Visualization, Simulation and Analysis of Reconfigurable Systems *

Claudia Ermel[1] and Karsten Ehrig[2]

[1] Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
Email: lieske@cs.tu-berlin.de,

[2] Department of Computer Science
University of Leicester, UK
Email: karsten@mcs.le.ac.uk

Abstract. Meta-modeling is well known to define the basic concepts of domain-specific languages in an object-oriented way. Based on graph transformation, an abstract meta-model may be enhanced with information on concrete visualization of objects and relations, and the language syntax is defined by a graph grammar. Moreover, graph transformation can also formalize the semantic aspects of models, thus providing a basis for model validation by simulation.
Apart from editing and simulating the behavior of a system, there may be necessary reconfiguration operations which change the underlying system structure at runtime. In this paper, we focus on the interrelation of simulation and reconfiguration operations using formal verification techniques based on graph transformation. Our approach is demonstrated by the definition of a domain-specific language for building, simulating and reconfiguring small railway systems, using the Tiger tool environment. For further verification, we define a model transformation from the railway domain to Petri nets.

Keywords: graph transformation, model transformation, reconfigurable system, visualization, simulation, analysis

1 Introduction

Domain-specific modeling (DSM) aims to model a system at the same level of abstraction with the domain itself. This reduces mental mapping by moving the modeling language closer to the domain as perceived by designers, and improves the model quality compared to using generic modeling languages. The disadvantage of DSM is that for each domain a different visual modeling tool is needed. Here, meta CASE tools can help (like e.g. *MetaEdit+* [1]), which generate e.g. a visual editor on the basis of a definition of the visual domain-specific language.

* This work has been partially sponsored by the IST-2005- 16004 Integrated Project SENSORIA (Software Engineering for Service-Oriented Overlay Computers).

Two main approaches to visual language definition can be distinguished: grammar-based approaches or meta-modeling. Using graph grammars [2], multi-dimensional representations are described by graphs. This allows not only a visual notation of the concrete syntax, but also a visualization of the abstract syntax. While the concrete syntax contains the concrete layout of a visual notation, the abstract syntax abstracts from the layout and provides a condensed representation to be used for further processing, e.g. behavior simulation or system reconfiguration. Graph rules are used to manipulate the graph representation of a language element. Meta-modeling (see e.g. [3]) is also graph-based, but uses constraints instead of a grammar to define a visual language. While visual language definition by graph grammars can borrow a number of concepts from classical textual language definition, this is not true for meta-modeling.

Graph transformation can also formalize the semantic aspects of models. There are numerous formal graph-transformation-based semantics definitions [4]. In this paper, we use graph transformation not only to construct and visualize domain-specific visual models, but also to simulate dynamic model behavior. Apart from operations for editing, there may be necessary operations to change the underlying system structure at runtime (i. e. during simulation). Systems allowing to be changed have become an important topic in recent years since the adaption of a system to a changing environment plays a significant role e. g. in computer supported cooperative work, multi agent systems or mobile networks. In our approach, such reconfiguration operations are modeled by *reconfiguration rules*, and the corresponding systems are called *reconfigurable systems*.

As running example, we model a toy railway system. The visualization shows different kinds of tracks and switches which can be glued at connection points. Simulation rules allow to move a train to an adjacent track, respecting the switch modes. Reconfiguration rules allow to toggle between two modes of a switch. Graph transformation as a formally defined calculus [2, 5] offers well-founded theoretical results that support the formal reasoning about graph-based models at all levels. We apply formal graph transformation techniques to reason about the independence of simulation and reconfiguration steps. For further verification, we define a model transformation from the railway system language to Petri nets. We apply the Tiger environment [10] for generating visual editor plug-ins in Eclipse [6] from graph grammars. Tiger is based on the graph transformation engine and analysis tool Agg [7].

The paper is structured as follows: Section 2 reviews the concepts for the graph-grammar based definition of visual languages, demonstrated by a domain-specific language to model small railway systems. In Section 3, concepts for simulation and reconfiguration of discrete-event systems by graph transformation are discussed, and the railway system is coming to life by operations for moving trains and changing switch modes. Section 4 applies verification techniques to analyze the interrelation of reconfiguration and simulation steps. Furthermore, a model transformation to Petri nets is defined, which allows to verify further dynamic system properties.

2 Defining Visual Domain-Specific Languages

Meta-modeling uses UML class diagrams to model a visual languages abstract syntax (see e.g. the MOF approach by the OMG [3]). While class diagrams appear to be more intuitive than graph grammars, they are also less expressive. Therefore, meta-modeling additionally uses context conditions to overcome the weaker expressive power. In the MOF approach, for instance, the Object Constraint Language (OCL) is used for this purpose. The advantage of meta-modeling is that UML users, who probably have basic UML knowledge, do not need to learn a new external notation to be able to deal with syntax definitions. Graph grammars are more constructive, i.e. closer to the implementation, and provide a formal basis for visualizing, validating and verifying model behavior. Hence, in our TIGER approach, we combine the visual definition of domain-specific languages by meta-modeling, and the definition of editing operations by graph transformation rules.

2.1 Graph Transformation

The main idea of graph grammars and graph transformation is the rule-based modification of graphs where each application of a graph transformation rule leads to a graph transformation step. Graph grammars can be used on the one hand to generate graph languages, and on the other hand to model state changes (operational behavior). Meanwhile, graph transformation has been investigated as a fundamental concept for programming, specification, concurrency, distribution, visual modeling and model transformation [2, 8].

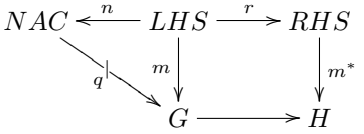
The core of a graph transformation rule ($LHS \xrightarrow{p} RHS$) is a pair of graphs (LHS, RHS), called left-hand side and right-hand side, and an injective (partial) graph morphism $p : LHS \to RHS$. Applying the rule ($LHS \xrightarrow{p} RHS$) means to find a match of LHS in the source graph and to replace this matched part in the source graph by the corresponding RHS, thus transforming the source graph into the target graph of the graph transformation.

Especially for the application of graph transformation techniques to visual language (VL) modeling, *typed attributed graph transformation systems* [2] have proven to be an adequate formalism. A VL is modeled by a type graph capturing the definition of the underlying visual alphabet, i.e. the symbols and relations which are available. Sentences or diagrams of the VL are given by graphs typed over (i.e. conforming to) the type graph. Such a VL type graph corresponds closely to a meta model. In order to restrict the visual sentences to valid visual models, a syntax graph grammar is defined, consisting of a set of language-generating graph transformation rules, typed over the abstract syntax part of the VL type graph. The rules describe editing operations which lead to the construction of valid visual models only.

Intuitively, the application of rule p to graph G via a match m from LHS to G deletes the image $m(LHS)$ from G and replaces it by a copy of the right-hand side $m^*(RHS)$. Note that a rule may only be applied if the so-called *gluing*

condition is satisfied, i.e. the deletion step must not leave *dangling edges*, and for two objects which are identified by the match, the rule must not preserve one of them and delete the other one.

Definition 1 (Graph Transformation). *Let* $(LHS \xrightarrow{p} RHS)$ *be a typed graph transformation rule and* G *a typed graph with a typed graph morphism* $LHS \xrightarrow{m} G$*, called match. A* graph transformation step $G \overset{p,m}{\Longrightarrow} H$ *from* G *to a typed graph* H *via rule* p*, match* m*, and co-match* m^* *is shown in the diagram to the right. The rule* r *may be extended by a set of* negative application conditions *(NACs) [9, 2]. A match* $LHS \xrightarrow{m} G$ *satisfies a NAC with the injective NAC morphism* $n : LHS \to NAC$*, if there is no injective graph morphism* $NAC \xrightarrow{q} G$*.*



A sequence $G_0 \Rightarrow G_1 \Rightarrow ... \Rightarrow G_n$ *of graph transformation steps is called* graph transformation *and denoted as* $G_0 \overset{*}{\Rightarrow} G_n$*.* △

The language of a graph grammar consists of the graphs that can be derived from the start graph by applying the transformation rules.

Although we do not define the attribution concept for graphs formally in this paper (see [2] for a complete definition of the theory), we use node attributes in our examples, e.g. text for the names of nodes, or integers for their positions. This allows us to perform computations on attributes in our rules and offers a powerful modeling approach.

2.2 Type Graph and Syntax Rules for a Railway System

Using graph transformation, a *type graph* defines the visual alphabet, i.e. the symbols and symbol relations of a visual language. Layout information is integrated in the type graph by special shape types connected to symbol nodes, and by constraints on the relations of visual representations. The shape types include information about the symbol's shape (any kind of graphical figure or line), and the constraints establish certain visual relations (like "The shape for this symbol type is always glued to the shape for another symbol type," or "The shape for this symbol type has always a minimal size of ...").

Fig. 1 shows the definition of the type graph of our domain-specific language for building railway systems (without trains so far) in TIGER *(Transformation-based Generation of Environments)* [10, 11], a visual editor generation tool. In the upper editor, we see the abstract syntax type graph with symbol types like Track, End and Buffer. For each type variant, a child inheriting from the corresponding abstract type is added to the type graph (e.g. StraightH for a horizontal straight track, Bend1 for a bend which is curved up-left/right-down, and HL for a track End which is the gluing point gluing two tracks at the first track's horizontal-left side. Note that the nodes in the abstract syntax type graph contain layout positions (x, y: int) allowing the editing rules to set the position of the corresponding figures in the editor accordingly. In the lower part of Fig. 1, editors for shape types are shown, depicting the visualization of different track types and the Buffer type.

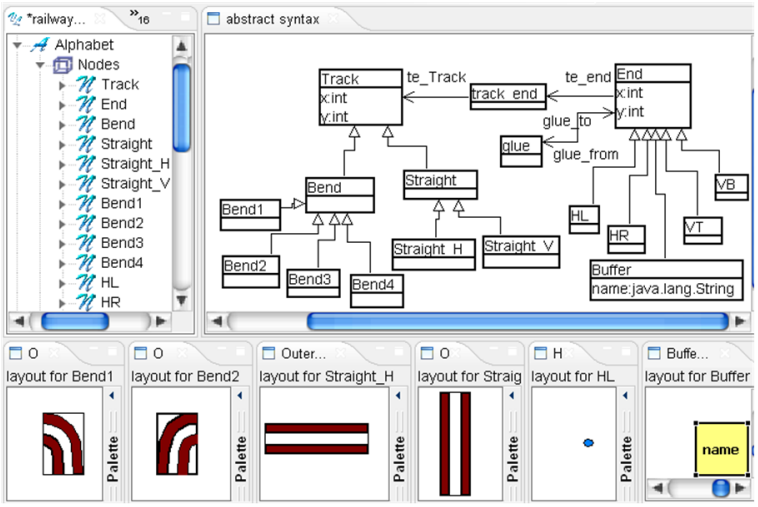


Fig. 1. Type Graph for the Domain-Specific Railway Language

A type graph together with a syntax graph grammar is used as high-level visual specification mechanism for VLs. The grammar restricts the allowed visual sentences conforming to the type graph to the meaningful ones. Grammar rules define syntactical editing operations. Such an operation is modeled as a graph rule typed over the VL type graph being applied to the syntax graph of the current diagram. Thus, only such syntactical changes are allowed which are described by a syntax rule and which result again in a valid VL diagram. An editing operation (i.e. the application of a syntax rule) results in a corresponding change of the internal abstract syntax graph of the diagram and the layout positions of the corresponding symbols.

Fig. 2 shows four of the syntax rules for the railway VL. Rule newStraightH produces an unconnected track, the other rules add tracks, switches and buffers by gluing them to tracks which already exist in the model. Numbers (m = ..) at objects indicate mappings from a rule's LHS to its RHS. Input parameters (objects to be identified for the match by mouse click) are indicated by numbers (in = ..) in a rue's LHS. NACs (not depicted) forbid gluing tracks to tracks at endpoints where already other tracks are glued. Positions relating objects to each other are defined in each rule's properties view.

A visual language (VL) definition based on a type graph and a set of syntax rules is used in Tiger to generate a corresponding visual editor. Tiger combines constructive VL specification using graph transformation with sophisticated graphical editor development features offered by the Eclipse Graphical Editing Framework (GEF) [12]. The execution of editor commands available in the generated editor correspond to the application of syntax rules to the underlying abstract syntax graph of a diagram. The rule application is performed by the graph transformation engine AGG [7]. Tiger extends AGG by a concrete

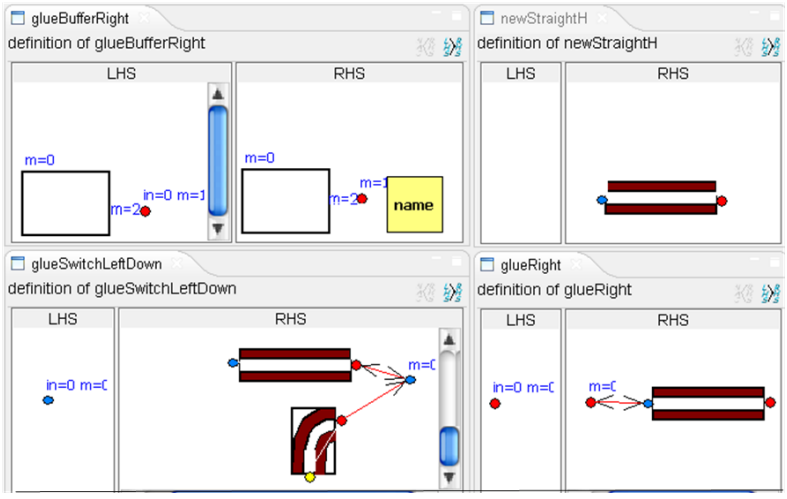


Fig. 2. Syntax Rules for the Railway Language

visual syntax definition for flexible means for visual model representation. From the definition of the VL, the TIGER *Generator* generates Java source code. The generated Java code implements an ECLIPSE visual editor plug-in based on GEF which makes use of a variety of GEF's predefined editor functionalities. Layout information (e.g. color, shape, and size, ..) are coded in the corresponding GEF editor classes. Fig. 3 shows the graphical user interface of the railway editor generated by TIGER from the VL specification consisting of the railway type graph similar to the one in Fig. 1, but now also allowing to edit train symbols (light-blue rectangles), and a railway syntax grammar. Basic editor operations are available in the tool palette on the left-hand side, or by the context menu which offers a list of operations depending on the selected symbol type.

3 Validation by Simulation

If a visual language models dynamic aspects of systems, visual simulation is interesting. Usually, a prerequisite for simulation is a (slight) extension of the visual language such that different execution states can be distinguished. In the case of our railway system, this is the addition of trains. Simulation then is specified by a set of *simulation rules*, typed over the extended VL type graph. The simulation rules specify the possible simulation steps (e.g. train movements) which do not change the underlying system structure. A sequence of simulation steps is called *simulation run* or *simulation scenario*. In general, we have non-determinism in simulation in the sense that there are more than one rules applicable at more than one possible matches. Up to now, TIGER supports stepwise simulation only, i.e. the user selects an applicable rule from the rule palette, and defines the match by clicking on relevant objects in the editor panel.

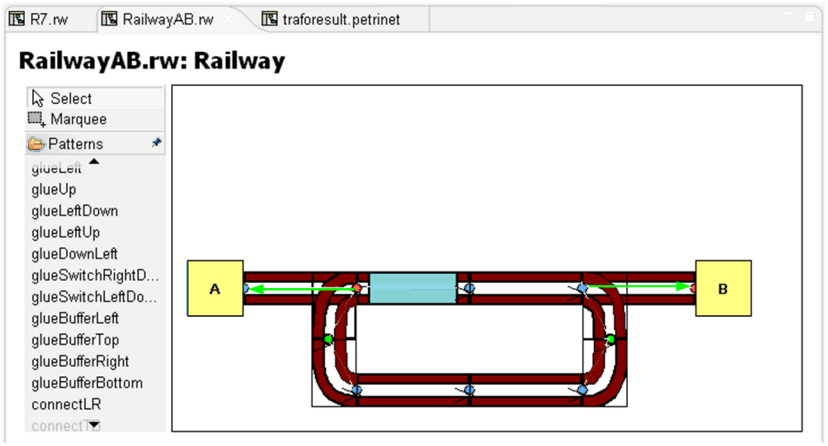


Fig. 3. TIGER-generated Visual Editor for the Railway Language

Fig. 4 shows the abstract syntax of the railway simulation rules. The first rule allows to add a train to a track, thus determining how many trains are distributed initially on which tracks in the railway system. The NAC (drawn as crossed-out part in the LHS) specifies that there must not be another train on this track. The second rule has to be applied after the first one, and models the movement of a train to the next track. Note that using the abstract nodes of type Track and Train, we only need one abstract rule for moving trains. Again, the NAC makes sure that the rule is only applied if there is not yet another train on the target track.

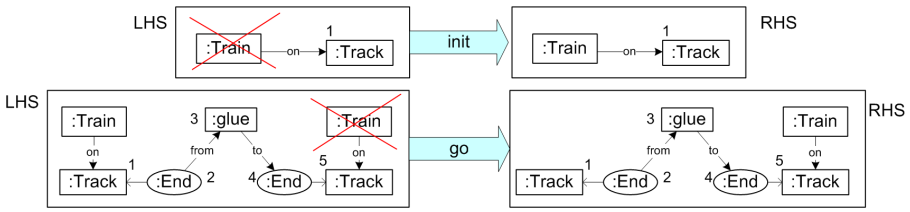


Fig. 4. Simulation Rules for Initial Train Distribution and for Train Movements

In railway simulation not only the position of trains is changing, but also the underlying net topology is adapted when a switch is changing its mode. In our approach, such reconfiguration operations are modeled by *reconfiguration rules*. Simulation and reconfiguration rules may be applicable to the same system states. In our railway system, changing the modes of switches is realized by applying a reconfiguration rule. A switch consists of two tracks (one bend and one straight track) and may be crossed by a train in only one way. The directions

a train is allowed to go are modeled by the glue edges connecting track end points. The reconfiguration rule switch is shown in the top row of Fig. 5, and the effect of its application (a transformation step changing the mode of a sample switch) is shown in the bottom row of Fig. 5, where the match mapping of the track end points is indicated by corresponding numbers. In the concrete syntax, a green arrow indicates the current switch mode.

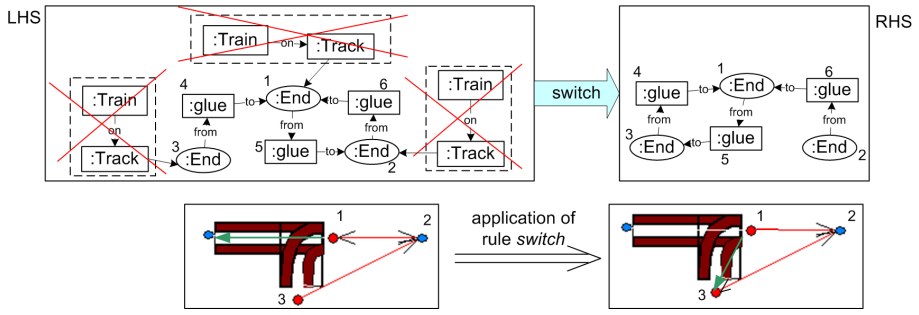


Fig. 5. Reconfiguration Rule realizing Switch Modes

4 Analysis

The aim of analyzing the railway specification is to avoid unsafe states in the simulation. For example, we would like to be sure that

(i) there are never more than one trains on a track,
(ii) a switch can only change its mode when there is no train on it.

In order to check condition (ii), we have to relate a reconfiguration operation (changing the switch mode) and a simulation operation (moving a train). We consider this relation in Section 4.1. Moreover, we are interested in safety properties like deadlocks which can best be analyzed using Petri net tools (see Section 4.4). Hence, we define a model transformation from the railway VL into the semantic domain of Petri nets (Sections 4.2 and 4.3).

4.1 Relation of Reconfiguration and Simulation

When reconfiguration of the system structure is allowed during runtime, the question arises under which conditions a simulation step is independent of a reconfiguration step, i.e. can the two transformations starting from the same system state be applied in any order, leading to the same result. The Local Church-Rosser Theorem for graph transformation systems [2] states that, for two parallel independent graph transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$, there is

a graph G' together with graph transformations $H_1 \overset{p_2,m_2'}{\Longrightarrow} G'$ and $H_2 \overset{p_1,m_1'}{\Longrightarrow} G'$. In our case, we need to analyze the parallel independence of rules belonging to two different rule sets (simulation and reconfiguration rules). To this end, we use the automatic *critical pair analysis* offered by AGG, where rule pairs are analyzed to find out critical pairs of rule matches. Each parallel dependent transformation is an extension of a critical pair. The result of the critical pair analysis applied to the reconfiguration rule switch and the simulation rule go yields e.g. the critical pair shown in Fig. 6.

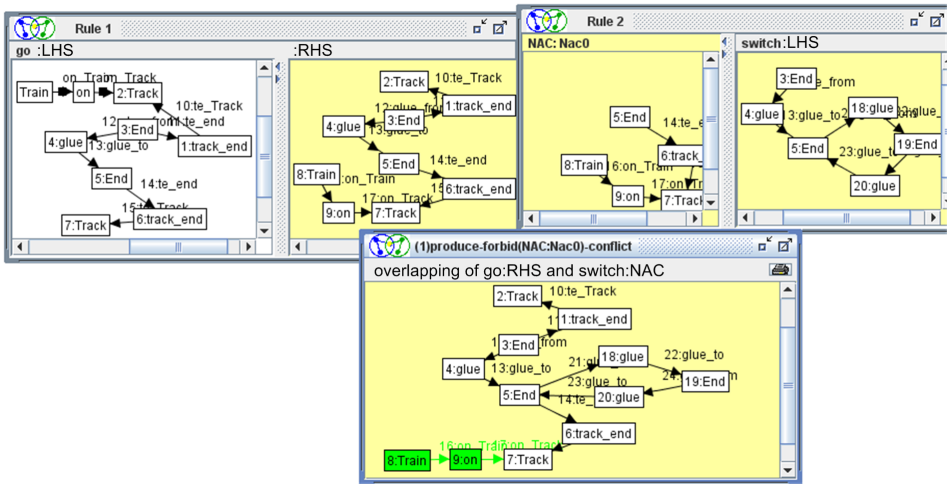


Fig. 6. Critical Pair Analysis of Railway Simulation and Reconfiguration

Analyzing this pair, we see that reconfiguration rule switch cannot be applied if simulation rule go has been applied before and has been moving a train onto a track which is part of the switch, because the NAC of rule switch forbids to reconfigure switches with trains on it. Here, we have a so-called *produce-forbid* conflict, where one rule produces an object which is forbidden in the match of the other rule. Applying critical pair analysis to the reconfiguration rule switch and the other simulation rule init yields a similar conflict. Both conflicts together confirm that condition (ii) is valid for all possible railway models. Here, conflict detection is used to analyze safety conditions. Analogously, critical pair analysis of both simulation rules can be performed to check condition (i).

4.2 Model Transformation from Railway Models to Petri Nets

In this section we present a model transformation from the railway system to Petri nets with the aim to use Petri net analysis and verification techniques to analyze the railway behavior. Surely, only a limited class of simulation problems is sufficiently "Petri net like" to allow transformation of the more powerful graph

rewriting model into Petri nets. Here, the distinction of simulation rules and reconfiguration rules helps to find the part of the system which behaves "Petri net like" (e.g. the trains moving along the tracks), and which can be translated and analyzed. The other parts describe reconfiguration operations (e.g. adding tracks or changing switches) and rather correspond to changes of the Petri net structure, but not to Petri net firing behavior.

Model transformations between visual languages is defined in our approach by graph transformation rules, as well. We transform the abstract syntax graph of a source model (e.g. a railway system state) by applying transformation rules resulting in the abstract syntax graph of the target model (e.g. a state of a Petri net). The abstract syntax of source and target models are specified by the type graphs TG_S and TG_T. A model transformation is defined by a graph transformation system $GTS = (TG, P)$ consisting of type graph TG and a set of TG-typed model transformation rules P, where both type graphs TG_S and TG_T have to be subgraphs of TG (see Fig. 7). The model transformation starts with the abstract syntax graph G_S of the source model. As TG_S is a subgraph of TG, G_S is also typed over TG. Please note that TG may contain not only TG_S and TG_T, but also additional types and relations which are needed for the transformation process.

After application of all model transformation rules P as long as possible, the resulting graph G_n is typed over TG, but not yet over the type graph TG_T of the target language. In order to delete all items in G_n which are not typed over TG_T we apply a restriction construction, which restricts G_n to those objects typed over TG_T. The model transformation process is visualized in Fig. 7.

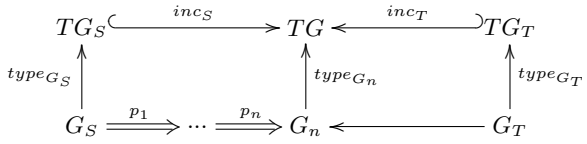


Fig. 7. Typing in the Model Transformation Process

Fig. 8 shows the type graph TG for the model transformation from railway systems to Petri nets. TG relates elements of the source type graph for railway systems (see Fig. 1) to elements of the target type graph for Petri nets, consisting of symbol types for Places, Transitions and Arcs in two directions. Tokens are modeled by an integer attribute of the Place type.

Two of the model transformation rules are shown in Fig. 9. Obviously, tracks are mapped to places (see rule *createPlace*), and trains to tokens (see rule *createToken*). Here, the possibility to use abstract types like Track or End in the rules proves to be very useful, since we do not have to relate e.g. all different types of tracks to places. When mapping connections between tracks to arcs and transitions in the Petri net, the gluing of ends must be considered to determine the direction of Petri net arcs. (Rule *createTransition* is not shown explicitly.)

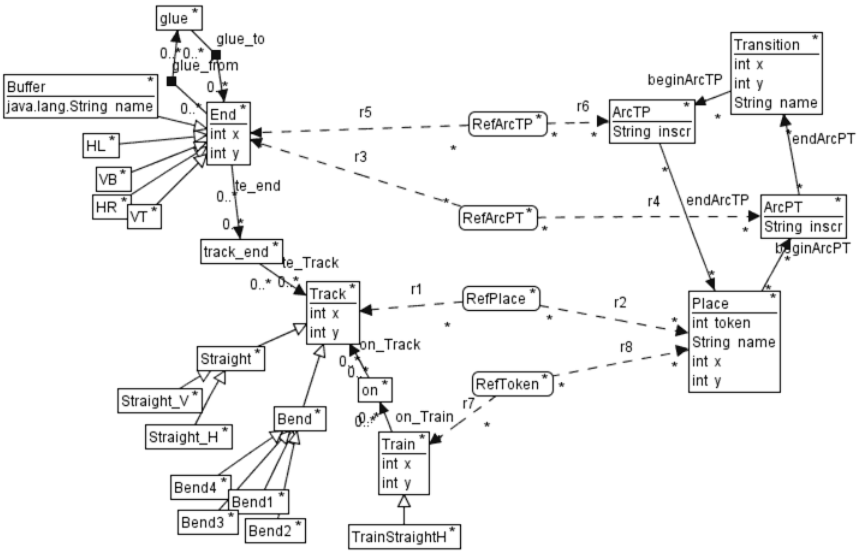


Fig. 8. Type Graph for the Model Transformation *Railway2Petri*

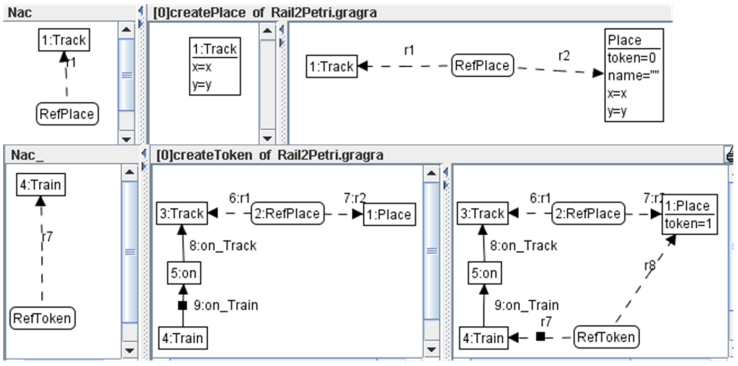


Fig. 9. Two Rules for the Model Transformation *Railway2Petri*

Model transformations based on graph transformation have been investigated e.g. in [13], where also techniques are presented to show that a model transformation has functional behavior, and is syntactically correct, i.e. for each diagram in the source language we obtain in a finite number of steps in a unique well-defined diagram in the target language. To execute model transformation rules and to check functional properties of model transformations (termination and confluence), the graph transformation engine AGG [7, 14] can be used. Furthermore, TIGER [11] also offers tool support for model transformation by graph transformation between two generated ECLIPSE editor plug-ins. Fig. 10 shows the

Petri net resulting from the model transformation of the train system depicted in Fig. 3 using the *Railway2Petri* model transformation rules.

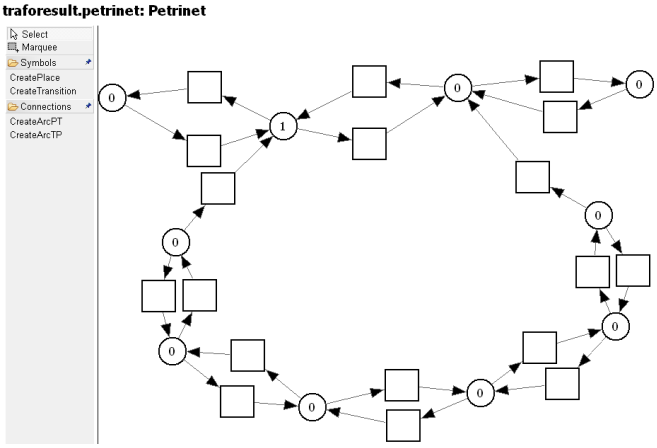


Fig. 10. Petri Net Obtained by Model Transformation of the Railway System in Fig. 3

4.3 Correctness of the Model Transformation

Apart from syntactical properties of a model transformation, we can argue about its semantical correctness if both the source and the target language have a semantics. In our case, the behavioral semantics of railway systems is given by the simulation rules, and the semantics of Petri nets is the well-known Petri net firing behavior. We have to show that for each simulation step in the source railway model, there is a corresponding simulation step in the target model, i. e. a firing step in the corresponding Petri net. Using formal properties of model-and-rule-transformation based on graph transformation [15, 16], we argue as follows: we perform a *rule transformation* of the simulation rules using the *Railway2Petri* model transformation rules. Basically, the model transformation rules are applied to the LHS, RHS and NACs of each simulation rule. This results in a transformed simulation rule consisting of the translated LHS, RHS and NACs. Applying such a rule transformation to the railway simulation rule go, we obtain the rule go' shown at the bottom of Fig. 11 which models the firing behavior of a transition with exactly one pre-domain place and one post-domain place which is enabled only if the post-domain place is unmarked.

All Petri nets which are results of a railway model transformation, have only transitions of that type. So, the firing rule go' in Fig. 11 describes the firing behavior for all possible resulting railway nets, provided that we assume elementary Petri nets (or *condition-event (C/E) nets* [17]) as underlying semantic domain, a restricted kind of place/transition nets where place capacity and arc

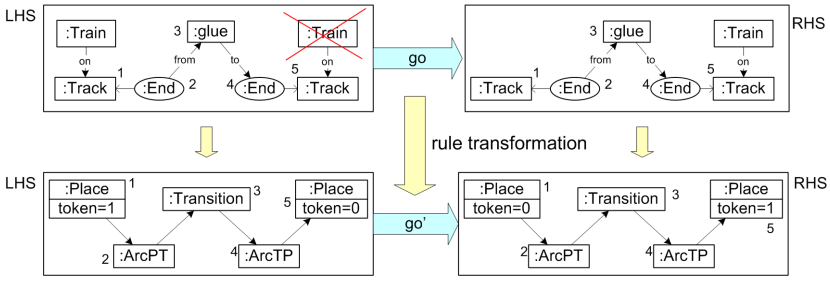


Fig. 11. Rule Transformation of the Railway simulation rule go in Fig. 3

weights are always one. The semantics of general place/transition nets would *not* correspond to the firing rule in Fig. 11, since in place/transition nets more than one token may be put to a post-domain place. In the case of C/E nets, each simulation step in the railway model (a train movement from one track to the next) corresponds to a transition firing step of the transition between the places corresponding to the two tracks, and we have the situation depicted in the following commuting correspondence diagram:

$$\begin{array}{ccc} G_1 & \overset{model\ trafo}{\overset{*}{\Longrightarrow}} & N_1 \\ \Big\downarrow{\scriptstyle Railway\ simulation\ step} & & \Big\downarrow{\scriptstyle C/E\ net\ simulation\ step} \\ G_2 & \overset{model\ trafo}{\overset{*}{\Longrightarrow}} & N_2 \end{array}$$

In this diagram, we start with a graph G_1 of the railway domain, such that the model transformation $G \overset{*}{\Longrightarrow} N_1$ yields the Petri net N_1, where a transition can be fired, leading to the Petri net N_2 with a different marking. Then, there exists a simulation step in the railway domain $G_1 \rightarrow G_2$ such that the model transformation of G_2 yields the same Petri net N_2. In fact, we have that the source railway simulation model and the target Petri net are always bisimilar. The model transformation establishes one equivalence relation relating railway graphs and marked C/E nets, and another one relating railway simulation rules and Petri net transitions. Then, given a railway graph G_1 and a corresponding Petri net N_1 resulting from the model transformation of G_1, i.e. $G_1 \sim N_1$, the equivalence is a bisimulation since for rule r used in the transformation step $G_1 \xrightarrow{r} G_2$ there exists a transition t with $r \sim t$ and $N_1 \xrightarrow{[t\rangle} N_2$ and $N_2 \sim G_2$.

4.4 Analysis in the Petri Net Domain

Now the resulting Petri net can be analyzed using Petri net techniques, e.g. for liveness (any transition can fire eventually), for place invariants (sets of places where the sum of tokens remains constant), transition invariants (sets of transitions the firing of which does not change the marking), deadlocks (sets of places that will never be marked again, once they are empty) or traps (sets of places

that will never loose their tokens). An example is the trap in the net in Fig. 10, consisting of the places corresponding to the horizontal tracks from A to B in Fig. 3, since in the current switch mode, the train will never leave those tracks.

An interesting aspect in model transformation for analysis is the back-annotation of analysis results to the source model. In our case, places can be traced back to the corresponding tracks easily, as we have a one-to-one correspondence between them (see Fig. 8 and rule createPlace in Fig. 9). Thus it is possible to visualize e. g. deadlocks in the railway system by highlighting the corresponding tracks in a certain color. Other interesting properties concern path finding (the shortest connection from point A to point B), and collision detection.

All these properties of reconfigurable systems should be analyzed having in mind the possible reconfiguration operations. For instance, more interesting than knowing whether there is a deadlock considering a fixed switch mode is it to know whether there are deadlocks independent of all possible switch modes. Here, the open problem is to find a way to generate all possible switch configurations inside the railroad domain and generate a Petri net for each case. A possible solution would be to include the switch behavior in the generated Petri net, which leads to more complex model transformation rules but allows one to analyze all possible switch configurations. As reconfiguration operations, then only rules for adding, removing or repairing tracks have to be considered.

5 Related Work

While Petri net modeling and analysis tools like Netlab [18] and CPNTools [19], are well known and frequently used, domain specific modeling languages as supported by TIGER may be generated using meta CASE tools like DiaGen [20] and AtoM^3 [21]. Those tools have no direct support for model driven analysis techniques and do not support reconfiguration of systems during runtime. Petri net transformations that aim at changing the net in arbitrary ways have been described in [22], and runtime system reconfiguration has been investigated in [23], but a user friendly, graphical environment for the design and analysis of reconfigurable systems is still missing.

Model transformations are supported from various tools like VIATRA2 [24], GrEAT [25], and other tools from the Eclipse Generative Modeling Tools (GMT) project [26]. In most cases these transformations have to be described textually, and user friendly support for visual analysis and testing is generally missing.

6 Conclusion

This paper gives an example for using the unifying approach of graph transformation to define the syntax and semantics of a domain-specific visual modeling language. The language models a small railway system, and from the graph-transformation based language definition, a visual editor is generated as ECLIPSE plug-in. The type hierarchy used for syntax definition provides a good basis also for describing the semantics of the system in terms of simulation rules, and for

a model transformation from the domain-specific language into Petri nets. Since many systems have to be reconfigurable during runtime, we have investigated the relation of reconfiguration operations (e.g. changing the mode of a switch) and simulation operations (e.g. move the train to the next track) by analyzing rule dependencies. Tool support for language definition, visualization and visual editor generation is available by the TIGER tool environment and the graph transformation engine AGG, providing support to analyze termination, conflicts and dependencies in graph transformation systems.

Using graph transformation for modeling and analyzing reconfigurable systems has shown to be a solid basis to reason about system properties in different reconfiguration modes. In this context, interactions between simulation states and structure should be investigated in more detail, since reconfiguration is often triggered by certain system state changes [27].

As future work concerning TIGER, we envisage an extension of TIGER towards more sophisticated editing and simulation. We intend to provide basic syntax-oriented operations automatically instead of requiring the language designer to specify them manually for each VL element. For simulation we aim at structuring simulation rules using control structures. Abstract rules and rule structuring techniques are the basis of a formal but scalable approach that hopefully will prove to be usable for modeling and analyzing also much larger case studies. Moreover, work is in progress to transfer results concerning dependencies of simulation and reconfiguration operations from graph transformation systems to Petri nets, such that *reconfigurable Petri nets* can be modeled [28].

Acknowledgements

The authors would like to thank Szilvia Varró-Gyapay and the anonymous referees for their useful comments.

References

1. Tolvanen, J., Rossi, M.: MetaEdit+: Defining and Using Domain-Specific Modeling Languages and Code Generators. In: Proc. Conf. on Object-oriented programming, systems, languages, and applications (OOPSLA '03). ACM Press (2003) 92–93
2. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theoretical Computer Science. Springer Verlag (2006)
3. Object Management Group: Meta-Object Facility (MOF), Version 1.4. (2005) http://www.omg.org/technology/documents/formal/mof.htm.
4. Kreowski, H.J., Hölscher, K., Knirsch, P.: Semantics of visual models in a rule-based setting. ENTCS **148**(1), Elsevier Science (2006) 75–88
5. Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformations, Vol. 1: Foundations. World Scientific (1997)
6. Eclipse Consortium: Eclipse – Version 3.2.1. (2007) http://www.eclipse.org.
7. Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: Proc. on Application of Graph Transformations with Industrial Relevance. Vol. 3062 of LNCS. Springer (2004) 446 – 456

8. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools. World Scientific (1999)
9. Habel, A., Heckel, R., Taentzer, G.: Graph Grammars with Negative Application Conditions. Special issue of Fundamenta Informaticae **26**(3,4) (1996) 287–313
10. Ermel, C., Ehrig, K., Taentzer, G., Weiss, E.: Object Oriented and Rule-based Design of Visual Languages using Tiger. In: Proc. Workshop on Graph-Based Tools. Vol. 1 of EC-EASST (2006)
11. Tiger Project, TU Berlin (2005) http://www.tfs.cs.tu-berlin.de/tigerprj.
12. Eclipse Consortium: Eclipse Graphical Editing Framework (GEF) – Version 3.2. (2006) http://www.eclipse.org/gef.
13. Ehrig, H., Ehrig, K.: Overview of Formal Concepts for Model Transformations based on Typed Attributed Graph Transformation. In: Proc. Workshop on Graph and Model Transformation. Vol. 152 of ENTCS (2005)
14. AGG, TU Berlin (2005) http://tfs.cs.tu-berlin.de/agg.
15. Ermel, C., Ehrig, H., Ehrig, K.: Semantical Correctness of Simulation-to-Animation Model and Rule Transformation. In: Proc. Workshop on Graph and Model Transformation Vol. 4 of EC-EASST (2006)
16. Ermel, C., Ehrig, H.: Behavior-preserving simulation-to-animation model and rule transformation. In Proc. Workshop on Graph Transformation for Verification and Concurrency. To appear in ENTCS (2007)
17. Reisig, W.: Systementwurf mit Netzen. Springer-Verlag, Springer Compass (1985)
18. RWTH Aachen: Petrinetz-Tool Netlab (Windows). (2007) http://www.irt.rwth-aachen.de/typo3/index.php?id=101&L=0.
19. CPN Group, University of Aarhus, Denmark: CPN Tools: Computer Tool for Coloured Petri Nets. (2005) http://wiki.daimi.au.dk/cpntools/cpntools.wiki.
20. Minas, M., Viehstaedt, G.: DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams. In: Proc. IEEE Symp. on Visual Languages (1995) 203–210
21. de Lara, J., Vangheluwe, H., Alfonseca, M.: Meta-Modelling and Graph Grammars for Multi-Paradigm Modelling in AToM^3. Software and System Modeling **3**(3) (2004) 194–209
22. Padberg, J., Urbášek, M.: Rule-Based Refinement of Petri Nets: A Survey. In: Advances in Petri Nets – Petri Net Technology for Communication Based Systems. Vol. 2472 of LNCS. Springer (2003) 161–196
23. Matevska-Meyer, J., Hasselbring, W., Reussner, R.: Software architecture description supporting component deployment and system runtime reconfiguration. In: Proc. Workshop on Component-Oriented Programming (2004)
24. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA: Visual automated transformations for formal verification and validation of UML models. In Proc. Automated Software Engineering. IEEE Press (2002) 267–270
25. Narayanan, A., Karsai, G.: Towards Verifying Model Transformations. In: Proc. Workshop on Graph Transformation and Visual Modeling Techniques. ENTCS, Elsevier Science (2006)
26. Eclipse Generative Modeling Tools (GMT) http://www.eclipse.org/gmt. (2007)
27. Wikipedia: Reconfigurable Computing (2007) [Online; accessed 28-August-2007].
28. Ehrig, H., Hoffmann, K., Padberg, J., Prange, U., Ermel, C.: Independence of Net Transformations and Token Firing in Reconfigurable Place/Transition Systems. In Proc. Conf. on Application and Theory of Petri Nets and Other Models of Concurrency. Vol. 4546 of LNCS, Springer (2007) 104–123