CONSTRUCTION AND CORRECTNESS ANALYSIS OF A MODEL TRANSFORMATION FROM ACTIVITY DIAGRAMS TO PETRI NETS

Hartmut Ehrig, Karsten Ehrig, Claudia Ermel, Julia Padberg
Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
email:{ehrig,karstene,lieske,padberg}@cs.tu-berlin.de

Abstract. With the growing importance of model-driven development, the ability of transforming models into well-defined semantic domains becomes a key to automated analysis and verification in the software development process. In this paper we use the concept of typed attributed graph transformation to construct a model transformation from a simple version of activity diagrams to Petri nets. Moreover our approach allows a correctness analysis which shows that this model transformation has functional behavior and is syntactically correct. This is the basis to use well-known analysis and verification techniques of Petri nets also for activity diagrams. The model transformation has been implemented in the TIGER environment developed at TU Berlin.

1. Introduction

Although visual modeling languages (VLs) are becoming increasingly popular in software and systems engineering, their formal underpinning is often not adequate. The Unified Modeling Language (UML) [20], for instance, offers various diagram types for all kinds of modeling tasks for different software engineering development phases. Tool support is offered for the syntactical definition of such diagrams, in form of visual editors, diagram parsers and diagram layouters. But the semantics of the defined diagrams is often ambigous and allows contradicting interpretations of one model. This problem can be solved by constructing a model transformation of the corresponding visual language into a well-known formal semantical domain, like Petri nets [16]. Model transformations thus enable the model designer to perform verification and consistency checks on the translated model [12], to simulate the model behavior, and to generate code.

Precise model transformation descriptions relate the abstract syntax elements of the source language to the elements of the target language. There are two competing approaches to define the abstract syntax of visual languages. One involves graph graph grammars [2] which extend grammar concepts from textual languages to diagrams. The other approach, called *metamodeling*, is based on MOF [14], and uses UML class diagrams to model a visual languages abstract syntax. While class diagrams appear to be more intuitive than graph grammars, they are also less expressive. Therefore, metamodeling also uses context conditions written in the Object Constraint Language (OCL) [13] that help to overcome the weaker expressive power. The advantage of metamodeling 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. But, however intuitive the metamodeling technique is, using it to define the UML is still limited to describing abstract syntax; the problems of diagram representations (concrete syntax) and of defining a formal semantics remain.

Automatically executable model transformations specified by means of graph transformation rules [12] have proven to be an adequate approach for visual languages. The graphical notation of transformation rules supports an intuitive understanding and the rule-based nature allows the flexibility to exchange and modify rules when the requirements for the mappings change. Especially for model transformations of UML behavioral diagrams (e.g. state diagrams) to Petri nets, there exist approaches based on graph transformation [3, 21] with the aim of model validation or verification. Model transformation based on graph transformation allows in principle checking functional properties formally, e.g. showing that the model transformation process terminates [6], and that the computed target model is unique [11]. In our paper [5] we have given an overview of formal concepts for model transformation based on typed attributed graph transformation. In this paper we construct a model transformation from a simple version of activity diagrams into Petri nets. The correctness analysis of this construction

shows that this model transformation has functional behavior and is syntactically correct using graph transformation techniques.

Since activity diagrams in general have only an informal semantics this model transformation allows providing at least simple activity diagrams with a formal semantics via the corresponding Petri net. Moreover, it is the basis to use well-known analysis and verification techniques of Petri nets also for activity diagrams. For this purpose it is essential that the model transformation has functional behavior, and is syntactically correct, i.e. for each activity diagram in the source language we obtain in a finite number of steps in a unique well-defined Petri net. An additional advantage of our approach is that tool support for model transformation by graph transformation is provided by the TIGER environment [19] for the generation of visual modeling tools in ECLIPSE [4]. To execute the model transformation rules and to check functional properties of model transformations, TIGER relies on the graph transformation engine AGG [18, 1].

The paper is structured as follows: Section 2 provides a short introduction of model transformations based on graph transformation. In Section 3 we give the construction for our model transformation by graph transformation where simple UML activity digrams are transformed to Petri nets. In Section 4 we show that our model transformation has functional behavior and is syntactically correct. Finally, we summarize the main points of our approach and outline some future work concerning model transformation to Petri nets in Section 5.

2. Model Transformation by Graph Transformation

For the application of graph transformation techniques to visual language modeling, *typed attributed graph transformation systems* [8, 7] have been proven to be an adequate formalism. A visual language is modeled basically by an *attributed type graph* ATG_{VL} which captures the definition of the underlying visual alphabet, i.e. the symbols and relations which are available. The concept of type graphs corresponds to the use of class diagrams in metamodeling. Visual sentences or diagrams of the VL are given by attributed graphs typed over the type graph.

Visual sentences can be manipulated by graph transformation rules. Roughly spoken a typed attributed graph transformation rule $p = (L \to R)$ consists of a pair of typed attributed graphs L and R (its left-hand and right-hand sides). It fixes a set of variables X and is attributed over the term algebra $T_\Sigma(X)$, meaning that the graphs in the rule may have as attributes values obtained from terms expressed over a signature Σ and variables in X. A typed attributed graph transformation system $GTS = (ATG, P)$ consists of a type graph ATG and a set of typed attributed graph transformation rules P typed over ATG. A direct graph transformation written $G \overset{p,m}{\Longrightarrow} H$, means that the graph G is transformed into the graph H by applying rule $p \in P$ at the match m. Fig. 1 illustrates the application of a rule insertArc at the occurrence marked by a dashed circle in graph G, resulting in the new graph H which differs from G in the newly inserted arc according to R between the two nodes of the types Place and Transition in the occurrence of L in G. In this example, the graph G represents the abstract syntax (without layout information) of a Petri net. The numbers in front of the node inscriptions define the different mappings $L \xrightarrow{p} R$, and $L \xrightarrow{m} G$ (the *match* of the rule in G), $R \xrightarrow{m^*} H$ and $G \xrightarrow{r^*} H$. Roughly spoken, the application of rule p to graph G deletes the image $m(L)$ from G and replaces it by a copy of the right-hand side $m^*(R)$.

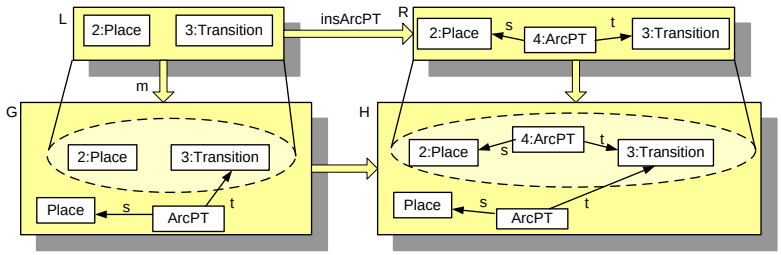

Figure 1: Application of Rule insertArc to Graph G

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. A rule p may be extended by a set of *negative application conditions (NACs)* [10, 7]. A NAC allows the rule application only if some context specified in an additional NAC-graph N does *not* occur in the current graph G.

A graph transformation $G_0 \stackrel{*}{\Longrightarrow} G_n = G_0 \stackrel{p_1,m_1}{\Longrightarrow} ... \stackrel{p_n,m_n}{\Longrightarrow} G_n$ in GTS is a sequence of direct graph transformations such that all rules p_i are from P. Briefly, we also write for $G \stackrel{p,m}{\Longrightarrow} H$ just $G \stackrel{p}{\Longrightarrow} H$.

For a detailed formal introduction to the topic of typed attributed graph transformation, the reader is referred to [7, 8].

In order to further restrict the visual sentences of a VL to valid visual models, a *syntax graph grammar GG* is defined. It consists of a start graph and a set of language-generating graph transformation rules that describe editing operations leading to the construction of valid visual models only. The rule insertArc in Fig. 1 is an example for a syntax rule from the syntax grammar defining the VL of Petri nets. A complete VL specification $VLspec = (ATG_{VL}, GG)$ is given by a VL alphabet ATG_{VL} together with a syntax grammar GG. Please note that in metamodeling, the restriction of the set of possible metamodel instances to valid models is done in a declarative way by using OCL constraints.

Not only visual language specification but also model transformations between visual languages can be constructed by graph transformation rules. For this purpose the abstract syntax graph of a source model is transformed by applying transformation rules resulting in the abstract syntax graph of the target model.

The abstract syntax graphs of the source models can be specified by all (or a subset of) instance graphs over a type graph ATG_S. Correspondingly, the abstract syntax graphs of the target models are specified by all (or a subset of) instance graphs over a type graph ATG_T. A model transformation based on graph transformation is defined by an attributed graph transformation system $GTS = (ATG, P)$ consisting of an attributed type graph ATG and a set of model transformation rules P typed over ATG, where both type graphs ATG_S and ATG_T have to be subgraphs of type graph ATG (see Fig. 2). The model transformation starts with graph AG_S typed over ATG_S. As ATG_S is a subgraph of ATG, AG_S is also typed over ATG. During the model transformation process the intermediate graphs are all typed over ATG. Please note that this type graph may contain not only ATG_S and ATG_T, but also additional types and relations which are needed for the transformation process.

After application of the model transformation rules P the resulting graph AG_n is typed over ATG, but not yet over the type graph ATG_T of the target language. In order to delete all items in AG_n which are not typed over ATG_T we can either provide corresponding deletion rules or apply a restriction construction, which deletes all these items in one step. In this paper we use the restriction construction. The model transformation process is visualized in Fig. 2, where the data types for node and edge attributes are preserved during the process.

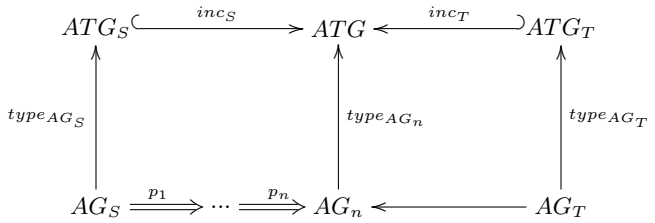

Figure 2: Typing in the Model Transformation Process

The model transformation $MT : VL_S \rightarrow VL_T$ is executed in this way: for each $AG_S \in VL_S$ first the rules P of the typed attributed graph transformation system $GTS = (ATG, P)$ are applied leading to a graph AG_n typed over ATG and then AG_n is restricted to the type graph ATG_T resulting in AG_T.

For general correctness requirements of model transformations based on graph transformation we refer to [5], for our model transformation from activity diagrams to Petri nets they will be discussed and analysed in Section 4.

3. Construction of a Model Transformation from Activity Diagrams to Petri Nets

In this section we present the construction for a model transformation between a simple version of UML *Activity Diagrams* given as the source language and *Petri Nets* given as the target language.

3.1 The Source and Target Languages

The source language alphabet for simple activity diagrams contains two kinds of symbol types, *activities* and *next-relations*. Next-relations begin and end at activities. Activities can be of different kinds, i.e. simple activities, start and end nodes as well as decision nodes. Simple activities are usually inscribed by their names. Moreover, next-relations may have inscriptions which are used to formulate conditions for decisions. The activity kind, the name and the conditions are given as attributes of the corresponding symbol types. The type graph of the activity diagram language is shown in the left-hand side of Fig. 3, where the numbers at the edge ends define multiplicities.

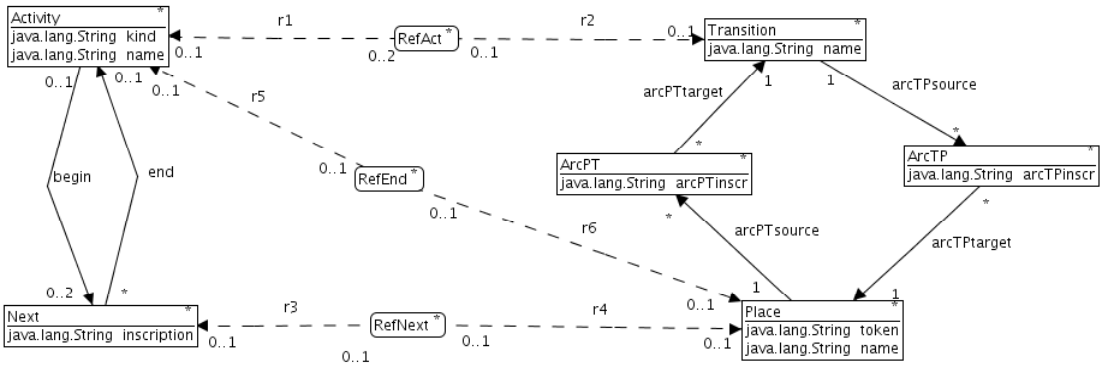

Figure 3: Model Transformation Type Graph *ATG*

Fig. 4 shows a sample source activity diagram for the model transformation, typed over the activity diagram type graph, both in its concrete and abstract syntax. The diagram describes the activities for the dispatching of a train. It is a slightly simplified version of the activity diagram given in [17].

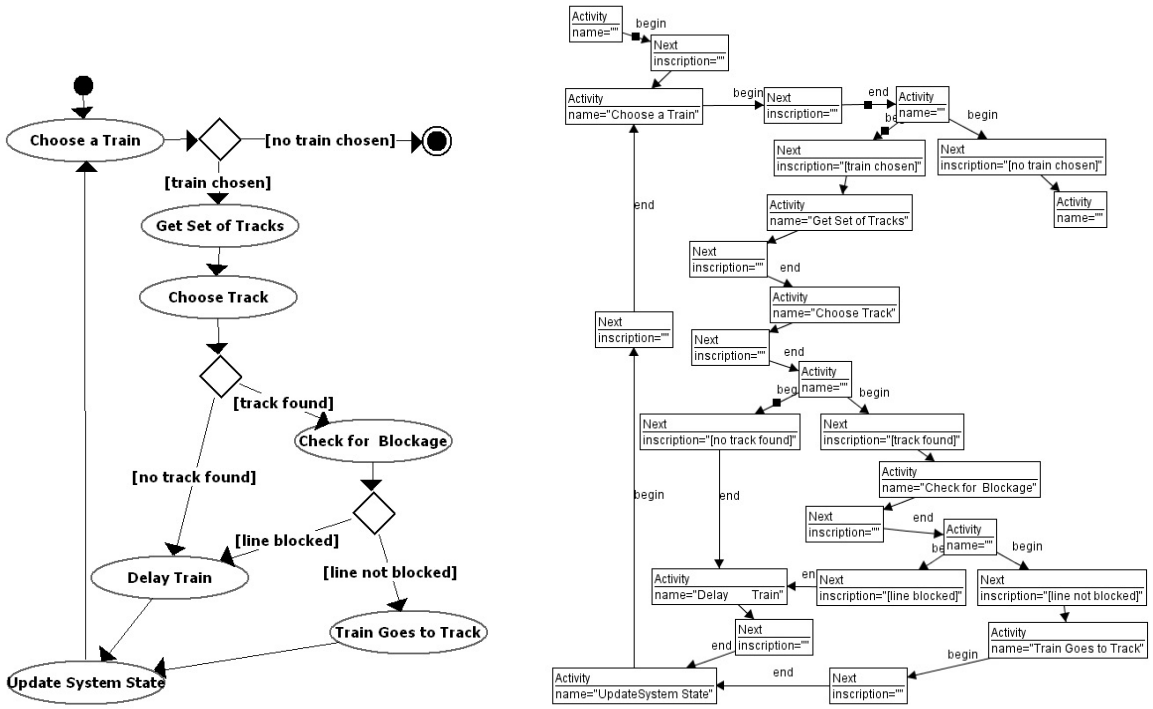

Figure 4: Activity Diagram for the Train Dispatcher: Concrete and Abstract Syntax

The first activity of the dispatcher is to choose a train, otherwise the whole dispatching is finished. Then after computing the feasable tracks, the dispatcher chooses one. If there is no feasable track, the train is delayed at the current location. Otherwise there is a check for the blockage of the line. If the

line is blocked, the train is delayed at the current location as well. Else the train proceeds to the chosen location. After the system state has been updated, the dispatching process may start again.

The syntax rules for simple activity diagrams decide important aspects of the visual language, e.g. the number of start and end activities which are allowed in one diagram. Our variant of simple activity diagrams allows only one start and one end activity. This is realized in the syntax grammar (see Fig. 5) by defining an activity diagram as start graph which consists of exactly one start and one end activity, connected by a next-relation. As none of the syntax rules adds or deletes start or end activities, their number will always be fixed to one each. The syntax rules shown in Fig. 5 allow the insertion of simple activities or decisions. There the cases are treated that a decision either leads to two simple states or to one simple state and to the final state. Moreover, two simple activities can be joined, and an activity can be relinked by a next-relation to another activity. The NAC of rule *JoinActivities* ensures that two branches of one decision cannot be merged. Please note that we can apply a rule only if the transformation does not leave dangling edges. Hence, rule *JoinActivities* is applicable only if the activity that is deleted has exactly one predecessor and one successor activity.

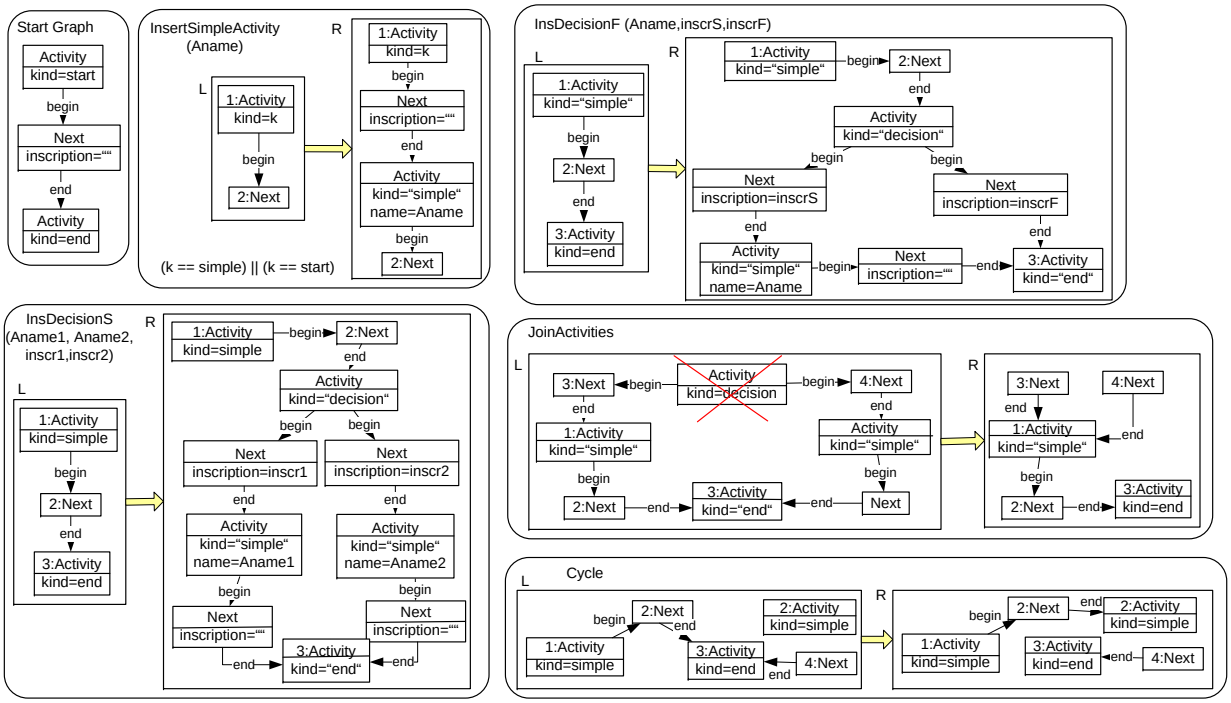

Figure 5: Syntax Grammar for Simple Activity Diagrams

The activity diagram for the train dispatcher has been created by applying the syntax rules *InsertSimpleActivity("Choose Train"), InsDecisionF("Get Set of Tracks", "[train chosen]","[no train chosen]"), InsertSimpleActivity("Choose Track"), InsDecisionS("Delay Train","Check for Blockage","[no track found]", "[track found]"), InsDecisionS("Delay2","Train Goes to Track","[line blocked]", "[line not blocked]"), JoinActivities* (which is applied to the simple activities *Delay Train* and *Delay2*), *InsSimpleActivity("Update System State"), InsSimpleActivity("Up2"),* (after activity *Train Goes to Track*), *JoinActivities* (applied to *Update System State* and *Up2*), and, finally, *Cycle* (applied to *Update System State* and *InsertSimpleActivity("Choose Train")*.

The alphabet for the Petri net target language contains the symbol types Place and Transition for the Petri net nodes, ArcPT for Petri net arcs from a place to a transition and ArcTP for arcs from a transition to a place. The arcs are connected to the respective source and target nodes by links of type arcPTsource, arcPTtarget, arcTPsource and arcTPtarget. Attributes are used for names of places and transitions and for the arc inscriptions. The type graph of the Petri net language is shown in the right-hand side of Fig. 3.

Petri net *places* model passive system parts (e.g., buffers and files), whereas *transitions* describe process activities. Thus, our model transformation maps activities to transitions and next-relations to places in between. The places can hold at most one token each, thus the token just shows how far

the process has reached. Petri nets of this special kind are called *elementary* or *condition-event nets*. Decision activities are translated to two transitions, one for each possible decision branch. To each of these transitions, an additional predomain place is assigned. The marking of this place models the evaluation of the corresponding guard to "true".

Fig. 6 shows the result of the model transformation transforming the activity diagram in Fig. 4 to a Petri net in the concrete syntax (the left-hand side) and in its abstract syntax (the right-hand side).

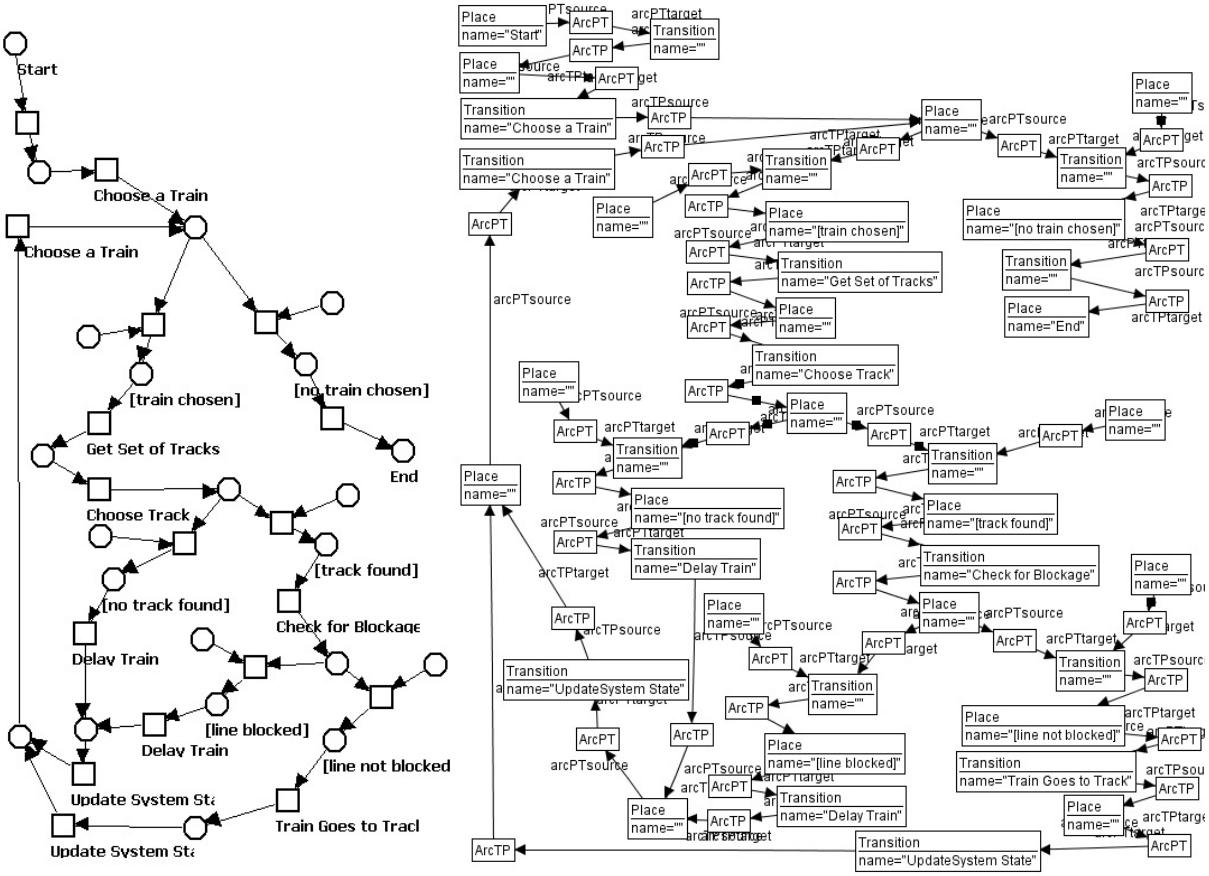

Figure 6: Petri Net for the Train Dispatcher: Concrete and Abstract Syntax

The model transformation type graph *ATG* (the complete graph shown in Fig. 3) is defined by the union of the source and target language alphabets plus two reference nodes. The adjacent arcs of these nodes connect the corresponding symbol types of both alphabets, i.e. activities to transitions and next-relations to places.

3.2 The Model Transformation Rules

The model transformation rules are defined by a graph transformation system typed over *ATG* in Fig. 3. Starting with the start graph in Fig. 4, the consecutive application of the model transformation rules results in the abstract syntax graph of the target diagram.

The following screenshots of the model transformation rules in Fig. 7 contain three graphs each (a negative application condition NAC, the left-hand side LHS and the right-hand side RHS of the rule). The NAC is present for each model transformation rule to ensure that Petri net elements are generated only once for each element of the activity diagram. The model transformation rules are structured in two layers for controlled rule application. Layer 0 contains the rules which create nodes: *createStart*, *createSimpleAct*, *createEnd*, *createEndPlace* and *createDecision*; the remaining rules *createEndArc*, and *createArc* insert arcs between nodes are in layer 1. Starting with layer 0, rules of the current layer are applied as long as possible. After the termination of all rules in the current layer the transformation continues with the next layer.

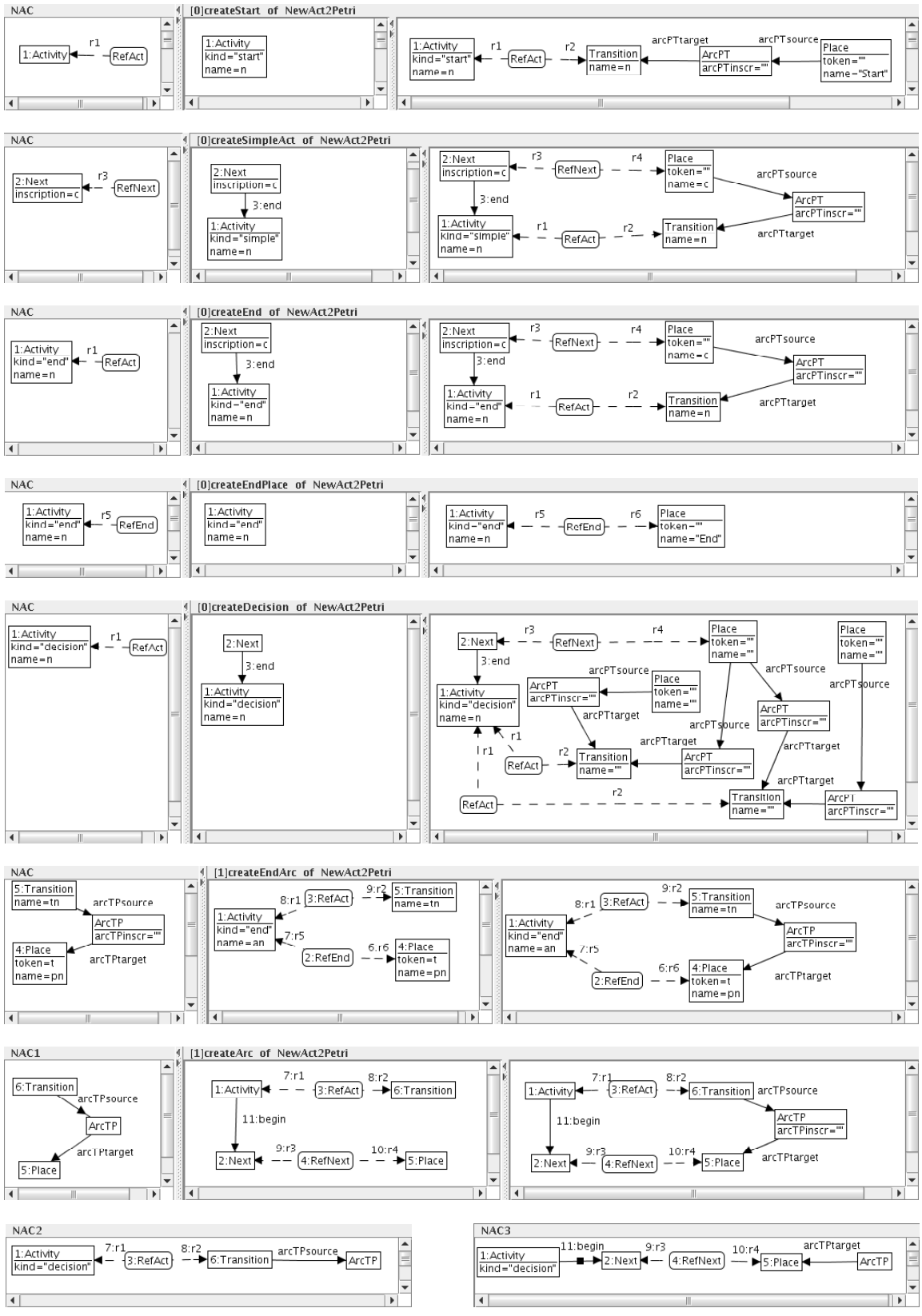

Figure 7: Model Transformation Rules to transform Activity Diagrams to Petri Nets

The first rule *createStart* of layer 0 translates the start activity into a place named "Start" connected to a transition. Simple and end activities are translated by the rules *createSimpleAct* and *createEnd*

to a *Place* and a *Transition* connected via an *ArcPT*. In addition, the final *Place* corresponding to the end activity is created by the rule *createEndPlace*. The rule *createDecision* translates a decision activity into the corresponding Petri subnet, containing two input *Places* to be used for controlling the decision application e.g. during simulation. Finally, layer 1 inserts the corresponding *ArcTPs* via the rules *createEndArc* and *createArc*. The rule *createArc* contains two additional NACs (NAC2,NAC3) depicted in Fig. 7 which ensure that the corresponding *ArcTP* may only be inserted once between the corresponding *Place* and *Transition* of the Petri decision subnet.

To delete all source and reference items, we use a restriction construction, resulting in a target graph typed over the target language.

4. Correctness Analysis of the Model Transformation

As pointed out in the introduction we want to show that our model transformation $MT : VL_S \to VL_T$ constructed in Section 3 is correct in the following sense:

The model transformation construction terminates, is confluent and syntactically correct. Moreover, we discuss briefly semantical correctness.

4.1 Termination

In [6] we have presented termination criteria for layered graph transformation systems, which have been extended in [7] to layered typed attributed graph transformation systems. These criteria have been verified for our model transformation in Section 3 using the termination checker in AGG.

4.2 Confluence

Confluence of model transformation means that the rules in each layer are confluent. In order to show confluence it is sufficient to have termination and local confluence, i.e. for each pair of direct transformations $H_1 \overset{p_1,m_1}{\Longleftarrow} G \overset{p_2,m_2}{\Longrightarrow} H_2$ there exists a graph X and a pair of transformations $H_1 \overset{*}{\Longrightarrow} X \overset{*}{\Longleftarrow} H_2$. Termination and local confluence implies functional behavior of the model transformation, i.e. for each $AG_S \in VL_S$ there is a unique result AG_T s.t. $MT(AG_S) = AG_T$. For the model transformation rules in Fig. 7 we are able to show that for each layer there are no essential critical pairs. This means that for each pair of direct transformations $H_1 \overset{p_1,m_1}{\Longleftarrow} G \overset{p_2,m_2}{\Longrightarrow} H_2$ we have either $G \overset{p_1,m_1}{\Longrightarrow} H_1 = G \overset{p_2,m_2}{\Longrightarrow} H_2$ or the pair is parallel independent, such that the *Local-Church Rosser Theorem* implies local confluence. We have to admit, however, that a correctness proof for the calculation of critical pairs for rulse with NACs in AGG is still missing. If there would be critical pairs one would have to show strict confluence of these critical pairs in order to obtain local confluence (see [6, 7]). This result, however, has been shown up to now only for rules without NACs.

4.3 Syntactical Correctness

It remains to show that the model transformation is syntactically correct, i.e. for each $AG_S \in VL_S$ we have $MT(AG_S) = AG_T \in VL_T$. Together with termination and confluence this implies that MT is a (total) function $MT : VL_S \to VL_T$. By restriction construction AG_T is typed already over AGT_T. But not all graphs G typed over AGT_T are Petri nets. We need to show in addition the constraint that there is at most one arc from each place to each transition and from each transition to each place. This can be concluded from the fact that in $AG_S \in VL_S$ there is at most one begin edge from an activity to a next node, and at most one end edge from a next to an activity node and – due to the NACs – each of the rules in Fig. 7 can be applied at most once for the same pair of activity and next nodes.

4.4 Semantical Correctness

For our source and target language VL_S and VL_T we have given no explicit semantics up to now. But for the target language VL_T of Petri nets there is a well-defined formal semantics given by the token game and the corresponding marking graph. This allows to define for each activity diagram $AG_S \in VL_S$ as semantics the marking graph of $AG_T = MT(AG_S)$. Moreover, it would make sense to extend the model transformation by constructing a suitable initial marking for $AG_T = MT(AG_S)$ s.t. the semantics might be given by the reachability graph. In both case we have trivial semantical correctness of the model transformation by construction.

On the other hand an explicit operational semantics for activity diagrams could be given by a simulation graph grammar based on the abstract syntax graph of activity diagrams. Similarily the token game of Petri nets can be expressed by a simulation graph grammar. In this case semantical correctness of the model transformation would mean that for each simulation step in AG_S there is a corresponding simulation step in AG_T. For basic ideas concerning semantical correctness criteria we refer to our paper [5].

5. Conclusion

In this paper we have shown how to construct a model transformation from activity diagrams to Petri nets using typed attributed graph transformation. The source language, a simple version of UML activity diagrams, is defined by a generating attributed graph grammar typed over ATG_S, the target language of Petri nets by an attributed type graph ATG_T with constraints and the model transformation by an attributed graph transformation system typed over an attributed type graph ATG which includes ATG_S and ATG_T. Moreover, we have verified that our model transformation has functional behavior and is syntactically correct. In our papers [5, 9] we have given already an overview concerning formal concepts for model transformations based on typed attributed graph transformation and how to use the AGG and TIGER environment to support the implementation of model transformation. These techniques have been applied for our model transformation in this paper.

There are several aspects how to extend and use this kind of model transformations. First of all we could consider more general activity diagrams in the sense of UML 2.0 by extending the generating graph grammar and the source attributed type graph ATG_S. Secondly we may generate Petri nets with initial marking corresponding to the begin of an activity and the choice for decisions, where the choice is triggered by the environment. The choice of Petri nets as target language allows the definition of the semantics of the source language via the model transformation and the semantics of the target language. In a similar way we can use well-known analysis and verification techniques for Petri nets to analyse and verify activity diagrams or other semiformal source languages which usually lack explicit analysis and verification techniques. So, in principle it is possible to use one of the many Petri net tools (see [15]) for invariant or deadlock analysis, model checking or code generation among others.

References

[1] AGG Homepage. `http://tfs.cs.tu-berlin.de/agg`.

[2] R. Bardohl, G. Taentzer, M. Minas, and A. Schürr. Application of Graph Transformation to Visual Languages. In 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.

[3] J. de Lara and H. Vangheluwe. Computer Aided Multi-Paradigm Modelling to Process Petri-Nets and Statecharts. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. 1st Int. Conf. on Graph Transformation (ICGT 2002)*, volume 2505 of *LNCS*, pages 239–253. Springer, 2002.

[4] Eclipse Consortium. *Eclipse – Version 2.1.3*, 2004. `http://www.eclipse.org`.

[5] H. Ehrig and K. Ehrig. Overview of Formal Concepts for Model Transformations based on Typed Attributed Graph Transformation. In *Proc. International Workshop on Graph and Model Transformation (GraMoT'05)*, ENTCS, Tallinn, Estonia, September 2005. Elsevier Science.

[6] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. Termination criteria for model transformation. In M. Wermelinger and T. Margaria-Steffen, editors, *Proc. Fundamental Approaches to Software Engineering (FASE)*, volume 2984 of *LNCS*, pages 214–228. Springer, 2005.

[7] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer, 2006. to appear.

[8] H. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graph transformation. In F. Parisi-Presicce, P. Bottoni, and G. Engels, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04), Rome, Italy*, volume 3256 of *LNCS*. Springer, 2004.

[9] K. Ehrig, C. Ermel, and S. Hänsgen. Towards Model Transformation in Generated Eclipse Editor Plug-Ins. In *Proc. International Workshop on Graph and Model Transformation (GraMoT'05)*, ENTCS, Tallinn, Estonia, September 2005. Elsevier Science.

[10] A. Habel, R. Heckel, and G. Taentzer. Graph Grammars with Negative Application Conditions. *Special issue of Fundamenta Informaticae*, 26(3,4):287–313, 1996.

[11] R. Heckel, J. Küster, and G. Taentzer. Confluence of Typed Attributed Graph Transformation with Constraints. In A. Corradini, H. Ehrig, H.-J. Kreowski, and Rozenberg. G., editors, *Proc. of 1st Int. Conference on Graph Transformation*, volume 2505 of *LNCS*. Springer, 2002.

[12] R. Heckel, J. Küster, and G. Taentzer. Towards Automatic Translation of UML Models into Semantic Domains . In H.-J. Kreowski, editor, *Proc. of APPLIGRAPH Workshop on Applied Graph Transformation (AGT 2002)*, pages 11 – 22, 2002.

[13] Object Management Group. *UML 2.0 OCL Specification*, 2003. `http://www.omg.org/docs/ptc/03-10-14.pdf`.

[14] Object Management Group. *Meta-Object Facility (MOF), Version 1.4*, 2005. `http://www.omg.org/technology/documents/formal/mof.htm`.

[15] Petri Nets World. *Petri Net Tools and Software*, 2005. `http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/`.

[16] W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, 1985.

[17] M. Rondon and F. Gomide. Line block analysis in railway dispatch and simulation systems. In E. Schnieder and U. Becker, editors, *Proc. 9th IFAC Symposium on Transportation Systems*, pages 405–409, 2000.

[18] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In J. Pfaltz, M. Nagl, and B. Boehlen, editors, *Application of Graph Transformations with Industrial Relevance (AGTIVE'03)*, volume 3062 of *LNCS*, pages 446 – 456. Springer, 2004.

[19] Tiger Project Team, Technical University of Berlin. *Tiger: Generating Visual Environments in Eclipse*, 2005. `http://www.tfs.cs.tu-berlin.de/tigerprj`.

[20] *Unified Modeling Language: Superstructure – Version 2.0*, 2004. Revised Final Adopted Specification, ptc/04-10-02, `http://www.omg.org/cgi-bin/doc?ptc/2004-10-02`.

[21] D. Varró, G. Varró, and A. Pataricza. Designing the Automatic Transformation of Visual Languages. *Journal Science of Computer Programming*, 44(2):205–227, 2002.