A Tutorial on Graph Transformation

Barbara König[1], Dennis Nolte[1(✉)], Julia Padberg[2], and Arend Rensink[3]

[1] Universität Duisburg-Essen, Duisburg, Germany
dennis.nolte@uni-due.de
[2] Hochschule für Angewandte Wissenschaften Hamburg, Hamburg, Germany
[3] University of Twente, Enschede, Netherlands

Abstract. Graph transformation or graph rewriting has been developed for nearly 50 years and has become a mature and manifold formal technique. Basically, rewrite rules are used to manipulate graphs. These rules are given by a left-hand side and a right-hand side graph and the application comprises matching the left-hand side and replacing it with the right-hand side of the rule.

In this contribution we give a tutorial on graph transformation that explains the so-called double-pushout approach to graph transformation in a rigorous, but non-categorical way, using a gluing construction. We explicate the definitions with several small examples.

We also introduce attributes and attributed graph transformation in a lightweight form. The paper is concluded by a more extensive example on a leader election protocol, the description of tool support and pointers to related work.

1 Introduction

A substantial part of computer science is concerned with the transformation of structures, the most well-known example being the rewriting of words via Chomsky grammars, string rewriting systems [9] or transformations of the tape of a Turing machine. We focus on systems where transformations are rule-based and rules consist of a left-hand side (the structure to be deleted) and a right-hand side (the structure to be added).

If we increase the complexity of the structures being rewritten, wse next encounter trees or terms, leading to term rewriting systems [2]. The next level is concerned with graph rewriting [42], which – as we will see below – differs from string and term rewriting in the sense that we need a notion of interface between left-hand and right-hand side, detailing how the right-hand side is to be glued to the remaining graph.

Graph rewriting is a flexible and intuitive, yet formally rigorous, framework for modelling and reasoning about dynamical structures and networks. Such dynamical structures arise in many contexts, be it object graphs and heaps, UML diagrams (in the context of model transformations [13]), computer networks, the world wide web, distributed systems, etc. They also occur in other domains,

R. Heckel and G. Taentzer (Eds.): Ehrig Festschrift, LNCS 10800, pp. 83–104, 2018.
https://doi.org/10.1007/978-3-319-75396-6_5

where computer science methods are employed: social networks, as well as chemical and biological structures. Specifically concurrent non-sequential systems are well-suited for modelling via graph transformation, since non-overlapping occurrences of left-hand sides can be replaced in parallel. For a more extensive list of applications see [12].

Graph rewriting has been introduced in the early 1970's, where one of the seminal initial contributions was the paper by Ehrig et al. [17]. Since then, there have been countless articles in the field: many of them are foundational, describing and comparing different graph transformation approaches and working out the (categorical) semantics. Others are more algorithmic in nature, describing for instance methods for analysing and verifying graph transformation. Furthermore, as mentioned above, there have been a large number of contributions on applications, many of them in software engineering [12], but in other areas as well, such as the recent growing interest from the area of biology in connection with the Kappa calculus (see for instance [7]).

Naturally, there are many other formalisms for describing concurrent systems, we just mention a few: Petri nets [38] can be viewed as a special case of graph transformation, missing the ability to model dynamic reconfigurations. There are however extensions such as reconfigurable Petri nets that extend nets with additional rules so that the net structure can be changed. An overview can be found in this collection [35]. Graph transformation is similar in expressiveness to process algebra [19,32,43], but is often more flexible, since a different choice of rules leads to different behaviours. Furthermore, behavioural equivalences, well-known from process algebra, can also be defined for graph transformation [14,24,29].

The aim of this paper is not to give a full overview over all possible approaches to graph transformation and all application scenarios. Instead, we plan to do quite the opposite: in the wealth of papers on graph transformation it is often difficult to discover the essence. Furthermore, readers who are not familiar with the categorical concepts (especially pushouts) used in the field can get easily intimidated. This is true for basic graph rewriting and is even more pronounced for enriched forms, such as attributed graph rewriting [11,18,34].

To solve this, in this paper we give a condensed version that can be easily and concisely defined and explained. For basic graph rewriting, we rely on the double-pushout (DPO) approach [4,17], which is one of the most well-known approaches to graph transformation, although clearly not the only one. In the definition we do not use the notion of pushouts, although we will afterwards explain their role.

For attributed graphs, we chose to give a lightweight, but still expressive version, which captures the spirit of related approaches.

Apart from spelling out the definitions, we will also motivate why they have a specific form. Afterwards, we will give an application example and introduce tool support.

Note that in the context of this paper we use the terms *graph rewriting* and *graph transformation* interchangeably. We will avoid the term *graph grammar*,

since that emphasizes the use of graph transformation to generate a graph language, here the focus is just on the rewriting aspect.

2 A Formal Introduction to Graph Transformation

We start by defining graphs, where we choose to consider directed, edge-labelled graphs where parallel edges are allowed.

Other choices would be to use hypergraphs (where an edge can be connected to any number of nodes) or to add node labels. Both versions can be easily treated by our rewriting approach.[1]

Throughout the paper, we assume the existence of a fixed set Λ from which we take our labels.

Definition 1 (Graph). *A* graph G *is a tuple* $G = (V, E, s, t, \ell)$*, where*

- V *is a set of nodes,*
- E *is a set of edges,*
- $s\colon E \to V$ *is the source function,*
- $t\colon E \to V$ *is the target function* and
- $\ell\colon E \to \Lambda$ *is the labelling function.*

Given a graph G, we denote its components by $V_G, E_G, s_G, t_G, \ell_G$. Given an edge $e \in E_G$, the nodes $s_G(e), t_G(e)$ are called *incident* to e.

A central notion in graph rewriting is a graph morphism. Just as a function is a mapping from a set to another set, a graph morphism is a mapping from a graph to a graph. It maps nodes to nodes and edges to edges, while preserving the structure of a graph. This means that if an edge is mapped to an edge, there must be a mapping between the source and target nodes of the two edges. Furthermore, labels must be preserved.

Graph morphisms are needed to identify the match of a left-hand side of a rule in a (potentially larger) host graph. As we will see below, they are also required for other purposes, such as graph gluing and graph transformation rules.

Definition 2 (Graph morphism). *Let* G*,* H *be two graphs. A* graph morphism $\varphi\colon G \to H$ *is a pair of mappings* $\varphi_V\colon V_G \to V_H$*,* $\varphi_E\colon E_G \to E_H$ *such that for all* $e \in E_G$ *it holds that*

- $s_H(\varphi_E(e)) = \varphi_V(s_G(e))$,
- $t_H(\varphi_E(e)) = \varphi_V(t_G(e))$ and
- $\ell_H(\varphi_E(e)) = \ell_G(e)$.

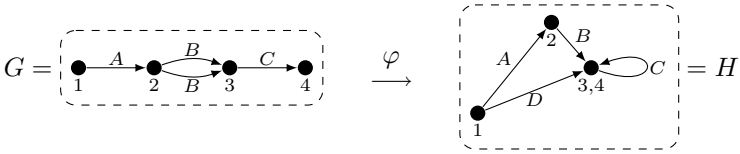
[1] Note that considerable part of graph transformation theory is concerned with making the results independent of the specific graph structure under consideration (see [28, 30]). This however depends on the use of category theory and we will not follow this path here.

A graph morphism φ is called injective (surjective) *if both mappings φ_V, φ_E are injective (surjective). Whenever φ_V and φ_E are bijective, φ is called an* isomorphism. *In this case we say that G_1, G_2 are isomorphic and write $G_1 \cong G_2$.*

Graph morphisms are composed by composing both component mappings. Composition of graph morphisms *is denoted by* $\circ$.

In the following we omit the subscripts in the functions φ_V, φ_E and simply write φ.

Example 1. Consider the following graphs G and H. Note that the numbers written at the nodes are not part of the graph: they are just there to indicate the morphism from G to H.



Here the edges of G are mapped with respect to their corresponding source and target node mappings. Note that the graph morphism φ is not surjective, since the D-labelled edge in H is not targeted. Furthermore, the morphism φ is not injective since the nodes 3 and 4 of the graph G are mapped to the same node in H and the two B-labelled edges in G are mapped to the same edge in H.

Now we come to another central concept that we here call *graph gluing*, but which is more conventionally called *pushout* in the literature. We will stick with the name graph gluing for now and will later explain the relation to categorical pushouts.

An intuitive explanation for the following construction is to think of two graphs G_1, G_2 with an overlap I. Now we glue G_1 and G_2 together over this common interface I, obtaining a new graph $G_1 +_I G_2$. This intuition is adequate in the case where the embeddings of I into the two graphs (called φ_1, φ_2 below) are injective, but not entirely when they are not. In this case one can observe some kind of merging effect that is illustrated in the examples below.

Graph gluing is described via factoring through an equivalence relation. We use the following notation: given a set X and an equivalence $\equiv$ on X, let $X/\equiv$ denote the set of all equivalence classes of $\equiv$. Furthermore $[x]_\equiv$ denotes the equivalence class of $x \in X$.

Definition 3 (Graph gluing). *Let I, G_1, G_2 be graphs with graph morphisms $\varphi_1 \colon I \to G_1$, $\varphi_2 \colon I \to G_2$, where I is called the* interface. *We assume that all node and edge sets are disjoint.*

Let $\equiv$ be the smallest equivalence relation on $V_{G_1} \cup E_{G_1} \cup V_{G_2} \cup E_{G_2}$ which satisfies $\varphi_1(x) \equiv \varphi_2(x)$ for all $x \in V_I \cup E_I$.

The gluing *of G_1, G_2 over I (written as $G = G_1 +_{\varphi_1,\varphi_2} G_2$, or $G = G_1 +_I G_2$ if the φ_i morphisms are clear from the context) is a graph G with:*

$$V_G = (V_{G_1} \cup V_{G_2})/\equiv \qquad E_G = (E_{G_1} \cup E_{G_2})/\equiv$$

$$s_G([e]_\equiv) = \begin{cases} [s_{G_1}(e)]_\equiv \text{ if } e \in E_{G_1} \\ [s_{G_2}(e)]_\equiv \text{ if } e \in E_{G_2} \end{cases} \qquad t_G([e]_\equiv) = \begin{cases} [t_{G_1}(e)]_\equiv \text{ if } e \in E_{G_1} \\ [t_{G_2}(e)]_\equiv \text{ if } e \in E_{G_2} \end{cases}$$

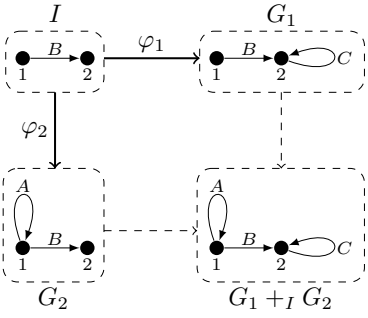
$$\ell_G([e]_\equiv) = \begin{cases} \ell_{G_1}(e) \text{ if } e \in E_{G_1} \\ \ell_{G_2}(e) \text{ if } e \in E_{G_2} \end{cases}$$

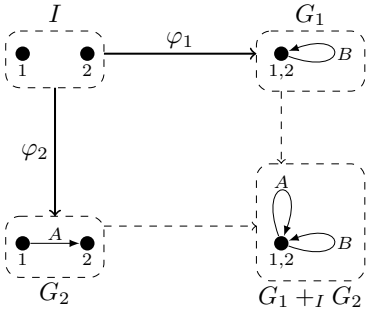
where $e \in E_{G_1} \cup E_{G_2}$.

Note that the gluing is well-defined, which is not immediately obvious since the mappings s_G, t_G, ℓ_G are defined on representatives of equivalence classes. The underlying reason for this is that φ_1, φ_2 are morphisms.

Example 2. We now explain this gluing construction via some examples.

– Let the two graph morphisms $\varphi_1 \colon I \to G_1$ and $\varphi_2 \colon I \to G_2$ to the right be given, where both φ_1 and φ_2 are injective. Since the interface I is present in both graphs G_1 and G_2, we can glue the two graphs together to construct a graph $G_1 +_I G_2$ depicted on the bottom right of the square.





– Now let the graph morphisms $\varphi_1 \colon I \to G_1$ and $\varphi_2 \colon I \to G_2$ to the left be given, where only φ_2 is injective. In the graph G_1, the interface nodes of I are merged via φ_1. The gluing graph $G_1 +_I G_2$ is constructed by merging all nodes in G_1, G_2, resulting in an A-labelled loop, together with the original B-labelled loop. This graph is depicted at the bottom right of the square.

We are now ready to define graph transformation rules, also called productions. Such a rule consists of a left-hand side graph L and a right-hand side graph R. However, as indicated in the introduction, this is not enough. The problem is that, if we simply removed (a match of) L from a host graph, we would typically have dangling edges, i.e., edges where either the source or the target node (or both) have been deleted. Furthermore, there would be no way to specify how the right-hand side R should be attached to the remaining graph.

Hence, there is also an interface graph I related to L and R via graph morphisms, which specify what is preserved by a rule.

Definition 4 (Graph transformation rule). *A* (graph transformation) rule r *consists of three graphs* L, I, R *and two graph morphisms* $L \overset{\varphi_L}{\leftarrow} I \overset{\varphi_R}{\rightarrow} R$.

Given a rule r, all nodes and edges in L that are not in the image of φ_L are called *obsolete*. Similarly, all nodes and edges in R that are not in the image of φ_R are called *fresh*.

After finding an occurrence of a left-hand side L in a host graph (a so-called match), the effect of applying a rule is to remove all obsolete elements and add all fresh elements. As indicated above, the elements of I are preserved, providing us with well-defined attachment points for R.

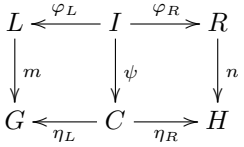
While this explanation is valid for injective matches and rule morphisms, it does not tell the full story in case of non-injective morphisms. Here rules might split or merge graph elements. Using the graph gluing defined earlier, it is easy to give a formal semantics of rewriting.

The intuition is as follows: given a rule as in Definition 4 and a graph G, we ask whether G can be seen as a gluing of L and an (unknown) context C over interface I, i.e., whether there exists C such that $G \cong L +_I C$. If this is the case, G can be transformed into $H \cong R +_I C$.

Definition 5 (Graph transformation). *Let* $r = (L \overset{\varphi_L}{\leftarrow} I \overset{\varphi_R}{\rightarrow} R)$ *be a rule. We say that a graph* G *is transformed via* r *into a graph* H *(symbolically:* $G \overset{r}{\Rightarrow} H$*) if there is a graph* C *(the so-called* context*) and a graph morphism* $\psi\colon I \to C$ *such that:*

$$G \cong L +_{\varphi_L,\psi} C \qquad H \cong R +_{\varphi_R,\psi} C$$

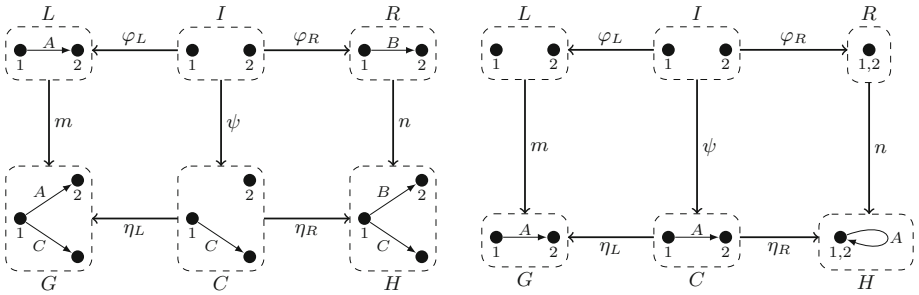
This situation can be depicted by the diagram to the right (also called double-pushout diagram).
The morphism m *is called the* match*,* n *the* co-match.



Depending on the morphisms φ_L and φ_R one can obtain different effects: whenever both φ_L and φ_R are injective, we obtain standard replacement. Whenever φ_L is non-injective we specify splitting, whereas a non-injective φ_R results in merging.

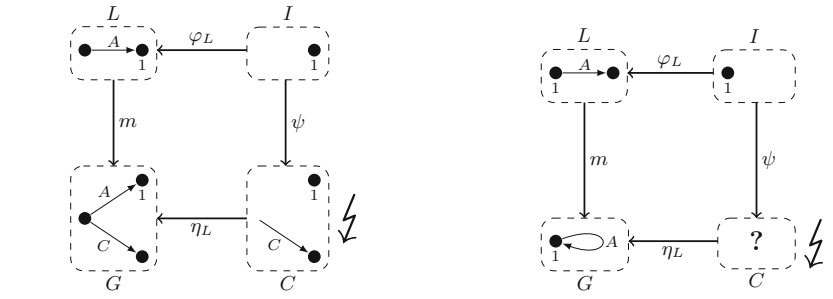
We now consider some examples. First, we illustrate the straightforward case where indeed the obsolete items are removed and the fresh ones are added, see Fig. 1a. Somewhat more elaborate is the case when the right leg φ_R of a rule is non-injective, which causes the merging of nodes, see Fig. 1b.

Different from string or term rewriting, in graph rewriting it may happen that we find a match of the left-hand side, but the rule is not applicable, because no context as required by Definition 5 exists. There are basically two reasons for



(a) Application of a GTS rule (b) Non-injective right leg φ_R

Fig. 1. GTS rule examples



(a) Dangling edge condition example (b) Identification condition example

Fig. 2. Gluing condition examples

this: either the rule removes a node, without removing all edges connected to that node (dangling edge condition, see Fig. 2a), or the match identifies two graph elements which are not preserved (identification condition) (see Fig. 2b).

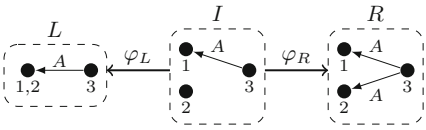
The following proposition [10] states under which circumstances the context exists.

Proposition 1 (Existence of a context, gluing condition). *Let $L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R$ be a graph transformation rule and let $m\colon L \to G$ be a match. Then a context C and a morphism $\psi\colon I \to C$ such that $G \cong L +_{\varphi_L,\psi} C$ exist if and only if the following holds:*

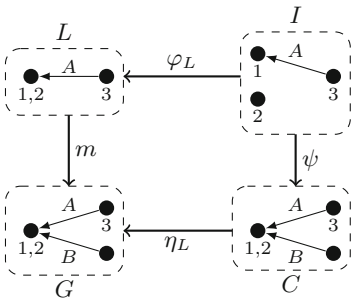
- *Every node $v \in V_L$, whose image $m(v)$ is incident to an edge $e \in E_G$ which is not in the image of m, is not obsolete (i.e. in the image of φ_L).*
- *Whenever two elements $x, y \in V_L \cup E_L$ with $x \neq y$ satisfy $m(x) = m(y)$, then neither of them is obsolete.*

However, even if the context exists, there might be cases where it is non-unique. This happens in cases where φ_L, the left leg of a rule, is non-injective. In this case one can for instance split nodes (see the rule in Fig. 3a) and the

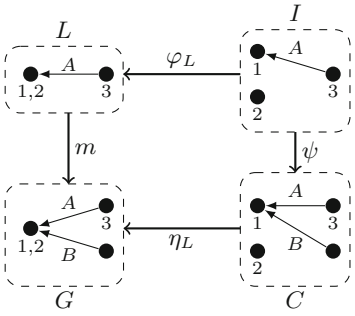
question is what happens to the incident edges. By spelling out the definition above, one determines that this must result in non-determinism. Either, we do not split (Fig. 3b) or we split and each edge can non-deterministically "choose" to stay either with the first or the second node (Fig. 3c and d). Each resulting combination is a valid context and this means that a rule application may be non-deterministic and generate several (non-isomorphic) graphs. In many papers such complications are avoided by requiring the injectivity of φ_L.



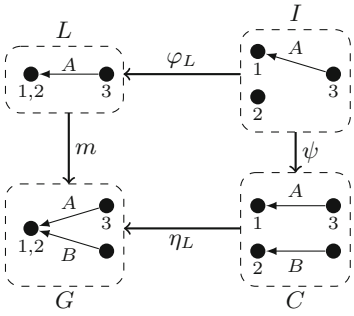
(a) Rule with non-injective left leg φ_L



(b) Valid context (i)



(c) Valid context (ii)



(d) Valid context (iii)

Fig. 3. Non-injective left leg rule with three valid contexts

Note that the specifics such as the dangling edge condition are typical to the double-pushout (or DPO) approach that we are following here. In other approaches (such as SPO, which we only discuss briefly in Sect. 7.2), whenever a node is deleted, all its incident edges are deleted as well (deletion in unknown contexts).

Finally, we can introduce the notion of a graph transformation system that is frequently used. Here we fix a start graph and a set of rules. This enables us to ask questions such as: Which graphs are reachable from the start graph via the given rules?

Definition 6 (Graph transformation system). *A* graph transformation system *is a tuple* $\mathcal{G} = (G_0, \mathcal{R})$ *where*

- G_0 *is an arbitrary graph, the so-called* initial graph *or* start graph, and
- $\mathcal{R}$ *is a set of* graph transformation rules.

3 Attributed Graph Transformation

For many applications one requires more than graphs labelled over a finite alphabet that we considered up to now. For instance, in Sect. 4 we will consider leader election on a ring where edges, representing processes, are labelled with natural numbers as Ids. Hence graphs should be attributed with elements of given data types (e.g. integer, string, boolean) and it should be possible to perform computations (e.g. add two integers) and define guards that restrict the applicability of rules (e.g. apply the rule only if a certain attribute is above some threshold).

In order to achieve this aim we now introduce attributed graph transformation. Choosing data types (also called sorts), carrier sets and operations amounts to defining a signature and a corresponding algebra [16,46] and we will start by introducing these concepts.

Definition 7 (Signature, Algebra). *A* signature Σ *is a pair* $(\mathcal{S}, \mathcal{F})$ *where* $\mathcal{S}$ *is a set of* sorts *and* $\mathcal{F}$ *is a set of function symbols equipped with a mapping* $\sigma\colon \mathcal{F} \to \mathcal{S}^* \times \mathcal{S}$. *Sorts are also called* types. *We require that* $\mathcal{S}$ *contains the sort bool.*

A Σ-algebra $\mathcal{A}$ *consists of carrier sets* $(\mathcal{A}_s)_{s\in\mathcal{S}}$ *for each sort and a function* $f^{\mathcal{A}} : \mathcal{A}_{s_1} \times \cdots \times \mathcal{A}_{s_n} \to \mathcal{A}_s$ *for every function symbol* f *with* $\sigma(f) = (s_1 \dots s_n, s)$.

By $T(\Sigma, X)$ *we denote the* Σ-term algebra, *where* X *is a set of variables, each equipped with a fixed sort. That is, the carrier sets of the term algebra consist of all terms of the corresponding sort.*

For an algebra $\mathcal{A}$ *we denote by* $\mathcal{A}_{\mathcal{S}}$ *the set* $\mathcal{A}_{\mathcal{S}} = \biguplus_{s\in\mathcal{S}} \mathcal{A}_s$, *i.e., the union of all carrier sets (under the implicit assumption that they are all disjoint).*

Example 3. As a typical example for an algebra assume that we have two sorts $\mathcal{S} = \{int, bool\}$ and function symbols *add*, *mult*, *eq* with $\sigma(add) = \sigma(mult) = (int\, int, int)$ and $\sigma(eq) = (int\, int, bool)$ (representing addition, multiplication and the equality predicate).

The carrier sets in an algebra $\mathcal{A}$ could be $\mathcal{A}_{int} = \mathbb{Z}$, $\mathcal{A}_{bool} = \{true, false\}$ and functions would be interpreted in the usual way, e.g. $add^{\mathcal{A}}(z_1, z_2) = z_1 + z_2$ and $eq^{\mathcal{A}}(z_1, z_2) = true$ whenever $z_1 = z_2$ and *false* otherwise.

On the other hand, in the term algebra $T(\Sigma, X)$ the carrier sets consist of terms, for instance $T(\Sigma, X)_{int}$ contains $add(mult(x, y), y)$ and $T(\Sigma, X)_{bool}$ contains $eq(add(x, x), y)$, where $x, y \in X$ are variables of sort *int*.

Algebras come equipped with their notion of morphism, so-called algebra homomorphisms. These are mappings between the carrier sets that are compatible with the operations.

Definition 8 (Algebra homomorphism). *Let $\mathcal{A}, \mathcal{B}$ be two Σ-algebras. An* algebra homomorphism *$h : \mathcal{A} \to \mathcal{B}$ is a family of maps $(h_s : \mathcal{A}_s \to \mathcal{B}_s)_{s \in \mathcal{S}}$ such that for each $f \in \mathcal{F}$ with $\sigma(f) = (s_1 \dots s_n, s)$ we have*

$$h_s(f^{\mathcal{A}}(a_1, \dots, a_n)) = f^{\mathcal{B}}(h_{s_1}(a_1), \dots, h_{s_n}(a_n)).$$

The next step is straightforward: add attributes, i.e., elements of a carrier set, to (the nodes or edges of) a graph. In the following we add attributes only to edges, however nothing prevents us from adding attributes also to nodes. In order to have a clean separation, we require that the edge label determines the sort of the corresponding attribute.

Definition 9 (Attributed graph). *Let $\mathcal{A}$ be a Σ-algebra with $\mathcal{A}_{bool} = \{true, false\}$. Let type$\colon \Lambda \to \mathcal{S}$ be a function that assigns a sort to every edge label. An* attributed graph over $\mathcal{A}$ *(G, att) consists of a graph $G = (V, E, s, t, \ell)$, together with a function att$\colon E_G \to \mathcal{A}_{\mathcal{S}}$ such that $att(e) \in \mathcal{A}_{type(\ell(e))}$.*

We first define the notion of attributed graph transformation rule, where left-hand side and right-hand side are attributed over the term algebra. Furthermore there is a guard condition of sort *bool*. We require that in the left-hand side terms are always single unique variables, which can then be used in terms in the right-hand side and in the guard.

Definition 10 (Attributed graph transformation rule). *Let Σ be a signature and let X be a set of variables. An* attributed rule *is a graph transformation rule $L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R$ with two functions $att_L\colon E_L \to T(\Sigma, X)_{\mathcal{S}}$, $att_R\colon E_R \to T(\Sigma, X)_{\mathcal{S}}$ and a guard $g \in T(\Sigma, X)_{bool}$. These attribution functions must respect sorts, i.e., for every $e \in E_L$ it holds that $att_L(e) \in T(\Sigma, X)_{type(\ell_L(e))}$ and analogusly for $e \in E_R$.*

We require that each term $att_L(e)$ for $e \in E_L$ is a single variable and all these variables occurring in the left-hand side are pairwise different. Furthermore, each variable in $att_R(e)$ for $e \in E_R$ and each variable in g occurs in the left-hand side.

Now we are ready to define attributed graph transformation: while the rule graphs L and R are attributed with elements from the term algebra, the graphs to be rewritten are attributed with elements from a carrier set that represents a primitive data type (such as integers or booleans).

Then a match determines the evaluation of the variables in the left-hand side, giving us a corresponding algebra homomorphism. This homomorphism is then used to evaluate the terms in the right hand sides and to generate the corresponding values. All other edges keep their attribute values.

Definition 11 (Graph transformation with attributed rules). *Given an attributed rule $L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R$ with functions att_L, att_R and guard g, it can be applied to a graph (G, att_G) attributed over $\mathcal{A}$ as follows: G is transformed to H as described in Definition 5. The match $m\colon L \to G$ induces an algebra homomorphism $h_m\colon T(\Sigma, X) \to \mathcal{A}$ by defining $h_m(x) = att_G(m(e))$ if $e \in E_L$ and $att_L(e) = x$. For each variable y not occurring in L the value $h_m(y)$ is arbitrary.*

The rule can be applied whenever $h_m(g) = true$*. In this case we define*

$$att_H(e') = \begin{cases} h_m(att_R(e)) \text{ if } e' = n(e), e \in E_R \\ att_G(\eta_L(e)) \text{ otherwise, if } e' = \eta_R(e), e \in E_C \end{cases}$$

where $e' \in E_H$*. Whenever* att_H *is not well-defined, the rule can not be applied.*[2]

Note that the algebra homomorphism h_m above is well-defined due to the requirement that each occurrence of a variable in the left-hand side is unique.

We start with a straightforward case where we apply an attributed graph transformation rule, see Fig. 4. The given rule shifts a B-labelled loop (which has an attribute y) over an A-labelled edge with corresponding attribute x. After the rule application the edge is attributed with the sum $add(x, y)$ and the loop inherits the former attribute x. (Note that in order to have a more compact notation we slightly abuse notation and write $x + y$ instead of $add(x, y)$.)

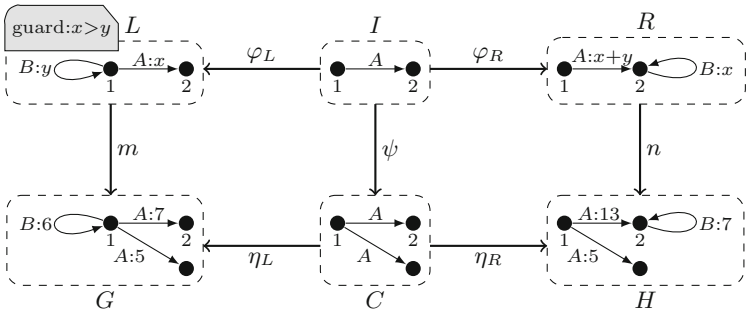


Fig. 4. Attributed graph transformation rule application example

Note that in the definition above, we have to require that the attribution function att_H of H is well-defined, here is an example that illustrates why: imagine a rule with two (equally labelled) edges in the left-hand side, which have attributes x, y. The edges are preserved and in the right-hand side they are attributed with the sum $add(x, y)$ and the product $mult(x, y)$ (see Fig. 5). Now, since we allow non-injective matches, such a rule can be applied to a single edge with attribute value 1 in the host graph.

The (preserved) edge has two different preimages under the co-match n. The first preimage would require to set the value $att_H(e)$ to $add^{\mathcal{A}}(1, 1) = 2$, the second to $mult^{\mathcal{A}}(1, 1) = 1$.

Here, the straightforward solution is to say that the rule is not applicable, since it would create an inconsistent situation. Such issues can be avoided by requiring that all morphisms (rule morphisms, match, etc.) are injective.

[2] The morphism n need not be injective, hence an edge e might have several preimages under n. In this case, it is possible that the new attribute of an edge cannot be uniquely determined.

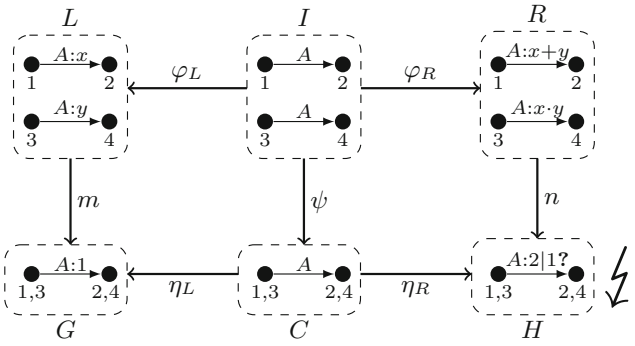


Fig. 5. Example for a non-applicable rule

4 Example: Leader Election

We now demonstrate the modelling power of attributed graph transformation systems, by modelling a variation of the leader election protocol. The protocol (according to Chang and Roberts [3]) works as follows: there is a set of processes arranged in a ring, i.e., every process has a unique predecessor and a unique successor. Furthermore, each process has a unique Id and there exists a total order on the Ids (which is easily achieved by assuming that Ids are natural numbers).

The leader will be the process with the smallest Id, however no process knows what is the smallest Id at the start of the protocol. Hence every process generates a message with its own Id and sends it to its successor. A received message with content *MId* is treated as follows by a process with Id *PId*:

- if $MId < PId$, then the message is forwarded to the successor
- if $MId = PId$, then the process declares itself the leader
- If $MId > PId$, then the message is not passed on (or alternatively discarded).

Whenever the Ids are unique, it can be shown that the protocol terminates with the election of a unique leader.

In this example we additionally assume that the topology of the ring changes. We extend the protocol allowing processes to enter and to leave the ring. Processes entering the ring obtain a unique Id via a central counter, larger than all other Ids existing so far in the ring. This additional feature does not interfere with the election of the leader, since we elect the process with the minimal Id. As any process might be deleted, we can not resort to the simplistic (and non-distributed) solution of choosing the first process that is created and has thus the lowest Id.

We now model this protocol as a graph transformation system (see Fig. 6). The start graph S consists of a single node and a loop labelled `count` which represents the counter, the current counter value `0` is stored in the attribute (where all attributes are natural numbers). In a first step, modelled by rule

first proc, if the counter attribute `count` satisfies the rule's guard `i=0`, then the first process labelled `proc:i+1` is created and the counter is set to `count:i+1`. Remember, only the edges are equipped with labels and corresponding terms. As required in Definition 10 the left-hand side is attributed only by variables. Again, the numbers in the nodes denote the morphisms from the interface to the left-hand and right-hand side. Then, in subsequent steps (rule *add proc*) other processes are created (incrementing the current counter and using it as the Id of the process) and are inserted after an arbitrary existing process. Processes may leave the ring, provided at least one other process is present, see rule *del proc*.

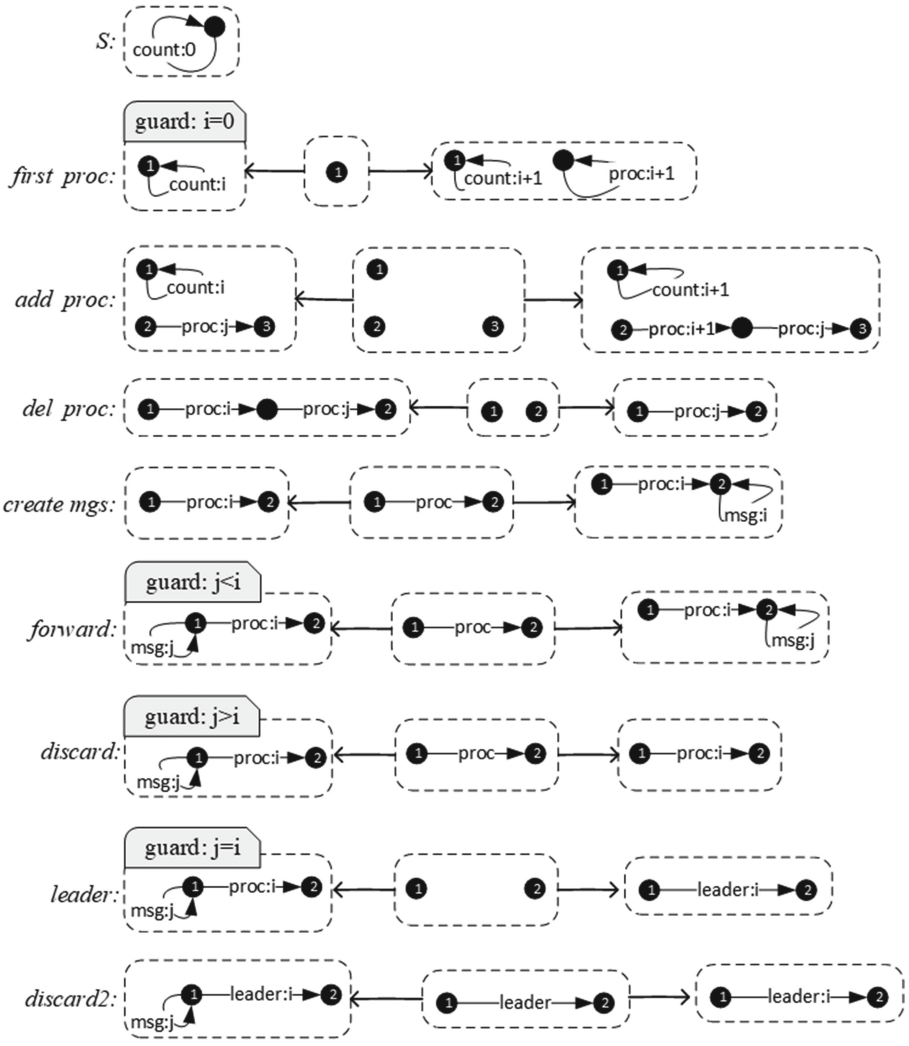


Fig. 6. Leader election in dynamic ring

Processes can create messages as described above, represented by `msg`-labelled loops (rule *create msg*). The attribute of the `msg`-loop is the Id of the sending process. The message is forwarded if its Id is less than the Id of the receiving process (rule *forward*) and if it is greater the message is discarded (rule *discard*). If a process receives a message with its own Id, it declares itself the leader (rule *leader*). Once the leader has been chosen, it cannot be deleted any longer since the rule *del proc* requires the label `proc`. Moreover, all subsequent messages arriving at the leader are discarded as well (rule *discard2*).

The application of rules may yield a graph G as given in Fig. 7. The match morphism m induces the algebra homomorphism $h_m \colon T(\Sigma, X) \to \mathcal{A}$ with $h_m(i) = 2$ and $h_m(j) = 2$. Obviously the guard is satisfied, so the edges with label `msg:2` and `prc:2` are deleted, yielding the graph C. The graph H is obtained by gluing an edge with label `leader:2` between the two nodes.

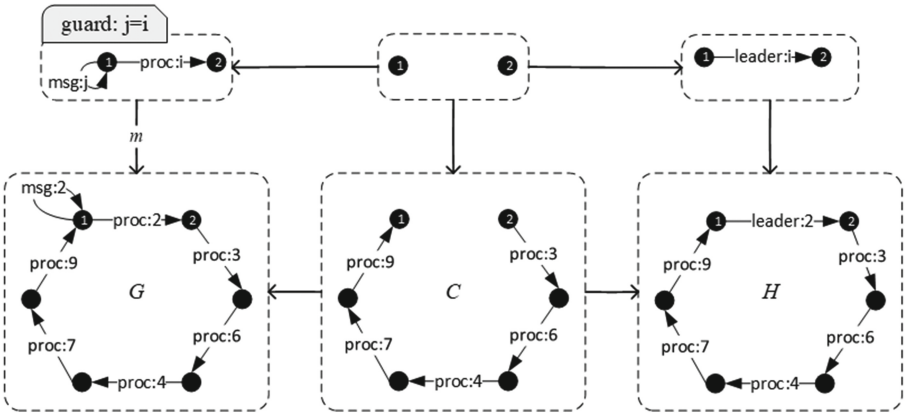


Fig. 7. Transformation step: declaring the leader process

The state space of this system is infinite, due to the capability of creating new processes with ever higher numbers, as well as unbounded numbers of messages. There are techniques for verifying infinite-state graph transformation systems (see for instance [26] where a similar system is analysed), but those are out of scope for this tutorial. Alternatively, if we restrict the number of processes to a fixed bound, then model checking becomes available as a technique, for instance as implemented in the tool GROOVE (see Sect. 5.1). Here are some example desirable properties, expressed in Computation Tree Logic (CTL) respectively Linear Temporal Logic (LTL):

1. Safety (CTL): AG ! twoLeaders
2. Reachability (CTL): AG EF hasLeader
3. Termination (LTL): G F hasLeader

Here, twoLeaders and hasLeader are propositions that are fulfilled by a given state graph if it contains, respectively, at least two leader-edges or at least one leader-edge. Those propositions can themselves be formulated as transformation rules that do not actually change the graph, but only check for applicability. Property 1 expresses the crucial safety property that there can never be two or more leaders elected; property 2 expresses that from every state, a future state is reachable in which a leader has been elected. Finally, property 3 expresses that all paths will eventually actually reach a point where a leader has been elected.

The protocol encoded in the rule system at hand satisfies properties 1 and 2, but not 3. Two reasons why termination fails to hold are that the process with the lowest number may forever be deleted *before* it receives back its own message and so is elected leader, or one process creates an infinite number of messages which are never forwarded.

The protocol satisfies the reachability property 2 even in the case of changes in topology. This can be reasoned out as follows: Since the added processes have higher Ids than already inserted processes, the message which the new process sends, will be discarded by the subsequent process. Another consequence of the changing topology could be that the process `proc:min1` with the minimal Id `min1` sends its message and is deleted before becoming leader. Then its message is forwarded along the ring as it always satisfies the condition of rule *forward*. But at some point the next minimal Id `min2` is sent by some other process and leads to the leader with Id `min2`. Then the message with `min1` will be eventually discarded with *discard2*.

5 Tools

Here, we merely hint at some of graph transformation tools that are available for many different purposes and describe two of them in more detail. We introduce AGG and GROOVE, since both of them can be considered to be general purpose graph transformation tools. Other graph transformation tools that are actively maintained are, ATOM[3] [8], VIATRA [6], FUJABA [33] and AUGUR [25].

5.1 GROOVE: Graphs for Object-Oriented Verification

The tool GROOVE[3] was originally created to support the use of graphs for modelling the design-time, compile-time, and run-time structure of object-oriented systems [39], but since then has grown to be a full-fledged general-purpose graph transformation tool. The emphasis lies on the efficient exploration of the state space, given a particular graph transformation system; see, for instance, [20]. While doing so, GROOVE recognises previously visited graphs modulo (graph) isomorphism, avoiding duplication during their exploration. GROOVE has a built-in model checker that can run temporal logic queries (LTL or CTL) over the resulting state space.

[3] http://groove.cs.utwente.nl/.

GROOVE has a very rich set of features to enable the powerful and flexible specification of transformation rules, including quantified rules [41] (so that a single rule may be applied in one shot to all subgraphs that satisfy a given property) and an extensive language to schedule rule applications (so that the default strategy of applying every rule to all reachable graphs can be modified).

GROOVE graphs do not conform precisely to the definition in this paper. Some important differences are:

- GROOVE graphs are typed; that is, all nodes have one of a set of types from a user-defined type graph (and so do all edges, but an edge type essentially corresponds to its label).
- More importantly, in GROOVE attributes are associated with nodes rather than edges; moreover, they are always named. Thus, rather than an edge `proc:1` between two untyped process nodes, one would have an unnumbered edge `next` (say) between two `Proc`-type nodes, in combination with a named attribute `nr = 1` on its target node.

However, the graphs of this paper can be easily mimicked. A rule system for leader election that corresponds to Sect. 4 is provided together with this paper.[4]

The results reported in the previous section on the safety, reachability and termination properties 1–3 can easily be checked on this rule system by disabling the rule *add proc* and starting with a graph that already has a given number of processes (so that the state space is finite), and then invoking the LTL or CTL model checker with the formulas given above. As stated before, the outcome is that properties 1 and 2 are satisfied, whereas 3 is violated.

5.2 AGG: The Attributed Graph Grammar System

The Attributed Graph Grammar System (AGG) [45] is a development environment for attributed graph transformation systems and aims at specifying and rapidly prototyping applications with complex, graph structured data. AGG[5] supports the editing of graphs and rules that can be attributed by Java objects and types. Basic data types as well as object classes already available in Java class libraries may be used. The graph rules may be attributed by Java expressions which are evaluated during rule applications. Additionally, rules may have attribute conditions that are boolean Java expressions. AGG provides simulation and analysis techniques, namely critical pair analysis and consistency checking. The application of rules can be manipulated using control structures such as negative application conditions to express requirements for non-existence of substructures. Further control over the rules is given by rule layers that fix the order in which rules are applied. The interpretation process applies rules of lower layers first, which means applying the rules of a layer as long as possible before applying those of the next layer. These rule layers allow the specification of a simple control flow.

[4] http://groove.cs.utwente.nl/wp-content/uploads/leader-electiongps.zip.
[5] http://www.user.tu-berlin.de/o.runge/agg/.

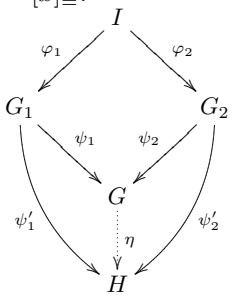
6 Some Remarks on the Categorical Background

In this section, we explain the name *double-pushout approach*. It gives some background information that is useful for understanding papers on the topic, but is not required for the formal definition of graph transformation given earlier.

Graph gluing as in Definition 3 can alternatively be characterized via the categorical notion of pushout. Category theory relies on so-called universal properties where, given some objects, one defines another object that is in some relation to the given object and is – in some sense – the most general object which is in this relation. The prototypical example is the supremum or join, where – given two elements x, y of a partially ordered set $(X, \leq)$ – we ask for a third element z with $x \leq z$, $y \leq z$ and such that z is the *smallest* element which satisfies both inequalities. There is at most one such z, namely $z = x \vee y$, the join of x, y.

In the case of graphs, the order relation $\leq$ is replaced by graph morphisms.

Proposition 2. *Let I, G_1, G_2 be graphs with graph morphisms $\varphi_1 \colon I \to G_1$, $\varphi_2 \colon I \to G_2$ as in Definition 3. Assume further that $G = G_1 +_{\varphi_1,\varphi_2} G_2$ is the gluing and that $\psi_i \colon G_1 \to G$, $i = 1, 2$, are graph morphisms that map each element $x \in V_{G_i} \cup E_{G_i}$ to its equivalence class $\psi_i(x) = [x]_{\equiv}$.*
The gluing diagram consisting of the morphisms $\varphi_1, \varphi_2, \psi_1, \psi_2$ commutes, i.e., $\psi_1 \circ \varphi_1 = \psi_2 \circ \varphi_2$ and it has the following universal property*: for any two morphisms $\psi_1' \colon G_1 \to H$, $\psi_2' \colon G_2 \to H$ satisfying $\psi_1' \circ \varphi_1 = \psi_2' \circ \varphi_2$, there exists a unique morphism $\eta \colon G_1 +_I G_2 \to H$ such that $\eta \circ \psi_1 = \psi_1'$ and $\eta \circ \psi_2 = \psi_2'$.*
Squares which commute and satisfy the universal property spelled out above are called pushouts*. The graph G is unique up to isomorphism.*



Intuitively, the pushout characterization says that G should be a graph where the "common" parts of G_1, G_2 must be merged (since the square commutes), but it should be obtained in the most general way by merging only what is absolutely necessary and adding nothing superfluous. This corresponds to saying that for any other merge H, G is more general and H can be obtained from G by further merging or addition of graph elements (expressed by a morphism from G to H).

7 Literature Overview

7.1 Introductory Papers

This paper is by no means the first introductory paper to graph transformation. It was our aim to write a paper that fills a niche and gives precise formal definitions, but does not rely on category theory. At the same time, we wanted to treat general rules and not restrict to injective matches or rule spans, which is often done in tutorial papers.

Since not all introductory papers are well-known, it is worth to make a meta-survey and to provide a list.

Of the original papers, the survey paper by Ehrig [10] is in any case worth a read, however the notation has evolved since the seventies.

A standard reference is of course the "Handbook of Graph Grammars and Computing by Graph Transformation", which appeared in three volumes (foundations [42] – applications, languages and tools [12] – concurrency, parallelism and distribution [15]). Strongly related to our exposition is the chapter on DPO rewriting by Corradini et al. [4], which is based on categorical definitions.

The well-known book by Ehrig et al. [11] revisits the theory of graph rewriting from the point of view of adhesive categories, a general categorical framework for abstract transformations. In an introductory section it defines the construction of pushouts via factorization, equivalent to our notion of graph gluing.

Nicely written overview papers that however do not give formal definitions are by Heckel [23] and by Andries et al. [1]. The paper by Giese et al. [21] is aimed towards software engineers and gives a detailed railcab example.

The habilitation thesis by Plump [36] and the introductory paper by Kreowski et al. [27] give very clear formal, but non-categorical, introductions. Both make injectivity requirements, either on the rule spans or on the match.

The paper by Löwe and Müller [31] makes a non-categorical, but slightly non-standard, introduction to graph transformation, while the introduction by Schürr and Westfechtel [44] is very detailed and treats set-theoretical, categorical and logical approaches. Both articles are however written in German.

7.2 Further Issues

Deletion in unknown contexts: In the DPO approach that was treated in this note, it is forbidden to remove a node that is still attached to an edge, which is not deleted. In this case, the rule is not applicable. In the single-pushout (or SPO) approach [30] however, the deletion of a node to which an undeleted edge is attached, is possible. In this case all incident edges are deleted as well. In contrast to DPO, SPO is based on partial graph morphisms.

Attributed graph rewriting: Our way of defining attributed graph rewriting was inspired by [26,37]. We provide some remarks on alternative approaches to attributed graph transformation: in an ideal world one would like to extend all the notions that we introduced previously (graph morphisms, gluing, rules, etc.) to this new setting. This would mean to extend graph morphisms to attributed graph morphisms by requiring algebra homomorphisms on the attributes.

This has been done [11,18,34], but unfortunately there are some complications. The first problem is that, as explained above, we want to work with two different algebras: the term algebra and an algebra representing primitive data types. This means that in the double-pushout diagrams, we would need algebra homomorphisms between different algebras on the vertical axis and identity algebra homomorphisms on the horizontal axis.

But this causes another problem: nodes or edges that are preserved by a rule, i.e., items that are in the interface usually should *not* keep their attribute value. Otherwise it would be impossible to specify attribute changes. (Note that it is possible to delete and recreate an edge, but not a node, since it is usually connected to unknown edges and the dangling condition would forbid this.) But this is contrary to the idea of having identity homomorphisms horizontally.

Hence, as announced above, we here opted for a lightweight approach where we do not define a new notion of attributed graph morphism, but only add algebra homomorphisms as required (for the match and co-match). Other options, which we do not pursue here, is to add the carrier sets to the graphs and add pointers from edges to the attribute values [18]. However, this formally turns graphs into infinite objects.

As a side remark, we would also like to mention that graphs themselves are two-sorted algebras with sorts node and edge and two function symbols (source and target). This view has been exploited in order to generalize the structures that can be transformed [30].

Application conditions: As we saw in the section on attributed graph transformation (Sect. 3) and in the example (Sect. 4), it is often useful to specify guards that restrict the applicability of rules. So far our guards talked about attributes, but it is also very useful to consider guards that refer to the structural properties of a graph, so-called application conditions [11,22].

A special case are negative applications conditions that inhibit the application of a rule whenever a certain structure is present in the vicinity of the left-hand side. This can for instance be used to specify rules for computing the transitive closure of a graph: whenever there exists an edge from node s to v and from v to t, add a direct edge from s to t, but only if such an edge is not already present. In order to gain expressiveness, application conditions can be nested and it has been shown that such conditions are equal in expressiveness to first-order logic [40].

8 Conclusion

Naturally, there are many topics related to graph transformation that we did not treat in this short tutorial. For instance, there exists a substantial amount of work on theory, generalizing graph transformation by means of category theory [11,28]. Furthermore, confluence or Church-Rosser theorems, in connection with critical pair analysis have been extensively studied. Other verification and analysis techniques have been studied as well (termination analysis, reachability analysis, model checking, etc.). Work on graph transformation is also closely connected to work on specification languages on graphs, such as nested application conditions [22] and monadic second-order logic [5].

Acknowledgements. We would like to thank all the participants of the North German GraTra Day in February 2017 in Hamburg for the discussion about this paper. Especially, we would like to acknowledge Berthold Hoffmann, Leen Lambers and Hans-Jörg Kreowski who contributed by commenting on our paper and giving valuable suggestions and hints.

References

1. Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.-J., Kuske, S., Plump, D., Schürr, A., Taentzer, G.: Graph transformation for specification and programming. Sci. Comput. Program. **34**(1), 1–54 (1999)
2. Bezem, M., Klop, J.W., de Vrijer, R. (eds.): Term Rewriting Systems. Cambridge University Press, Cambridge (2003)
3. Chang, E.J.H., Roberts, R.: An improved algorithm for decentralized extrema-finding in circular configurations of processes. Commun. ACM **22**(5), 281–283 (1979)
4. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic approaches to graph transformation–part I: basic concepts and double pushout approach. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations. World Scientific (1997). Chapter 3
5. Courcelle, B., Engelfriet, J.: Graph Structure and Monadic Second-Order Logic, A Language-Theoretic Approach. Cambridge University Press, New York (2012)
6. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA - visual automated transformations for formal verification and validation of UML models. In: 17th IEEE International Conference on Automated Software Engineering, pp. 267–270. IEEE Computer Society (2002)
7. Danos, V., Feret, J., Fontana, W., Harmer, R., Hayman, J., Krivine, J., Thompson-Walsh, C.D., Winskel, G.: Graphs, rewriting and pathway reconstruction for rule-based models. In: Proceedings of the FSTTCS 2012. LIPIcs, vol. 18. Schloss Dagstuhl - Leibniz Center for Informatics (2012)
8. Lara, J., Vangheluwe, H.: AToM^3: a tool for multi-formalism and meta-modelling. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45923-5_12
9. Dershowitz, N., Jouannaud, J.-P.: Rewrite systems. In: van Leeuwen, J. (eds.) Formal Models and Semantics. Handbook of Theoretical Computer Science, vol. B, pp. 243–320. Elsevier (1990). Chapter 6
10. Ehrig, H.: Introduction to the algebraic theory of graph grammars (a survey). In: Claus, V., Ehrig, H., Rozenberg, G. (eds.) Graph Grammars 1978. LNCS, vol. 73, pp. 1–69. Springer, Heidelberg (1979). https://doi.org/10.1007/BFb0025714
11. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
12. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation. Applications Languages and Tools. World Scientific, Singapore (1999)
13. Ehrig, H., Ermel, C., Golas, U., Hermann, F.: Graph and Model Transformation - General Framework and Applications. Monographs in Theoretical Computer Science. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47980-3
14. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting with borrowed contexts. Math. Struct. Comput. Sci. **16**(6), 1133–1163 (2006)
15. Ehrig, H., Kreowski, H.-J., Montanari, U., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation. Concurrency, Parallellism, and Distribution. World Scientific, Singapore (1999)

16. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1, Equations and Initial Semantics. Monographs in Theoretical Computer Science. Springer, Heidelberg (1985). https://doi.org/10.1007/978-3-642-69962-7
17. Ehrig, H., Pfender, M., Schneider, H.: Graph grammars: an algebraic approach. In: Proceedings of the 14th IEEE Symposium on Switching and Automata Theory, pp. 167–180 (1973)
18. Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 161–177. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30203-2_13
19. Fokkink, W.: Introduction to Process Algebra. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-662-04293-9
20. Ghamarian, A.H., de Mol, M.J., Rensink, A., Zambon, E., Zimakova, M.V.: Modelling and analysis using groove. Int. J. Soft. Tools Technol. Transf. **14**(1), 15–40 (2012)
21. Giese, H., Lambers, L., Becker, B., Hildebrandt, S., Neumann, S., Vogel, T., Wätzoldt, S.: Graph transformations for MDE, adaptation, and models at runtime. In: Bernardo, M., Cortellessa, V., Pierantonio, A. (eds.) SFM 2012. LNCS, vol. 7320, pp. 137–191. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30982-3_5
22. Habel, A., Pennemann, K.-H.: Nested constraints and application conditions for high-level structures. In: Kreowski, H.-J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) Formal Methods in Software and Systems Modeling. LNCS, vol. 3393, pp. 293–308. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31847-7_17
23. Heckel, R.: Graph transformation in a nutshell. In: Bezivin, J., Heckel, R. (eds.) Language Engineering for Model-Driven Software Development, number 04101 in Dagstuhl Seminar Proceedings (2005)
24. Høgh Jensen, O., Milner, R.: Bigraphs and mobile processes (revised). Technical report UCAM-CL-TR-580, University of Cambridge (2004)
25. König, B., Kozioura, V.: Augur - a tool for the analysis of graph transformation systems. Bull. EATCS **87**, 126–137 (2005)
26. König, B., Kozioura, V.: Towards the verification of attributed graph transformation systems. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 305–320. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87405-8_21
27. Kreowski, H.-J., Klempien-Hinrichs, R., Kuske, S.: Some essentials of graph transformation. In: Ésik, Z., Martin-Vide, C., Mitrana, V. (eds.) Recent Advances in Formal Languages and Applications, pp. 229–254. Springer, Heidelberg (2006). https://doi.org/10.1007/978-3-540-33461-3_9
28. Lack, S., Sobociński, P.: Adhesive and quasiadhesive categories. RAIRO - Theor. Inf. Appl. **39**(3), 511–545 (2005)
29. Leifer, J.J., Milner, R.: Deriving bisimulation congruences for reactive systems. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 243–258. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44618-4_19
30. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theor. Comput. Sci. **109**, 181–224 (1993)
31. Löwe, M., Müller, J.: Algebraische Graphersetzung: mathematische Modellierung und Konfluenz. Forschungsbericht des Fachbereichs Informatik, TU Berlin, Berlin (1993)

32. Milner, R. (ed.): A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10235-3
33. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: Ghezzi, C., Jazayeri, M., Wolf, A.L. (eds.) Proceedings of the 22nd International Conference on on Software Engineering, pp. 742–745. ACM (2000)
34. Orejas, F.: Symbolic graphs for attributed graph constraints. J. Symbolic Comput. **46**(3), 294–315 (2011)
35. Padberg, J., Kahloul, L.: Overview of reconfigurable Petri nets. In: Heckel, R., Taentzer, G. (eds.) Ehrig Festschrift. LNCS, vol. 10800, pp. 201–222. Springer, Cham (2018)
36. Plump, D.: Computing by Graph Rewriting. Habilitation thesis, Universität Bremen (1999)
37. Plump, D., Steinert, S.: Towards graph programs for graph algorithms. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 128–143. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30203-2_11
38. Reisig, W.: Petri Nets: An Introduction. EATCS Monographs on Theoretical Computer Science. Springer, Heidelberg (1985). https://doi.org/10.1007/978-3-642-69968-9
39. Rensink, A.: The GROOVE simulator: a tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25959-6_40
40. Rensink, A.: Representing first-order logic using graphs. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 319–335. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30203-2_23
41. Rensink, A., Kuperus, J.-H.: Repotting the geraniums: on nested graph transformation rules. In: Boronat, A., Heckel, R. (eds.) Graph Transformation and Visual Modelling Techniques (GT-VMT). Electronic Communications of the EASST, vol. 18 (2009)
42. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations. World Scientific, Singapore (1997)
43. Sangiorgi, D., Walker, D.: The π-calculus-A Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)
44. Schürr, A., Westfechtel, B.: Graph grammars and graph rewriting systems (in German). Technical report AIB 92–15, RWTH Aachen (1992)
45. Taentzer, G.: AGG: a tool environment for algebraic graph transformation. In: Nagl, M., Schürr, A., Münch, M. (eds.) AGTIVE 1999. LNCS, vol. 1779, pp. 481–488. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45104-8_41
46. Wirsing, M.: Algebraic specification. In: van Leeuwen, J. (ed.) Formal Models and Semantics. Handbook of Theoretical Computer Science, vol. B, pp. 675–788. Elsevier (1990). Chapter 13