Graph Transformation with Variables

Berthold Hoffmann

Technologiezentrum Informatik, Universität Bremen
Postfach 330 440, D-28334 Bremen, Germany
hof@informatik.uni-bremen.de

Abstract. Variables make rule-based systems more abstract and expressive, as witnessed by term rewriting systems and two-level grammars. In this paper we show that variables can be used to define advanced ways of graph transformation as well. Taking the gluing approach to graph transformation [7, 3] as a basis, we consider extensions of rules with attribute variables, clone variables, and graph variables, respectively. In each case, the variables in a rule are instantiated in order to obtain a set of rule instances that in turn defines the transformation relation. By combining different kinds of variables, we define very expressive rules, and reduce them to plain rules by instantiation. Since gluing graph transformation has a well developed theory, this opens the door to lift results of that theory from instances to rules with variables.

1 Introduction

Rules are frequently used in computer science, for specifying the behavior of systems in an axiomatic way. Rules do often contain variables. *Term rewriting systems*, for instance, specify the evaluation of functions by rewrite rules such as

$$\mathsf{fib}(\mathsf{s}(\mathsf{s}(\mathsf{N}))) \to \mathsf{fib}(\mathsf{s}(\mathsf{N})) + \mathsf{fib}(\mathsf{N})$$

wherein the substitution of variables like N by terms yields ground rules like

$$\mathsf{fib}(\mathsf{s}(\mathsf{s}(\mathsf{0})))) \to \mathsf{fib}(\mathsf{s}(\mathsf{s}(\mathsf{0}))) + \mathsf{fib}(\mathsf{s}(\mathsf{0}))$$

that define the term rewrite relation [19]. *Two-level grammars*, another example, derive languages of words by rules such as

$$\langle \mathsf{T}_0\ \mathsf{expression} \rangle \ ::= \ \langle \mathsf{T}_1\ \mathsf{to}\ \mathsf{T}_0\ \mathsf{operator} \rangle\ \langle \mathsf{T}_1\ \mathsf{expression} \rangle$$

wherein variables like T_0 and T_1 are substituted by words of a context-free meta grammar in order to obtain context-free production rules like

$$\langle \mathsf{bool\ expression} \rangle \ ::= \ \langle \mathsf{int\ to\ bool\ operator} \rangle\ \langle \mathsf{int\ expression} \rangle$$

which in turn define the derivation relation of the grammar [2]. In both cases, rewriting is used twice: On the *meta level*, rule are instantiated by substituting

variables, producing rule instances that generate the rewrite relation on the *object level*. Variables make rules more abstract and more expressive: term rewriting systems define functions on infinite sets in a finite way, and two-level grammars derive recursively enumerable languages on the basis of context-free derivation.

This paper is about the use of variables in the area of graph transformation. Surprisingly, this concept has hardly been used in the major approaches of graph transformation that are documented in the handbook [27]. Early attempts by H. Göttler [11] and W. Hesse [17] to extend two-level word grammars to graphs were not successful. This work was inspired by three papers: D. Plump and A. Habel have devised variable hyperedges as placeholders for hypergraphs [25]; N. van Eetvelde and D. Janssens have introduced variable nodes as placeholders for graphs in [32]; and, D. Plump and S. Steinert have proposed variable labels as placeholders for attribute values [26]. These proposals follow the two-level model outlined above, but are based on different approaches to graph transformation. We catch on their ideas, but reformulate them so that they coherently use a single way of graph transformation on the object level. As a common basis, we choose the well-known gluing approach to graph transformation [7, 3] (also known as the algebraic, or double-pushout approach). In addition, we define clones, which are nodes that stand for sets of similar nodes within the same framework. The unified notions of variables allow to model several advanced concepts of graph transformation, such as the connection instructions of [10], and object set nodes and path expressions devised for programmed graph transformation [30]. The two-level model is modular so that different kinds of variables can also be combined easily. In this way, even the very advanced rules of [32] can be reduced to sets of simple gluing rules. Since variable instantiation is simply defined and easily understood, the resulting definitions are easily understood as well. And, since gluing transformation, the common basis of the instantiations, has a rich theory, results of this theory can help to prove properties of the rules as well.

The paper may raise a fundamental objection: *Need rules be that sophisticated?* Indeed, generative power is no issue, as gluing rules without variables already derive the recursively enumerated languages. However, in complex applications like software refactoring [22, 32], operations can be developed and verified more easily if they are expressed as a single rule – even a complex one – rather than as programs that control the application of simple rules in order to achieve the same effect.

The paper is structured as follows. We first recall graph transformation by gluing rules with relabeling in Section 2. This is the basis for discussing the use of variables in graph transformation in Section 3. Three kinds of variables are considered in Sections 4 to 6: attribute variables, clones, and graph variables. Section 7 discusses how instantiations can be combined, and how they may help to lift properties and results known from rule instances to rules. In Section 8 we conclude with pointers to related work and a discussion of further research.

2 Basic Graph Transformation

Rules in the gluing approach to graph transformation [7, 3] shall be used as rule instances on the object level. We have chosen this approach as it is not committed to a particular notion of graph (not even to graphs!), is widely used and has a rich theory. We confine the occurrences of rules to injective morphisms. It has been shown in [14] that this is no restriction; on the contrary, it permits a finer control of rule application. As in [15], we allow partially labeled graphs in rules so that nodes and edges may be relabeled in a transformation step.

Graphs. A *(partially labeled) graph* $G = \langle V_G, E_G, s_G, t_G, \ell_G \rangle$ over a set $\mathcal{C}$ of *labels* consists of disjoint finite sets V_G of *nodes* and E_G of *edges*, *source* and *target functions* $s_G, t_G \colon E_G \to V_G$ for edges, and a partial *labeling function* $\ell_G \colon V_G \cup E_G \to \mathcal{C}$. [1] An edge $e \in E_G$ is called *incident* to its source and target $s_G(e)$ and $t_G(e)$ and makes these nodes *adjacent* with each other. G is called *totally labeled* if the function ℓ_G is total.

A *premorphism* $m \colon G \to H$ between two graphs G and H consists of two functions $m_V \colon V_G \to V_H$ and $m_E \colon E_G \to E_H$ that preserve sources and targets, i.e., $s_H \circ m_E = m_V \circ s_G$ and $t_H \circ m_E = m_V \circ t_G$. If m also preserves defined labels, i.e., if $\ell_H(m(n)) = \ell_G(n)$ for all $n \in \mathrm{Dom}(\ell_G)$, it is called a *morphism*. A morphism m is *injective* (*surjective*) if both m_V and m_E are injective (surjective, resp.), and it is an *inclusion* if $m(n) = n$ for all nodes and edges n in G. Two graphs G and H are *isomorphic*, written $G \cong H$, if there is an injective and surjective morphism $m \colon G \to H$ that preserves all labels, i.e., $\ell_H(m(n)) = \ell_G(n)$ for all $n \in V_G \cup E_G$.

Rules. A *rule* $t = (L \leftarrow I \rightarrow R)$ consists of two inclusions $I \to L$ and $I \to R$ between partially labeled graphs such that for all $n \in V_L \cup E_L$, $\ell_L(n) = \bot$ implies $n \in V_I \cup E_I$ and $\ell_R(n) = \bot$, and, vice versa, $\ell_R(n) = \bot$ implies $n \in V_I \cup E_I$ and $\ell_L(n) = \bot$.

Example 1 (Rule). Assuming that the set $\mathcal{C}$ of labels contains natural numbers, the rule in Fig. 1, taken from [26], relabels the target of an edge in a graph. In our examples, numbers attached to the nodes in the graphs of a rule define the morphisms between them.

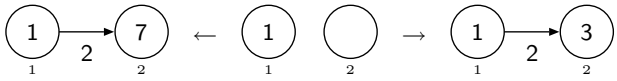
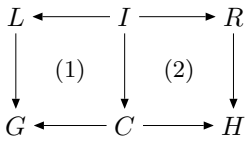


Fig. 1. A rule

[1] The set $\mathrm{Dom}(f) = \{x \in A \mid f(a) \text{ is defined}\}$ denotes the *domain* of a partial mapping f; we write $f(x) = \bot$ if $f(x)$ is undefined.

Transformation. Let G and H be totally labeled graphs and $t = (L \leftarrow I \rightarrow R)$ a rule. We say that t *transforms* G *to* H and write $G \Rightarrow_t H$ if there exists a graph C with two natural pushouts



so that the vertical morphisms are injective.[2] If $\mathcal{T}$ is a set of rules, we write $G \Rightarrow_{\mathcal{T}} H$ if $G \Rightarrow_t H$ for some rule $t \in \mathcal{T}$, and call $\Rightarrow_{\mathcal{T}} \subseteq \mathcal{G} \times \mathcal{G}$ the *transformation relation* induced by $\mathcal{T}$.

Rule Application. The above definition does not tell how a rule t is actually applied to a graph G in order to transform it into a graph H. For a totally labeled graph G and a rule $t = (L \leftarrow I \rightarrow R)$, an injective morphism $m\colon L \rightarrow G$ is called a *match* of t in G if it satisfies the following *dangling condition*: No node in $m(L \setminus I)$ is incident to an edge in $G \setminus m(L)$. Using this definition, a transformation $G \Rightarrow_t H$ is constructed as follows:

- Find a match $m\colon L \rightarrow G$ of t in G (if it exists).
- Remove all nodes and edges in $m(L \setminus I)$ from G, yielding the context graph C.
- Obtain H from the disjoint union of R and C by identifying the corresponding nodes and edges of $m(I)$ and R.

This construction defines H uniquely up to isomorphism.

3 A Framework for Graph Transformation with Variables

In the general setting for graph transformation with variables, a rule scheme will be instantiated to a set of rule instances that in turn defines the transformation relation. Below we outline general properties of rules, instantiation, and rule application that will be used in the following sections.

Rules. The set $\mathcal{C}$ of labels is extended by a set X of *variable names*. Graphs with labels from $\mathcal{C}$ and X are called *graph patterns* (or just *patterns*). In a pattern G, a variable name $x \in X$ may occur as a label, or be part of a label; it designates the label, or the so labeled node or edge as a placeholder. The *kernel* $\underline{G}$ of a pattern G is the graph obtained by removing all placeholders.

Then a *rule scheme* is a rule $t = (L \leftarrow I \rightarrow R)$ where L, I, and R are patterns.

[2] A pushout is natural if it is a pullback as well. The construction of natural pushouts is described in [15].

Instantiation. A *substitution function* σ specifies how variable names occurring in a rule shall be substituted.

Instantiation of a rule scheme t according to some substitution σ defines a particular *rule instance* t^σ. Then $\mathcal{T}(t) = \{t^\sigma \mid \sigma \text{ is a substitution}\}$ defines the *set of rule instances* for t. $\mathcal{T}(t)$ is a set of rules without variables that defines a transformation relation as described in the previous section.

Rule Application. The application of a rule t cannot generate the set $\mathcal{T}(t)$ in order to apply one of the resulting rules because this set is infinite in general. Instead, rule application proceeds as follows.

Let G be a graph, and $t = (L \leftarrow I \rightarrow R)$ a rule scheme.

1. Identify a *kernel match* $\underline{m}\colon \underline{L} \to G$ of the kernel $\underline{L}$ of L in G (if it exists).
2. Induce a substitution σ such that the kernel match $\underline{m}$ extends to a *full match* $m\colon L^\sigma \to G$ of t (if such a substitution exists).
3. Construct the instance R^σ and apply t^σ to construct the instance application $G \Rightarrow_{t^\sigma} H$.

Step 2 need not succeed in all cases; it may also be nondeterministic. We require that rule schemes used in the sequel satisfy the following two conditions:

- A rule scheme t is *left-linear* if every variable name occurs at most once in L.
- A rule scheme t is *closed* if every variable name occurring in R occurs in L as well.

These conditions make rule application easier. If a rule scheme is left-linear, the substitution σ can be induced (in step 2) by considering the unique occurrences of variable names in L one after the other. If a rule scheme is closed, the substitution σ induced in step 2 determines R^σ uniquely and completely.

4 Attribute Variables

In many applications of graph transformation, the nodes and edges of graphs have attributes like numbers or strings that shall be computed by functions during transformation. The attribute model of D. Plump and S. Steinert [26] represents attribute values as labels. The rule schemes on the meta-level are labeled with terms that specify how these values are computed. This is close to the models proposed in [28, 21].

Attributed Rules. *Attributed patterns* are graphs that are partially labeled with terms over a family $\mathcal{F} = (\mathcal{F}_n)_{n \geqslant 0}$ of graded *function symbols* that is disjoint to a set X of *variable names*. The set $\mathcal{T}(X)$ of *terms* is the least set satisfying (i) $x \in \mathcal{T}(X)$ for all variable names $x \in X$, (ii) $c \in \mathcal{T}(X)$ for all constant symbols $c \in \mathcal{F}_0$, and (iii) $f(t_1, \ldots, t_k) \in \mathcal{T}(X)$ for all function symbols $f \in \mathcal{F}_k$ and $k > 0$ terms $t_1, \ldots, t_k \in \mathcal{T}(X)$.

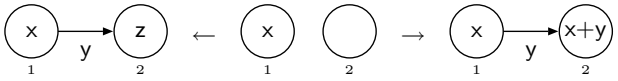


Fig. 2. An attributed rule

A rule scheme $t = (L \leftarrow I \rightarrow R)$ is an *attributed rule* if L, I, and R are attributed patterns. (Note that t is left-linear and closed, as every rule scheme.) We also require that t is *deterministic*, meaning that all terms used as labels in L are variables.

Instantiation. The meaning of function symbols $\mathcal{F}$ is given by an *algebra* A that consists of a *carrier set* A with elements $c_A \in A$ for all $c \in \mathcal{F}_0$, and functions $f_A \colon A^k \to A$ for all $f \in \mathcal{F}_k$ with $k > 0$.

A function $\alpha \colon X \to A$ is called an *assignment*. The extension $\hat{\alpha} \colon \mathcal{T}(X) \to A$ of α is defined by (i) $\hat{\alpha}(x) = \alpha(x)$ for all variable names $x \in X$, (ii) $\hat{\alpha}(c) = c_A$ for all constant symbols $c \in \mathcal{F}_0$, and (iii) $\hat{\alpha}(f(t_1, \ldots, t_k)) = f_A(\hat{\alpha}(t_1), \ldots, \hat{\alpha}(t_k))$ for all function symbols $f \in \mathcal{F}_k$ and all terms $t_1, \ldots, t_k \in \mathcal{T}(X)$.

For an attributed pattern G and an assignment $\alpha \colon X \to A$, its *instance* G^α is the partially labeled graph over A obtained by replacing the labeling function ℓ_G by $\hat{\alpha} \circ \ell_G$.[3] The instance of an attributed rule $t = (L \leftarrow I \rightarrow R)$ is the rule $t^\alpha = (L^\alpha \leftarrow I^\alpha \rightarrow R^\alpha)$ with partially labeled graphs over A.

Example 2 (Computing Path Weights). The attributed rule in Fig. 2 computes node attributes that represent the *weight* of paths in a graph. The assignment $\alpha = \{x \mapsto 3, y \mapsto 2, z \mapsto 7\}$ instantiates the attributed rule to the instance shown in Fig. 1 above (supposing that $+_A$ implements addition).

Rule Application. A transformation step $G \Rightarrow_t H$ via some attributed rule $t = (L \leftarrow I \rightarrow R)$ is constructed by finding a *pre-morphism* $m \colon L \to G$. Since t is left-linear and all defined labels in L are variables, mapping $\ell_L(n)$ onto $\ell_G(m(n))$ for every labeled node or edge n in L, uniquely defines a partial assignment $\alpha \colon X \to A$ that is defined for all $x \in X$ occurring in L. Then α determines the instantiation R^α completely since t is closed, and the transformation step $G \Rightarrow_t H$ is uniquely defined as well (up to isomorphism). The induction of assignments is thus deterministic and always successful.

Discussion. For simplicity, our definitions deal with untyped terms and algebra whereas the labels in [26] are many-sorted. In that paper, attribute values are also used to define *conditional rules* of the form $t = (L \leftarrow I \rightarrow R$ **where** $c)$ with a boolean expression c; instantiation yields an instance $t = (L^\alpha \leftarrow I^\alpha \rightarrow R^\alpha)$ if and only if $\hat{\alpha}(c)$ evaluates to true. Then the induction of assignments is still deterministic, but partial, as some assignments α yield invalid rule instances t^α.

[3] The composition $\hat{\alpha} \circ \ell_G$ is undefined at the nodes and wedges where ℓ_G is undefined.

The algebra A can be implemented by some library in a programming language. It can also be defined by rewriting itself: Every confluent and terminating term rewriting system defines an algebra. The values of A may as well be graphs, and the operations can be defined by (confluent and terminating) graph transformations. This idea has been considered in [29] for the first time.

Attribution by instantiation is considerably simpler than other models, such as the one proposed in [16], where every graph is burdened by an infinite set of nodes that represents all values of A, and cluttered up with edges that point from nodes to their actual attribute values. It is also more general as edges may be labeled as well as nodes.

5 Clone Variables

Rules in programmed graph transformation [30] may contain "object set identifiers", which are nodes in a rule pattern that shall match the set of all nodes in a graph that are connected to the nodes of the rule in the same way. We call such nodes *clones*.

Cloning Rules. Extend the label set $\mathcal{C}$ by the Cartesian product $\mathcal{C}\times X$, where X is a set of variable names disjoint to $\mathcal{C}$. A partially labeled graph G over $\mathcal{C}\cup(\mathcal{C}\times X)$ is a *clone pattern* if labels of the form (c, x) with $c \in \mathcal{C}$ and $x \in X$ are only used on nodes. A node v in a clone pattern G with $\ell_G(v) = (c, x)$ is called an *x-fold c-clone*, or just a *clone* if c and x do not matter. (Labels "$(, x)$" indicate clones with undefined label.) All other nodes are called *constant*. The clone variable x in a label (c, x) stands for a number of c-nodes that are instantiated for v. Although this number is arbitrary for every variable name x, it restricts x-fold clones v and w to be instantiated by the same number of nodes.

A rule scheme $t = (L \leftarrow I \rightarrow R)$ consisting of clone patterns is a *cloning rule*. (It is left-linear and closed.)

Instantiation. Let $\mu\colon X \to \mathbb{N}$ be a *multiplicity function* which specifies how many instances of clones shall be inserted in a graph.

For a clone pattern G and a multiplicity function μ, the *instance* G^μ is defined as follows:

1. Replace every x-fold clone v by a set of $\mu(x)$ nodes, which are called the *instances* of v and are denoted by $v^{\mu(x)}$.
2. Replace every edge between a clone v and a constant node w by $\mu(x)$ edges between every instance in $v^{\mu(x)}$ and w, with the same label and direction.
3. Replace every edge between two x-fold clones v and w by $\mu(x)$ edges between corresponding instances in $v^{\mu(x)}$ and in $w^{\mu(x)}$, with the same label and direction. (See Fig. 3 on the left.)
4. Replace every edge between an x-fold clone v and a y-fold clone w (with different variables $x \neq y$) by $\mu(x) \times \mu(y)$ edges between every instance in $v^{\mu(x)}$ and every instance of $w^{\mu(y)}$, with the same label and direction. (See Fig. 3 on the right.)

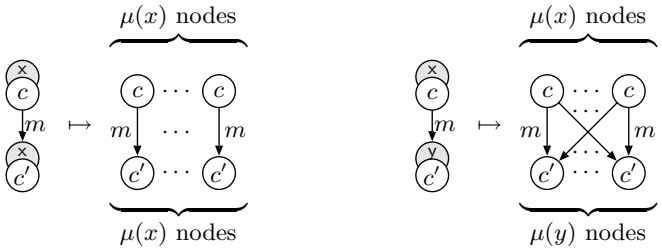


Fig. 3. Instantiation of edges between multiple nodes, case 3 (left) and case 4 (right)

The rule $t^\mu = (L^\mu \leftarrow I^\mu \rightarrow R^\mu)$ is the instance of a cloning rule for a multiplicity function μ.

Example 3 (A Pull-Down-Method Refactoring). The cloning rule in Fig. 4 has been adapted from [32]. It describes a transformation of graphs representing object-oriented programs where a method definition (denoted by the square δ-node) is pulled down from a class (represented by the constant C-node) to its subclasses (represented by the n-fold C-clone). For the transformation rule, the neighborhood of the class and the method definition have to be considered; this is done by the unlabeled clones with the clone variables x, y, and z.

Instantiating clone variables by $\mu = \{n \mapsto 3, x \mapsto 0, y \mapsto 1, z \mapsto 2\}$ yields the instance shown in Fig. 5. The i-edges are induced by case 2 of the instantiation, whereas m-edges are obtained according to case 3, and the unlabeled edges to the δ-clones are generated according to case 4.

Rule Application. The *kernel match* of a cloning rule $t = (L \leftarrow I \rightarrow R)$ is an injective morphism $\underline{m}\colon \underline{L} \rightarrow G$, where $\underline{L}$ is the kernel of L from which all clones and their incident edges have been removed. The kernel match uniquely

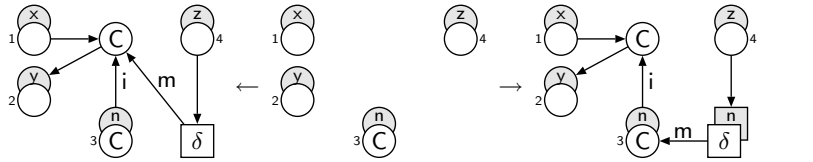


Fig. 4. Cloning rule for the pull-down-method refactoring

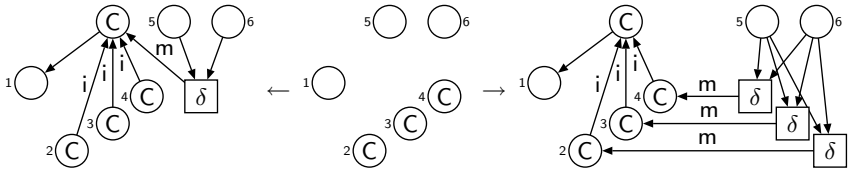


Fig. 5. Cloning rule instance for the pull-down-method refactoring

determines the instances of the clones in L as follows: If $\{\bar{v}_1, \ldots, \bar{v}_k\}$ is the set of nodes adjacent with an x-fold c-clone $\bar{v}$ in L, search the adjacent nodes of their kernel matches $\underline{m}(\bar{v}_1), \ldots, \underline{m}(\bar{v}_k)$ for the set $\{v_1, \ldots, v_k\}$ of all c-nodes that are connected to $\underline{m}(\bar{v}_1), \ldots, \underline{m}(\bar{v}_k)$ in G in the same way as $\bar{v}$ is connected to $\bar{v}_1, \ldots, \bar{v}_k$ in L. This defines $\mu(x) = k$, $v^{\mu(x)} = \{v_1, \ldots, v_k\}$, and L^μ. The interface instance I^μ is included in L^μ, and the right hand side R can be instantiated by making $\mu(x)$ copies of all x-fold clones that are fresh in R, and connecting them accordingly. It is important that the clone variable x is defined by the left hand side of the rule in order to know how many copies shall be made for every fresh clone. (In Fig. 5, e.g., the clone variable n of the δ-clone on the right hand side is bound by the C-clone on the left hand side.)

The multiplicity function μ can be uniquely determined for every kernel match $\underline{m}$. However, the instance t^μ may not apply because it violates the dangling condition. For instance, the cloning rule in Fig. 4 cannot be applied to a graph where the match of the constant C-node has an incoming edge e that is not labeled with i because deletion of that node would leave e dangling.

Simulating Connecting Graph Transformation. Cloning rules can simulate connecting graph transformation in the sense of [22, 32]. The *connecting rules* in that paper take the form (L, R, in, out) where L is the graph to be matched (and deleted), R is the graph, a copy of which has to be replaced for the match of L, and the *connection instructions* $in, out \subseteq V_L \times V_R \times \mathcal{C}_E \times \mathcal{C}_E$ specify how the nodes incident to the match $m(L)$ are to be connected to the nodes in the copy of R:

- An instruction $(v, w, c, d) \in in$ says that every neighbor node v' pointing to v by a c-edge shall point to the node w in R by a d-edge.
- An instruction $(v, w, c, d) \in out$ says that if v points to some neighbor node v' by a c-edge, the node w in R shall point to the node v' by a d-edge.
- All other edges connecting nodes in L to neighbor nodes (with labels or directions not mentioned in connection instructions, that is) are deleted.

The cloning rule $t = (\tilde{L} \leftarrow \tilde{I} \rightarrow \tilde{R})$ simulating a connecting rule (L, R, in, out) is defined as follows:

- $\tilde{L}$ is obtained by extending L with a set of mutually distinct clones so that every node v in L is adjacent with two clones $x_{v,c,in}$ and $x_{v,c,out}$, for every edge label c in $\mathcal{C}_E$.
- $\tilde{I}$ consists of all clones of $\tilde{L}$.
- $\tilde{R}$ is obtained by extending R with $\tilde{I}$, and by connecting the clones of $\tilde{I}$ to nodes w in R according to the connection instructions:
  - If $(v, w, c, d) \in in$, let the clone $x_{v,c,in}$ point to w by a d-edge.
  - If $(v, w, c, d) \in out$, let w point to the clone $x_{v,c,out}$ by a d-edge.

Admittedly, this simulation is rather clumsy because $2 \times |\mathcal{C}|$ variables are needed for every node of a left hand side. Especially if $\mathcal{C}$ is infinite, we need a notation for clones that match "all other neighbor nodes" of some node v, i.e., all adjacent nodes that are connected by labels and directions not mentioned in the other clones at v.

6 Graph Variables

In [25], D. Plump and A. Habel have devised rules wherein variable hyperedges are placeholders for hypergraphs that are substituted by hyperedge replacement [5]. We "translate" hyperedge replacement to a simple way of node replacement in graphs.

Rules with Variable Nodes. The label set $\mathcal{C}$ is extended by a disjoint set X of *variable names*. Every variable name $x \in X$ comes with a *type* $\text{type}(x) \in \mathcal{C}^* \times \mathcal{C}^*$ that specifies the number and labels of its incident edges as follows. Let G be a graph over $\mathcal{C} \cup X$. A node v in G is a *node variable* if $\ell_G(v) = x \in X$. A node variable v with $\ell_G(v) = x$ and $\text{type}(x) = (c_1 \cdots c_n, \bar{c}_1 \cdots \bar{c}_{\bar{n}})$ is *well-typed* if v is the target of n edges $e_1 \cdots e_n$ with $\ell_G(e_i) = c_i$ for $1 \leqslant i \leqslant n$, and the source of $\bar{n}$ edges $\bar{e}_1 \cdots \bar{e}_{\bar{n}}$ with $\ell_G(\bar{e}_j) = \bar{c}_j$ for $1 \leqslant j \leqslant \bar{n}$; v is *straight* if all sources of its ingoing edges and all targets of its outgoing edges are pairwise distinct, and finally, v is *apart* if there is no variable node among these adjacent nodes.

A graph G over $\mathcal{C} \cup X$ is a *graph pattern* if all edges have constant labels, and all variable nodes are well-typed, straight, and apart.

A rule scheme $t = (L \leftarrow I \rightarrow R)$ with graph patterns L, I, and R is a *rule with variable nodes* if I is constant. As usual, we also assume that t is left-linear and closed.

Instantiation. Variables in a graph pattern G are instantiated by replacing variable nodes by graphs.

With $\langle x^\star \rangle$ we denote the *star graph* of a variable name x, which is the graph with an x-labeled center node and incident edges according to $\text{type}(x)$, plus unlabeled nodes at the other ends of these edges. The graph $\langle x^\circ \rangle$ is the discrete subgraph of $\langle x^\star \rangle$ that consists just of its border nodes.

A rule of the form $t = (\langle x^\star \rangle \leftarrow \langle x^\circ \rangle \rightarrow S)$ is called a *simple node replacement rule*.[4] (Fig. 6 shows such a rule.) A graph substitution γ maps variable names $x \in X$ onto node replacement rules $\gamma(x) = (\langle x^\star \rangle \leftarrow \langle x^\circ \rangle \rightarrow S)$.

The instantiation of a graph pattern G according to a graph substitution γ applies the simple node replacement rules $\gamma(\ell_G(v))$ to all variable nodes v in G in parallel.

In a graph pattern G, straightness guarantees that the simple node replacement rule $\gamma(x)$ applies to every x-node in G. Apartness ensures that the matches of two nodes replacement rules t and t' either overlap only in their border nodes, or they overlap completely, and $t = t'$. In the first case, the steps commute by the parallel independence results for gluing rules; in the second case, the definition of γ makes sure that the rules are equal. Thus simple node replacement via γ is strongly confluent for graph patterns, and the instance H^γ is unique up to isomorphism.

[4] This definition covers a restricted form of node replacement that corresponds to hyperedge replacement [5]. Full node replacement rules have connection instructions, and may be applied to variables that are not apart.

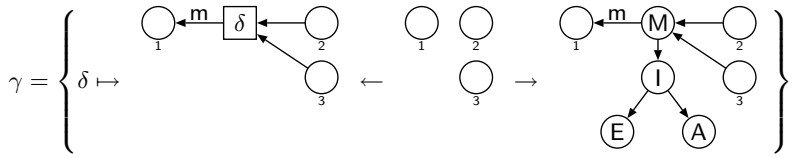


Fig. 6. A graph substitution

Example 4 (A Pull-Down-Method Refactoring). The substitution γ shown in Fig. 6 with $\mathrm{Dom}(\gamma) = \{\delta\}$ specifies a simple node replacement rule for the variable name δ with $\mathrm{type}(\delta) = (\bot\bot, \mathsf{m})$ that replaces δ-stars by a method definition. In this case, the method body is a tree representing an if-statement **if** E **then** A with a condition E and a simple assignment A.

The substitution γ instantiates the rule with variable nodes in Fig. 5 to the rule shown in Fig. 7 below.

Rule Application. For constructing a transformation of a graph G via a rule t with variable nodes, we define the kernel of the left hand side pattern L in such a rule as the subgraph $\underline{L}$ where all variable nodes and their incident edges are removed, and determine a kernel match $\underline{m}\colon \underline{L} \to G$.

We then attempt to induce a substitution γ by starting, for every variable node v in L, at the matches of its adjacent nodes, say $v_1, \ldots, v_k$. Every candidate for $\gamma(\ell_L(v))$ has to be isomorphic to a subgraph $S \subseteq G$ that overlaps with $\underline{m}(\underline{L})$ and with the substitution candidates for other variables only in the nodes $\underline{m}(v_1), \ldots, \underline{m}(v_k)$.

One the one hand, there need not be such a graph, for instance if one v_i is not in the interface I, not adjacent with any other variable node in L, and if its kernel match is source or target of an edge e outside $\underline{m}(\underline{L})$ that is incident to the kernel match $\underline{m}(v_i)$ of a node $\bar{v}$ in $\underline{m}(\underline{L})$ other than $\underline{m}(v_1), \ldots \underline{m}(v_k)$. This edge cannot belong to S, it cannot belong to the substitution of another variable, and it may also not belong to the context of the transformation, since deletion of $\underline{m}(v_i)$

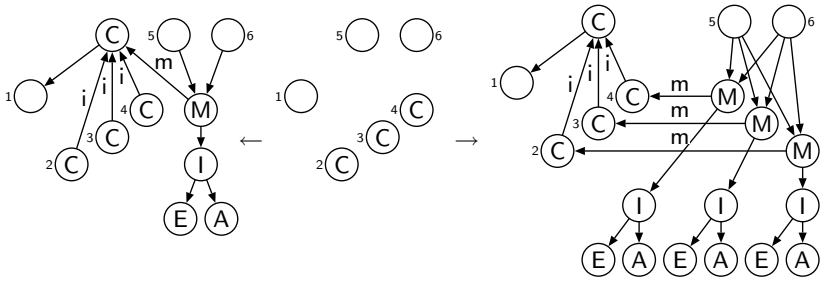


Fig. 7. Rule instance for the pull-down-method refactoring

would leave e dangling. On the other hand, there may be several candidates for substituting a variable in general.

Instantiation of the right hand side R is unique once a graph substitution γ has been found. The transformation $\mathcal{G} \Rightarrow_{t^\gamma} H$ and the resulting graph H are then unique up to isomorphism.

Discussion. Variable nodes can also be substituted if they are not apart, and by applying full node replacement with connection instructions [10]. However, this requires some precaution as node replacement is not confluent in general.

Programmed graph transformation rules [30] feature *path expressions* by which the existence of paths can be specified. Path expressions can be considered as variable edges that may be substituted by chain-like graphs which are defined by edge replacement.

7 Combining Variables

It is fairly straight-forward to combine different kinds of variables in order to get even more powerful graph transformation rules. Simulation of the rules proposed in [32], for instance, requires at least two kinds of variables: The rules need clone variables as they use connection instructions, and variable nodes as they copy subgraphs of arbitrary size (like the method bodies). The rule in Fig. 4 contains clone variables and a variable node (δ) which are instantiated by making clones first, and instantiating the variable node δ afterwards. The combination requires some care in order to achieve the desired results: The clone variables like δ on the right hand side of Fig. 4 need to be cloned before the resulting set of variable nodes is substituted itself.

The rules in [32] might also take advantage of attribute variables if computations on primitive values shall be performed. In this case, attributes should be instantiated last.

Lifting Properties of Rule Instances. Given that instantiation is a simple concept, there is hope that some results for the underlying rule instances can be lifted to the level of the rule schemes. Confluence, for instance, is relevant for many applications. Fortunately, the theory of gluing graph transformation provides criteria for the parallel independence of transformations [7], and there is also a critical pair lemma [24]. In [13], parallel independence has been lifted to transformation with variable nodes. We think that confluence for the kind of rules used in [32] can be proved more easily in the two-level model than in the original definition.

8 Conclusions

In this paper we have studied how variables can be used in graph transformation. The general framework of two-level transformation – instantiation of rule

schemes to rule instances which in turn define transformations – applies to several kinds of variables: attribute variables, clones, and graph variables. Since the model is modular, different kinds of variables can be combined. This yields very expressive rules that are still comprehensible, because variables are a familiar concept in specification and programming. Instantiation reduces rule schemes to standard gluing rules, and thereby allows to add important concepts of connecting and programmed graph transformation to the gluing approach – such as connection instructions and path expressions. Gluing graph transformation has a rich theory that is automatically available on the level of rule instances. Some of this theory can probably be lifted to the level of rule schemes as well.

Related Work. Several authors have studied advanced graph transformation rules. The early book [23] by M. Nagl defines a very general way of operational graph transformation. The structured rules of H.-J. Kreowski and G. Rozenberg [20] combine gluing rules with connection instructions. Both approaches do not consider graph variables.

Future Work. The work started in this paper can be continued in several directions.

The basic transformation model can be extended by types and shapes [18], by negative application conditions [12], and so on.

We would also like to loosen the standard conditions on rule schemes. For left-linearity, this is rather easy; but then, the induction of substitutions can no longer be done independently for every variable, since different occurrences of the same variable must have equal substitutions. Dropping the closedness condition is only feasible if transformation is lifted to graphs with variables. Then matching of graphs with variables against graphs has to be replaced by *unification* of two graphs with variables. However, it is unknown whether graph unification is decidable.

An important issue is to make the induction of substitutions for graph variables less nondeterministic, because variable nodes may have many different substitutions. The paper [6] gives rather strict conditions for the unique induction of substitutions. In [18], the nondeterminism of induction is reduced by requiring substitutions to be "shaped" graphs that are generated by a hyperedge replacement grammar. Then, substitution candidates can be found by parsing according to the grammar. In the rule in Fig. 5, for instance, the substitutions of δ could be restricted to the syntax trees of method definitions. A substitution for δ can be induced by parsing according to the syntactic rules.

Of course, we are also interested in evaluating how useful variables are for modeling realistic case studies. Refactoring seems to be an area where graph transformation can be applied successfully, and where advanced concepts are needed for the transformation rules. Such case studies could also show whether it is really possible to lift theorems from the underlying world of gluing rules to the high-level rules.

Acknowledgment. In 1978, Hartmut Ehrig urged I.-R. Schmiedecke and me to define extended affixgrammars (a variant of the two-level grammars mentioned in the introduction) so that he could understand them. This resulted not only in a graph grammar definition [9], but also introduced me to the field of graph transformation which has fascinated me ever since. Now the topics of this paper brought me back to two-level grammars where all this began. *Thank you, Hartmut!*

References

1. Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg, editors. *Proc. Graph Grammars and Their Application to Computer Science and Biology*, number 73 in Lecture Notes in Computer Science. Springer, 1979.
2. C.J. Cleaveland and R.C. Uzgalis. *Grammars for Programming Languages*. Elsevier, New York, 1977.
3. Andrea Corradini, Hartmut Ehrig, Reiko Heckel, Michael Löwe, Ugo Montanari, and Francesca Rossi. Algebraic approaches to graph transformation, part I: Basic concepts and double pushout approach. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*. World Scientific, 1997.
4. Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *1st Int'l Conference on Graph Transformation (ICGT'02)*, number 2505 in Lecture Notes in Computer Science. Springer, 2002.
5. Frank Drewes, Annegret Habel, and Hans-Jörg Kreowski. Hyperedge replacement graph grammars. In Rozenberg [27], chapter 2, pages 95–162.
6. Frank Drewes, Berthold Hoffmann, and Mark Minas. Constructing shapely nested graph transformations. In Hans-Jörg Kreowski and Peter Knirsch, editors, *Proc. Int'l Workshop on Applied Graph Transformation (AGT'02)*, 2002. 107–118.
7. Hartmut Ehrig. Introduction to the algebraic theory of graph grammars. In Claus et al. [1], pages 1–69.
8. Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors. *2nd Int'l Conference on Graph Transformation (ICGT'04)*, number 3256 in Lecture Notes in Computer Science. Springer, 2004.
9. Hartmut Ehrig, Berthold Hoffmann, and Ilse-Renate Schmiedecke. A Graph-Theoretical Model for Multi-Pass Parsing. In J.R. Mühlbacher, editor, *Proc. 7th Conf. on Graph-Theoretical Concepts in Comp. Sci. (WG'81)*, pages 19–32, München-Wien, 1982. Hanser Verlag.
10. Joost Engelfriet and Grzegorz Rozenberg. Node replacement graph grammars. In Rozenberg [27], chapter 1, pages 1–94.
11. Herbert Göttler. *Zweistufige Graphmanipulationssysteme für die Semantik von Programmiersprachen*. Dissertation, Universität Erlangen-Nürnberg, 1977. [In German].
12. Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26:287–313, 1996.
13. Annegret Habel and Berthold Hoffmann. Parallel independence in hierarchical graph transformation. In Ehrig et al. [8], pages 178–193.
14. Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.

15. Annegret Habel and Detlef Plump. Relabelling in graph transformation. In Corradini et al. [4], pages 135–147.
16. Reiko Heckel, Jochen M. Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In Corradini et al. [4], pages 161–176.
17. Wolfgang Hesse. Two-level graph grammars. In Claus et al. [1], pages 255–269.
18. Berthold Hoffmann. Shapely hierarchical graph transformation. In *Proc. IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 30–37. IEEE Computer Press, 2001.
19. Jan Willem Klop. Term rewriting systems. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.
20. Hans-Jörg Kreowski and Grzegorz Rozenberg. On structured graph grammars, I and II. *Information Sciences*, 52:185–210 and 221–246, 1990.
21. Michael Löwe, Martin Korff, and Annika Wagner. An algebraic framework for the transformation of attributed graphs. In Sleep et al. [31], pages 185–199.
22. Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour-preserving transformation. In Corradini et al. [4], pages 286–301.
23. M. Nagl. *Graph-Grammatiken: Theorie, Anwendungen, Implementierungen*. Vieweg-Verlag, Braunschweig, 1979. In German.
24. Detlef Plump. Hypergraph rewriting: Critical pairs and undecidability of confluence. In Sleep et al. [31], pages 201–213.
25. Detlef Plump and Annegret Habel. Graph unification and matching. In Janice E. Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, *Proc. Graph Grammars and Their Application to Computer Science*, number 1073 in Lecture Notes in Computer Science, pages 75–89. Springer, 1996.
26. Detlef Plump and Sandra Steinert. Towards graph programs for graph algorithms. In Ehrig et al. [8], pages 128–143.
27. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations*. World Scientific, Singapore, 1997.
28. Georg Schied. *Über Graphgrammatiken, eine Spezifikationsmethode für Programmiersprachen und verteilte Regelsysteme*. Dissertation, Universität Erlangen-Nürnberg, 1992. [In German].
29. Hans-Jürgen Schneider. On categorical graph grammars integrating structural transformations and operations on labels. *Theoretical Computer Science*, 109:257–274, 1993.
30. Andy Schürr. Programmed graph replacement systems. In Rozenberg [27], chapter 7, pages 479–546.
31. M. Ronan Sleep, Rinus Plasmeijer, and Marko van Eekelen, editors. *Term Graph Rewriting, Theory and Practice*. Wiley & Sons, Chichester, 1993.
32. Niels van Eetvelde and Dirk Janssens. Extending graph rewriting for refactoring. In Ehrig et al. [8], pages 399–415.