Instance-Aware Model Checking of Graph Transformation Systems using Henshin and mCRL2

Christian Krause Stefan Neumann

Hasso Plattner Institute for Software Systems Engineering, Prof.-Dr.-Helmert-Str. 2-3, D-14482 Potsdam, Germany
{christian.krause|stefan.neumann}@hpi.uni-potsdam.de

Network topologies in distributed and mobile systems can be naturally described using graph-based models. Specifying configurations of such systems is realized by assigning nodes modeling entities in the network to logical locations in the graph. The operational semantics of such models can be formally described using graph transformation systems by modeling the interactive behavior of the entities in the network with rewrite rules. As an example, we model here the movement of autonomous railway shuttles along tracks and the creation of temporary convoys between shuttles.

As a standard approach in formal methods, model checking facilitates the analysis of the dynamics modeled by graph transformation systems. However, in many settings such as in mobile and distributed systems, the model checking yields useful results only if it allows to reason about individual entities, e.g. specific railway shuttles, and their interactive behavior during the execution. Since the existing tools lack support for reasoning about individual entities, we introduce an approach and a toolchain for instance-aware model checking of graph transformation systems. We use HENSHIN as a modeling language and state space generation tool and MCRL2 as model checking back-end.

1 Introduction

Graph transformation systems constitute a formal modeling approach for systems with structural state models and operational semantics that is defined in terms of rule-based rewriting of these models. Due to the graph-based nature of the state models, graph transformation is particularly well-suited to specify applications where the network topologies must be modeled explicitly, such as in distributed, mobile and reconfigurable systems (see, e.g., [8]). As a running example in this paper, we consider the RailCab system [12] in which small, intelligent, autonomous shuttles provide on-demand passenger transfers. In our modeling, the shuttles operate on an existing and fixed railway network and share the tracks with other vehicles, e.g. trains. The dynamics of the system is defined using graph transformation rules that model the movement of vehicles on the tracks and the forming of convoys between shuttles.

As a standard approach in formal verification methods, finite graph transformation systems can be analyzed by constructing an explicit state space and applying model checking. However, the standard approach of using atomic propositions or simple transition labels for the model checking of graph transformation systems is rather unsatisfactory, since it does not provide a means to refer to entities in the structural state models. In this paper, we therefore describe an approach and a toolchain for *instance-aware* model checking of graph transformation systems, which allows to refer to specific nodes in the graph models and to use quantification over node types in the specification of formulas. As technical contributions of this paper, we describe (1) our work on the state space generation tools for the graph transformation-based modeling language and toolset HENSHIN [1], and (2) an adapter and user front-end which allows to automatically perform instance-aware model checking using the MCRL2 [7] toolsuite.

Unpublished document. Cite as: http://www.eclipse.org/modeling/emft/henshin/documents/henshin_mcrl2.pdf

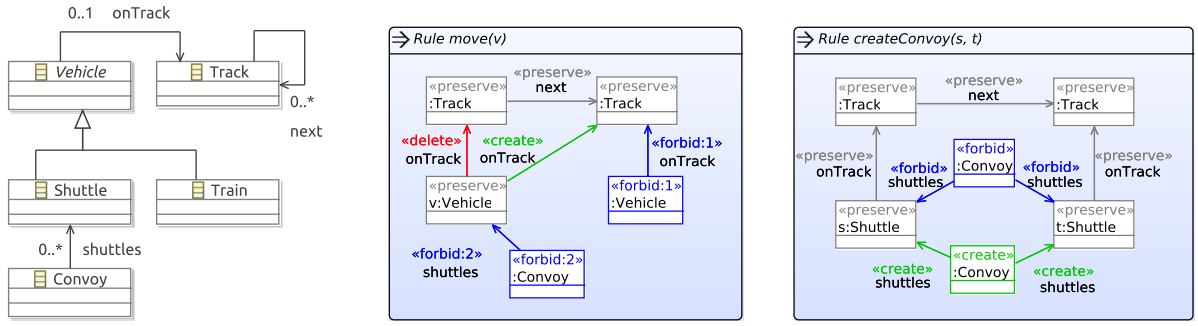

Figure 1: Type graph (left) and transformation rules (mid and right) for the RailCab system in HENSHIN

2 Modeling with HENSHIN

HENSHIN [1, 9] is a modeling language and toolset that implements the double-pushout approach [3] for attributed typed graph transformation, based on structural data model in the Eclipse Modeling Framework [4]. Due to lack of space, we describe the relevant concepts only informally here. A *graph transformation system* (GTS) consists of (1) an (attributed) type graph, (2) a set of parameterized[1] transformation rules, and (3) a graph representing the initial state. Figure 1 depicts the type graph and two parameterized transformation rules for modeling the RailCab system [12] in HENSHIN.

(1) Type graph We distinguish five different node types: Track, *Vehicle*, Shuttle, Train and Convoy. *Vehicle* denotes an abstract node type and thus cannot be instantiated. HENSHIN supports node type inheritance which we use here to define Shuttle and Train as concrete realizations of *Vehicle*. The edge type next:Track→Track is used to model the topology of the railway system. The edge type onTrack:*Vehicle*→Track is used to specify the location of vehicles and shuttles:Convoy→Shuttle models the existence of convoys between shuttles.

(2) Parameterized transformation rules The parameterized transformation rules *move(v:Vehicle)* and *createConvoy(s:Shuttle,t:Shuttle)* in Figure 1 are used here to model the operational semantics of the RailCab system. In the syntax of HENSHIN, rewrite patterns are denoted using the stereotypes ⟨⟨preserve⟩⟩, ⟨⟨create⟩⟩, ⟨⟨delete⟩⟩ and ⟨⟨forbid⟩⟩, respectively for matching and preserving, creating, deleting, and forbidding the existence of nodes or edges of a given type. The rule *move(v:Vehicle)* moves the shuttle or train given by the parameter *v* from its current track to one of its successor tracks. This is possible only if (1) no other vehicle is on the next track, and (2) the vehicle is currently not part of a convoy. The second constraint is added because modeling the movement of convoys is more involved and is for simplicity reasons not modeled here. The rule *createConvoy(s:Shuttle,t:Shuttle)* creates a convoy node between two shuttles which are located at adjacent tracks, if there is no convoy existing yet between them. We also introduce the rule *deleteConvoy(s:Shuttle,t:Shuttle)* which removes a convoy node again (not shown here).

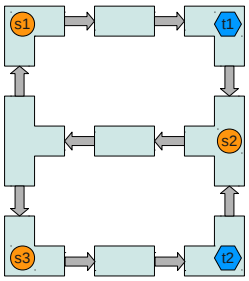

Figure 2: Initial state

(3) Initial state As initial state we use the typed traph which is schematically depicted in Figure 2. It consists of 9 interconnected tracks, 3 shuttles (depicted as circles) and 2 trains (depicted as hexagons) distributed on the tracks.

[1] For a formal definition of graph transformation systems with parameterized rules see, e.g., [8].

3 State space generation and visualization with HENSHIN

The HENSHIN toolset contains an interpreter engine implementing graph matching algorithms based on constraint solving for executing transformations. In addition, a number of tools for state space generation, visualization, and verification using invariant checking and model checking are included.

State space generation State spaces are generated as labeled transition systems (LTSs) where the states are graphs. Two different modes are supported: without and with nodes IDs. In the mode without node IDs, transition labels consist only of rule names, without parameters. In this mode, states are compared using mere graph isomorphy checking and thus, different instances of the same node type cannot be distinguished. In the mode with node IDs, all nodes whose type occurs as a parameter type in a rule, get a unique ID, e.g. all nodes of type Train and Shuttle in the RailCab example. These node IDs are then used as identifiers in the parameters of transition labels. The mode with node IDs yields significantly larger state spaces. However, it is required to distinguish between different nodes of the same type and to track their history, which is a key ingredient for the instance-aware model checking as we argue in Section 4. The sizes of the generated LTSs for the RailCab example without and with node IDs are shown in Table 1 together with the time required for generating the complete state spaces.[2] HENSHIN also supports interactive visualization and exploration of small state spaces (for $\sim 10^4$ states).

IDs	States	Trans.	Time
no	1,274	4,297	1s
yes	15,030	51,384	20s

Table 1: RailCab LTS details

Optimizations The state space generation tool is based on a memory-efficient state space model and makes extensive use of caching. On multi-core processors, the state space generation is performed using multiple threads. The state space generation is implemented as a hybrid approach of depth-first and breadth-first search.[3] Indexing of states is crucial to avoid unnecessary isomorphy checks and is implemented using a graph hash function. For the graph isomorphy checking, the rule match finding engine of the HENSHIN interpreter is used. Node IDs are generated only for those nodes whose type is used as a parameter in a rule. Thereby, symmetries in the underlying system topology can be used to reduce the size of the state space. For instance, in the RailCab example the topology of the track network can be mirrored vertically (see Figure 2), which reduces the size of the state space by a factor of 2.

4 Instance-aware model checking with MCRL2

MCRL2 [7] is a behavioral specification language and associated toolset with support for model checking based on the modal μ-calculus extended with regular expressions, data and time [6]. MCRL2 can indirectly model check explicit state spaces as generated by HENSHIN using the import tool `lts2lps` which converts a given LTS into MCRL2's own format (so-called *linear process specifications*).

The HENSHIN state space tools include an adapter and a front-end for instance-aware model checking of generated state spaces using MCRL2. State spaces are first converted into an LTS format readable by MCRL2. In the mode with node IDs, the adapter extracts all used node IDs from the state space first and generates a data type definition in the format of MCRL2. For the RailCab example, the generated data type specification is shown in Listing 1. As data types in MCRL2 cannot make use of inheritance, the adapter identifies Vehicle as common supertype of Shuttle and Train. The adapter found five node IDs

[2] The tests were conducted on a 64-bit Intel(R) Xeon(R) CPU with 4×2.50GHz and 8GB memory running Linux 2.6.26.

[3] Depth-first search can make better use of caching, but cannot be parallelized. Therefore, the tool unfolds a fixed number of new states at once using multiple threads. Thus, the tool implements a hybrid approach of depth-first and breadth-first search.

Listing 1: Generated MCRL2 data type specification for the RailCab example

```
1 sort Vehicle = struct s1 | s2 | s3 | t4 | t5;
2 act move : Vehicle;  createConvoy : Vehicle#Vehicle; deleteConvoy : Vehicle#Vehicle;
3 var x : Vehicle;
4 map isShuttle : Vehicle -> Bool;    eqn isShuttle(x) = (x==s1) || (x==s2) || (x==s3);
5 map isTrain : Vehicle -> Bool;      eqn isTrain(x) = (x==t4) || (x==t5);
```

Listing 2: Example property *a)*

```
1 forall s:Vehicle.val(isShuttle(s)) =>
2  ([true*] (exists t:Vehicle.val(isShuttle(t)) && (<true*.createConvoy(s,t)> true)))
```

for this type and generates the data type `Vehicle` accordingly as a sum over five different identifiers, modeling the instances. These identifiers are also used in the parameters of the transitions and can be therefore used for the model checking (see also the generated actions and their parameter types in line 2). The adapter also generates the helper predicates `isShuttle` and `isTrain` (see line 4–5) which can be used to check for the real data type of instances in formulas.

Using the generated LTS and data type definition in Listing 1, we can now directly verify μ-calculus formulas with data using MCRL2. For our running example we check the following two properties:

a) For every shuttle it holds in every state that it can eventually form a convoy with another shuttle. In MCRL2-syntax, the property can be written as shown in Listing 2. This property is satisfied.

b) For every shuttle it holds in every state that it can eventually form a convoy with another shuttle only by moving itself (meaning that no other vehicle stands in the way). We can adapt the formula in Listing 2 by replacing `<true*.createConvoy(s,t)>` with `<move(s)*.createConvoy(s,t)>`. This property is not satisfied, meaning that it is possible that a train blocks a shuttle.

Note that this approach allows us to use existential and forall-quantification over node types in the original graph transition system and to track transitions involving bound node variables.

5 Other model checkers supported by HENSHIN

OCL invariants The Object Constraint Language (OCL) [11] can be used in HENSHIN to specify and check structural state invariants. For small state spaces, found counterexamples are highlighted in the state space explorer and can be inspected in more detail.

CADP Besides MCRL2, model checking of modal μ-calculus formulas for explicit state spaces, e.g., given as LTSs, is also supported by the the CADP toolsuite [5]. The HENSHIN state space tools include an adapter which translates state spaces generated from graph transformation systems into the input format of CADP, and invokes the `evaluator` model checking tool. In contrast to MCRL2, CADP can output counterexamples which are automatically extracted and –if possible– shown in the graphical state space explorer. However, CADP can currently not be used for instance-aware model checking.

PRISM Based on the approach of stochastic graph transformation [8], HENSHIN rules can be annotated with stochastic delays (formally, rates of exponential distributions). The rates together with the generated state space give rise to a Continuous Time Markov Chain. The HENSHIN state space tools can generate such a stochastic model in the input format of the PRISM model checker. Using a front-end in the state space explorer, the steady state probabilities can be computed, formulas specified in Continuous Stochastic Logic (CSL) can be verified, and plots can be generated using PRISM.

6 Related approaches and tools

In the MOMENT2 [2] tool, verification of graph transformation systems is supported using OCL invariant checking and LTL model checking. To distinguish between different instances of the same type in an LTL formula, node identities must be encoded using ID-attributes which must be defined already in the initial state. Thus, only nodes which are already included in the start graph can be distinguished in the model checking. To the best of our knowledge, existential and forall-quantification over node types are also not supported. Another tool for LTL and CTL model checking of graph transformation systems is GROOVE [10]. Model checking in GROOVE is based on atomic propositions which are defined using graph patterns. An atomic proposition is fulfilled for a given state graph iff the pattern can be matched. Therefore, it is not possible to reason about specific instances or to quantify over node types.

7 Conclusions and future work

We have introduced an approach and a toolchain for instance-aware model checking of graph tranformation systems using HENSHIN and MCRL2, which increases the expressiveness of model checking for graph transformation systems by allowing to refer to specific instances and to quantify over node types. As future work, we plan to use CADP as model checking back-end (since it supports counterexample extraction), and to investigate ways to accommodate for the increased state space sizes.

References

[1] T. Arendt, E. Bierman, S. Jurack, C. Krause & G. Taentzer (2010): HENSHIN*: Advanced Concepts and Tools for In-Place EMF Model Transformations*. In: *Proc. MoDELS'10*, LNCS 6394, Springer, pp. 121–135, doi:10.1007/978-3-642-16145-2_9.

[2] A. Boronat, R. Heckel & J. Meseguer (2009): *Rewriting Logic Semantics and Verification of Model Transformations*. In: *Proc. FASE'09*, LNCS 5503, Springer, pp. 18–33, doi:10.1007/978-3-642-00593-0_2.

[3] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel & M. Löwe (1997): *Handbook of Graph Grammars and Computing by Graph Transformation*, chapter Algebraic approaches to graph transformation I: Basic concepts and double pushout approach, pp. 163–245. World Scientific.

[4] EMF: *The Eclipse Modeling Framework*. `http://www.eclipse.org/emf`.

[5] H. Garavel, F. Lang, R. Mateescu & W. Serwe (2011): *CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes*. In: *Proc. TACAS'11*, LNCS 6605, Springer, pp. 372–387, doi:10.1007/978-3-642-19835-9_33.

[6] J. F. Groote & R. Mateescu (1999): *Verification of Temporal Properties of Processes in a Setting with Data*. In: *Proc. AMAST'98*, LNCS 1548, Springer, pp. 74–90.

[7] J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko & M.J. van Weerdenburg (2007): *The Formal Specification Language* MCRL2. In: *Methods for Modelling Software Systems*, IBFI, Schloss Dagstuhl.

[8] R. Heckel, G. Lajios & S. Menge (2006): *Stochastic Graph Transformation Systems*. *Fundamenta Informaticae* 74(1), pp. 63–84.

[9] HENSHIN project homepage: `http://www.eclipse.org/modeling/emft/henshin`.

[10] H. Kastenberg & A. Rensink (2006): *Model Checking Dynamic States in* GROOVE. In: *Proc. SPIN'06*, LNCS 3925, Springer, pp. 299–305, doi:10.1007/11691617_19.

[11] OCL: *Object Constraint Language*. `http://www.omg.org/technology/documents/formal/ocl.htm`.

[12] *RailCab project homepage*. `http://www.railcab.de/`.