Electronic Communications of the EASST
Volume X (2008)

Proceedings of the Third Workshop on Petri Nets and Graph Transformations (PNGT 2008)

Implementing Petri Net Transformations using Graph Transformation Tools

Enrico Biermann, Claudia Ermel, Tony Modica and Peggy Sylopp

14 pages

Guest Editors: Paolo Baldan, Barbara König
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Implementing Petri Net Transformations using Graph Transformation Tools

Enrico Biermann, Claudia Ermel, Tony Modica and Peggy Sylopp

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
formalnet@cs.tu-berlin.de

Abstract: Petri net transformations have been defined formally in the abstract framework of adhesive HLR categories, which allows rule-based rewriting of graph-like structures, similar to graph transformation. In this paper we discuss differences between Petri net rewriting and graph rewriting which makes it necessary to add checks and conditions when implementing Petri net transformations using an existing graph transformation tool like AGG. The extensions concern the preservation of Petri net transition firing behavior and the mapping of markings. As a running example, we present the RON environment, a visual editor, simulator and net transformation tool for reconfigurable Petri nets which has been developed as a plug-in for ECLIPSE based on the graph transformation engine AGG.

Keywords: Petri nets, net transformation, graph transformation, visual editor, reconfigurable Petri nets, Petri net tool

1 Introduction

Modeling the adaption of a system to a changing environment becomes more and more important. Application areas cover e.g. computer-supported cooperative work, multi agent systems, dynamic process mining or mobile networks. One approach to combine formal modeling of dynamic systems and controlled model adaption are *Reconfigurable Petri nets* [HEP07, EHP+07]. The main idea is the stepwise reconfiguration of place/transition nets by given net transformation rules [EHP+08, EHP06]. Think of these rules as replacement systems where the left-hand side is replaced by the right-hand side while preserving a context. This approach increases the expressiveness of Petri nets and allows in addition to the well known token game a formal description of structural changes.

Since Petri nets can be considered as bipartite graphs the concept of graph transformations [EEPT06] can be applied in principle to define also transformations of Petri nets [EP04]. Unfortunately, there are some differences between graph morphisms and our notion of Petri net morphisms which lead to different transformation behaviors. The aim of this paper is to identify the differences in both approaches and to offer solutions that allow a simulation of Petri net transformations by graph transformation. Note that other graph transformation-based simulation tools also deal with structural transformations of Petri nets (e.g. [LVA04]), but they do not guarantee the preservation of firing behavior in Petri net transformations. Similarly, in the approach of Llorens and Oliver [LO04], rewriting of Petri nets by graph transformation rules is used for

the reconfiguration of nets, but does not preserve the firing behavior. Instead, reconfiguration focuses on the modification of the flow relation, i.e. only arcs are added / deleted or reconnected.

Our paper is structured as follows: In Section 2 we review the formal definitions of graph and net morphisms, as well as of graph and net transformations. Section 3 outlines the main differences between both transformation formalisms, dealing in particular with markings and the preservation of transition-firing behavior. We present solutions that allow a simulation of Petri net transformations by graph transformation. Finally, in Section 4, we sketch a prototypical implementation of a visual Petri net simulation and transformation tool, which converts Petri nets and net transformation rules to graphs and graph transformation rules and uses the graph transformation engine AGG [AGG] to compute net transformations.

2 Graph and Petri Net Transformation

2.1 Graph Transformation

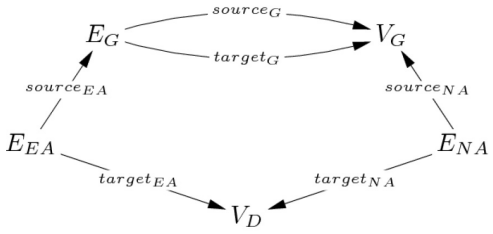
The research area of graph transformation [EEPT06, Roz97] dates back to the early seventies. Methods, techniques, and results from the area of graph transformation have already been studied and applied in many fields of computer science such as formal language theory, pattern recognition and generation, software engineering, concurrent and distributed system modeling, model transformation, and visual language design.

For our aim to represent Petri net transformations by graph transformation we need typed, attributed graphs and graph morphisms which are especially suited for visual language modeling: a visual language (VL) is modeled by an attributed 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 attributed graphs typed over (i.e. conforming to) the type graph. Such a VL type graph corresponds closely to a meta model.

The following definitions are taken from [EEPT06]. The key idea is to model an attributed graph with node and edge attribution, where the underlying graph G is a new kind of graph, called an E-graph, which allows attribution edges not only from graph nodes to attribute nodes but also from graph edges to attribute nodes. This new kind of attributed graph, combined with the concept of typing, leads to a category $\mathbf{AGraphs}_{ATG}$ of attributed graphs typed over an attributed type graph ATG. This category proved to be an adequate formal model not only for various applications in software engineering and visual languages but also for the internal representation of attributed graphs in our graph transformation tool AGG [TER99, AGG].

Definition 1 (E-graph and E-graph morphism) An *E-graph* G with $G = (V_G, V_D, E_G, E_{NA}, E_{EA}, (source_j, target_j)_{j\in\{G,NA,EA\}})$ consists of the sets
V_G and V_D, called the graph and data nodes (or vertices), respectively;

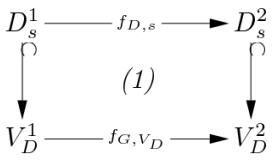
E_G, E_{NA}, and E_{EA} called the graph edges, node attribute edges, and edge attribute edges;
$src_G : E_G \to V_G$, $tar_G : E_G \to V_G$, source and target functions for graph edges;
$src_{NA} : E_{NA} \to V_G$, $tar_{NA} : E_{NA} \to V_D$, source and target functions for node attribute edges;
$src_{EA} : E_{EA} \to E_G$, $tar_{EA} : E_{EA} \to V_D$, source and target functions for edge attribute edges.



Consider the E-graphs G^1 and G^2 with $G^k = (V_G^k, V_D^k, E_G^k, E_{NA}^k, E_{EA}^k, (source_j^k, target_j^k)_{j\in\{G,NA,EA\}})$ for $k = 1,2$. An *E-graph morphism* $f : G^1 \to G^2$ is a tuple $(f_{V_G}, f_{V_D}, f_{E_G}, f_{E_{NA}}, f_{E_{EA}})$ with $f_{V_i} : V_i^1 \to V_i^2$ and $f_{E_j} : E_j^1 \to E_j^2$ for $i \in \{G,D\}$, $j \in \{G,NA,EA\}$ such that f commutes with all source and target functions, for example $f_{V_G} \circ source_G^1 = source_G^2 \circ f_{E_G}$.

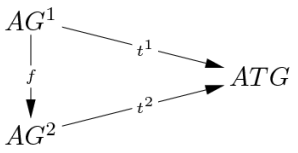
An attributed graph is an E-graph combined with an algebra over a data signature *DSIG*. In the signature, we distinguish a set of attribute value sorts. The corresponding carrier sets in the algebra can be used for the attribution.

Definition 2 (Attributed graph and attributed graph morphism) Let $DSIG = (S_D, OP_D)$ be a data signature with attribute value sorts $S'_D \subseteq S_D$. An *attributed graph* $AG = (G,D)$ consists of an E-graph G together with a $DSIG$-algebra D such that $\dot{\bigcup}_{s\in S'_D} D_s = V_D$.
For two attributed graphs $AG^1 = (G^1, D^1)$ and $AG^2 = (G^2, D^2)$, an *attributed graph morphism* $f : AG^1 \to AG^2$ is a pair $f = (f_G, f_D)$ with an E-graph morphism $f_G : G^1 \to G^2$ and an algebra homomorphism $f_D : D^1 \to D^2$ such that (1) commutes for all $s \in S'_D$, where the vertical arrows are inclusions:



For the typing of attributed graphs, we use a distinguished graph attributed over the final $DSIG$-algebra Z (see [EEPT06]). This graph defines the set of all possible types.

Definition 3 (Typed attributed graph and typed attributed graph morphism) Given a data signature *DSIG*, an *attributed type graph* is an attributed graph $ATG = (TG, Z)$, where Z is the final $DSIG$-algebra.
A *typed attributed graph* (AG, t) over ATG consists of an attributed graph AG together with an attributed graph morphism $t : AG \to ATG$.
A *typed attributed graph morphism* $f : (AG^1, t^1) \to (AG^2, t^2)$ is an attributed graph morphism $f : AG^1 \to AG^2$ such that $t^2 \circ f = t^1$:



As an example for typed, attributed graphs we refer to Def. 11 in Section 3.

The main idea of graph transformation is the rule-based modification of graphs where each application of a graph transformation rule leads to a graph transformation step. The core of a graph transformation rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ is a pair of graphs (L,R), called left-hand side (LHS) and right-hand side (RHS), an interface $K = L \cap R$ and an two injective graph morphisms $L \xleftarrow{l} K$ and $K \xrightarrow{r} R$ embedding the interface in the left- and right-hand sides. Applying the rule p means to find a match of L in the source graph G and to replace this matched part by R, thus

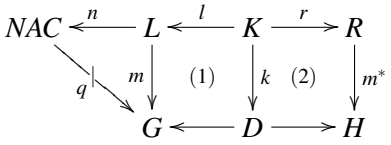
transforming the source graph into the target graph of the graph transformation.

Intuitively, the application of rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ 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 $m^*(R)$. 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 4 (Typed attributed graph rule) Given an attributed type graph ATG with a data signature $DSIG$, a typed attributed graph rule, or "rule" for short, $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ consists of typed attributed graphs L, K, and R with a common $DSIG$-algebra $T_{DSIG}(X)$, the $DSIG$-termalgebra with variables X, and injective typed attributed graph morphisms $l : K \to L$, $r : K \to R$, where the $DSIG$-part of l and r is the identity on $T_{DSIG}(X)$.

For the definition of graph transformations we need pushouts in the category $\mathbf{AGraphs_{ATG}}$, the existence of which is shown in [EEPT06].

Definition 5 (Typed attributed graph transformation) Given a rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ as defined above and a typed attributed graph G with a typed attributed graph morphism $m : L \to G$, called match, a direct typed attributed graph transformation, or "direct graph transformation" for short, $G \overset{p,m}{\Longrightarrow} H$ from G to a typed attributed graph H is given by the following double pushout (DPO) diagram in the category $\mathbf{AGraphs_{ATG}}$, where (1) and (2) are pushouts. Informally, a pushout in a category **CAT** is a gluing construction of two objects over a specific interface. The rule r may be extended by a set of *negative application conditions* (NACs) [HHT96, EEPT06]. A match $L \xrightarrow{m} G$ satisfies a NAC with the injective NAC morphism $n : L \to NAC$, if there is no injective graph morphism $NAC \xrightarrow{q} G$ with $q \circ n = m$.



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

Note that for flexible rule application, variables for attributes can be used in rules, which are instantiated by concrete values by the rule match. Moreover, Boolean attribute conditions (expressions over attributes) may be combined with a rule to control rule application.

2.2 Petri Net Transformation

Reconfigurable place/transition systems are Petri nets with initial markings and a set of rules which allow the modification of the net during runtime in order to adapt the net to new requirements of the environment. Thus, not only the successor marking can be computed but also the structure can be changed by rule application to obtain a new P/T-system that is more appropriate with respect to some requirements of the environment. Moreover these activities can be interleaved [EHP+07]. For rule-based transformations of P/T-systems we use the framework of net transformations from [EHP+08, EHP+07].

Definition 6 (P/T net, P/T system) A *P/T-net* is given by $PN = (P, T, pre, post)$ with sets of places P and transitions T, and pre- and post domain functions $pre, post : T \to P^{\oplus}$, where $P^{\oplus}$ is the free commutative monoid over the set P of places with binary operation $\oplus$ (e.g. the monoid notation $M = 2p_1 \oplus 3p_2$ means that we have two tokens on place p_1 and three tokens on p_2).

A *P/T-system* is given by (PN, M) with marking $M \in P^{\oplus}$, Note that M can also be considered as function $M : P \to \mathbb{N}$ where only for a finite set $P' \subseteq P$ we have $M(p) \geq 1$ with $p \in P'$. We can switch between these notations by defining $\sum_{p \in P} M(p) \cdot p = M \in P^{\oplus}$. Moreover, for $M_1, M_2 \in P^{\oplus}$ we have $M_1 \leq M_2$ if $M_1(p) \leq M_2(p)$ for all $p \in P$.

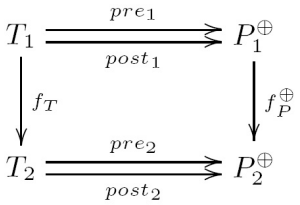
A transition $t \in T$ is M-enabled for a marking $M \in P^{\oplus}$ if we have $pre(t) \leq M$, and in this case the successor marking M' is given by $M' = M \ominus pre(t) \oplus post(t)$ and $(PN, M) \xrightarrow{t} (PN, M')$ is called firing step. Note that the inverse $\ominus$ of $\oplus$ is only defined in $M_1 \ominus M_2$ if we have $M_2 \leq M_1$.

In order to define rules and transformations of P/T-systems we introduce P/T-morphisms which preserve firing steps by Condition (1) below. Additionally they require that the initial marking at corresponding places is increasing (Condition (2)) or even stronger (Condition (3)).

Definition 7 (P/T-morphisms) Given P/T-systems $PN_i = (PN_i, M_i)$ with $PN_i = (P_i, T_i, pre_i, post_i)$ for $i = 1, 2$, a P/T-morphism $f : (PN_1, M_1) \to (PN_2, M_2)$ is given by $f = (f_P, f_T)$ with functions $f_P : P_1 \to P_2$ and $f_T : T_1 \to T_2$ satisfying

(1) $f_P^{\oplus} \circ pre_1 = pre_2 \circ f_T$ and $f_P^{\oplus} \circ post_1 = post_2 \circ f_T$ and
(2) $M_1(p) \leq M_2(f_P(p))$ for all $p \in P_1$.

Note that the extension $f_P^{\oplus} : P_1^{\oplus} \to P_2^{\oplus}$ of $f_P : P_1 \to P_2$ is defined by $f_P^{\oplus}(\sum_{i=1}^{n} k_i \cdot p_i) = \sum_{i=1}^{n} k_i \cdot f_P(p_i)$. (1) means that f is compatible with pre- and post domain (as shown in the diagram to the right), and (2) that the initial marking of PN_1 at place p is smaller or equal to that of PN_2 at $f_P(p)$.



Moreover the P/T-morphism f is called strict if f_P and f_T are injective and

(3) $M_1(p) = M_2(f_P(p))$ for all $p \in P_1$.

The category defined by P/T-systems and P/T-morphisms is denoted by **PTSys** where the composition of P/T-morphisms is defined componentwise for places and transitions.

Remark 1 *For our morphisms we do not always have $f_P^{\oplus}(M_1) \leq M_2$. E.g. $M_1 = p_1 \oplus p_2, M_2 = p$ and $f_P(p_1) = f_P(p_2) = p$ implies $f_P^{\oplus}(M_1) = 2p > p = M_2$, but $M_1(p_1) = M_1(p_2) = 1 = M_2(p)$.*

We now define the gluing of P/T-systems via P/T-morphisms by pushouts in the category **PTSys**. Given the left-hand side of a rule $(K, M_K) \xrightarrow{l} (L, M_L)$ (see Def. 9) and a match m: $(L, M_L) \to (PN_1, M_1)$, we have to construct a P/T-system (PN_0, M_0) such that (1) becomes a pushout. In particular, we are interested in pushouts with l being a strict morphism and c being a general morphism (due to the construction of the category **PTNet** as weak adhesive HLR category [EHP+07]).

This requires the following gluing condition, which has to be satisfied in order to apply a rule at a given match. The characterization of specific points is a sufficient condition for the existence and uniqueness of the pushout complement (PN_0, M_0), because it allows checking for the applicability of a rule to a given match.

$$\begin{array}{ccc} (K,M_K) & \xrightarrow{l} & (L,M_L) \\ {\scriptstyle c}\downarrow & (1) & \downarrow{\scriptstyle m} \\ (PN_0,M_0) & \longrightarrow & (PN_1,M_1) \end{array}$$

Definition 8 (Gluing condition for P/T-systems) Let $(L,M_L) \xrightarrow{m} (PN_1,M_1)$ be a P/T-morphism and $(K,M_K) \xrightarrow{l} (L,M_L)$ a strict morphism. We define the gluing points GP, dangling points DP, and identification points IP of L as in [EHP+07]:

$$\begin{array}{ll} GP = & l(P_K \cup T_K) \\ DP = & \{p \in P_L | \exists t \in (T_1 \setminus m_T(T_L)) : m_P(p) \in pre_1(t) \oplus post_1(t)\} \\ IP = & \{p \in P_L | \exists p' \in P_L : p \neq p' \wedge m_P(p) = m_P(p')\} \\ & \cup \{t \in T_L | \exists t' \in T_L : t \neq t' \wedge m_T(t) = m_T(t')\} \end{array}$$

A P/T-morphism $(L,M_L) \xrightarrow{m} (PN_1,M_1)$ and a strict morphism $(K,M_K) \xrightarrow{l} (L,M_L)$ satisfy the gluing condition, if

1. all dangling and identification points are gluing points, i.e $DP \cup IP \subseteq GP$, and
2. m is strict on places to be deleted (see [EHP+07]), i.e. $\forall p \in P_L \setminus l(P_K) : M_L(p) = M_1(m(p))$.

Note that only places are considered as dangling points because a transition can be matched only if its complete environment is matched too (see Def. 7).

Next, we present rule-based transformations of P/T-systems following the DPO approach of graph transformations, which is restrictive concerning the treatment of unmatched transitions at places which should be deleted. Here, the gluing condition forbids the application of rules in this case. Furthermore, items which are identified by a non-injective match are both deleted by or preserved by rule applications.

Definition 9 (P/T-system rule) A rule $p = ((L,M_L) \xleftarrow{l} (K,M_K) \xrightarrow{r} (R,M_R))$ of P/T-systems consists of P/T-systems (L,M_L), (K,M_K), and (R,M_R), and two strict P/T-morphisms $(K,M_K) \xrightarrow{l} (L,M_L)$ and $(K,M_K) \xrightarrow{r} (R,M_R)$ [EHP+07].

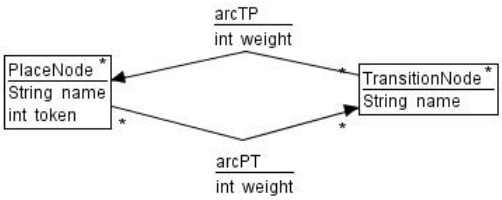
Definition 10 (P/T-system transformation) A rule $p = ((L,M_L) \xleftarrow{l} (K,M_K) \xrightarrow{r} (R,M_R))$ is called applicable at the match $(L,M_L) \xrightarrow{m} (PN_1,M_1)$ if the gluing condition is satisfied for l and m. In this case we obtain a P/T-system (PN_0,M_0) leading to a net transformation step $(PN_1,M_1) \xRightarrow{p,m} (PN_2,M_2)$ consisting of the following pushout diagrams (1) and (2). The P/T-morphism $n : (R,M_R) \to (PN_2,M_2)$ is called comatch of the transformation step.

$$\begin{array}{ccccc} (L,M_L) & \xleftarrow{l} & (K,M_K) & \xrightarrow{r} & (R,M_R) \\ {\scriptstyle m}\downarrow & (1) & \downarrow{\scriptstyle c} & (2) & \downarrow{\scriptstyle n} \\ (PN_1,M_1) & \xleftarrow[l^*]{} & (PN_0,M_0) & \xrightarrow[r^*]{} & (PN_2,M_2) \end{array}$$

3 Simulating Petri Net Transformations by Graph Transformation

For the simulation of Petri net transformations with graph transformations, we need a suitable Petri net representation in the domain of graphs. Hence, we define a type graph for the VL of Petri nets.

Definition 11 (Type graph for Petri nets) The attributed graph below shows the type graph modelling the Petri net VL. We use the concrete syntax from AGG, i.e. attribute type names and data types are written inside their node type nodes or below their arc type names.
A *PlaceNode* has two attributes, a *name* (a String), and a *token* number (for simplicity we took the Java datatype *int* instead of defining a more suitable datatype *nat*). Arc edges are attributed by a weight number, and Transition nodes also carry a name attribute. All graphs typed over this type graph are considered as graph representations of Petri nets.



On the basis of the Petri net type graph, we now define the obvious translation from Petri nets and Petri systems to typed attributed graphs. This translation can be extended in a straightforward way to the translation of P/T-morphisms and rules. We then discuss the problems with this translation leading to graph transformations which do not correspond to original Petri net transformations. As a solution, we propose extensions of the rule translation by adding negative application conditions and attribute conditions to the graph rules. We claim that these extensions lead to an equivalence of Petri net transformations by Petri net rules on the one hand and the transformations of graphs (the translated Petri nets) by graph rules (the extended translated Petri net rules) on the other hand (see Section 5).

Definition 12 (Translating P/T systems to typed, attributed graphs) Let (N,M) be a P/T system with P/T-net $N = (P,T,pre,post)$ and marking $M \in P^{\oplus}$ according to Def. 6. Then, the PT2Graph translation of (N,M) is an attributed graph $AG = (G,D)$ with D being a $DSIG$-algebra over the datatype signature $DSIG$ with sorts $String$ and $Integer$, and the obvious operations, and G being an E-graph according to Def. 1 which is typed over the type graph for Petri nets (Def. 11). G is defined as follows:

- $V_G = PlaceNode \cup TransitionNode$, with $PlaceNode = P$ and $TransitionNode = T$.
- $E_G = ArcPT \cup ArcTP$ is constructed from the pre- and post-domain functions:
  - $\forall t \in T : \forall p \in pre(t) : \exists e \in ArcPT$ with $source_G(e) = p$ and $target_G(e) = t$ and $\exists e_{weight} \in E_{EA}$ with $source_{EA}(e_{weight}) = e$ and $target_{EA}(e_{weight}) = pre(t)(p)$
    (an ArcPT edge is attributed by the weight of the corresponding arc)
  - $\forall t \in T : \forall p \in post(t) : \exists e \in ArcTP$ with $source_G(e) = t$ and $target_G(e) = p$ and $\exists e_{weight} \in E_{EA}$ with $source_{EA}(e_{weight}) = e$ and $target_{EA}(e_{weight}) = post(t)(p)$
    (an ArcTP edge is attributed by the weight of the corresponding arc)
- $\forall p \in PlaceNode : \exists e_{token} \in E_{NA}$ with $source_{NA}(e_{token}) = p$ and $target_{NA}(e_{token}) = M(p)$
  (a PlaceNode is attributed by the number of tokens on the corresponding place).

Note that this translation is *not* a functor between the corresponding categories **PTSys** and $\mathbf{AGraphs}_{\mathbf{ATG}}$, because Petri net morphisms allow to map a place marking to a different marking, but the corresponding graph morphisms do not allow to map place nodes with different token attributes. The extension of Def. 12 to translate P/T morphisms and rules to graph morphisms and rules is straightforward only for the net components, since place mappings are translated to PlaceNode mappings, and transition mappings to TransitionNode mappings. The typing of graphs over the P/T type graph in Def. 11 ensures that net components are mapped according to the P/T morphism parts f_P and f_T. If we restrict to injective P/T morphisms only, then we get ArcPT mappings and ArcTP mappings with identical weight attributes, which corresponds to translated pre and post-domains. We would run into a problem if we had non-injective morphisms since P/T morphisms allow to merge arcs, provided that the sum of arc weights remains constant. This problem is still open (see Section 5).

In this section we discuss and solve two other problems:

1. Given a translated rule, a match into the graph of a translated net might be found, where the match morphism does not correspond to a firing-behaviour-preserving P/T morphism;
2. The token attribute of a PlaceNode must be mapped to an identical token attribute of another PlaceNode, which does not correspond to more general mapping of markings in P/T morphisms.

3.1 Problem 1: Preserving Firing Behavior of Petri Net Transitions

As described in Section 2, Petri net morphisms preserve the firing behavior of the Petri nets by preserving the pre- and post domains of transitions. Graph morphisms are not restricted in this way and graph morphisms could match a *TransitionNode* even if its pre- and post domain in the transition's image is different from that of the original transition's pre- or post domain.

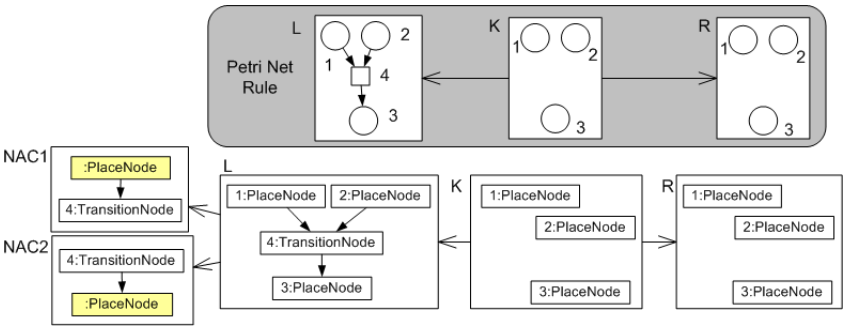


Figure 1: Translation to firing behavior preserving graph rules

To solve this problem, we extend the translation of rules. We create two additional negative application conditions for each TransitionNode in an LHS of a graph rule (see Fig.1, where mappings are depicted by numbering of corresponding nodes): one NAC that forbids an additional place (different to 1 and 2) in the pre-domain of the transition and a second NAC for the post-domain, respectively. Obviously, a transition in the LHS of a rule now can be matched only if its pre- and post-domain are preserved by the match.

3.2 Problem 2: Mapping the Marking for Petri Systems

Another problem is the use of tokens on places. By modeling tokens as attributes of PlaceNodes, we restrict the matching of PlaceNodes to other PlaceNodes with the same number of tokens. But Petri net morphisms (see Def. 7) allow matching of places to places that contain the same number of tokens *or more*.

Let us consider a different possibility to model tokens that would solve this mapping problem: In contrast to their definition as *PlaceNode* attributes, tokens could alternatively be represented as nodes typed over an additional *Token* node type. As shown in Fig. 2, this would result in valid graph morphisms that allow matching a PlaceNode with less tokens to a PlaceNode that contains more tokens. However, in graph rules that delete PlaceNodes it would now be required to specify

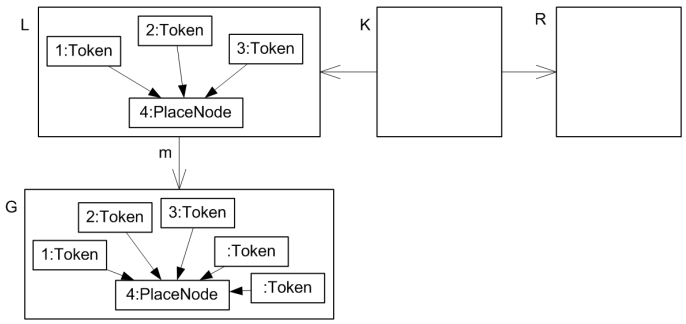


Figure 2: Marking modeled as nodes

the exact number of token nodes connected to the PlaceNode that will be deleted. Otherwise the graph rule is either not applicable (because of the gluing condition of the DPO approach) or unconnected token nodes would remain in the graph (in the case of the SPO approach).

For this reason, we keep our attribute representation of tokens but replace the fixed number of tokens in a PlaceNode in the LHS by a variable. Additionally, we add an attribute condition which ensures that a match is valid only if the number of tokens in the PlaceNode in the graph is at least as large as required (see Fig.3). For example, for a PlaceNode with 3 tokens in the

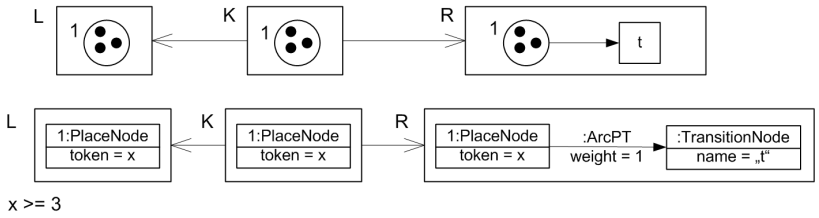


Figure 3: Marking modeled as token variables and attribute conditions in rules

LHS, the attribute value is replaced by a variable x, and the attribute condition $x \geq 3$ is added to the rule. Attribute conditions must be true for a given match for the rule to be applicable. In the example, the PlaceNode from the LHS can be matched to any PlaceNode with 3 or more tokens

which is exactly the expected behavior of the corresponding Petri net match morphism.[1]

4 Implementing Petri Net Transformations based on AGG

The main idea behind Reconfigurable Object Nets (RONs) is to support the visual specification of controlled rule-based net transformations of place/transition nets (P/T nets) [BM08, BEHM07]. RONs are high-level nets with two types of tokens: object nets (place/transition nets) and net transformation rules (a dedicated type of graph transformation rules).

4.1 RON Architecture

In a student project in summer 2007 a visual editor for RONs has been realized as a plug-in for ECLIPSE using the ECLIPSE Modeling Framework (EMF) [EMF08] and Graphical Editor Framework (GEF) [GEF06] plug-ins. (GEF) [GEF06] is based on ECLIPSE's Standard Widget Toolkit (SWT) and provides implementing a visual editor based on the Model-View-Controller pattern. It supports many operations and features common to most graphical editors like zooming, various layouting of figures, support for drag and drop etc.

The RON environment [BEHM07] consists of four main components, i.e. the RON tree view based on an EMF model for RONs, and the visual editors for object nets, for transformation rules and for high-level nets. The visual net editor also supports the simulation of an edited object net or high-level net. Fig. 4 shows the RON environment including all views and editors.

RON Tree View. View 1 in Fig. 4 shows the main editor component, a tree view for the complete RON model from which the graphical views can be opened by double-click.

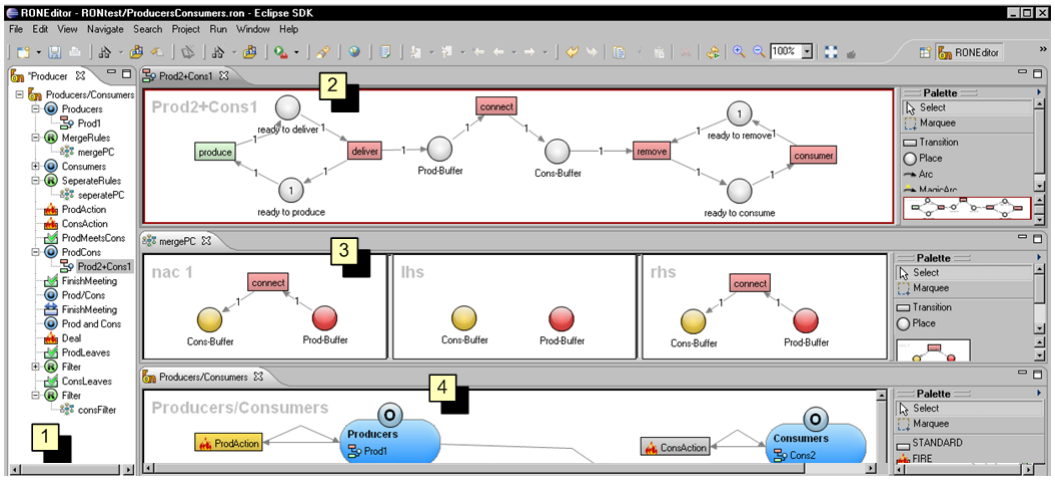


Figure 4: The RON Environment for Editing and Simulating Reconfigurable Object Nets

[1] The same restricting effect could be achieved by x NACs in this rule, each forbidding a certain number of tokens less than x on the place. We use attribute conditions for the sake of lucidity and because AGG allows to use them.

Object Net Editor. Fig. 4, View 2 shows the kernel component, a place/transition net editor with the possibility of the simulation of transition firing. The object net *ProdCons* models producer-consumer interaction.

Transformation Rule Editor. The editor for transformation rules 3 in Fig. 4 consists of three panels, one for the left-hand side (LHS), one for the right-hand side (RHS), and one showing one of the rule's negative application conditions (NACs). With the *mapping tool* in the palette, you can define mappings from LHS objects to RHS objects and from LHS objects to NAC objects.

High-Level Net Editor. High-Level Net Editor, shown in Fig. 4, View 4 allows the editing of a high-level net, which controls object net behavior and rule applications to object nets. The blue containers marked by an "O" for *Object Nets* are NET places, the green marked by an "R" for *Rules* are RULE places. Enabled HL-transitions are colored, disabled ones are gray.

4.2 Defining ON Transformation Rules

The rule editor (Fig. 5) provides a palette with the objects to generate: places. transitions and arcs. In order to define mappings, the palette entry Map has to be selected, clicking on places first in LHS, then RHS or NAC. Mapped places are high-lighted in the same color. By right-clicking with the mouse on a panel there are several copy services provided like copying the object net from the LHS to the RHS or to the current NAC.

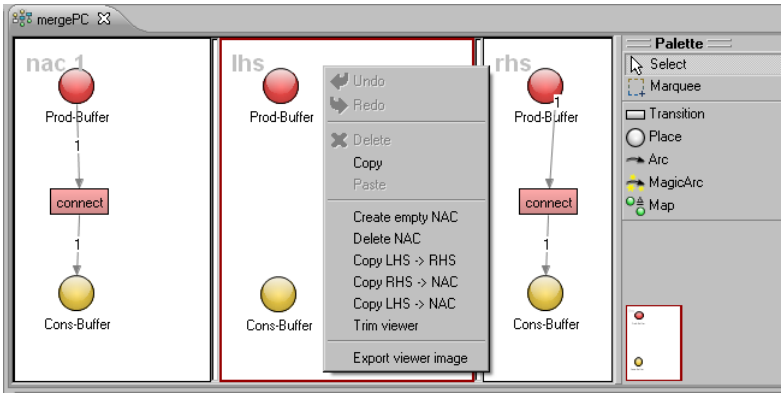


Figure 5: Defining Rules

4.3 Translating ON Transformation Rules to AGG: The *RON2AGG* Converter

In order to calculate possible rule matches or transformation results, the graph transformation engine AGG is used. Therefore, the RON editor is extended by a *RON2AGG*converter, which converts object nets to graphs and net transformation rules to graph rules using the extensions of simple translations discussed in Sections 3.1 and 3.2.

If there are more than one possible match a match-dialog is opened. A possible match is shown and can be confirmed or, by pushing the *next*-button, the next possible match is shown. The

function calculate completions lets AGG compute all matches for the selected rule. For example in Fig. 6 a), there are two places in the object net (p1 and p2) with equal and more tokens than the place in the LHS of the rule. Hence, AGG computes two matches: $p1 \rightarrow p1$ and $p1 \rightarrow p2$.

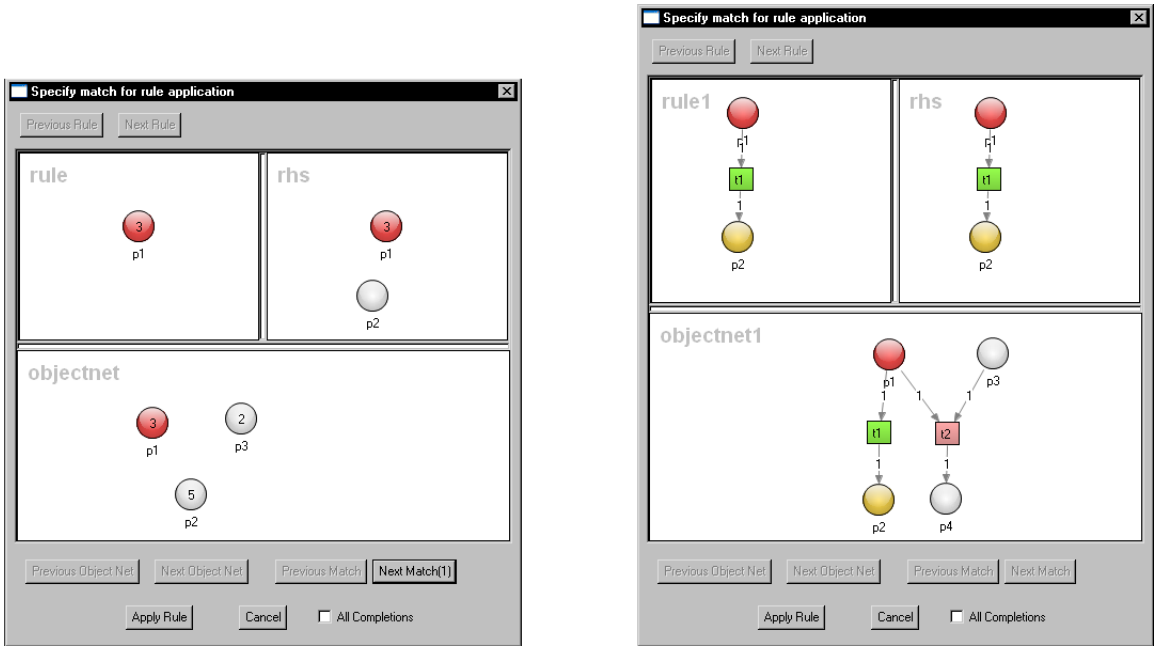


Figure 6: Match Dialogs: a) places with different markings and b) transitions with different pre- and post-domains

To ensure the user-specified mapping being a valid Petri net morphism, each mapped transition's pre- and post-domain is checked if it is preserved by the mapping. E.g., consider Fig. 6 b) showing the state after application of calculate completions. A match from t1 to t2 would be invalid, because t1's pre-domain could not be preserved. Hence, AGG offers only the currently high-lighted match.

5 Conclusion

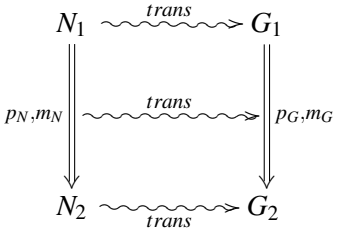
We have defined a formal translation of Petri nets and Petri net transformation rules to graphs and graph transformation rules. The result graphs of the application of translated rules to translated Petri nets can be interpreted as Petri nets. With this encoding, transformation of Petri nets can be simulated in common graph transformation tools. We presented a prototype that simulates Petri net transformation systems by first translating rules and Petri nets to AGG rules and graphs and then invoking the AGG graph transformation engine.

As future work, we plan to cover also non-injective matches of Petri nets with arc weights. Merging weighted arcs has further compatibility properties, which constrain the applicability of rules. These restrictions would have to be reflected by corresponding translated graph rules.

Furthermore, a formal proof showing the correctness and completeness of our translation is

desirable to know for sure that the behaviour defined by a graph transformation rule (the result of the translation) and the behaviour of the original Petri net rule are equivalent.

This means, we have to show that the diagram to the right commutes, i.e. that a translation of a P/T-system followed by an application of a translated rule results in the same graph as first applying the Petri net rule and afterwards translating the resulting net.



As an important benefit of our approach, we could use the graph transformation analysis features offered by AGG (e.g. Critical Pair Analysis and dependence analysis for sets of rules) also for Petri net transformations.

Bibliography

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

[BEHM07] E. Biermann, C. Ermel, F. Hermann, T. Modica. A Visual Editor for Reconfigurable Object Nets based on the ECLIPSE Graphical Editor Framework. *Proc. Workshop on Algorithms and Tools for Petri Nets (AWPN'07)*. 2007. http://tfs.cs.tu-berlin.de/roneditor

[BM08] E. Biermann, T. Modica. Independence Analysis of Firing and Rule-based Net Transformations in Reconfigurable Object Nets. *Proc. Workshop on Graph Transformation and Visual Modeling Techniques. (GT-VMT'08)*. Vol. 10. EC-EASST, 2008. http://tfs.cs.tu-berlin.de/gtvmt08/GTVMT-program.htm

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer, 2006.

[EHP06] H. Ehrig, K. Hoffmann, J. Padberg. Transformations of Petri Nets. *Proc. School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques*. ENTCS 148 / 1, pp. 151–172. Elsevier Science, 2006.

[EHP+07] H. Ehrig, K. Hoffmann, J. Padberg, U. Prange, C. Ermel. Independence of Net Transformations and Token Firing in Reconfigurable Place/Transition Systems. *Proc. Conf. on Application and Theory of Petri Nets and Other Models of Concurrency*. LNCS 4546, pp. 104–123. Springer, 2007.

[EHP+08] H. Ehrig, K. Hoffmann, J. Padberg, C. Ermel, U. Prange, E. Biermann, T. Modica. Petri Net Transformations. In *Petri Net Theory and Applications*. Pp. 1–16. I-Tech Education and Publication, 2008.

[EMF08] Eclipse Consortium. Eclipse Modeling Framework (EMF) – Version 2.4. 2008. http://www.eclipse.org/emf.

[EP04] H. Ehrig, J. Padberg. Graph Grammars and Petri Net Transformations. In *Lectures on Concurrency and Petri Nets*. LNCS 3098, pp. 496–536. Springer, 2004.

[GEF06] Eclipse Consortium. Eclipse Graphical Editing Framework (GEF) – Version 3.2. 2006. http://www.eclipse.org/gef.

[HEP07] K. Hoffmann, H. Ehrig, J. Padberg. Flexible Modeling of Emergency Scenarios using Reconfigurable Systems. *Proc. Conf. on Integrated Design & Process Technology*. 2007.

[HHT96] A. Habel, R. Heckel, G. Taentzer. Graph Grammars with Negative Application Conditions. *Special issue of Fundamenta Informaticae* 26(3,4):287–313, 1996.

[LO04] M. Llorens, J. Oliver. Structural and Dynamic Changes in Concurrent Systems: Reconfigurable Petri Nets. *IEEE Transactions on Computers* 53(9):1147–1158, 2004.

[LVA04] J. de Lara, H. Vangheluwe, M. Alfonseca. Meta-Modelling and Graph Grammars for Multi-Paradigm Modelling in AToM^3. *Software and System Modeling: Special Section on Graph Transformations and Visual Modeling Techniques* 3(3):194–209, 2004.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations, Vol. 1: Foundations*. World Scientific, 1997.

[TER99] G. Taentzer, C. Ermel, M. Rudolf. The AGG-Approach: Language and Tool Environment. In Ehrig et al. (eds.), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*. Pp. 551–603. World Scientific, 1999.