Towards Translating Graph Transformation Approaches by Model Transformations

Frank Hermann* and Harmen Kastenberg and Tony Modica***
*Institut für Softwaretechnik und Theoretische Informatik, Technische Universität Berlin, Germany
{frank,modica}@cs.tu-berlin.de
**Department of Computer Science, University of Twente, The Netherlands
h.kastenberg@cs.utwente.nl

Abstract. *Recently, many researchers are working on semantics preserving model transformation. In the field of graph transformation one can think of translating graph grammars written in one approach to a behaviourally equivalent graph grammar in another approach. In this paper we translate graph grammars developed with the* GROOVE *tool to* AGG *graph grammars by first investigating the set of core graph transformation concepts supported by both tools. Then, we define what it means for two graph grammars to be behaviourally equivalent, and for the regarded approaches we actually show how to handle different definitions of both - application conditions and graph structures. The translation itself is explained by means of intuitive examples.*

Keywords: graph transformation, bridging languages, preserving semantics, tools

1 Introduction

Models in general are representations of certain structures, fulfilling some properties, which may be given by a specification. Transforming those models can be defined in many ways, e.g. by XSLT style sheets [18] developed by the W3C to keep the format but possibly change the internal structure or semantics. Furthermore the finalization of OMG's language QVT [10] is underway; there are also implementations, e.g. for Eclipse: GMT [3]. Besides, there is also graph transformation, putting model transformation on a formal basis. The transformation is defined by rules, which are defined by pure mathematical constructs, but using a very intuitive visual notation. During the last three decades there has been much interest in developing suitable approaches and analyzing their properties in means of correctness, concurrency, termination and confluence. Most of them consider categorical constructs like the single, double, and triple pushout approach, single and double pullback, but also triple graph grammars [14] being especially suitable for specifying the connections between a source and target model. Transforming UML models by graph transformation rules is especially supported by the tool VIATRA [2], which is part of the mentioned GMT project.

Again graph transformation systems are models itself, so they can be translated to other models. And there are several tools supporting the definition of a graph transformation system as well as their simulation and analysis. Some of them also support an export to XMI or to XML formats like the Graph eXchange Language (or GXL for short) [15], but the problem is to import such a model in a tool, which uses a different approach. Even if an import is possible, then the behaviour of the system is not necessarily equal.

Two existing graph transformation tools are GROOVE [11], which is mainly developed for state space generation and model checking, and AGG [17] supporting simulation and analysis of graph transformation systems. The translation between them shows on the one hand, how to bridge gaps between their differences in the formal approaches, and on the other hand it facilitates the elaboration of core concepts of graph transformation in general. The main goal of the translation is the possibility to combine the features of both tools.

There are already special file formats to exchange graphs and whole graph transformation systems between tools. GXL supports the exchange of graphs, which is used e.g. between FUJABA [9] and PROGRES [13]. However, its extension GTXL [16] for graph transformation systems stores graph transformation systems syntactically, but the behaviour in other tools is often far from equivalent. Special attention is asked to encode rules in a different approach, as the tools base on different approaches in general.

In this paper we describe the translation between the mentioned tools GROOVE and AGG and define what it means for graph grammars in both approaches to be behaviourally equivalent. Despite the fact that these tools look quite similar when comparing the approaches – both use SPO – there have been interesting challenges. As GROOVE handles simple graphs, i.e. graphs with simple edges only, we had to ensure that AGG will not create parallel edges, which it allows in general. Additionally, the definitions of (negative) application conditions are different, slightly when looking at the mathematical definitions, but complex when trying to translate them while preserving the semantics.

The paper is structured as follows. Sect. 2 gives a short introduction to the basics of graph transformation. In Sect. 3 we introduce the two approaches playing a central role in this paper, discuss their differences and define an equivalence relation. In Sect. 4 and Sect. 5 we elaborate on the translation from one approach to the other and vice versa, illustrating the necessary steps using simple examples. We will conclude with a short discussion and some remarks about further work.

2 Preliminaries

The foundations of graph transformation were developed in the early seventies, e.g. in [4], to extend the common formalisms of one-dimensional textual rewriting to a more complex level. In the following years various approaches were defined keeping one big advantage in common – they automatically preserve the specified graph structure.

In general, a graph $G = \langle N, E, \mathsf{src}, \mathsf{tgt} \rangle$ consists of a set N of *nodes* and a set E of *edges*, with *source* and *target* functions $\mathsf{src}, \mathsf{tgt} \colon E \to N$. The global set of graphs is denoted $\mathcal{G}$ and ranged over by G, H.

Modelling system states as graphs facilitates the specification of system evolution. Graph transformation rules are used to intuitively define in what sense the system state changes. Such a rule $p : L \to R$ consists of a graph L_p (the left-hand-side, or LHS) and a graph R_p (the right-hand-side, or RHS) together with a graph morphism r_p mapping nodes and edges of L_p to those of R_p, and a set of so-called *application conditions* (AC_p, which are supergraphs of L_p). The application of a graph transformation rule p transforms a graph G, the *source graph*, into a graph H, the *target graph*, by looking for an occurrence of L_p in G (specified by a graph matching m that satisfies the extension conditions of all AC_p) and then replacing that occurrence with R_p, resulting in H. Such a rule application is denoted as $G \xRightarrow{p,m} H$. Here we present the most general definitions using the SPO approach, which can be restricted to the DPO approach by adding two conditions to the application of a rule: (1) identification and (2) dangling condition. The former requires that every element that should be deleted in the source graph has only one pre-image in the LHS of the rule; the latter requires that if a node n is deleted, the rule must specify the deletion of all edges incident to n. In both approaches transformation rules can be equipped with appropriate application conditions, which is described in [6]. For formal analysis and a concurrent semantics definition by processes we refer to [12].

Definition 1 (Graph Production System) *A* graph production system $P = \langle \mathcal{R}, G \rangle$ *consists of a set* $\mathcal{R} = \{p : L \rightharpoonup R \mid L, R \in \mathcal{G}\}$ *of graph transformation rules and a graph* G*;* G *is said to be the* initial graph*.*

As rules of a graph production system (or GPS for short) may delete nodes and edges of a graph, the defining morphism consists of two partial functions, which is indicated by a halved arrow head.

Each graph production system P specifies a (possibly infinite) state space which can be generated by repeatedly applying the graph transformation rules on the graphs, starting from the initial graph G_P. This results in a *graph transition system*.

Definition 2 (Graph Transition System) *The* graph transition system $T = \langle S, \rightarrow, I \rangle$ *generated by a graph production system* $P = \langle \mathcal{R}, G \rangle$ *consists of a set* S *of states which are actually graphs* ($S \subseteq \mathcal{G}$)*; a transition relation* $\rightarrow \subseteq S \times \mathcal{R} \times [\mathcal{G} \rightarrow \mathcal{G}] \times S$*, such that* $\langle G, p, m, H \rangle \in \rightarrow$ *iff there is a rule application* $G \xRightarrow{p,m} H'$ *with* H' *isomorphic to* H*; and an initial state* I*.*

3 Approaches

The two approaches for which we define a semantics preserving translation are discussed in this section. We will mention the formalisms used for representing models and for specifying their transformations.

3.1 The GROOVE Approach

The main goal of the GROOVE tool [11] is to use graphs as a formalism to model system states and graph transformation rules to specify system behaviour and perform model checking on state spaces that can be generated by repeatedly applying the rules on the states. The main advantage of using graphs for modelling system states, instead of bit vectors, as used by many other model checking approaches, is the possibility to cope with the dynamic character of systems more naturally [7].

The Formalism. In GROOVE we support the use of non-typed attributed graphs, where graphs are set-based models consisting of three distinct sets: a set N of *nodes*, a global set L of *labels*, and a set $E \subseteq N \times L \times N$ of *edges*. Nodes are non-structured elements having a unique identity; edges, on the other hand, are identified by means of their end-points and their label, i.e. for an edge $e = \langle n_1, l, n_2 \rangle \in E$ we distinguish its *source*, *label*, and *target*, denoted by $\mathsf{src}(e)$, $\mathsf{lab}(e)$, and $\mathsf{tgt}(e)$. As a consequence, it is not possible to have *parallel edges*, i.e.

$$\forall e_1, e_2 \in E : \mathsf{src}(e_1) = \mathsf{src}(e_2) \wedge \mathsf{lab}(e_1) = \mathsf{lab}(e_2) \wedge \mathsf{tgt}(e_1) = \mathsf{tgt}(e_2) \Rightarrow e_1 = e_2$$

Currently, GROOVE supports the use of negative application conditions (or NAC for short) in conjunctive form, but one NAC cannot contain more than one connected component. The rule morphism may be non-injective as well as the matchings to the host graphs. GROOVE does not support typing, but uses the edge-labelling as a typing-mechanism instead.

For performing graph transformations, GROOVE applies the SPO-approach.

Input/Output. A GROOVE graph grammar is saved in the GXL [15] XML-format. Rules are saved as single graphs, in which the rule-roles (preserve, create, delete, and NAC) are encoded in edge-labels by adding structure to the labels. A graph production system consists of all the rules in a single directory (as well as its subdirectories). In the future we plan to support the special-purpose format GTXL.

Special features. The main feature of the GROOVE engine is its ability to generate state spaces from graph grammars. During state space generation, it checks for the occurrence of isomorphic states. Furthermore, a CTL [1] model checking algorithm has been implemented checking temporal properties in which graph structures can be used as atomic propositions. In the future we plan to implement partial order reduction techniques based on confluence properties of transformation rules as well as abstraction techniques which enable the verification of larger (or possibly infinite) system models.

3.2 The AGG Approach

According to the complete formalization of the SPO approach by Michael Löwe [8] in 1990 the AGG tool [17] was developed to support an editor for graph grammars, which also offers simulation and analysis capabilities by certain criteria including termination and confluence as the most important ones. Further developments integrated high level features being for instance attribution and typing. Therefore, AGG builds a basis for various fields, e.g. formal model transformation controlled by graph transformation, but also the definition of visual modelling languages with the possibility of an automatic editor generation, using Tiger [5].

The Formalism. Similar to GROOVE the AGG tool uses the SPO approach to perform graph transformations, but the fundamental description of graphs differs from that in GROOVE. Graphs with multiple edges between nodes are allowed as long as the multiplicity constraints in the type graph are fulfilled. The typing itself is handled by a type graph T, which includes all node and edge types and for each graph G there is a graph morphism $t : G \to T$ to this type graph. In this way, the type graph defines the general structure of all instances. Rules are visualized by separate graphs for the left and right hand side as well as for the application conditions. The morphisms of a rule, which define the deletion, preservation, creation and forbidden parts, are indicated by a unique naming of the nodes. Additionally, negative application conditions are handled differently in comparison to GROOVE.

$$\begin{tikzcd}
N \arrow[dr, "\nexists\, i"'] & L \arrow[l, "n"'] \arrow[d, "m"] \arrow[dl, phantom, "(=)"] \\
 & G
\end{tikzcd}$$

(a) Simplified definition of NACs

$$\begin{tikzcd}
N \arrow[d, "n'"'] \arrow[dr, phantom, "PO"] & L \arrow[l, "n"'] \arrow[d] \arrow[dd, bend left=60, "m"] \\
N' \arrow[dr, "\nexists\, i"'] \arrow[dr, phantom, "(=)"] & m(L) \arrow[l, hook'] \arrow[d, hook] \arrow[r, phantom, "="] & {} \\
 & G
\end{tikzcd}$$

(b) Complete definition of NACs

Figure 1: NACs in AGG

Fig. 1(a) explains a simplified view to NACs in AGG. The LHS L is embedded in the NAC N, where identifications of parts in LHS are possible. Given a match m for the rule, a negative application condition N is satisfied, if there is no morphism i with additional restrictions from N to the G making the triangle commutative. This means that the condition forbids a certain structure around the image of L in G. The morphism i has to be injective on the part of N, which is not reached by n and of course identifies the same things as m does. In other words, i is m extended injectively by the remaining elements. Fig. 1(b) shows the complete formal version, where N' is created at runtime, if it exists. It represents N after identifying the elements in N according to the identifications by m for the corresponding elements in L. Now the rule is applicable at m, if there is no injective morphism i making the outer arrows commuting: $i \circ n' \circ n = m$. It is sufficient that the bottom left triangle commutes, because $L \to m(L)$ is just m restricted to its image and therefore surjective. In comparison to GROOVE, where the morphism i is not necessary injective, this definition of NACs is a restriction, but also an extension as multiple forbidden identifications of elements in L cannot be handled in one GROOVE NAC.

Input/Output. AGG features several XML-based file formats to exchange graphs and transformation systems. Internally, AGG uses the GGX format. It can import and export graphs in GXL, also used by GROOVE. AGG can also export graph grammars using the special format GTXL which is an extension of GXL for storing entire graph transformation systems. Finally, models generated by Eclipse modelling plugin Omondo (in OMONDO XMI) can be imported by AGG.

Special features. One main advantage of AGG in comparison to other tools is its possibility to specify the desired graph transformation approach. The DPO approach can be used by activating the dangling and the identification condition. Possible extensions are attribution, typing, node type inheritance, and multiplicities for the structure part and rule amalgamation as well as application levels for the control part. Those extensions are also of help when using the second advantage of AGG: its analysis capabilities. Critical pairs between rules can be computed and help to show confluence, while termination can be checked, which can be supported by the usage of levels. Finally the simulation part allows applying formal model transformation with the certainty to reach a valid result in means of typing in the target language.

3.3 Behavioural Equivalence

In this paper, graph production systems in the different approaches are related to each other by defining a bi-directional equivalence (or simulation) relation between the graph transition systems generated by them.

Definition 3 (behavioural equivalence) *Given two graph production systems* $P_1 = \langle \mathcal{R}_1, G_1 \rangle$ *and* $P_2 = \langle \mathcal{R}_2, G_2 \rangle$ *and their generated graph transition systems* $T_1 = \langle S_1, \rightarrow_1, I_1 \rangle$ *and* $T_2 = \langle S_2, \rightarrow_2, I_2 \rangle$*, respectively, we say that* P_2 *is* behaviourally equivalent *to (or* simulates*)* P_1 *if:*

$$\forall t_i = \langle s_i, p, m, s_{i+1} \rangle \in \rightarrow_1 \exists t'_i = \langle s'_i, p', m', s'_{i+1} \rangle \in \rightarrow_2 \colon p' \in \mathsf{tr}(p) \wedge s'_i = \mathsf{tr}(s_i) \wedge s'_{i+1} = \mathsf{tr}(s_{i+1})$$

where tr is the translation function, $s_i, s_{i+1} \in \mathcal{G}$ and $p \in \mathcal{R}$.

Restrictions. In the translation from graph production systems in GROOVE to behaviourally equivalent ones in AGG, and vice versa, we restrict to those GPSs not using node or edge-attribution. Typing in AGG is flattened to labelled-edges in GROOVE. For both approaches we require injective rule-morphisms, while non-injective rule matchings are allowed. When translating GPSs from AGG to GROOVE we require $n \colon L \rightarrow N$ (see Fig. 1(b)) to be an inclusion, because in GROOVE it is only possible to express so called *merge-embargoes* pair-wise.

Translation Issues. When translating between GROOVE and AGG, we need to pay special attention to: (1) parallel edges and (2) application conditions. In the GROOVE approach parallel edges are not supported. For rules it is allowed to specify the creation of specific edges without requiring their absence. The application of such rules identifies the freshly created edges with the already existing ones.

On the other hand, when translating AGG rules to GROOVE we need to introduce a mechanism which enables the creation of parallel edges. Therefore, we will create a 'structured edge' in GROOVE to reflect the original AGG edges (see Fig. 6 in Sect. 5).

As mentioned in Sect. 3, both approaches use a slightly different definition of application conditions. Where GROOVE only allows negative application conditions containing a single connected component, AGG supports both positive and negative application conditions containing multiple components. Additionally, in AGG the matchings from L to G via the NACs must be injective on the elements that are only in the NAC, where GROOVE allows any matching. Therefore, the translation of NACs from GROOVE to AGG is more involved than simply copying. In the translation from AGG to GROOVE we need to handle NACs with multiple components and make sure that the NAC elements are matched injectively by merge embargoes.

4 GROOVE to AGG

When translating GROOVE rules to AGG, the basic idea is to keep track of edges being created by rules not having a negative application condition which forbids this edge to be present already between the incident nodes. In general, we have to create a number of AGG rules together describing the same behaviour as the original rule. This number is (in the worst case) exponential in the number of edges created by the GROOVE rule.

The Algorithm. Given a GROOVE rule p, we first take the nodes of L_p and R_p and create counterparts in $L_{p'}$ and $R_{p'}$ for each node and build up the rule-morphism. When translating the edges we have to deal with preserved, deleted, and created edges. The first two types can be translated easily. Before taking care of the created edges we first translate the NACs. Every NAC in a GROOVE rule, in general, results in a set of NACs in the AGG rule, in such a way that the AGG NACs are all possible identifications of the NAC-only elements with other NAC elements from the GROOVE rule.

For the edges that are created we need to perform some additional checks. If at least one of the incident nodes of the created edge is also created, we can do the translation straightforward. The same holds for the case when the rule contains a NAC prohibiting this edge. In all other situations, we have to copy the AGG rule created so far. The rules will create a subset of those edges, contain a corresponding NAC prohibiting them and the rest of those edges are preserved, instead of creating parallel edges.

Example 4 *Consider the* GROOVE *rules depicted in Fig. 2, modelling a person who can get a driving license by attending driving school, but possibly lose it again. The first rule applied to a person object creates the edge* gotDLicence *at it. As in real life, applying this rule to a person who already has a driving license will not create another edge because of* GROOVE *not supporting parallel edges, i.e. one can not gather several driving licenses.*

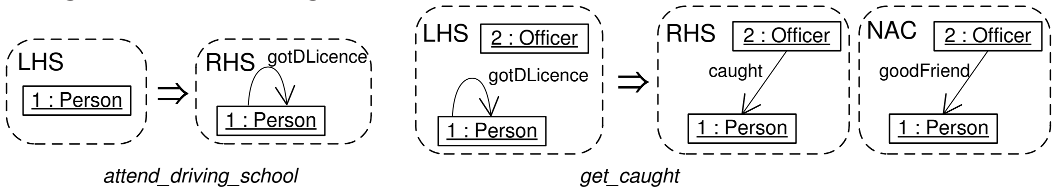

Figure 2: Rules specifying a person getting and losing a driving license.

The second rule states that a police officer can catch a person with a driving license breaking some law. The person will lose his license in this case. The negative application condition prohibits the rule to be applied when the person is a good friend of the police officer. In GROOVE NAC*s can be non-injectively matched. In this example, the officer himself could be the driving person and assuming most people like themselves, the officer would not fine himself, i.e. the rule would not be applicable.*

The resulting rules created by the algorithm for the left rule of Fig. 2 are shown in Fig. 3. The left rule of Fig. 3 shows the rule creating the gotDLicence*-edge with a corresponding* NAC*; the right rule represents the case in which* AGG *must preserve the edge when it is already there, instead of creating a parallel edge.*

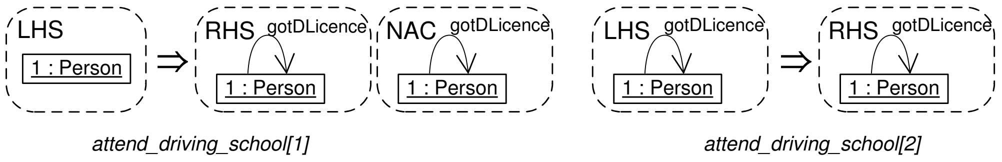

Figure 3: AGG rules created for the rule *attend_driving_school*

Fig. 4 shows the two rules together behaving equivalent to the right rule of Fig. 2. The upper rule creates the caught*-edge, the lower rule preserves it.*

Simulation. In order to prove the correctness of the translation we show behavioural equivalence as defined in Definition 3, i.e. a derivation can be performed in AGG if and only if there is a corresponding derivation in GROOVE. The next theorem also shows that this simulation starts with the same graph and also ends with the same resulting graph.

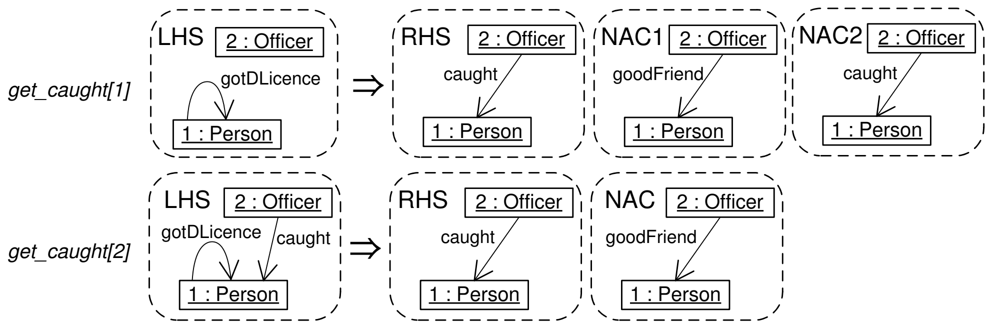

Figure 4: AGG rules created for the rule *get_caught*

Theorem 5 *Given a derivation* $G \overset{r,m}{\Longrightarrow} G'$ *of a grammar in* GROOVE *there is exactly one corresponding derivation starting at* G*, which can be performed in* AGG*. Furthermore, the resulting graph in* AGG *is again* G' *and if the rule is not applicable in* GROOVE *it is not in* AGG *via the corresponding match.*

$$\text{GROOVE:}\quad \begin{tikzcd} N[i] & L \arrow[l, "n_i"'] \arrow[r, tail, "r"] \arrow[d, "m"'] \arrow[dr, phantom, "PO"] & R \arrow[d, "o"] \\ & G_1 \arrow[r, tail, "p"'] & G_2 \end{tikzcd} \quad \longrightarrow \text{AGG:}\quad \begin{tikzcd} N'[k] & L' \arrow[l, "n'_k"'] \arrow[r, tail, "r'"] \arrow[d, "m'"'] \arrow[dr, phantom, "PO"] & R \arrow[d, "o"] \\ & G_1 \arrow[r, tail, "p"'] & G_2 \end{tikzcd}$$

Proof:
(Sketch). Let $G_1 \overset{r,m}{\Longrightarrow} G_2$ be a derivation in GROOVE as depicted in Fig. 5(a). The type graph in AGG

$$\begin{tikzcd} L \arrow[r, "r"] \arrow[d, "m"'] \arrow[dr, phantom, "(=)"] & R \arrow[d, "n"] \arrow[dl, bend left, "m_c"] \\ G \arrow[r, "p"'] & G' \end{tikzcd}$$

(a) Construction of m'

$$\begin{tikzcd} N[i] \arrow[d] & & L \arrow[ll, hook', "n_i"'] \arrow[dl, "n_1"'] \arrow[dll, dotted, "\hat{n}_i"] \\ \hat{N}[i] & n(L) \arrow[ul, hook'] \arrow[l, "n_2"'] \arrow[u, phantom, "(=)"] \arrow[l, phantom, "(=)", near end] & \end{tikzcd}$$

(b) Construction of NACs $\hat{N}$

Figure 5: Creation of m' and NACs

is just the initial graph. When translating one rule to a set of new rules the right hand side is preserved and also the nodes of L and the nodes of the different NACs $N[i]$ are translated identically. New NACs and additional edges may be added.

A NAC $N[i]$ is transformed to a set of NACs for each possible overlapping of NAC only elements with elements in $N[i]$, which are allowed to be identified. This is shown in Fig. 5(b), where $\hat{N}[i]$ is one of the new NACs with $\hat{n}_i : L \to \hat{N}[i]$. Additionally, for each merge embargo NAC between two nodes in L, one separate NAC is created, forbidding this identification. In the following, the new NACs are denoted by $\hat{N}[j]$.

Additional edges have to be created when preserving edges found in a graph instead of creating parallel ones. Therefore, we consider the edges E_c, which are created by the rule r between two existing nodes, and which are not forbidden by any NAC:

$$E_c = \{e \in E_R \backslash r(E_L) \mid \mathsf{src}(e) \in r(V_L),\ \mathsf{tgt}(e) \in r(V_L),\ \forall \hat{N}[j] : e \notin E_{\hat{N}[j]}\}$$

For each subset $E_c' \subseteq E_c$ containing the edges to be preserved, a new rule is created, which extends the identical rule by the following regulations.

- $E_{L'} = E_L \uplus E'_c$, s.t. this rule can be applied if the new edges occur in G,
- $E_{N'[j]} = E_{\hat{N}[j]} \uplus E'_c$, therefore, the NACs are extended by the new edges,
- $\forall e \in E_c \backslash E'_c$: create an extra $NAC\ N_C,\ V_{N_C} = V_L,\ E_{N_C} = E_{L'} \uplus \{e\}$, so adding an extra NAC to prevent the application of the rule, if more edges of E_c are present in G, and
- $r' = r \cup id_{E'_c}$, $n'_j = \hat{n}_j \cup id_{E'_c}$, this way the rule morphism r' and the morphisms to $\hat{N}[i]$ are extended by the new edges.
- If edges are added, the functions src and tgt for L' and the NACs are extended according to the morphisms r, n_i for the nodes and src, tgt of R.

The partial morphism m_c will be used to extend the original m and it is defined for all edges of the biggest $E' \subseteq E_c$ and their source and target nodes, s.t. the triangle in Fig. 5(a) commutes: $m_c \circ r = m$. All rule instances, which contain an edge $e \in E'$ in a new NAC are not applicable and cannot produce a parallel edge. Only the rule with E' as an extension of L' in L is applicable and as theses edges are preserved, no parallel edge is produced. Note that this holds also, if m is not injective. The matching is extended by the new edges: $m' = m \cup (m_c \circ r)$. When applying this rule via m' the result is the same as in GROOVE, because the rule deletes and produces the same, except that the preserved edges in E' are not created. The transformation $G \xRightarrow{r',m'} G'$ can be executed in AGG and because of the translated NACs, only if the corresponding rule was applicable in GROOVE via its match. $\surd$

5 AGG to GROOVE

In the translation from AGG to GROOVE we need to introduce a mechanism in GROOVE to simulate the parallel edges that are possible in AGG. Next to that, we need to handle the injectivity of NAC-only elements and NACs consisting of several components. Since the latter cannot be simulated in GROOVE by one rule with separate NACs, we have to create a set of GROOVE rules, which together behaving equivalent (see Definition 3). The number of GROOVE rules together describing the equivalent behaviour is exponential in the number of NACs.

The Algorithm. From an AGG rule p, we can first simply iterate over the nodes of L_p and R_p just as we described in the above algorithm. When iterating over the edges, we need to create the structure shown in Fig. 6 for every edge we encounter. For every $NAC\ N$ in the AGG rule we have to perform some computations. First of all, the set C_N of connected components of N needs to be determined, since for every $c_i \in C_N$ (with $0 \leq i \leq n$ and n being the number of connected components in N) we have to create a separate rule p_i, such that $L_{p_i} = L_p$ and $R_{p_i} = R_p$ and $NAC_{p_i} = c_i$. Furthermore, we have to ensure that every element in $NAC_{p_i} \setminus L_{p_i}$ is mapped injectively. This is achieved by creating NACs for all possible identifications of NAC-only nodes with all other nodes in that NAC. Finally, we have to add an extra NAC for each original node, forbidding it to match to a proxy node.

We have to admit that representing parallel edges as shown in Fig. 6 has one serious drawback. Since we use the SPO approach to perform graph transformations, deleting the source (or target) node of the original edge in AGG, will not result in the deletion of the entire edge-structure in GROOVE. This means that in such cases, the resulting graph in GROOVE still contains some *garbage*. However, since

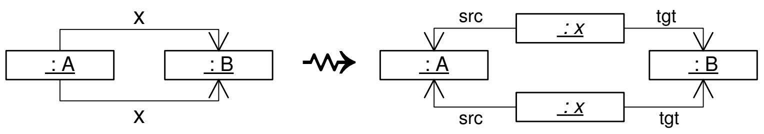

Figure 6: Graph structure in GROOVE to represent parallel edges

these ‘dead-edges’ will never be involved in derivations they can actually be ignored or collected and removed by special rules.

Example 6 *The rule in Fig. 7 creates a node* OR *with an* true*-edge for two formulae* 1 *and* 2*. It can only be applied, if not both of the matched formulas have a* false*-edge. Logically, this could be expressed as* $\neg(\neg 1 \wedge \neg 2)$ *which is equivalent to* $1 \vee 2$*.*

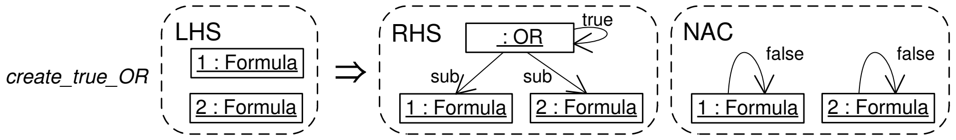

Figure 7: AGG rule for disjunctive formulae

The algorithm creates the two rules shown in Fig. 8. The main focus of this conversion is on solving the NAC *with two components and possibilities of creating parallel edges. The* NAC *is split meaning that a disjunction of two formulae* 1 *and* 2 *is evaluated to* true*, if one of them is* true*. Naturally, if both are, both generated rules are applicable, but lead to the same result:* $\neg(\neg 1 \wedge \neg 2) \equiv \neg(\neg 1) \vee \neg(\neg 2)$*.*

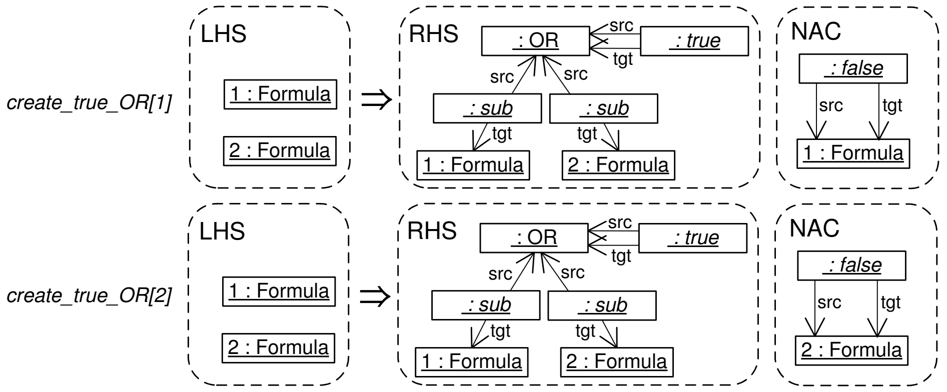

Figure 8: GROOVE rules created for the rule *create_true_OR*

Simulation. Analogue to the previous section the next theorem shows the behavioural equivalence of a system in AGG and its translation in GROOVE, according to Definition 3. Therefore, a derivation can be performed in AGG if and only if there is a corresponding derivation in the translated system in GROOVE. Again this simulation is stronger as it starts and ends with the same graph (up to the different edge-structure).

Theorem 7 *Given a derivation $G \overset{r,m}{\Longrightarrow} G'$ of a grammar in* AGG *there are corresponding derivations starting at the translated graph* $\mathtt{tr}(G)$*, which can be performed in* GROOVE*. Furthermore, all resulting graphs in* GROOVE *are again* $\mathtt{tr}(G')$ *and if the rule is not applicable in* AGG *it is not in* GROOVE *via the corresponding match.*

AGG:
$$\begin{tikzcd}
N'[k] & L' \arrow[l, hook', "n'_k"'] \arrow[r, tail, "r'"] \arrow[d, "m'"'] \arrow[dr, phantom, "PO"] & R \arrow[d, "o'"] \\
 & G'_1 \arrow[r, tail, "p'"'] & G'_2
\end{tikzcd}$$
$\longrightarrow$ GROOVE:
$$\begin{tikzcd}
N[i] & L \arrow[l, hook', "n_i"'] \arrow[r, tail, "r"] \arrow[d, "m"'] \arrow[dr, phantom, "PO"] & R \arrow[d, "o"] \\
 & G_1 \arrow[r, tail, "p"'] & G_2
\end{tikzcd}$$

Proof:
(Sketch). Let $G_1 \overset{r,m}{\Longrightarrow} G_2$ be a derivation in AGG with type graph TG. The set of labels $L = V_{TG} \uplus E_{TG} \uplus \{p, src, tgt\}$ contains all node and edge types and a special flag p indicating that a node in GROOVE is a proxy for modelling parallel edges as visualized in Fig. 6. Let G' be one of $N'[k],\ L',\ R'$ and G'_1 from AGG, its corresponding graph G in GROOVE is constructed as follows:

- $V_G = V_{G'} \uplus E_{G'}$, nodes are all original nodes and edges,
- $E_G \subseteq V_G \times L \times V_G, E_G = \{(n,l,n) \mid n \in V_G, l = type(n)\} \cup \{(e,p,e) \mid e \in E_{G'}\} \cup \{(e,src,src(e)) \mid e \in E_{G'}\} \cup \{(e,tgt,tgt(e)) \mid e \in E_{G'}\}$, new naming edges and new edges with source and target distinction,

The rule morphism $r = (r_V, r_E)$ is constructed as follows: $r_V(v) = r'(v), v \in V_L$,

$$r_E(e) = \begin{cases} (r'(e), l, r'(e)) & , \quad e = (e,l,e), l \notin \{src, tgt\} \\ (r'_E(e), src, r'_V(n)) & , \quad e = (e, src, n) \\ (r'_E(e), tgt, r'_V(n)) & , \quad e = (e, tgt, n) \end{cases}$$

The morphisms n_i and m are created analogously. For each node in $V_{L'}$ a NAC is generated, which forbids a proxy edge (e,p,e) on such a node in L. To prevent identification of NAC only elements, new NACs are created by putting merge embargo edges between all NAC only nodes. Finally the NACs of AGG containing several components are divided and distributed to identical rules using each combination of components of the different NACs. Therefore, several new rules may be applicable along m, but all lead to $G_2 = \mathtt{tr}(G'_2)$. If a NAC in AGG forbids the application there is no rule applicable via the corresponding match in GROOVE, because the NACs of a specific rule imply that all NACs in AGG are satisfied. $\surd$

6 Discussion and Conclusion

Discussion. In this paper we have shown that rules cannot be translated one-to-one between GROOVE and AGG rules behaving equivalent. In particular cases, however, the semantic domain may not allow to create parallel edges of a particular kind. Take for example a file access protocol, where processes can have different rights for accessing a file. In such systems it does not make sense to store an access right twice. GROOVE automatically ensures this constraint, while in AGG one can include a type graph in a graph production system to which all graphs and rules must have a typing morphism. In this setting, the type graph could constraint the number of specific edges. Then, applying a rule could make the graph invalid, but in such cases that rule is not applicable.

Conclusion. By comparing two graph transformation approaches, one comes to the roots of both approaches. In this paper we have shown how to transform particular graph production systems specified in GROOVE to behaviourally equivalent ones in AGG and vice versa, by transforming their building blocks: the transformation rules. We have explained how to deal with two concepts on which both approaches differ essentially, being parallel edges and application conditions. Furthermore, we have illustrated how to apply the mentioned algorithms on simple and intuitive examples. The major part of the translation has already been implemented.

Further Work. The given particular translation considered on the one hand different categories of graphs: graphs with simple or parallel edges, and on the other hand different definitions for the used control structures for the transformation: negative application conditions with partly different interpretation and expressiveness. It should be possible to extend the work on both, more differences in the graph and the control structure of transformation systems. For example the concepts of attribution and typing with inheritance seem to be a straight forward extension and they are already available in AGG. Furthermore, the generalization to arbitrary rule-morphisms instead of injective morphisms only is worth to investigate. Generalizing beyong GROOVE and AGG is difficult, since every other approach brings its own underlying formalism. The main motivation for this work was, eventually, to be able to make optimal use of the features provided by both approaches, i.e. the analysis techniques implemented in AGG and the model checking algorithm(s) implemented in GROOVE. In order to reach this, both tools need to be extended to support export and import to GXL, or better GTXL.

Acknowledgements. This work is partially supported by the SegraVis Research Training Network. The second author is employed in the GROOVE-project funded by the Dutch NWO (project number 612.000.314).

References

[1] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the IBM Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1982.

[2] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA – visual automated transformations for formal verification and validation of uml models. In *Proc. of the 17th IEEE Int. Conf. on Automated Software Engineering (ASE 2002)*, pages 267–270. IEEE Computer Society, 2002.

[3] Eclipse. The generative modeling tools. `http://www.eclipse.org/gmt/`.

[4] H. Ehrig, M. Pfender, and H.-J. Schneider. Graph grammars: An algebraic aapproach. In *Proc. of the* 14^{th} *Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.

[5] K. Ehrig, C. Ermel, S. Hänsgen, and G. Taentzer. Generation of visual editors as eclipse plug-ins. In D. F. Redmiles, T. Ellman, and A. Zisman, editors, *Proc. of the 20th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2005)*, pages 134–143. ACM, 2005.

[6] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Special issue of Fundamenta Informaticae*, 26(3,4):287–313, 1996.

[7] H. Kastenberg and A. Rensink. Model checking dynamic states in GROOVE. In A. Valmari, editor, *Proc. of the* 13^{th} *Int. SPIN Workshop on Software Model Checking (SPIN'06)*, volume 3925 of *Lecture Notes in Computer Science*, pages 299–305. Springer, 2006.

[8] M. Löwe. *Extended Algebraic Graph Transformation*. PhD thesis, Technical University of Berlin, 1990.

[9] U. A. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The FUJABA environment. In *Proc. of the* 22^{nd} *Int. Conf. on Software Engineering (ICSE)*, pages 742–745. ACM Press, 2000.

[10] QVT-Merge Group. Query/View/Transformations. `http://www.omg.org/`.

[11] A. Rensink. The GROOVE Simulator: A tool for state space generation. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE'03)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer, 2004.

[12] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume I: Foundations. World Scientific, 1997.

[13] A. Schürr. PROGRES: A VHL-language based on graph grammars. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. of the* 4^{th} *Int. Workshop on Graph-Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 641–659. Springer, 1991.

[14] A. Schürr. Specification of graph translators with triple graph grammars. In G. Tinhofer, editor, *Proc. of the* 20^{th} *Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG'94)*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 1994.

[15] A. Schürr, S. E. Sim, R. Holt, and A. Winter. The GXL graph exchange language. `http://www.gupro.de/GXL`.

[16] G. Taentzer. Towards common exchange formats for graphs and graph transformation systems. In J. Padberg, editor, *Proc. of the Workshop on Uniform Approaches to Graphical Process Specification Techniques (UNIGRA'01)*, volume 44 of *Electronic Notes in Theoretical Computer Science*, 2001.

[17] G. Taentzer, C. Ermel, and M. Rudolf. The AGG Approach: Language and tool environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformations*, volume II: Applications, Languages and Tools, pages 163–246. World Scientific, 1999.

[18] W3C. XSL Transformations. `http://www.w3.org/TR/xslt`.