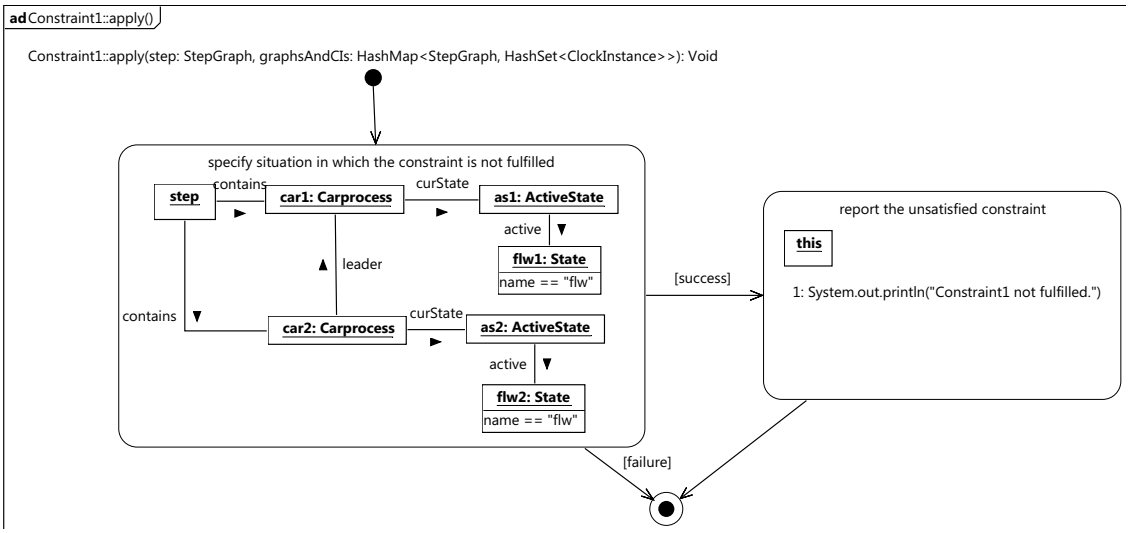

Fig. 10. Graph pattern match rule for constraint 1

Checking for deadlocks can easily performed by a post-processing searching for graphs having no successor and the reached graph transition system.

4 Evaluation

Our system can analyze topologies with up to eight car processes. Runtime measurements are presented in Table 1. Each line represents a full analysis with the maximum number of car processes limited to `max processes`. Column two shows the number of non-isomorphic graphs resulting for that analysis run and

column three shows the time needed for the computation. Column four denotes the number of nodes per graph while Column five shows the memory consumption of the computation.

max processes	graphs	Runtime (s)	graphsize	memory
2	7	0	21	?
3	26	0	26	?
4	96	1	31	7MB
5	348	2	36	42MB
6	1317	12	41	163MB
7	5100	347	46	733MB
8	20353	14046	51	5.8GB

Table 1. Evaluation results

Figure 11 shows the nodes per graph. This grows linear as expected. In contrast to that, our analysis runtime grows exponentially, as shown in figure 12. Analyzing the system with up to eight processes already requires almost 4 hours runtime, so calculating nine seems to be impossible with the current approach, as well as with the current memory requirements. The graph isomorphism check still seems to be inefficient when many isomorphisms are expected.

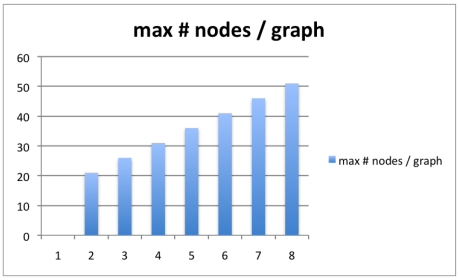

Fig. 11. Number of nodes per graph

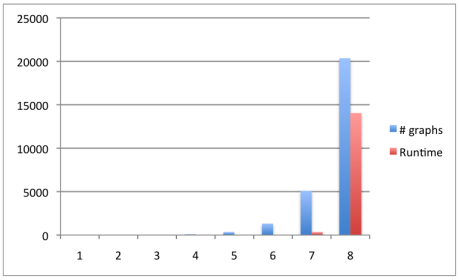

Fig. 12. Problem size versus runtime (in seconds)

5 Conclusions

We have shown an approach to model the topology analysis case study by using concrete statechart syntax. The statecharts are automatically transformed into Story Diagrams and executed by a framework computing the set of reachable graphs. Our evaluation results have shown, that checking more than 8 car processes is currently not possible using our tool regarding time and memory consumption for the computations with 8 cars. Although not requested in this case study, our framework is able to handle timing constraints which would lead to a more realistic specification of the merge protocol.

In FujabaRT, we are now able to use a combination of real-time statecharts and real-time transformation rules to specify the behavior of a system. Using our generators and the reachability graph framework, we

are able to compute the reachability graph for the specified system. In addition, one may add constraint checking rules to the reachability graph framework.

Currently, we use normal rules to specify our constraints that check for the existence or absence of certain invariant structures. This enables us to check CTL formulas of the kind $EF\varphi$ and $\neg AG\varphi$ for some graph invariant φ. We plan to extend our checking approach to a greater class of CTL formulas in future work. It is also possible to extend our approach by using inductive invariants to verify properties for infinite state systems ([4]).

The performance of our reachability analysis has been improved since last year. However, our framework introduces a certain overhead due to the unused timing capabilities which require some additional search operations in the graph. Additionally, our current Timed Story Chart approach uses one object for each state of the statechart which increases the number of objects per graph by a constant and we plan to investigate whether a more compact statechart encoding yields better results. Finally, this case study clearly identified the isomorphism checks of our graph lib framework as a bottleneck and we have developed a number of ideas for further improvements for the next year.

References

1. A. David, M. O. Möller, and W. Yi. Formal Verification of UML Statecharts with Real-Time Extensions. In *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 208–241. Springer Berlin / Heidelberg, 2002.
2. H. Giese and S. Burmester. Real-time statechart semantics. Technical Report tr-ri-03-239, Lehrstuhl für Softwaretechnik, Universität Paderborn, Paderborn, Germany, June 2003.
3. 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.
4. H. Giese and D. Schilling. Towards the Automatic Verification of Inductive Invariants for Infinite State UML Models. Technical Report tr-ri-04-252, University of Paderborn, Paderborn, Germany, December 2004.
5. C. Heinzemann, S. Henkler, and M. Hirsch. Refinement checking of self-adaptive embedded component architectures. Technical Report tr-ri-10-313, University of Paderborn, 2010.
6. 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.
7. S. Henkler, M. Hirsch, C. Priesterjahn, and W. Schäfer. Modeling and verifying dynamic communication structures based on graph transformations. In *Proc. of the Software Engineering 2010 Conference, Paderborn, Germany, 22.-26.2.2010*, 2010. accepted.
8. L. Geiger, C. Schneider, C. Record. Template- and modelbased code generation for MDA-Tools. *3rd International Fujaba Days 2005, Paderborn, Germany*, September 2005.
9. K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, Oct. 1997.
10. C. Priesterjahn, M. Tichy, S. Henkler, M. Hirsch, and W. Schäfer. Fujaba4eclipse real-time tool suite. In *Model-Based Engineering of Embedded Real-Time Systems (MBEERTS)*, LNCS, pages 1–7. Springer, 2009. accepted.
11. The EDobs Dynamic Object Browser. http://www.se.eecs.uni-kassel.de/typo3/index.php?edobs, 2006.
12. A. Zündorf. Model Checking the Leader Election Protocol with Fujaba. In *GraBaTs 2009, 5th International Workshop on Graph-Based Tools*, Zurich, Switzerland, 2009.

A Appendix

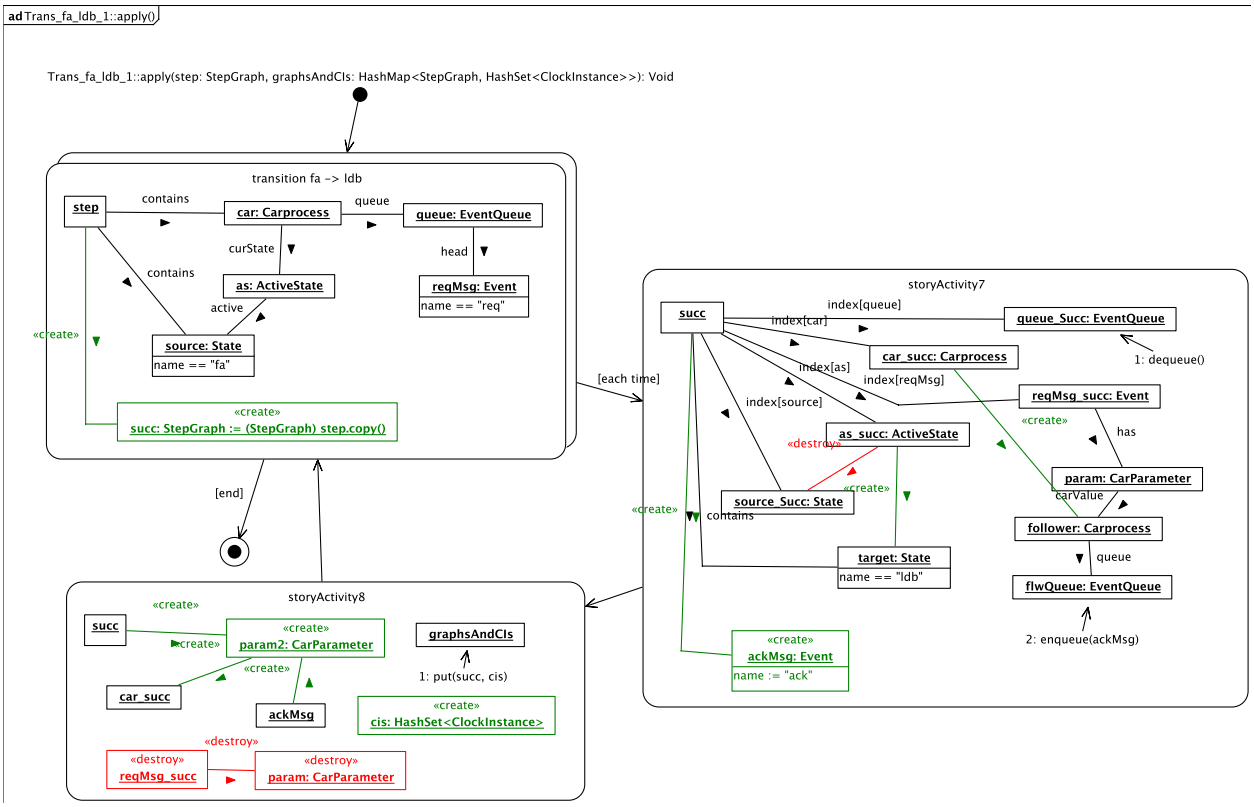

Fig. 13. Example transformation rule rule: fa to ldb

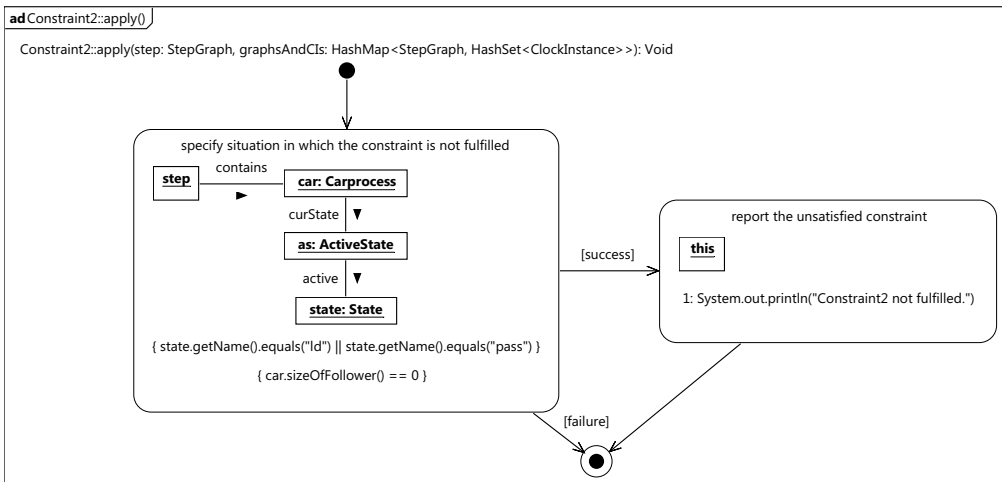

Fig. 14. Graph pattern match rule for constraint 2

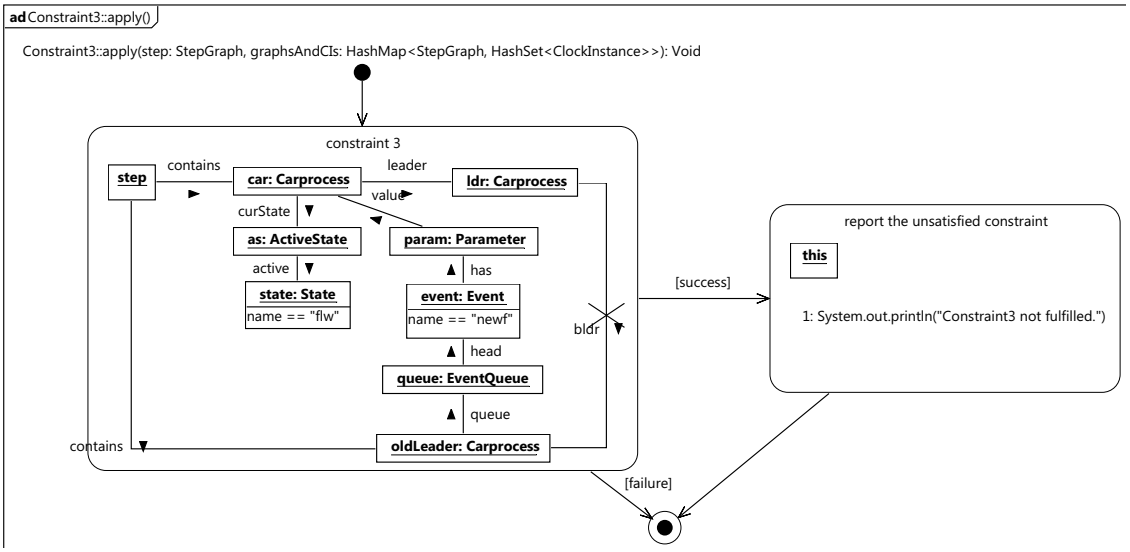

Fig. 15. Graph pattern match rule for constraint 3

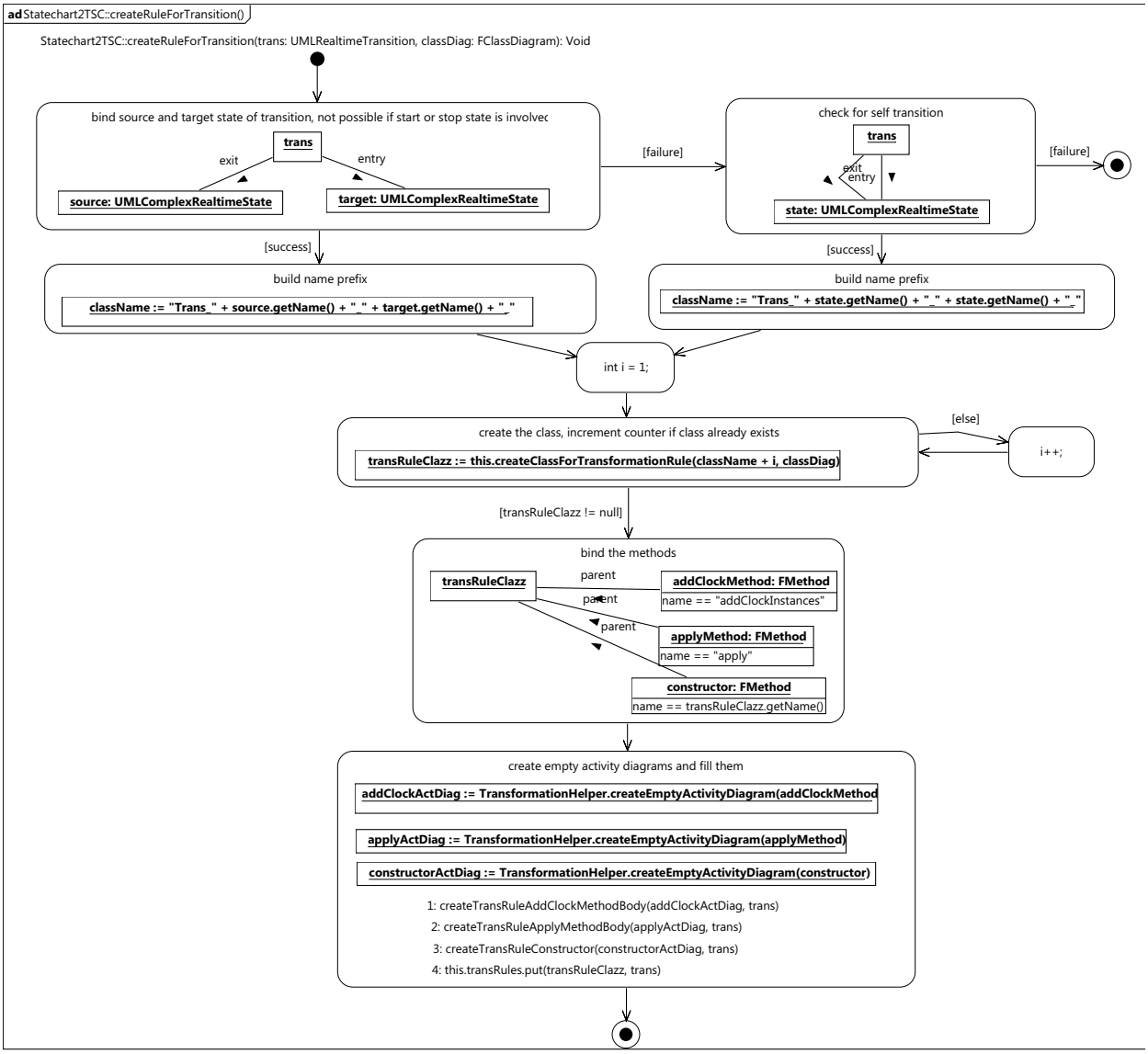

Fig. 16. Top-level Story Diagram translating a transition into a Story Diagram rule