# A Simulator for Probabilistic Timed Graph Transformation Systems with Complex Large-Scale Topologies

Christian Zöllner(✉), Matthias Barkowsky, Maria Maximova, Melanie Schneider, and Holger Giese

Hasso Plattner Institute at the University of Potsdam, Potsdam, Germany
{christian.zoellner,matthias.barkowsky,maria.maximova,melanie.schneider, holger.giese}@hpi.de

**Abstract.** Future cyber-physical systems, like networks of autonomous vehicles, will result in a huge number of collaborating systems acting together on large-scale topologies. Modeling them requires capturing timed and probabilistic behavior as well as structure dynamics. In [9], we introduced Probabilistic Timed Graph Transformation Systems (PTGTSs) as a means of modeling a high-level view of these systems of systems and provided model checking support. However, given the scale of emerging systems of systems and their often complex topologies, analyzing only small or medium size models using model checking is insufficient. To close this gap, we developed a simulator for PTGTSs that can import real-world topologies, automatically detect violations of state properties, and handle the graph pattern matching as well as time and probabilities efficiently so that complex large-scale topologies can be considered.

## 1 Introduction

In future large-scale cyber-physical systems, such as networks of autonomous vehicles, the interconnection of the autonomous systems via complex software and networking will result in massive systems of systems where a huge number of systems collaborate and act together on complex large-scale topologies.

Since these systems of systems are often real-time critical and exhibit probabilistic phenomena like failures, modeling them requires capturing timed and probabilistic behavior. In addition, structure dynamics needs to be taken into account since the interconnections between autonomous subsystems may change at runtime. Finally, given the scale of emerging systems of systems and their complex topologies, the modeling must also allow for capturing the complex large-scale topologies in which these systems will operate.

In [9], we introduced Probabilistic Timed Graph Transformation Systems (PTGTSs) as a means for modeling a high-level view of these systems of systems

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 241885098.


F. Gadducci and T. Kehrer (Eds.): ICGT 2020, LNCS 12150, pp. 325–334, 2020.
https://doi.org/10.1007/978-3-030-51372-6_20

and provided model checking support. However, with model checking, only small or medium size models could be analyzed, which is insufficient since the small models will (1) likely not exhibit all characteristics of complex topologies that can lead to failures and (2) likely will not allow to study emergent phenomena and failures that result from the interaction of many autonomous systems.

To close this gap and to enable the analysis of large-scale systems of systems, we developed a simulator for PTGTSs that can import complex real-world topologies, can automatically detect violations of state properties, and handles the graph pattern matching as well as the concepts of time and probabilities efficiently. The simulator maps the application of rules of a PTGTS to the probabilistic application of graph transformation (GT) rules and a dedicated time management. Scalability is achieved by exploiting the local nature of changes and by managing time in a way that avoids global updates.

Employing graph transformation systems (GTSs) and incremental graph pattern matching techniques for the simulation of complex systems has been proposed in [13]. A link between GTSs and discrete event simulation has been considered in [14]. Also, an extension of GTSs with stochastic behavior and related simulators like GraSS [15] and SimSG [4] have been developed. However, to the best of our knowledge, no simulator for GTS variants that support timed and probabilistic behavior (like PTGTSs [9]) has been presented so far.

This tool paper is structured as follows. The preliminaries, such as a running example and the PTGTS formalism, are introduced in Sect. 2. The simulator's concept is outlined in Sect. 3. An evaluation in Sect. 4 shows that the tool can import complex real-world topologies, can automatically detect violations of state properties, and can handle graph pattern matching as well as time and probabilities so efficiently that complex large-scale topologies can be considered. The paper is closed with a conclusion and an outlook on future work in Sect. 5.

## 2 Preliminaries

In this section, we introduce our running example, briefly recall the framework of GTSs, and recap the formalism of PTGTSs. As a running example, we model a scenario inspired by the RailCab project [12] where autonomous shuttles on a track topology form a system of systems.

In PTGTSs, we use the formalism of typed graphs [5] to describe the states of the systems and their structure. A *graph* $G = (G_V, G_E, s_G, t_G)$ is given by a set $G_V$ of nodes, a set $G_E$ of edges, and source and target functions $s_G, t_G : G_E \to G_V$. Let $G = (G_V, G_E, s_G, t_G)$ and $H = (H_V, H_E, s_H, t_H)$ be two graphs, then a *graph morphism* $f : G \to H$ is defined as a pair of mappings $f_V : G_V \to H_V$, $f_E : G_E \to H_E$ that are compatible with the source and target functions, i.e., $f_V \circ s_G = s_H \circ f_E$ and $f_V \circ t_G = t_H \circ f_E$.

Let $TG$ be a distinguished graph, called a *type graph*. Then a *typed graph* $(G, type)$ consists of a graph $G$ and a graph morphism $type : G \to TG$. For two given typed graphs $G'_1 = (G_1, type_1)$ and $G'_2 = (G_2, type_2)$, a *typed graph morphism* $f : G'_1 \to G'_2$ is a graph morphism $f : G_1 \to G_2$ that is compatible with the typing functions, i.e., $type_2 \circ f = type_1$.

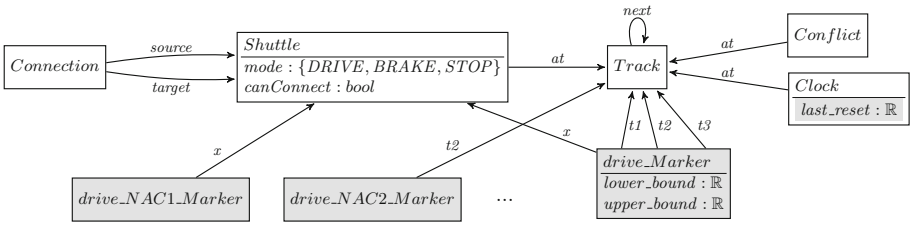



**Fig. 1.** Shuttle scenario type graph and generated extensions (grey, see Subsect. 3.2).

The type graph of the running example is given in Fig. 1 (without the grey extensions). In the context of this scenario, track nodes are connected to the adjacent tracks by *next* edges. *Shuttle* nodes are located on tracks, which is represented by *at* edges. Shuttles can move forward on tracks being in *DRIVE* mode or can stop resp. brake by changing into *STOP* resp. *BRAKE* mode. To avoid collisions and unnecessary braking maneuvers, shuttles can communicate and establish connections. For this, adjacent tracks are marked by *Conflict* nodes.

PTGTSs are typed over some type graph $TG$ containing at least a type node *Clock*. Furthermore, for every graph $G$ we use the function $CN(G) = \{n \mid n \in G_V \ \wedge \ type_V(n) = Clock\}$ to identify in every graph the nodes used for time measurement only. In the following, we call such identified nodes simply *clocks*.

The type graph in Fig. 1 thus equips tracks with *clocks* needed for time measurement to be able to control the time for rule applications.

The adaptation of graphs is realized using GT rules, which are to be understood as local rule-based modifications defining additions and removals of substructures. A *rule* $\rho = L \xleftarrow{l} K \xrightarrow{r} R$ is given by a span of injective typed graph morphisms with the graphs $L$ and $R$ called the left-hand side and the right-hand side of the rule, respectively. A *match* for a rule is a graph morphism from $L$ to the current graph $G$ describing one option where the rule could be applied in $G$. The transformation procedure defining a GT step is formally introduced by the *DPO approach* [5].

According to [9], PTGTSs are a combination of Probabilistic Graph Transformation Systems (PGTSs) and Timed Graph Transformation Systems (TGTSs). Similarly to PGTSs, transformation rules in PTGTSs can have multiple right-hand sides where each of them is annotated with a probability. While the choice for a rule match remains nondeterministic, the effect of a rule becomes probabilistic. Similarly to TGTSs, each probabilistic timed graph transformation (PTGT) rule has a guard formulated over clocks contained in the left-hand side of the rule, which is used to control the rule application. Moreover, each rule contains the information about clocks that have to be reset during the rule application.

A *probabilistic timed graph transformation (PTGT) rule* $R$ is a tuple $(L, P, \mu, \phi, r_C)$ where $L$ is a common left-hand side graph, $P$ is a finite set of graph transformation rules with the left-hand side $L$, $\mu \in Dist(P)$ is a probability distribution, $\phi \in \Phi(CN(L))$ is a guard over nodes of the type *Clock* contained in $L$, and $r_C \subseteq CN(L)$ is a set of nodes of the type *Clock* in $L$ to be reset (see [9]).

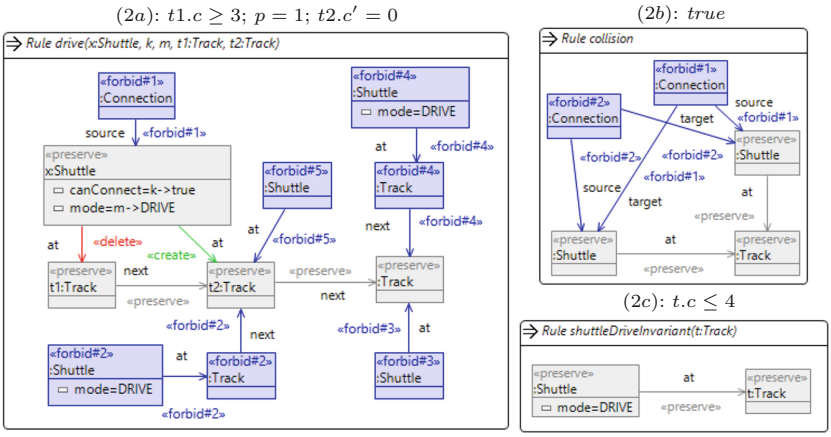



**Fig. 2.** PTGT rule *drive (a)*, atomic proposition *collision (b)*, and invariant *shuttleDriveInvariant (c)* of the shuttle scenario PTGTS in HENSHIN syntax.

In PTGTSs, we also employ negative application conditions (NACs) [7] and attributes. They allow to increase the descriptive expressiveness of the rules and can be added straightforwardly to the presented formalization.

The behavior of the shuttle scenario is modeled using 14 PTGT rules in HENSHIN [3]. In the following, we only discuss one of them in more detail and give an intuition for the other rules due to space restrictions (see more details in [11]). Shuttles can drive alone or can build convoys to reduce the energy consumption. The rule *drive* (see Fig. 2a) allows a shuttle leading a convoy or a shuttle driving without a convoy to move forward if there are no shuttles located too close in front of it. The restrictions for the location of other shuttles are given by NACs of the rule. To reflect real-time behavior, we require that moving on a single track can take between 3 and 4 time units, which we express using the corresponding guards and invariants, respectively, formulated over the track clocks for the driving rules. For the rule *drive* in Fig. 2a, the corresponding guard is given by the annotation $t1.c \geq 3$. For brevity, we refer to a clock $c$ linked to an element $e$ as $e.c$ and omit the extra node $c$. After rule application, we refer to $c$ as $e.c'$. To measure the time spent on a track, we reset the clock of the track to which a shuttle is moving when applying the rule *drive* (annotation $t2.c' = 0$). Other rules of the scenario handle the connection attempts between shuttles as well as situations when shuttles have to brake or stop. Some rules, such as the rules for connection attempts, have higher priorities to ensure their timely application. Furthermore, probabilistic effects are used to model connection failures.

State properties in the form of invariants and atomic propositions are both given for PTGTSs as conditions (non-changing rules) over clocks, the satisfaction of which can be checked for a given state. In the context of our shuttle scenario, we consider an atomic proposition *collision* that is depicted in Fig. 2b and that identifies a collision whenever two shuttles are at the same track without being

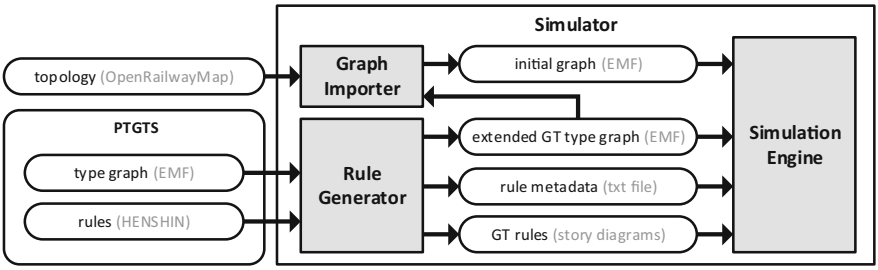



**Fig. 3.** Architecture of the PTGTS simulator.

connected. The invariant *shuttleDriveInvariant* in Fig. 2c ensures that a shuttle in mode DRIVE should not remain longer than 4 time units on a track ($t.c <= 4$).

A *probabilistic timed graph transformation system (PTGTS)* $S$ is then a tuple $(TG, G_0, v_0, \Pi, I, AP, prio)$ where $TG$ is a finite type graph including the type node *Clock*, $G_0$ is a finite initial graph over $TG$, $v_0 : CN(G_0) \to \mathbb{R}$ is the initial clock valuation assigning the clock value 0 to every clock, $\Pi$ is a finite set of PTGT rules, $I$ is a finite set of probabilistic timed invariants, $AP$ is a finite set of probabilistic timed atomic propositions, and $prio : \Pi \to \mathbb{N}$ is a priority function assigning a priority to each rule (see [9]).

## 3 Simulator

In this section, we present the concepts behind our PTGTS simulator [11]. Each PTGT rule is translated into multiple typed GT rules. During the simulation, only specific GT rules must be applied to specific subgraphs. Structural matches are marked to avoid searching large parts of the graph after a local change.

Our simulator consists of three active components highlighted in Fig. 3. The rule generator creates GT rules from a PTGTS and the simulation engine selects and applies these GT rules. The graph importer constructs input graphs based on real-world public transport network topologies from OpenRailwayMap [10].

### 3.1 Simulation Engine

The simulation engine's algorithm for applying GT rules is sketched in Fig. 4. To select applicable rules and affected subgraphs, the engine keeps track of so-called *markers*. The engine is implemented in Java. It uses the Eclipse Modeling Framework (EMF) and an interpreter for story diagrams [6]. The interpreter allows for graph pattern matching starting with a fixed partial match, which, together with the engine's marker bookkeeping, makes the algorithm incremental.

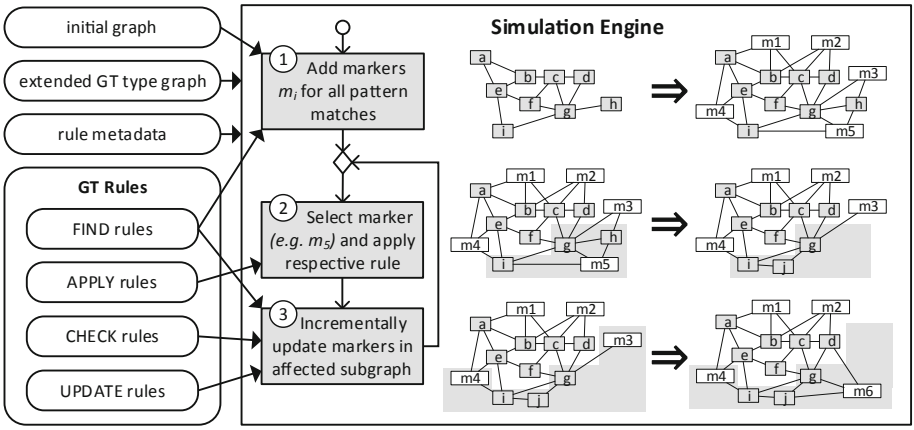



**Fig. 4.** PTGTS simulation algorithm based on marking pattern matches.

**Step 1: Add initial markers** Patterns occurring in the input graph are marked by generated FIND rules for PTGT rules, invariants, atomic propositions, and NACs. Since the rules for NACs are applied first, NAC markers can be used in other rules [2]. In the top example in Fig. 4, markers $m_1$–$m_5$ are created.

**Step 2: Apply a rule** Out of the created markers, the engine selects one that represents an enabled rule application with highest available priority and satisfied time bounds. Afterwards, it computes a new global time $t'_g$ s.t. no invariants are violated. Then, the engine uses a generated APPLY rule to apply the actual PTGT rule at the marked pattern, and, finally, resets clocks. In the middle example in Fig. 4, node $h$ is deleted while node $j$ and two edges are created.

**Step 3: Update affected subgraph** After a rule application, the subgraph affected by the application (incl. all markers) is determined so that the necessary updates to the markers can be conducted incrementally. As can be seen in the bottom example in Fig. 4, CHECK rules remove markers that became invalid (e.g. $m_3$), FIND rules mark new patterns with new markers (e.g. $m_6$), and UPDATE rules update the time constraints of remaining markers (e.g. $m_4$).

**Termination.** The simulation engine stops when no rules are applicable (due to a lack of markers or due to violated invariants or time constraints) or when an atomic proposition (e.g. *collision* from Fig. 2b) is matched.

**Handling of Timed Behavior.** Simulating the timed behavior of a PTGTS requires according to PTGTS semantics the advancement of all clock values

whenever time elapses. To avoid changing a potentially huge number of clock values each time, our simulation engine only maintains a global simulation time $t_g$. Instead of a time value $t(c)$, each clock $c$ in the model has a last reset time value $t_r(c)$. Whenever a rule mandates a clock reset for $c$, the last reset time value $t_r(c)$ is set to the global simulation time $t_g$. Whenever the time value $t(c)$ is needed to evaluate a guard or invariant, it can be computed as $t(c) = t_g - t_r(c)$.

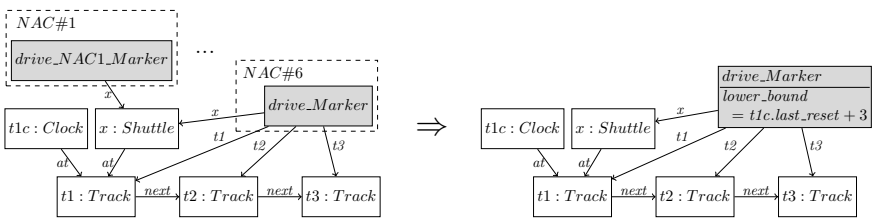
To handle guards and invariants even more efficiently, they are translated into lower and upper bounds, respectively. For example, the guard $t1.c \geq 3$ of the rule *drive* (see Fig. 2a) is translated into $lower\_bound = t_r(t1.c) + 3$, which can then be compared to the current global simulation time $t_g$.

### 3.2 Generation of GT Type Graph and Rules

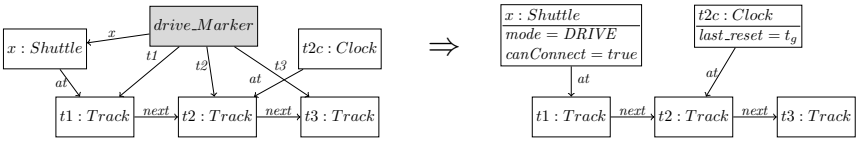
In this subsection, we describe the generation of the type graph and GT rules based on the running example of the PTGT rule *drive* (see Fig. 2a). The GT rules are generated once and stored in the form of story diagrams [6].

**Extended Type Graph.** Markers for all possible pattern matches are added to the type graph. Moreover, a *last_reset* attribute is added to the *Clock* node in order to store the values of $t_r$ as well as lower and upper bound attributes to marker types. Fig. 1 shows the type graph extensions for the PTGT rule *drive* (see Fig. 2a). Similar extensions are made for all other rules but omitted here.

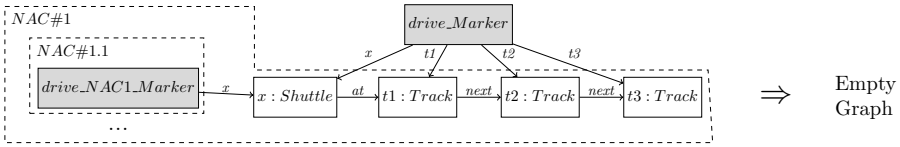
**FIND: Identifying Pattern Matches.** The FIND rules create markers for pattern matches. Their left-hand side is equal to that of the respective PTGT rule, with the exception that instead of NAC patterns, NAC markers are employed. Similar FIND rules are generated for the NAC patterns themselves. To ensure that NACs are found first, the ordering of FIND rules is stored in the rule metadata. FIND rules also assign lower resp. upper time bounds to a marker, which are computed from guards resp. invariants as described above.



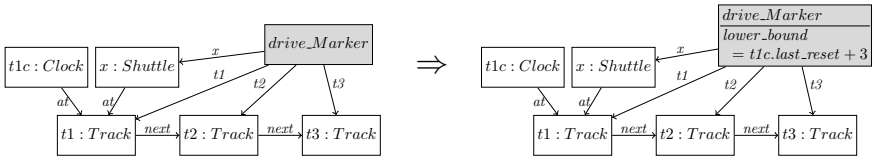

**APPLY: Applying a Rule.** The APPLY rules are similar to the PTGT rules, with the exception that they require a marker on the left-hand side and perform clock resets. If a PTGT rule has more than one right-hand side, multiple APPLY rules are created. Their probabilities are stored in the rule metadata.




**CHECK: Checking Completeness of Pattern Matches.** The CHECK rules remove markers for matches that have become invalid after a rule application. A NAC of the whole original pattern ensures that unless the complete pattern is found, the marker is deleted.




**UPDATE: Updating Time Bounds.** The UPDATE rules recompute lower and upper bounds of markers affected by the update of *last_reset* attributes.




## 4 Evaluation

For evaluation, we constructed input graphs from the tram networks of four different German cities, including Europe's largest connected tram network in Berlin, which we modeled with 9184 track nodes. We assumed a density of one shuttle per 10 tracks and, in case of Potsdam, created an additional topology with doubled density. For each topology, we generated three sets of initial shuttle positions and ran each of these experiments three times, leading in total to 45 runs for up to 25.000 steps each (most ended earlier due to invariant violations).

We were able to use the simulator to improve the PTGTS by discovering and analyzing situations where invariants were violated. These situations were too complex to be efficiently discovered by our previous model checking approach in [9] using PRISM [8] e.g. when a violation is caused by three shuttles approaching two subsequent crossroads with a specific timing.

Also, we tested whether the average runtime for a simulation step does not change according to a trend (i.e., it is stationary) after an initial interval. For that, we ran three different stationarity tests (ADF, KPSS and PP, see [1]). All tests showed statistically significant results (i.e., $p\text{-}value < 0.05$), except for a single simulator run in Frankfurt where one of the three tests had a $p\text{-}value$ of 0.09.

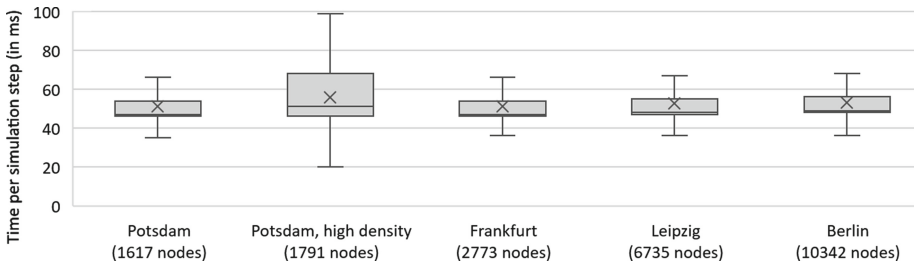



**Fig. 5.** Distribution of runtime per simulation step after non-stationary first interval.

As can be seen in Fig. 5, when excluding the non-stationary first interval, the size of our example models has no significant impact on the average runtime per simulation step. However, the higher shuttle density appears to have an influence on the runtime, which can be explained by a higher rate of rule applications for the connection of shuttles that affect a larger subgraph.

## 5 Conclusion and Future Work

We presented a simulator for PTGTSs [9] and demonstrated that it can import complex real-world topologies, automatically detect violations of state properties, and handle the graph pattern matching as well as the concepts of time and probabilities so efficiently that complex large-scale topologies can be considered. As future work, we plan to formally analyze and further improve the efficiency of our tool, provide more mature tool support covering, in particular, the transition to model checking, and support checking for more than state properties.

**Acknowledgments.** We thank our colleague Christian Medeiros Adriano who supported us in the statistical evaluation of the experiment results.

## References

1. Banerjee, A., Dolado, J.J., Galbraith, J.W., et al.: Co-integration, Error Correction, and the Econometric Analysis of Non-stationary Data. OUP Catalogue (1993)
2. Beyhl, T., Blouin, D., Giese, H., Lambers, L.: On the operationalization of graph queries with generalized discrimination networks. In: Echahed, R., Minas, M. (eds.) ICGT 2016. LNCS, vol. 9761, pp. 170–186. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40530-8_11
3. The Eclipse Foundation: EMF Henshin (2013). https://www.eclipse.org/modeling/emft/henshin
4. Ehmes, S., Fritsche, L., Schürr, A.: SimSG: rule-based simulation using stochastic graph transformation. J. Object Technol. **18**, 1:1–17 (2019). The 12th International Conference on Model Transformations
5. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. MTCSAES. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2_15

6. Giese, H., Hildebrandt, S., Seibel, A.: Improved flexibility and scalability by interpreting story diagrams. In: Magaria, T., Padberg, J., Taentzer, G. (eds.) Proceedings of the 8th International Workshop on Graph Transformation and Visual Modeling Techniques, vol. 18. Electronic Communications of the EASST (2009)
7. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae **26**(3, 4) (1996)
8. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
9. Maximova, M., Giese, H., Krause, C.: Probabilistic timed graph transformation systems. J. Log. Algebraic Methods Program. **101**, 110–131 (2018)
10. OpenRailwayMap. https://www.openrailwaymap.org
11. PTGTS Simulator Project Website. https://mdelab.de/ptgts-simulator
12. RailCab project. https://www.hni.uni-paderborn.de/cim/projekte/railcab
13. Ráth, I., Vago, D., Varró, D.: Design-time simulation of domain-specific models by incremental pattern matching. In: 2008 IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 219–222 (2008)
14. Syriani, E., Vangheluwe, H.: A modular timed graph transformation language for simulation-based design. Softw. Syst. Modeling **12**(2) (2013)
15. Torrini, P., Heckel, R., Ráth, I.: Stochastic simulation of graph transformation systems. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 154–157. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12029-9_11