Simulation and Animation of Visual Models of Embedded Systems

A Graph-Transformation-Based Approach Applied to Petri Nets

Hartmut Ehrig, Claudia Ermel, and Gabriele Taentzer

Theoretical Computer Science – Formal Specification Techniques
Technische Universität Berlin, Germany
{ehrig,lieske,gabi}@cs.tu-berlin.de

Summary. Behavior specification techniques like Petri nets provide a visual description of software and embedded systems as basis for behavior validation by simulation. Graph transformation systems can be used as a unifying formal approach to define various visual behavior modeling languages including different kinds of Petri nets, activity diagrams, Statecharts etc., and to provide models with an operational semantics defining simulations of visual models based on graph transformation rules. Moreover, simulation of visual models can be extended by animation which allows to visualize the states of a model simulation run in a domain-specific layout which is closer to the problem domain than the layout of the abstract diagrammatic notation of the specification technique. This kind of model transformation is defined also within the framework of graph transformation, which allows to show interesting properties like semantical correctness of the animation with respect to simulation. In this paper we give an overview of simulation and animation of visual models based on graph transformation and discuss corresponding correctness issues.

As running example we use a high-level Petri net modeling the basic behavior of an elevator. We show how Petri nets are mapped to graph transformation systems, and how the elevator system is extended using an animation view which shows the movements of an elevator cabin between different floors.

1 Introduction

Visual modeling techniques provide an intuitive, yet precise way in order to express and reason about concepts at their natural level of abstraction. The success of visual techniques in computer science and engineering resulted in a variety of methods and notations addressing different application domains and different phases of the development process.

Nowadays two main approaches to visual language definition can be distinguished: grammar-based approaches or meta-modeling. Using graph grammars, visual sentences are described by graphs and visual languages by graph

grammars. Meta-modeling is also graph-based, but uses constraints instead of a grammar to define the 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.

In this paper, we apply the graph grammar-approach to visual language definition [1]. The concrete as well as the abstract syntax of visual notations are described by typed attributed graphs. The type information given in the type graph, captures the definition of the underlying visual alphabet, i.e. the symbols and relations which are available. A visual language is defined by an alphabet and a typed attributed graph grammar, the *VL syntax grammar*.

The behavior of visual models (e.g. Petri nets or Statecharts) can be described by graph transformation as well. Graph rules called *simulation rules* are used to define the operational semantics of the model. Simulation means to show the before- and after-states of an action as diagrams. In Petri nets, for example, a simulation is performed by playing the token game. Different system states are modeled by different markings, and a scenario is determined by a firing sequence of transitions resulting in a sequence of markings. In the formal framework of graph transformation, a scenario corresponds to a transformation sequence where the simulation rules are applied to the graphs representing system states.

However, for validation purposes simulation is not always adequate, since system states are visualized in simulation runs as diagrams. On the one hand, these state graphs abstract from the underlying system domain (as modeling VLs serve many purposes), on the other hand they may become rather complex as complete system states involve auxiliary data and constructs which are necessary to control the behavior but which do not provide insight in the functional behavior of the specific model.

In this paper we extend the simulation for a visual model by *animation views* which allow to define domain-specific scenario animations in the layout of the application domain (cf. [7]). We call the simulation steps of a behavioral model *animation steps* when the states before and after a step are shown in the animation view.

To define the animation view, the VL alphabet first is extended to an animation alphabet by adding symbols representing the problem domain. Second, the set of simulation rules for a specific visual model is extended, resulting in a new set of graph transformation rules, called *animation rules*.

The paper is organized as follows: In Section 2 we introduce our running example, namely the basic behavior of an elevator modeled as high-level Petri net, and present a domain specific animation view of the system as motivation for the further sections. In Section 3 we define the syntax of visual languages and simulation of visual models, and discuss the correctness of simulation. The extension from simulation to animation is presented in Section 4 together with a discussion of the correctness of animation. In the conclusion in Section 5 we discuss related work and implementation issues.

2 Example: An Elevator Modeled as Petri Net

Petri nets are a standard modeling technique for reactive and concurrent systems. In our running example, we use high-level Petri nets to model the basic behavior of an elevator which could be easily extended to a system of concurrent elevators.

In high-level Petri nets, structured tokens are used to represent the data of a system, whereas the Petri net captures the modification of the data. An *Algebraic High-Level net* (AHL net for short) [8] consists of a Place/Transition net for the process description and an algebraic specification $SPEC$ for the data type description describing operations used as arc inscriptions. Tokens are elements of a corresponding $SPEC$-algebra. In order to keep the example simple, the AHL net does not model the control mechanism to call the elevator but only the movements up and down (see Fig. 1 (a)). In general, the system states show in which floor the elevator cabin is, and whether the elevator is moving or not. The data type specification $SPEC$ consists of the specification Nat of natural numbers and the two constants $MaxFloor = 4$ and $MinFloor = 0$. Thus, in Fig. 1 (a), a house with five floors is modeled. The variable f used as arc inscription is holding the number of the current floor of the elevator cabin. The initial marking specifies that in the beginning the elevator is in the ground floor, in the state not moving (token 0 on place not moving).

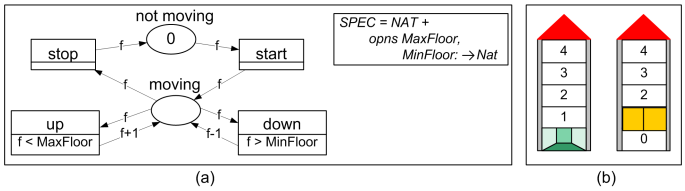

Fig. 1. A Basic Elevator as AHL Net (a) and Snapshots of its Animation View (b)

A domain-specific animation view of the AHL net is illustrated in Fig. 1 (b). We show two snapshots of the animation view according to two possible markings of the net in Fig. 1. The elevator is illustrated as part of a building showing the actual number of floors. The elevator cabin is visualized as box with doors when moving, and as an open box when the elevator is standing, The state not moving is visualized in the left snapshot of Fig. 1 (b), where the cabin is positioned in the respective floor and the doors are open. This snapshot corresponds to the initial state of the AHL net shown in Fig. 1 (a). When the elevator is moving between floors (state moving), the cabin doors are closed. This is shown in the right snapshot of Fig. 1 (b), and corresponds to a token "1" on place moving. We will ensure that the actions that can be performed in the animation view correspond to the transitions in the AHL net.

3 Visual Languages and Simulation of Visual Models

We start with a short introduction to graph transformation which is our basis for modeling visual languages.

3.1 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 similar to Chomsky grammars in formal language theory. On the other hand, graphs can be used to model the states of all kinds of systems and graph transformation to model state changes.

Especially for the application of graph transformation techniques to visual language (VL) modeling, *typed attributed graph transformation systems* have proven to be an adequate formalism [3, 4].

The core of a graph transformation rule $p = (L, R)$ is a pair of graphs called left-hand side L and right-hand side R, and an injective (partial) graph morphism $r : L \to R$. Intuitively, the application of rule p to graph G via a match m from L to G deletes the image $m(L)$ from G and replaces it by a copy of the right-hand side R. A typed graph grammar $GG = (TG, P, S)$ consists of a type graph TG, a set of rules P, and a start graph S. The language of a graph grammar consists of the graphs that can be derived from the start graph by applying the transformation rules arbitrarily often.

Moreover, we use node attributes in our examples, e.g. text for the names of nodes, or numbers for their positions. This allows us to perform computations on attributes in our rules and offers a powerful modeling approach. For flexible rule application, variables for attributes can be used, which are instantiated by concrete values in the rule match.

An example for a graph grammar with node attributes is the visual syntax grammar for AHL nets which is explained in Section 3.2. For a formal theory of typed attributed graph transformation systems we refer to [3].

3.2 Syntax of Visual Languages

A visual language is specified by an alphabet for symbol and link types (the type graph), and a grammar consisting of a start sentence and a set of syntax rules. A visual language VL is then the set of all visual sentences that can be derived by applying syntax rules. The abstract syntax of the alphabet for the AHL net language is shown in Fig. 2 (a).

At the abstract syntax level, there are symbols as Place, Transition, etc. in Fig. 2 which are attributed by datatypes like PlName of type String or Value of type Nat. These abstract syntax items are linked by directed edges. We distinguish arcs running from places to transitions (type PreArc) and arcs from transitions to places (type PostArc). The concrete syntax level is shown

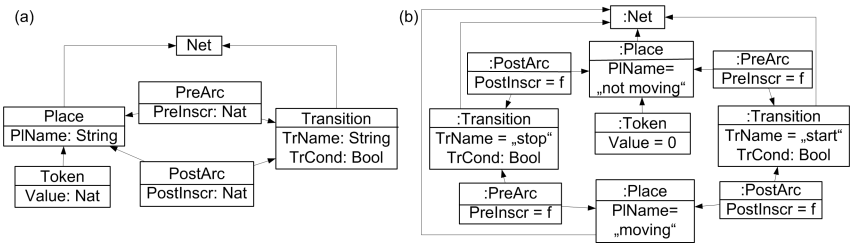

Fig. 2. Abstract Syntax of Visual AHL Net Alphabet (a) and Visual Sentence (b)

in Fig. 7. An example for a visual sentence in its abstract syntax is depicted in Fig. 2 (b). It shows a part of the abstract syntax of the elevator net in Fig. 1.

Syntax rules for AHL nets define the generation of places and transitions, and of arcs between them. A sample syntax rule insPreArc(i) the abstract syntax of which is depicted in Fig. 3, defines the operation for inserting an arc from a place to a transition. Note that graph objects which are preserved by the rule (the place and the transition), occur in both L and R (indicated by equal numbers for the same objects). The newly created arc is inserted together with the arc inscription defined by the rule parameter i (see [6] for more details.) A suitable rule application control has to ensure that rule insPreArc(i) is applied only once for a given place and transition.

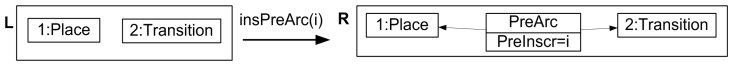

Fig. 3. Sample Abstract Syntax Rule from the AHL Net Syntax Grammar

3.3 Simulation of Visual Models

As discussed already in the introduction, we can use graph grammars not only to define the syntax of visual languages, but also for the simulation of visual models. A visual model is given by the subset of all visual sentences of a given visual language, which correspond to a specific model, like the AHL net model of our elevator in Fig. 1. The behavior of the AHL net in Fig. 1 is given by the well-known token game of AHL nets (see [8]).

In our framework the behavior of a visual model is defined by graph transformation rules based on the abstract syntax of the corresponding visual sentences. These rules are called *simulation rules* in contrast to the syntax rules of the visual language. In order to define the simulation rules we use a suitable encoding of Petri nets into graph grammars (see also [11]). A very natural encoding of nets into grammars regards a net as a graph grammar acting on graphs representing the markings of a net. A Petri net transition is represented as a graph transformation rule consuming the tokens in the pre domain and

generating the tokens in the post domain of the transition. Places are represented as graph nodes which are preserved by the rule. Fig. 4 shows the schema for the translation of an arbitrary AHL net transition to a graph transformation rule $L \rightarrow R$, where L holds the pre domain tokens to be consumed and R contains the post domain tokens to be generated.

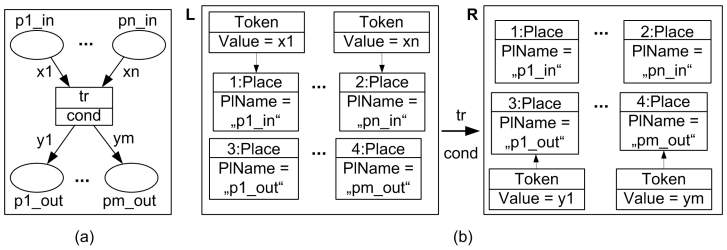

Fig. 4. (a) AHL Net Transition, (b) Corresponding Graph Transformation Rule

For our elevator net in Fig. 1, we have four simulation rules, one for each transition of the net. Three of them are shown in Fig. 5 (the rule stop equals the inverse rule of rule start).

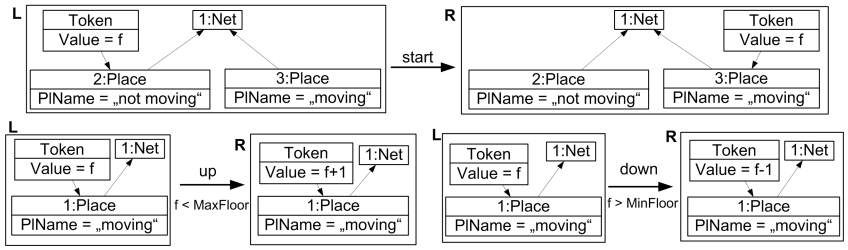

Fig. 5. Simulation Rules for the Elevator Model

If the visual model is given by a specification using a formal specification technique like Petri nets, which have already a well-defined operational semantics, it is important to show that the visual modeling and simulation using graph grammars is correct w.r.t. the given formal specification technique. Otherwise, it is problematic to apply formal analysis and verification techniques. In our case, the formal relationship between AHL nets and graph grammars has been analyzed in [8], where the semantical compatibility of AHL nets and the corresponding attributed graph grammar with simulation rules has been shown as one of the main results.

4 From Simulation to Animation of Visual Models

In order to bridge the gap between the underlying formal, descriptive specification as a Petri net and a domain-specific dynamic visual representation of processes being simulated, we suggest the definition of animation views for Petri nets. Of course, the behavior shown in the animation view has to correspond to the behavior defined in the original Petri net.

4.1 Animation of Visual Models

The elevator net as illustrated in Fig. 1 (a), is a visual sentence of our AHL net language. The animation view of this sentence has been already motivated by Fig. 1 (b), where an elevator cabin moving between the floors of a house is visualized. The concrete and abstract syntax of the alphabet of the animation view is depicted in Fig. 6. It contains a symbol type for the elevator shaft (House), types for the different states of the elevator cabin (Open Cabin and LeftDoor and RightDoor, which are positioned within the corresponding symbol of type Floor). The concrete syntax both for the AHL net elements and for the new elements of the animation view is described by further graphic nodes and by graphical constraints between them specifying the intended layout. We use dotted arrows at the concrete syntax level in order to indicate the graphical constraints, e.g. that a token value is always written *inside* the corresponding Place ellipse, or that the left elevator door is positioned in the left part of the floor rectangle.

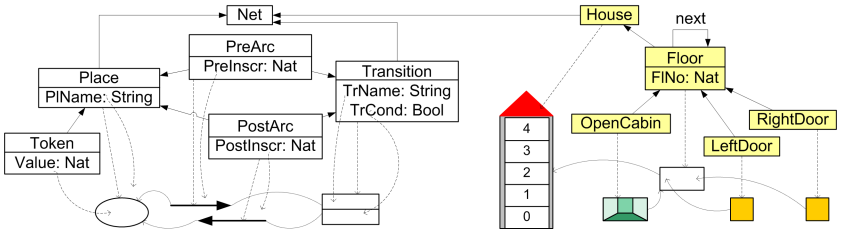

Fig. 6. Animation Alphabet for the Elevator Model

Based on the extension of the visual alphabet for AHL nets in Fig. 2 to the animation alphabet in Fig. 6, we have to extend now the simulation rules in Fig. 5 correspondingly. The resulting animation rules for the transitions start and up are presented in Fig. 7, while those for stop and down are given by the inverse rules for start and up, respectively. By applying first start and then up, the left snapshot of the animation view in Fig. 1 (b) is transformed into the right snapshot.

Moreover, by adding continuous animation operations to animation rules the resulting scenario animations are not only discrete-event steps but can

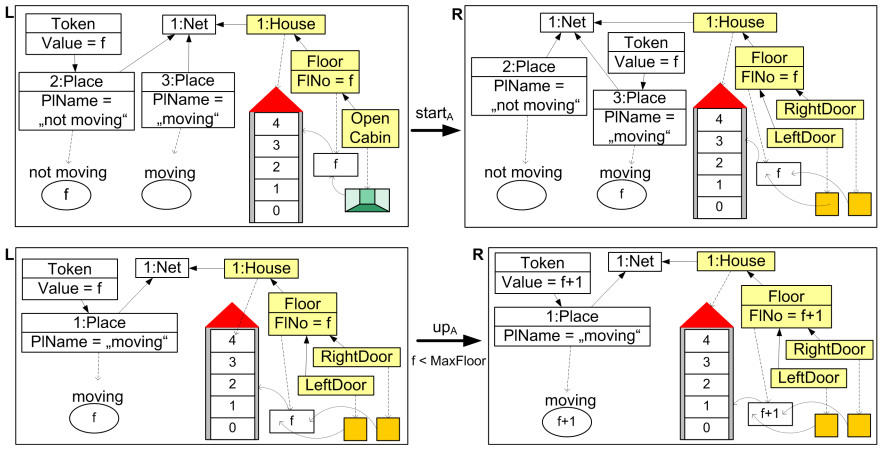

Fig. 7. Resulting Animation Rules for the Elevator Model

show the model behavior in a movie-like fashion. For the elevator, such animation operations are the opening and closing of the cabin doors (for the animation rules start and stop), and the up- and downward movements of the cabin doors (for the animation rules up and down).

4.2 General Construction and Correctness of Animation Rules

Now we present the main ideas for a general construction of animation rules from simulation rules as presented above. This construction is based on a model transformation from simulation to animation, short *S2A* model transformation. This *S2A* model transformation is again given by a typed attributed graph grammar. In general, *S2A* model transformation rules are nondeleting and are applied exactly once at each match to the graphs L and R of each simulation rule. *S2A* rules do not delete elements from the original simulation rules but only add elements from the domain specific extension of the animation alphabet.

For our elevator example, the *S2A* model transformation rules are given in Fig. 8. The tokens of the AHL net model the different locations of the elevator cabin. A token on place not moving corresponds to a cabin graphic with open doors, whereas a token on place moving is visualized by a graphic showing a cabin with closed doors. The position of the graphics relative to the elevator shaft depends on the value of the floor number. The *S2A* rules first add the basic elements for the animation view, i.e. the house symbol containing the elevator shaft. These elements are not changed during animation, and form the *animation background* (rule build house in Fig. 8).

After the generation of the animation background, depending on the marking of a place, an elevator cabin has to be drawn inside the floor corresponding to the token value on the place. For a marked place not moving, the symbol

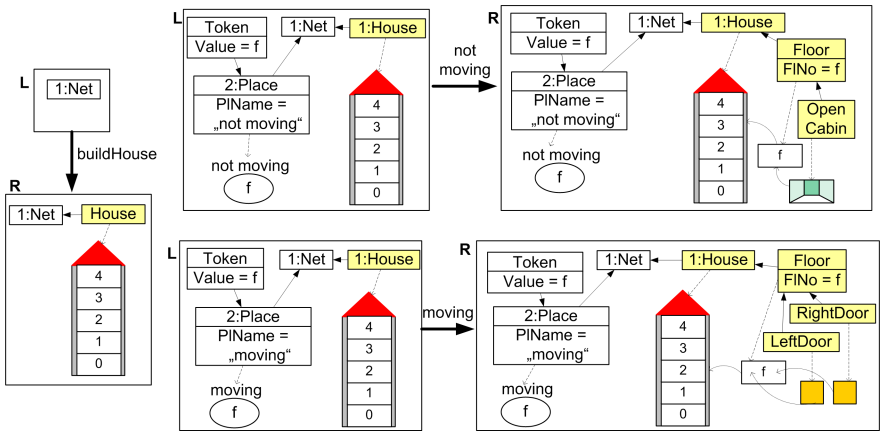

Fig. 8. S2A Rules for the Elevator Model

OpenCabin is generated. For a marked place moving, two doors of type LeftDoor and RightDoor are added (rules moving and not moving in Fig. 8).

In our paper [5], we give a general construction how to apply the *S2A* model transformation rules to simulation rules like those in Fig. 5, in order to obtain animation rules, like those in Fig. 7. Moreover, we present general criteria for the semantical correctness of animation w.r.t. simulation in the following sense, which are satisfied for our running example.

For each simulation step $S_1 \Longrightarrow S_2$ via a simulation rule sr with corresponding animation rule $ar = S2A(sr)$, there are animation models $A_1 = S2A(S_1)$ and $A_2 = S2A(S_2)$ and a corresponding animation step $A_1 \Longrightarrow A_2$ via ar, as depicted in the diagram to the right.

$$\begin{array}{ccc} S_1 & \xrightarrow{S2A} & A_1 \\ {\scriptstyle sr}\downarrow & & \downarrow{\scriptstyle ar} \\ S_2 & \xrightarrow{S2A} & A_2 \end{array}$$

5 Conclusion

In this paper we have given an overview of concepts for simulation and animation of visual models based on the formal theory of typed attributed graph transformation from [3]. This theory allows to show also formal correctness of visual modeling and simulation, as well as correctness of animation w.r.t. simulation (see [8, 5]).

The running example for simulation and animation of Petri nets has been modeled using the GenGED environment for visual language definition, simulation and animation based on graph transformation and graphical constraint solving [9, 1]. Animation operations are defined by coupling the animation rules to animations in SVG format (Scalable vector graphics), and can be viewed using suitable SVG viewers. In contrast to related approaches and tools, e.g. for the animation of Petri net behavior (like the SimPEP-tool for

the animation of low-level nets in PEP [10]), the graph transformation framework offers a basis for a more general formalization of model behavior which is applicable to various Petri net classes and other visual modeling languages. Recently, the new tool environment TIGER [6, 12] has been developed at TU Berlin, supporting the generation of visual environments, based on the one hand on recent MDA development tools integrated in the development environment ECLIPSE [2], and on the other hand on typed attributed graph transformation to support syntax-directed editing and simulation.

References

1. R. Bardohl. A Visual Environment for Visual Languages. *Science of Computer Programming (SCP)*, 44(2):181–203, 2002.
2. Eclipse Consortium. *Eclipse – Version 2.1.3*, 2004. http://www.eclipse.org.
3. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer, 2006.
4. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.
5. H. Ehrig and C. Ermel. From Simulation to Animation: Semantical Correctness of S2A Model and Rule Transformation. in preparation, 2006.
6. K. Ehrig, C. Ermel, S. Hänsgen, and G. Taentzer. Generation of Visual Editors as Eclipse Plug-ins. In *Proc. IEEE/ACM Intern. Conf. on Automated Software Engineering*, IEEE Computer Society, Long Beach, California, USA, 2005.
7. C. Ermel and R. Bardohl. Scenario Animation for Visual Behavior Models: A Generic Approach. *Software and System Modeling: Special Section on Graph Transformations and Visual Modeling Techniques*, 3(2):164–177, 2004.
8. C. Ermel, G. Taentzer, and R. Bardohl. Simulating Algebraic High-Level Nets by Parallel Attributed Graph Transformation. In H.-J. Kreowski, U. Montanari, F. Orejas, G. Rozenberg, and G. Taentzer, editors, *Formal Methods in Software and Systems Modeling*, vol. 3393 of *LNCS*. Springer, 2005.
9. GenGED Homepage. http://tfs.cs.tu-berlin.de/genged.
10. B. Grahlmann. The State of PEP. In *Proc. Algebraic Methodology and Software Technology*, vol. 1548 of *LNCS*. Springer, 1999.
11. H. J. Schneider. Graph Grammars as a Tool to Define the Behaviour of Process Systems: From Petri Nets to Linda. In *Proc. Graph Grammars and their Application to Computer Science*, vol. 1073 of *LNCS* pp. 7–12. Springer, 1994.
12. Tiger Project Team, Technical University of Berlin. *Tiger: Generating Visual Environments in Eclipse*, 2005. http://www.tfs.cs.tu-berlin.de/tigerprj.