Termination Criteria for Model Transformation*

Hartmut Ehrig[1], Karsten Ehrig[1], Juan de Lara[2], Gabriele Taentzer[1], Dániel Varró[3], and Szilvia Varró-Gyapay[3]

[1] Technische Universität Berlin, Germany
{ehrig,karstene,gabi}@cs.tu-berlin.de
[2] Universidad Autonoma of Madrid, Spain
Juan.Lara@ii.uam.es
[3] Budapest University of Technology and Economics, Hungary
{varro,gyapay}@mit.bme.hu

Abstract. *Model Transformation* has become central to most software engineering activities. It refers to the process of modifying a (usually graphical) model for the purpose of analysis (by its transformation to some other domain), optimization, evolution, migration or even code generation. In this work, we show our approach to express model transformation based on *graph transformation.* This framework offers visual and formal techniques based on rules, in such a way that model transformations can be subject to analysis. Previous results on graph transformation are extended by proving the termination of a transformation if the rules applied meet certain criteria. We show the suitability of the approach by an example in which we translate a simplified version of Statecharts into Petri nets for functional correctness analysis.

1 Introduction

Diagrams are ever more frequently used in our everyday work as a means for problem solving, specification and comprehension. Their use is pervasive in areas such as computer science, with the increasing tool support and popularity of notations (such as UML), and model-based development processes (such as the one proposed by the MDA [24]). In this area, we are witnessing a paradigm shift, where models are no longer mere (passive) documentation, but are used for code generation, analysis and simulation as well.

Whereas the syntax of most notations is usually well-defined (sometimes by means of a meta-model), semantics are often specified in a semi-formal way, which prevents the use of analysis methods. Moreover, sometimes modelling is easier using a certain notation, but the formalism lacks certain analysis techniques to solve some of the user problems. One way to solve these difficulties is by specifying transformations from the initial source formalism into a target notation [9]. Once the model is translated, we can use the target notation analysis techniques to solve the initial problem. There are many other scenarios in

* Juan de Lara, Dániel Varró and Szilvia Varró-Gyapay were partially supported by the Segravis Research Training Network.

which model transformations are present, such as model evolution, migration (for example between different database schemata or between different versions of the UML meta-model) or model optimization. Even code generation can be seen as a transformation into the abstract syntax of the target textual language.

Problem statement. An important question is *how to specify such model transformations.* A recent initiative of the Object Management Group aims at developing a standard for describing Queries, Views and Transformations (QVT) [20] for UML (in fact, any MOF-based) models. Although the submitted approaches vary a lot (e.g. in providing textual [14] vs. graphical specifications [23] for transformations), high-level, graph-based and declarative specifications are proposed in many of the submissions.

The correctness of model transformations, namely, to *guarantee that certain semantic properties hold for a transformation*, is also a crucial aspect of transformation engineering. For instance, when transforming UML models into mathematical domains, the results of a formal analysis can be invalidated by erroneous model transformations as the systems engineers cannot distinguish whether an error is in the design or in the transformation. Most typical correctness properties of a model transformation are termination, uniqueness (confluence) and behaviour preservation.

Objectives. In the paper, we propose the use of graph transformation [25] over typed and attributed graphs that provides rule and pattern-based manipulation of graph models generalizing Chomsky grammars from strings to graphs. The algebraic approach to graph transformation is based on concepts of *category theory* (see [11]), and has a rich body of theoretical results that have been developed in the last 30 years (see [25]). In this way, transformations expressed as graph grammars become not only graphical and intuitive but also formal, declarative and high-level models, subject themselves to analysis.

While the use of graph transformation for specifying model transformations has been under intensive research, the main result of the paper is concerned with the *termination of model transformations.* Although termination is undecidable for graph grammars in general [22], in this paper we show that if graph grammars with negative application conditions (see [15]) meet suitable termination criteria, we can conclude that they are terminating. The criteria we propose are based on assigning a *layer* to each rule, node and edge label (type).

Structure of the paper. The rest of the paper is organized as follows: Sec. 2 presents a running example, in which we specify (with graph grammar rules) a transformation from a restricted version of Statecharts into Petri nets, with the aim of subsequent analysis. Sec. 3 details the critera for termination of layered graph grammars. Sec. 4 discusses the application of the criteria to the running example, and sketches how the criteria can easily be applied to other interesting model transformation examples. Sec. 5 discusses related work and finally Sec. 6 presents our conclusions and proposals for future work.

2 Motivating Example: From Statecharts to Petri Nets

In order to illustrate the idea of the proposed criteria for termination of model transformation we introduce a model transformation from UML statecharts into Petri nets. The running example is a simplified version of the original transformation that was designed and implemented in the VIATRA system [28] as part of a Hungarian research project (IKTA 065/2000 – A framework for the modelling and analysis of dependable and safety critical systems) and discussed in more details in [27]. The transformation aims at formal verification of safety critical applications designed by UML statecharts using semi-decision analysis methods of Petri nets [21]. Similar transformations into various classes of Petri nets could carry out dependability and performance analysis for the system model in early stages of design.

2.1 Source modelling language: UML statecharts

UML statecharts are an object-oriented variant of classical Harel statecharts [16] that describe behavioural aspects of (any instance of) a class in the system under design. In fact, the statechart formalism itself is an extension of finite state machines to allow a decomposition of states into a state hierarchy with parallel regions that greatly enhance the readability and scalability of state models.

An extract of the metamodel of UML statecharts is depicted in the upper left part of Fig. 1 (abbreviated as SC). In fact, this metamodel is a proper extension of the standard UML metamodel (for which we assume the reader's familiarity) that explicitly introduces several notions of statecharts that are only implicitly present in the standard (such as state configurations, queues, etc.). The necessity and the guideline of these extensions to obtain a formal operational semantics of statecharts is discussed in [27, 26].

In the paper, we consider a network of statemachines SM, each of which having an associated event queue. A single statemachine captures the behaviour of any object of a specific class by flattening the state hierarchy into *state configurations* and grouping parallel transitions into *steps*. [4]

A Configuration is composed of a set of States that can be active at a time. The activeness of a state is indicated by the isAct edge, while the initial configuration is identified by the initConf association.

A Step is composed of non-conflicting Transitions (which are, in turn, binary relations between states) that can be fired in parallel. A step is leading from a configuration fromConf to a configuration toConf, and it is triggered by a common Event for all its transitions. The effect of a step is a sequence of Actions. For the paper, we only consider *send actions* which send a message to a target (receiver) queue in the form of a corresponding event.

Each statemachine has exactly one associated event Queue (handled as sets and not FIFOs for presentation purposes) that store Events. The inQueue associ-

[4] Note that configurations and steps can be collected at compile time, i.e. prior to the statecharts to Petri nets model transformation (see [27] for further details).

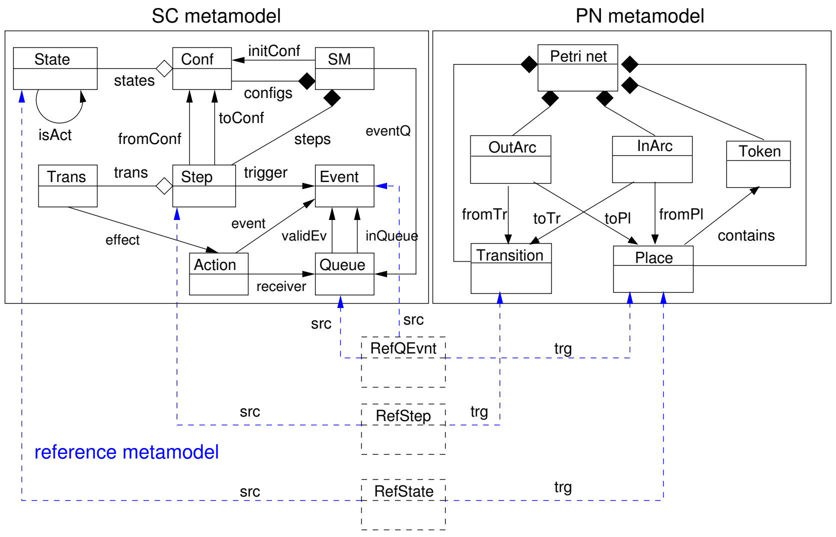

Fig. 1. The combined metamodel of statecharts and Petri nets

ation denotes if a certain event is present in the corresponding event queue. The set of acceptable events in a certain queue is denoted by the association validEv.

2.2 Target modelling language: Petri nets

Petri nets (abbreviated as PN) are widely used to formally capture the dynamic semantics of concurrent systems due to their easy-to-understand visual notation and the wide range of available analysis tools. From a system modelling point of view, transforming UML models to Petri nets may provide correctness, dependability and performance analysis for the system model in early stages of design.

Petri nets are bipartite graphs, with two disjoint sets of nodes: Places and Transitions. Places may contain an arbitrary number of Tokens. A token distribution defines the state of the modelled system. The state of the net can be changed by firing enabled transitions. A transition is *enabled* if each of its input places contains at least one token (if no arc weights are considered). When firing a transition, we remove a token from all input places (connected to the transition by InArcs) and add a token to all output places (as defined by OutArcs). A Petri net metamodel is shown in the upper right corner of Fig. 1.

Reference metamodel. In order to interrelate the source and target modelling languages, we use reference metamodels [28]. For instance, a reference node of type RefState (in Fig. 1) relates a source State to a target Place.

2.3 Transforming state machines into Petri nets

An informal overview of graph transformation. The model transformation from state machines into Petri nets is specified by graph transformation rules.

Graph transformation [25] (for the formal treatment see Sec. 3) provides a rule-based manipulation of graph models. A *graph transformation rule* consists of a left-hand side (LHS) graph L, right-hand side (RHS) graph R, and (an optional) negative application condition N. Informally, L and N of a rule define the *precondition* while R defines the *postcondition*.

The *application* of a rule to a *host model graph* G (e.g., a UML model built by the user) alters the model graph by replacing the pattern defined by L with the pattern of the R. This is performed by (i) *finding a match* of the L pattern in model M; (ii) *checking the negative application conditions* N which prohibits the presence of certain model elements; (iii) *removing* a part of the model M that can be mapped to the L pattern but not the R pattern yielding an intermediate graph D; (iv) *adding* new elements to the intermediate graph D which exist in the R but not in L yielding the derived graph H. In our example we follow the *Double Pushout Approach* [7, 13]. Technical details are given as far as necessary in Sec. 3.

For a more compact presentation of the rules, we abbreviate the L, N and R graphs of a rule into one, and we only mark which (the images of) graph elements need to be removed (del), or created (new). Due to the special structure imposed by nondeleting rules (to be discussed in Sec. 3), all elements in the negative application condition N should also be present in R. Therefore, we assume for the current model transformation that R and N are isomorphic, and we simply omit the neg tags for the sake of clarity. The graph transformation rule generating a PN transition for a SC step is depicted in both the mathematical and the abbreviated notation in the upper-most part of Fig. 2.

The UML statechart to Petri net transformation. Transforming our flattened UML statechart representation (with configurations and steps) into Petri nets is relatively simple (see the transformation rules and an example in Fig. 2).

- Each SC state is modeled with a respective place in the target PN model where a token in such a place denotes that the corresponding state is active initially (rules ActState2TokenR, State2PlaceR). In addition, places are generated to model messages stored in event queues of a state machine. A separate place is generated for each valid event accepted by a certain queue, and initialized according to the presence of corresponding events (QueueEvent2PlaceR: the general case; InQueueEvent2TokenR: a special case).
- Each SC step is projected into a PN transition (Step2TransR). Naturally, the Petri net should simulate how to exit and enter the corresponding states in the statechart, therefore input and output arcs of the transition should be generated accordingly (see StepFrom2InArcR and StepTo2OutArcR). Furthermore, firing a transition should consume the token of the trigger event (Trigger2InArcR), and should generate tokens to (the places related to) the target (receiver) event queues according to the actions (Action2OutArc).

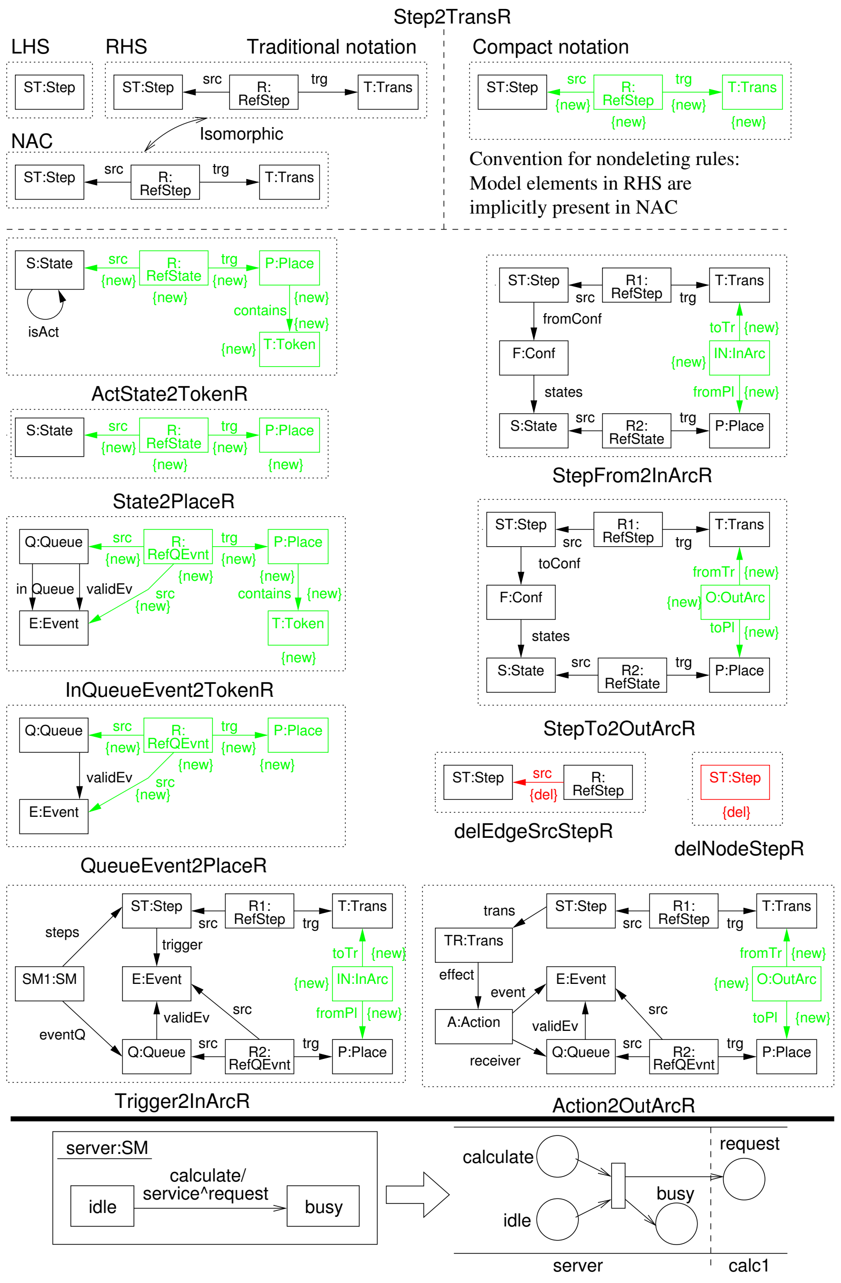
Convention for nondeleting rules: Model elements in RHS are implicitly present in NAC

Fig. 2. From state machines to Petri nets

– Finally, we clear up the joint model by removing all model elements from the source and the reference metamodel by another set of graph transformation rules. For instance, rule DelEdgeSrcStepR deletes all the src edges leading to a Step, while rule DelNodeStepR deletes all Step nodes from the graph. All the other deleting rules of similar kind (including those removing reference nodes and edges) are omitted for space considerations.

3 Termination Criteria for Layered Graph Transformation Systems

In this section we present and prove termination criteria for layered graph transformation systems, which will be used in the next section to show termination of our running example. In fact, our termination criteria are valid for a broad class of graph transformation systems. The criteria for nondeleting rules are based on the single – or double pushout approach (see [7]). For the applications in Sec. 2 and 4 we use in this paper typed attributed graph transformation (see [17, 12]) with injective rule morphisms $l : K \to L, r : K \to R$ and injective matches $m : L \to G$. Moreover we use negative application conditions (NACs) given by an injective morphism $n : L \to N$. The match $m : L \to G$ satisfies the NAC if there is no injective morphism $q : N \to G$ with $m = q \circ n$. Rule morphisms are depicted (in Fig. 2) by using the same node identifiers in LHS, NAC and RHS. Labels LAB can be defined in the traditional way by label sets or in correspondence with the metamodel.

Now we define layered graph grammars with deletion and nondeletion layers. Informally, the deletion layer conditions express that the last creation of a node with a certain label should precede the first deletion of a node with the same label. On the other hand, nondeletion layer conditions ensure that if an element of label l occurs in the LHS of a rule then all elements of the same label were already created in previous layers.

Definition 1 (Layered Graph Grammar). *A graph grammar with rules RUL and labels LAB is called layered graph grammar if for each rule $r \in RUL$ we have a rule layer $rl(r) = k$ with $0 \leq k \leq k_0$ $(k, k_0 \in \mathbb{N})$ where k_0 is the number of layers and for each label $l \in LAB$ a creation and a deletion layer $cl(l), dl(l) \in \mathbb{N}$ and a deletion layer or a nondeletion layer satisfying the following conditions for all $r \in RUL$:*

If k is a deletion layer *Deletion Layer Conditions*	*If k is a nondeletion layer* *Nondeletion Layer Conditions*
1. r is deleting at least one item	*1. r is nondeleting, i.e. $r : L \to R$ with r total and injective*
2. $0 \leq cl(l) \leq dl(l) \leq k_0$ for all $l \in LAB$	*2. r has NAC $n : L \to N$ with $n' : N \to R$ injective s.t. $n' \circ n = r$*
3. r deletes $l \Rightarrow dl(l) \leq rl(r)$	*3. $x \in L \Rightarrow cl(label(x)) \leq rl(r)$*
4. r creates $l \Rightarrow cl(l) > rl(r)$	*4. r creates $l \Rightarrow cl(l) > rl(r)$*

For the SC2PN transformation the layer conditions mean, for instance, that rules creating Place nodes cannot be in the same layer with rule StepFrom2InArcR: since Place nodes can be used as a pre-condition by a rule (StepFrom2InArcR) only if its creation has finished, i.e. there are no more rules creating Place nodes in the same layer or above. Thus rules ActState2TokenR, State2PlaceR, InQueueEvent2TokenR, and QueueEvent2PlaceR has to precede rule StepFrom2InArcR.

The termination of layered graph grammars expresses that no infinite derivation sequences exist starting from an initial graph if rules are applied within layers as long as possible.

Definition 2 (Termination of Layered Graph Grammars). *A layered graph grammar with finite start graph G_0 and rules RUL terminates, if there is no infinite derivation sequence from G_0 via $RUL = [RUL_k = \{r \in RUL \mid rl(r) = k\}]_{k=0..k_0}$, where starting with layer $k = 0$ rules $r \in RUL_k$ are applied as long as possible before going over to layer $k + 1 \leq k_0$.*

The termination of layered graph grammars are proved separately for the deletion and the nondeletion layers.

Lemma 1 (Termination of Layered Graph Grammars with Deletion). *Each layered graph grammar with deletion terminates.*

Proof (Lemma 1).
Step 0: Let $c_0 = card\{x \in G_0 | dl(label(x)) = 0\}$.
By deletion layer conditions 1,3 the application of a rule r to G_0 with $rl(r) = 0$ deletes at least one item $x \in G_0$ with label $l = label(x)$ and $dl(l) = 0$.
Moreover by deletion layer condition 4 each of the rules r can only create items x with $label(x) = l$, where $cl(l) > 0$. This means by using deletion layer condition 2 that only items x with $label(x) = l$ and $dl(l) \geq cl(l) > 0$ can be created. Hence at most c_0 applications of rules $r \in RUL_0$ are possible in layer 0 leading to $G_0 \Rightarrow^* G_1$ via RUL_0.
Step k: Given graph G_k as result of step $(k-1)$ for $1 \leq k \leq k_0$ then define $c_k = card\{x \in G_k \mid dl(label(x)) \leq k\}$. Using now rules r with $rl(r) = k$ each $r \in RUL_k$ deletes at least one item $x \in G_k$ with $dl(label(x)) \leq k$ by deletion layer conditions 1 and 3 and creates at most items x with $cl(label(x)) > k$ by deletion layer condition 4 which implies $dl(label(x)) \geq cl(label(x)) > k$ by deletion layer condition 2. Hence at most c_k applications of rules $r \in RUL_k$ are possible in layer k leading to $G_k \Rightarrow^* G_{k+1}$ via RUL_k.
After step n we have at most $c = \Sigma_{k=0}^{k_0} c_k$ applications of rules $r \in R$ leading to $G_0 \Rightarrow^* G_{k_0+1}$, which implies termination. □

Before proving termination for nondeletion layers, we need to define the notion of essential matches. Informally, an essential match m_0 of a match $m_1 : L \to H_1$ for a transformation $G_0 \Rightarrow^* H_1$ with $G_0 \subseteq H_1$ means that m_1 can be restricted to $m_0 : L \to G_0$.

Definition 3 (Transformation and Essential Match). *Given a nondeleting graph grammar with injective matches a nondeleting rule r is given by an* *injective morphism $r : L \to R$, and a match $m : L \to G$ is an injective morphism leading to a transformation step $G \Rightarrow H$ via (r, m) defined by the pushout* (1) *of r and m, where $d : G \to H$ is called* <u>*tracking morphism*</u> *of $G \Rightarrow H$ via (r, m).*

$$\begin{tikzcd} L \arrow[r, "r"] \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & R \arrow[d, "m^*"] \\ G \arrow[r, "d"'] & H \end{tikzcd}$$

Since r and m are injective morphisms, pushout properties (1) *imply that also d and m^* are injective. Given a transformation $G_0 \Rightarrow^* H_1$ i.e. a sequence of transformation steps with induced injective tracking morphism $d_1 : G_0 \to H_1$ a match $m_1 : L \to H_1$ of L in H_1 has an* <u>*essential match*</u> *$m_0 : L \to G_0$ of L in G_0 if we have $d_1 \circ m_0 = m_1$. Note, that there is at most one essential match m_0 for m_1, because d_1 is injective.*

The following lemma (which is proved in Appendix A) states that rules can be applied at most once with the same essential match.

Lemma 2. *In each derivation sequence starting from G_0 of a nondeleting layered graph grammar with injective matches, each rule $r : L \to R$ with $r \in RUL_0$ can be applied at most once with the same essential match $m_0 : L \to G_0$ and $m_0 \models NAC$.*

Lemma 3 (Termination of Nondeleting Layered Graph Grammars). *Each nondeleting layered graph grammar with injective matches terminates.*

Proof (Lemma 3).
<u>Step 0</u> Given the start Graph G_0 we count for each $r \in RUL_0$ with $r : L \to R$ and NAC the number of possible matches $m : L \to G_0$ with $m \models NAC$

$$c_r^0 = card\{m_0 | m_0 : L \to G_0 \text{ match with } m_0 \models NAC\}$$

We will show the following:
The application of rules $r \in RUL_0$ creates by nondeletion layer condition 4 only new items x with $cl(label(x)) > rl(r) = 0$, while each item $x \in L$ for any rule $r \in RUL_0$ has $cl(label(x)) \leq rl(r) = 0$ by nondeleting layer condition 3. This means that for each derivation sequence $G_0 \Rightarrow^* H_1$ via RUL_0 with injective matches and injective morphism $d_1 : G_0 \to H_1$ (induced from $G_0 \Rightarrow^* H_1$ by nondeleting layer condition 1) each match $m_1 : L \to H_1$ of some $r \in RUL_0$ must have an 'essential match' $m_0 : L \to G_0$ with $d_1 \circ m_0 = m_1$.

From Lemma 2 we conclude that in step 0 we have at most

$$c_0 = \sum_{r \in RUL_0} c_r^0$$

application of rules $r \in RUL_0$ leading to $G_0 \Rightarrow^* G_1$ via RUL_0.
<u>Step k</u> Given graph G_k as result of step $(k-1)$ for $1 \leq k \leq k_0$ then define for each $r \in RUL_k$ with $r : L \to R$ and NAC

$$c_r^k = card\{m_k \mid m_k : L \to G_k \text{ match with } m \models NAC\}.$$

Similar to step 0 each $r \in RUL_k$ creates only new items x with $cl(label(x)) > rl(r) = k$, while each item $x \in L$ has $cl(label(x)) \leq rl(r) = k$. Now we can apply Lemma 2 for G_k, RUL_k, and m_k instead of G_0, RUL_0, and m_0 and can conclude to have at most $c_k = \sum_{r \in RUL_k} c_r^k$ application of rules leading to $G_k \Rightarrow^* G_{k+1}$, via RUL_k.
After step n we have at most $c = \sum_{k=0}^{k_0} c_k$ applications of rules $r \in RUL$ leading to $G_0 \Rightarrow^* G_{k_0+1}$, which implies termination.
This completes the proof of Lemma 3. □

Theorem 1 (Termination of Layered Graph Grammars). *Each layered graph grammar with injective matches terminates.*

Proof (Theorem 1). Starting with $k = 0$ we can apply for each deletion layer the deletion layer conditions (see Lemma 1) and for each nondeletion layer the nondeletion layer conditions (see Lemma 3). □

4 Termination Analysis

4.1 Termination analysis of the running example

Now we apply the results (of Sec. 3) to prove the termination of the model transformation of Sec. 2 from UML statecharts to Petri nets. Therefore, we first assign the rules of Fig. 2 to four layers (three nondeletion and one deletion layer). Then the creation and deletion layers of labels (types) in the metamodel of Fig. 1 are set to respect Def. 1. Finally, the check of the conditions in Def. 1 yields the termination of the transformation according to Theorem 1.

Assigning rule layers. Let us define four layers for the model transformation rules of Fig. 2 as follows:

Layer 0	Layer 1	Layer 2	Layer 3
nondeletion	nondeletion	nondeletion	deletion
$rl(r) = 0$	$rl(r) = 1$	$rl(r) = 2$	$rl(r) = 3$
Step2TransR ActState2TokenR	State2PlaceR QueueEvent2PlaceR InQueueEvent2TokenR	StepFrom2InArcR StepTo2OutArcR Trigger2InArcR Action2OutArcR	delNodeStepR delEdgeSrcStepR

Assigning layers to labels (types). We define a possible way to automatically assign creation and deletion layers to each label (type) in the metamodel based upon the previous layer definitions for rules.

Definition 4 (Layer assignments). *If we have a start graph G_0 with start labels $T_0 \subseteq LAB$ and then we can define for each $l \in LAB$ the creation and deletion layers as follows*

$cl(l) = \underline{if}\ l \in T_0\ \underline{then}\ 0\ \underline{else}\ max\{rl(r)|r\ creates\ l\} + 1$
$dl(l) = \underline{if}\ l\ is\ deleted\ by\ some\ r\ \underline{then}\ min\{rl(r)|r\ deletes\ l\}\ \underline{else}\ n$

As only the elements in the source language are present initially in a model transformations, exactly those labels are included in the start labels T_0. Now the creation and deletion layer of labels are assigned as follows (only a subset of labels are considered due to space limitations).

Src label l_s	$cl(l)$	$dl(l)$	Ref label l_r	$cl(l)$	$dl(l)$	Trg label l_t	$cl(l)$	$dl(l)$
State	0	3	RefState	2	3	Place	2	4
Step	0	3	RefStep	1	3	Trans	1	4
Queue	0	3	RefQEvnt	2	3	Token	2	4
Event	0	3				InArc	3	4
Conf	0	3				OutArc	3	4

Checking conditions of termination. Finally we show how the sufficient conditions of deletion and nondeletion layers in Def. 1 are fulfilled by the previous layer assignments.

- *Nondeletion Layer Conditions.* First, we notice that Conditions 1 and 2 are straightforwardly guaranteed by the construction (as NAC is isomorphic/identical with RHS). Now we only show the validity of Condition 3 and 4 for a single rule, namely, r = StepFrom2InArcR (while the rest of the rules can be checked similarly). In Condition 3, for each graph element x in the LHS, we need to check $cl(label(x)) \leq rl(r)$, which holds according to the layer assignments above (as $max_{x \in L}\{cl(label(x)\} = 2$ and $rl(r) = 2$). Condition 4 states that $cl(l) > rl(r)$ for all l created by r which is justified by $cl(\mathsf{InArc}) = 3$ and $rl(r) = 2$ (and similar reasoning on edges).
- *Deletion Layer Conditions.* As a first observation, Condition 1 trivially holds for the deletion rules of Layer 3. Condition 2 can be verified according to the table above. Since all deletion rules in Fig. 2 are included in the last layer, Condition 3 holds directly. Finally, the fact that these deletion rules do not create new elements implies Condition 4.

Furthermore, we carried out critical pair analysis using the AGG system [1], which builds upon the confluence results of [13]. As the graph transformation rules within a single layer are free of critical pairs (potential conflicts), we can conclude that our model transformation is a well-defined function from the class of statecharts to that of Petri nets (i.e. it yields a unique result).

4.2 Further case studies

From the General Resource Model (GRM) to Petri nets. In order to provide Petri net-based simultaneous optimization and verification of resource allocation problems, in [8] we aim at generating the application specific Petri net model from a variant of the *General Resource Modeling framework (GRM)* [19] using attributed graph transformation. The graph transformation system (implemented in AGG [1]) consists of five rule layers as follows (where layers 0 and 2 are nondeletion layers while the others are deletion layers):

0. Target model elements are derived from core GRM elements like resource types, activities, and control flow elements;
1. Petri net transitions and arcs are created between the already transformed Petri net items according to the control flow in the source model;
2. The start and the end points of the process are marked by auxiliary edges;
3. The quantitative attributes of the Petri net elements are set;
4. All the auxiliary edges and the source model elements are deleted.

Since the rules are applied to the host graph using injective matches only, and the GTS with a valid source model satisfies the Layered Graph Grammar Definition (Def. 1), the graph grammar fulfills the termination criteria of Theorem 1 hence the graph grammar terminates.

From Process Interaction Diagrams to Timed Petri nets. In [9], a model transformation from a Process Interaction notation to Timed Transition Petri nets is specified using graph transformation. The source language is customized towards the area of manufacturing and allows building and simulating networks of machines and queues through which pieces can flow. For the mapping, timed transitions depict service times of machines, places are used to model queues and machine states, and finally pieces are mapped to tokens. The transformation was divided in four layers, the first one being nondeleting, while the rest are deleting. The first layer creates Petri net elements connected to the source elements. Rules in the second layer delete the pieces in the model, creating tokens in the appropriate places. In the third layer, we connect the Petri net elements following the connectivity of the source language elements. In addition, the connectivity of the Process Interaction elements is deleted. Finally, the last layer deletes the Process Interaction elements. The languages and the transformation were defined with the AToM^3 tool [10], and then analyzed using AGG.

5 Related Work

Termination of graph transformation systems is undecidable in general [22], but several approaches have been considered to restrict a graph transformation system such that termination can be shown. The classical approach of proving termination is to construct a monotone function that measures graph properties, and to show that the value of such a function decreases with every rule application. Concrete criteria such as the number of nodes and edges of certain types have already been considered by Aßman in [2]. However, he sticks to these concrete criteria, while Bottoni et.al. [5] developed a general approach to termination based on measurement functions.

With respect to termination for graph transformation systems, the current work generalizes and formalizes the work begun at [9]. This, in fact, is an extension of the layering conditions for deleting grammars proposed in [6], which were used for parsing.

With respect to the transformation from Statecharts into Petri nets, in [10] graph grammars were also used to describe the translation. In that approach, Statecharts were restricted to be flat (no hierarchy), termination was not proven and intermediate elements for linking source and target language elements were not formally defined.

6 Conclusion

In this paper, we have presented termination criteria for model transformation expressed as graph transformation. The criteria are based on dividing the grammar in (deleting or nondeleting) layers. A running example, showing a transformation from Statecharts into Petri nets was verified to be terminating. The applicability of the criteria to other examples was also discussed. The proposed termination checks will be available in AGG soon.

In the future, it will be interesting to extend the termination criteria to graph grammars with *abstract rules* [3]. These rules may contain nodes whose typing is abstract, and are equivalent to all the rules resulting from the substitution of the abstract nodes by nodes in its inheritance tree. This extension would allow to use type graphs with inheritance for the definition of the source and target languages in a model transformation.

A further direction of future research is to prove semantic compatibility between statecharts and Petri nets. A potential solution is to first capture the operational semantics of statecharts and Petri nets in the form of graph transformation rules (see [27, 26]) and then show the bisimilarity of the two graph transformation systems at a certain level of abstraction.

References

1. AGG Homepage, `http://tfs.cs.tu-berlin.de/agg`.
2. Aßmann, U. 2000. *Graph Rewrite Systems for Program Optimization.* ACM TOPLAS, vol. 22(4), pp. 583–637, ACM Press, New York.
3. Bardohl, R., Ehrig, H., de Lara J., and Taentzer, G. 2004. *Integrating Meta Modelling with Graph Transformation for Efficient Visual Language Definition and Model Manipulation.* In proceedings of ETAPS/FASE'04, LNCS 2984, pp.: 214-228. Springer.
4. Baresi, L., Pezzé, M. 2001. *Improving UML with Petri-Nets.* ENTCS 44 (4).
5. Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G. 2004. *Termination of High-Level Replacement Units with Application to Model Transformation.* In proceedings of VLFM'04, ENTCS.
6. Bottoni, P., Taentzer, G., Schürr, A. 2000. *Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation.* In Proc. Visual Languages 2000 IEEE Computer Society. pp.: 59-60.

7. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. *In [25]*, chap. Algebraic Approaches to Graph Transformation — Part I: Basic Concepts and Double Pushout Approach, pp. 163–245. World Scientific, 1997.
8. S. Gyapay. Model transformation from General Resource Models to Petri nets using graph transformation. Tech. rep., Technical University of Berlin, Dept. of Computer Science, July, 2004.
9. de Lara, J., Taentzer, G. 2004. *Automated Model Transformation and its Validation with AToM^3 and AGG.* In DIAGRAMS'2004 (Cambridge, UK). Lecture Notes in Artificial Intelligence 2980, pp.: 182-198. Springer.
10. de Lara, J., Vangheluwe, H. 2002 *Computer Aided Multi-Paradigm Modelling to process Petri-Nets and Statecharts.* In ICGT'2002. LNCS 2505. Pp.: 239-253. Springer.
11. Ehrig, H. 1979. *Introduction to the Algebraic Theory of Graph Grammars.* In V. Claus, H. Ehrig, and G. Rozenberg (eds.), 1st Graph Grammar Workshop, pages 1-69. Springer LNCS 73, 1979.
12. Ehrig, H., Habel, A., Padberg, J., Prange, U. 2004. *Adhesive High-Level Replacement Categories and Systems.* In Proc. 2nd Int. Conference on Graph Transformation (ICGT'04). Springer LNCS 3256, pp. 144–160.
13. Ehrig, H., Prange, U., Taentzer, G. 2004. *Fundamental Theory for Typed Attributed Graph Transformation.* In Proc. 2nd Int. Conference on Graph Transformation (ICGT'04). Springer LNCS 3256, pp. 161–177.
14. Guy, J., Leicher, A., Busse, S. 2003. *OCLPrime – Environment and Language for Model Query, Views and Transformations.* In OCL Workshop at UML'2003. San Francisco.
15. Habel, A., Heckel, R., Taentzer, G. 1996. *Graph Grammars with Negative Application Conditions*, Special issue of Fundamenta Informaticae 26(3-4): 287-313.
16. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, vol. 8(3):pp. 231–274, 1987.
17. Heckel, R., Küster, J., Taentzer, G. 2002. *Towards Automatic Translation of UML Models into Semantic Domains* . In Proc. of APPLIGRAPH Workshop on Applied Graph Transformation (AGT 2002).
18. R. Heckel, J. M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg (eds.), *Proc. ICGT 2002: First International Conference on Graph Transformation*, vol. 2505 of *LNCS*, pp. 161–176. Springer, Barcelona, Spain, 2002.
19. Object Management Group. *UML Profile for Schedulability, Performance and Time.* `http://www.omg.org`.
20. Object Management Group. *QVT: Request for Proposal for Queries, Views and Transformations.* `http://www.omg.org`.
21. Pataricza, A. Semi-decisions in the validation of dependable systems. In *Suppl. Proc. DSN 2001: The International IEEE Conference on Dependable Systems and Networks*, pp. 114–115. Göteborg, Sweden, 2001.
22. Plump, D. 1998. *Termination of Graph Rewriting is Undecidable.* Fundamenta Informaticae 33(2):201-209.
23. QVT-Partners. *Revised submission for MOF 2.0 Query / Views / Transformations RFP.* `http://qvtp.org`.
24. Raistrick, C., Francis, P., Wright, J., Carter, C., Wilkie, I. 2004. *Model Driven Architecture with Executable UML.* Cambridge University Press. Cambridge, UK. See also the MDA home page at OMG web site: `http://www.omg.org/mda/`.
25. Rozenberg, G. (ed) 1997. *Handbook of Graph Grammars and Computing by Graph Transformation.* World Scientific. Volume 1.

26. Varró, D. A formal semantics of UML Statecharts by model transition systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg (eds.), *Proc. ICGT 2002: 1st International Conference on Graph Transformation*, vol. 2505 of *LNCS*, pp. 378–392. Springer-Verlag, Barcelona, Spain, 2002.
27. Varró, D. *Automated Model Transformations for the Analysis of IT Systems.* Ph.D. thesis, Budapest Univ. of Technology and Economics, Dept. of Measurement and Information Systems, 2004.
28. Varró, D., Varró, G., and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, vol. 44(2):pp. 205–227, 2002.

A Proofs

Lemma 2. *In each derivation sequence starting from G_0 each rule $r : L \to R$ with $r \in RUL_0$ can be applied at most once with the same 'essential match' $m_0 : L \to G_0$ and $m_0 \models NAC$.*

Proof (Lemma 2). Assume that in $G_0 \Rightarrow^* H_1$ rule r has been applied with the same 'essential match' m_0 already. This means we can decompose $G_0 \Rightarrow^* H_1$ into $G_0 \Rightarrow^* G \Rightarrow H \Rightarrow^* H_1$ with pushout(1) and injective morphisms $G_0 \xrightarrow{g} G \xrightarrow{d} H \xrightarrow{h_1} H_1$ satisfying $d_1 = h_1 \circ d \circ g$ and $d_1 \circ m_0 = m_1$ in Figure 3.

$$\begin{tikzcd}
& L \arrow[r, "r"] \arrow[d, "m"] \arrow[dl, "m_0"'] \arrow[dr, phantom, "(1)"] & R \arrow[d, "m^*"] & L \arrow[r, "r"] \arrow[d, "m_1"'] \arrow[dr, "n"] \arrow[dlll, bend right, "m_0"'] & R \\
G_0 \arrow[r, "g"'] \arrow[rrr, bend right, "d_1"'] & G \arrow[r, "d"'] & H \arrow[r, "h_1"'] & H_1 & N \arrow[l, dotted, "q_1"] \arrow[u, "n'"']
\end{tikzcd}$$

Fig. 3. Second Application of Rule r with same essential match m_0

In order to prove the lemma now it is sufficient to show that
$m_1 : L \to H_1$ does not satisfy the NAC of r, i.e. $m_1 \not\models NAC$, where the NAC is given by an injective morphism $n : L \to N$ with $n' : N \to R$ injective satisfying $n' \circ n = r$ by condition 2. In fact we are able to construct an injective $q_1 : N \to H_1$ with $q_1 \circ n = m_1$.
Let $q_1 = h_1 \circ m^* \circ n'$, then q_1 is injective because n', m^* and h_1 are injective and injectivity of m^* follows from injectivity of match m. Moreover we have:

$$q_1 \circ n = h_1 \circ m^* \circ n' \circ n = h_1 \circ m^* \circ r = h_1 \circ d \circ m = h_1 \circ d \circ g \circ m_0 = d_1 \circ m_0 = m_1$$

This completes the proof of lemma 2. □