Adaptive Graph Transformation Rules*

Berthold Hoffmann[1], Dirk Janssens[2],
Mark Minas[3], and Niels Van Eetvelde[2] **

[1] Universität Bremen, Germany
[2] Universiteit Antwerpen, Belgium
[3] Universität der Bundeswehr München, Germany

Abstract. The rule-based transformation of graphs has many applications in computer science and beyond. The adaptive graph transformation rules proposed in this paper contain variables as placeholders for attribute values and subgraphs of varying number and shape. Replacing these variables by concrete values derives sets of double pushout rules that actually transform graphs. Even rather complex transformations occurring in real-life applications, such as the *Pull-Up-Method* refactoring operation, can be specified by a single adaptive rule.

1 Introduction

Many systems in computer science (and other parts of science) have structured states that gradually evolve according to rules. Graphs and graph transformation are natural candidates for modeling states and behavior of such systems in an axiomatic way. Unfortunately, the rules of classical formalisms are rather restricted. E.g., double pushout (DPO) rules [8] just allow to remove a constant subgraph from a host graph, and insert another constant graph for it. For describing the behavior of complex real-life systems, one needs a large number of such rules, and may have to program their application with control structures. In this paper we pursue another idea: we make rules *adaptive* by introducing variables as placeholders for values of different kinds. Every adaptive rule abstracts from a set of simple rules, one for every assignment of values to its variables. In [31, 19, 20] we proposed *cardinality variables* and a *cloning* concept that allows to make copies of subgraphs, and *graph variables* that allow to expand nodes by graphs of different shape. Now we add *attributes* by which computations can be defined on labels, and refine the way how variables can be combined: Attribute and graph variables may be cloned *individually*, yielding a set of different variable names of the same type, or *uniformly*, yielding a set of identical names. In an ongoing case study of using graph transformation for refactoring [22, 31, 32, 20],[4] these concepts turned out to be useful: Even rather complex transformations occurring in real-life applications, like the *Pull-Up-Method* refactoring

* Supported by SeGraVis (www.segravis.org), a European research training network.
** On leave to Universität Bremen on a SeGraVis grant (October 2005–January 2006).
[4] **Niels**: *Shall we cite your PhD thesis here, as "ongoing work"?*

operation, can be specified by a single adaptive rule. Other examples have shown that graph variable expansion [18] or cloning [23] alone are useful for specifying graph transformations.

The paper is structured as follows: The next section recalls a simple kind of DPO graph transformation, and discusses how far it meets the needs to express refactoring by examining the *Pull Up Method* refactoring operation. In Section 3, we develop three adaptations of graph transformation rules—attribute evaluation, expansion of nodes by graphs, cloning of subgraphs—and discuss their interplay, in particular the individual cloning of graph and attribute variables. Then adaptive graph transformation is defined, in Section 4. The conclusions in Section 5 point out related and future work.

2 Graph Transformation

Refactoring prepares object-oriented software for evolution by improving its structure without changing its behavior. Since software models—specifications in UML, or representations of program code—are often graph-like, graph transformation is a natural candidate for specifying refactorings. One of the refactoring operations defined by Martin Fowler [13] will be our running example.

Example 1 (Pull-Up-Method). This refactoring applies to a method definition d in a class c where all sibling classes define methods that have the same name and parameter list, and bodies that are semantically equivalent. Note that this does not imply that they have identical syntax trees. *Pull-Up-Method* moves d to c's superclass and removes the equivalent method definitions from the sibling classes. However, checking semantical equivalence of all these method bodies remains the programmer's task, and is beyond the scope of our considerations here.

For the case study, the structure of programs is represented by labeled graphs.

Definition 1 (Graph). Let Σ be a set of *labels* that may be infinite.

A *graph* $G = \langle \dot{G}, \bar{G}, s_G, t_G, \dot{\ell}_G, \bar{\ell}_G \rangle$ (*labeled with* Σ) consists of disjoint finite sets $\dot{G}$ of *nodes* and $\bar{G}$ of *edges*, of two functions $s_G, t_G \colon \bar{G} \to \dot{G}$ defining the *source* and *target* nodes of its edges, and of two functions $\dot{\ell}_G \colon \dot{G} \to \Sigma$ and $\bar{\ell}_G \colon \bar{G} \to \Sigma$ that assign *labels* to its nodes and edges.

A *morphism* $m \colon G \to H$ from a graph G to a graph H consists of two functions $\dot{m} \colon \dot{G} \to \dot{H}$ and $\bar{m} \colon \bar{G} \to \bar{H}$ that preserve sources, targets, and labels:

$$s_H \circ \bar{m} = \dot{m} \circ s_G,\ t_H \circ \bar{m} = \dot{m} \circ t_G, \text{ and } \ell_H \circ m = \ell_G$$

$\mathcal{G}_\Sigma$ shall denote the class of graphs labeled with Σ. An edge in a graph is *incident* with its source and target nodes, and makes these nodes *adjacent* to each other. We often do not distinguish between nodes and edges if a statement holds analogously for both sets; then "$g \in G$" refers to the (*graph*) *item* "$g \in \dot{G} \cup \bar{G}$". A graph morphism $m \colon G \to H$ is injective (surjective) if its functions have the respective property; an injective morphism m is an inclusion if $m(g) = g$ for

every item $g \in G$; then we write $G \subseteq H$ and call G a *subgraph* of H. An injective and surjective morphism $m\colon G \to H$ is an *isomorphism*; it makes G and H *isomorphic*, written $G \cong H$.

Example 2 (Program Graphs). Fig. 1 shows two *program graphs*, a language-independent representation for object-oriented programs is introduced in [22]. Nodes (rounded boxes) represent program entities, and edges (arrows from their sources to their targets) represent syntactical and contextual relations among entities. Labels (which are inscribed to nodes, and ascribed to edges) classify the types of entities and relations, and may contain attribute values (like the name of an entity). The typing of program graphs is discussed in [22].

A simple notion of graph transformation will be used to perform refactorings.

Definition 2 (Transformation). A (*graph transformation*) *rule* $r = L/R$ consists of *left hand side* L and *right hand side* R that are graphs in $\mathcal{G}_\Sigma$ sharing a subset $I = \dot{L} \cap \dot{R}$ of *interface nodes*.

Let $G \in \mathcal{G}_\Sigma$ and $r = L/R$ a rule. An injective morphism $m\colon L \to G$ is a *match* of r in G if it satisfies the following *dangling condition*:

$$\text{No node in } \dot{m}(\dot{L} \setminus I) \text{ is incident with an edge in } \bar{G} \setminus \bar{m}(\bar{L}).$$

Then r *transforms* G (via the match m) to a graph $H \in \mathcal{G}_\Sigma$ that is constructed as follows:

1. Clip $m(L)$ off G at the nodes $\dot{m}(I)$ to obtain the *context graph* C.
2. Glue R to C by identifying $\dot{m}(I)$ and I in the disjoint union $C \uplus R$, and label the node $\dot{m}(v)$ with $\ell_R(v)$ for every $v \in \dot{I}$.

Then we write $G \Rightarrow_{m,r} H$ or just $G \Rightarrow_r H$.

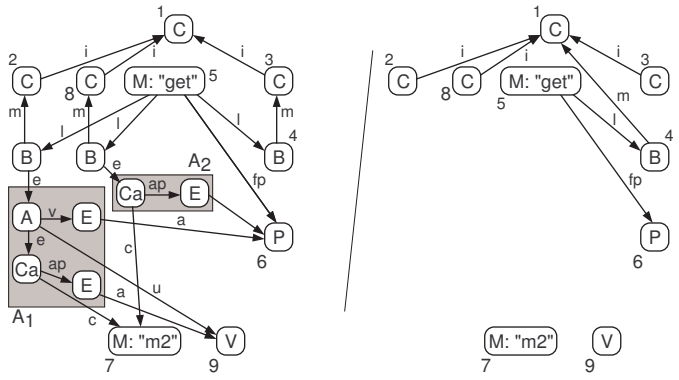

Fig. 1. A concrete rule for *Pull-Up-Method* refactoring

This notion of transformation corresponds to DPO graph transformation with relabeling [16].[5] The non-injective matches allowed in "classical" DPO transformation [8] can be simulated by taking *quotients* of rules as proposed in [15]: If $r = L/R$ is a rule, every rule $r' = L'/R'$ obtained by identifying pairs of nodes $v, v' \in I$ that satisfy $\ell_L(v) = \ell_L(v')$ and $\ell_R(v) = \ell_R(v')$ is a *quotient* of r. Since r has finitely many quotients (as long as r is finite), the set $Q(r)$ of quotient rules can be used instead of just r. Moreover, unwanted elements may be removed from $Q(r)$ in order to control how parts of r may be identified in transformations.

Example 3 (A Refactoring Rule). Fig. 1 shows a rule performing a *Pull-Up-Method* refactoring on program graphs. (The interface nodes of the rule are specified by annotating them with the same number in L and R.)

The C-nodes represent a class (3) with its superclass (1) and all its sibling classes (2,8). A method signature (5) with one parameter (6) has overloaded bodies (B-nodes) that are members of all sibling classes (2,8); both implementations make a call to another method m2 (7). One makes a call (Ca-node) to m2 using the formal parameter of get as its actual parameter whereas the other first assigns (A-node) this parameter to a variable (9) and then calls m2 with this variable. The E-nodes represent expressions, which are accesses to variables and parameters in this case. Obviously, these implementations have the same semantics but differ syntactically.

On the right hand side of the rule, the implementations of get in the sibling classes (2,8) are removed, and its body (4) is moved to the superclass (1). (The expressions defining the body (4) need not be mentioned in the rule as they are not changed by the refactoring.)

This rule does not define *Pull-Up-Method* in general, however, as it only applies to particular situations:

- Here the class (3) has two sibling classes; the method has one parameter, and its bodies access two names. In general, there can be any number of sibling classes, parameters, and names.
- The syntactical structure of the method bodies may vary; a general rule should apply to bodies of any form.
- This rule pulls up a method named get, but this name should not matter for the general rule.

Note that a set of simple DPO rules would be needed in order to express the general refactoring: Some for checking that the method is implemented in all sibling classes, other ones for removing all but one implementation recursively, and finally the one pulling up the remaining implementation. The applications of these rules have to be controlled in a non-trivial way, and it will not be easy to see that they do what they should, let alone to show it.

[5] Actually, it corresponds to the special case where, in the DPO rule $L \leftarrow I \rightarrow R$, I is completely unlabeled and discrete, while L and R are totally labeled. Since edges are not in the interface, they are never relabeled, but just removed and (re-) inserted by a transformation step.

Therefore we propose to define this refactoring by a single general rule that is then *adapted* with respect to attribute values, the form of certain subgraphs, and the number of copies of certain subgraphs. The rest of the paper describes these adaptations and their combination.

3 Graph Adaptation

The graphs occurring in adaptive rules shall be variable in the following sense:

- Their edges and nodes are labeled by terms that specify attribute values, referring to values of other nodes and edges by attribute variables.
- They may contain *graph variables* which designate *stars* (nodes with their incident edges and adjacent nodes) that may be expanded to varying graphs.
- They may contain nodes that may occur once, several times, or not at all. *Cardinality variables* distinguish these clonable nodes and specify which nodes shall have the same number of clones.
- Graph and attribute variables may be qualified as *individual* if they are contained in a clonable node; clones of such variables get different fresh names in each of their copies so that they can be substituted by different graphs and attribute values, respectively. Other graph and attribute variables are *uniform*, and keep the same name so that all clones of these variables will have equal substitutions.

Attribute variables, graph variables, and cardinality variables trigger derivations of a graph, called attribute evaluation, expansion, and cloning.

3.1 Attribute Evaluation

Attribute evaluation needs a semantic basis that provides values and computations. We define this basis by the carrier sets and functions of a many-sorted algebra. The evaluation of attributes is then specified by the interpretation of terms that label nodes and edges of a graph, as in [25].

Definition 3 (Order-Sorted Algebra). Let S be a finite set of *sorts* wit a *subsort relation* $\leqslant\ \subseteq S \times S$, and let $\Omega = (\Omega_{w,s})_{w \in S^*, s \in S}$ be a family of sets of many-sorted *operation symbols* over S that are pairwise disjoint.[6]

An order-sorted *algebra* $\mathcal{A}$ (over S and Ω) consists of a family $(\mathcal{A}_s)_{s \in S}$ of non-empty *carrier sets* so that $\bar{s} \leqslant s$ if and only if $A_{\bar{s}} \subseteq A_s$, and of *functions* $op_{\mathcal{A}} \colon \mathcal{A}_{s_1} \times \cdots \times A_{s_k} \to \mathcal{A}_s$ for every operation symbol $op \in \Omega_{s_1 \cdots s_k, s}$. Elements $c \in \Omega_{\varepsilon, s}$ are called *constant symbols*, and associated with *values* $c_{\mathcal{A}} \in \mathcal{A}_s$.

Let $X = (X_s)_{s \in S}$ be an S-sorted family of *variable names* that are pairwise disjoint and disjoint with every set in Ω. Then the *terms* (with variables X over S and Ω) are the least family $\mathcal{T} = (\mathcal{T}_s)_{s \in S}$ of sets so that $x \in \mathcal{T}_s$ if $x \in X_s$, and $op\ t_1 \cdots t_k \in \mathcal{T}_s$ if $op \in \Omega_{s_1 \cdots s_k, s}$ and $t_i \in \mathcal{T}_{s_i}$ for $1 \leqslant i \leqslant k, k \geqslant 0$.

[6] **Bert**: *Order-sortedness probably requires another property here.*

The terms $\mathcal{T}$ induce a *free term algebra* where every operation symbol is interpreted as a term-constructing function: $op_{\mathcal{T}}(t_1, \ldots, t_k) = op\ t_1 \cdots t_k$ for all $op \in \Omega_{s_1 \cdots s_k, s}$ where $t_i \in \mathcal{T}_{s_i}$ for $1 \leqslant i \leqslant k, k \geqslant 0$.

Assumption 1 (Semantic Basis). For our case study, we assume that the many-sorted algebra contains the following sorts, carrier sets, and operations:

1. The sort Nat has the carrier set $\mathcal{A}_{\mathsf{Nat}} = \mathbb{N}$ and terms t^n (without variables) such that $t^n_{\mathcal{A}} = n$ for all $n \in \mathbb{N}$.[7]
2. The sort String has the carrier set $\mathcal{A}_{\mathsf{String}} = C^*$, i.e., words over a set C of characters; it has a concatenation operator $+\!+ \in \Omega_{\mathsf{String}\,\mathsf{String},\mathsf{String}}$.
3. The sorts $\mathsf{C}, \ldots$ are used as node labels without attributes, where $\mathcal{A}_x = \{x\}$.
4. The sort M represents method signatures, with the carrier set $\mathcal{A}_{\mathsf{M}} = \{\mathsf{M}\ s \mid s \in \mathcal{A}_{\mathsf{String}}\}$.
5. The abstract sort N is a supersort of C, V, M, and P, with a carrier set $\mathcal{A}_{\mathsf{N}}$ that is the (disjoint) union of the carrier sets of its subsorts.
6. The sorts $\mathsf{i}, \mathsf{m}, \ldots$ are used as edge labels with numbers as attributes, where $\mathcal{A}_x = \{x\ n \mid n \in \mathbb{N}\}$.[8]

We assume that the labels of $\mathcal{G}_\Sigma$ defined in Def. 1 are the carrier sets of an algebra, and define term-labeled graphs and attribute evaluation.

Definition 4 (Attribute Evaluation). $\mathcal{G}_\mathcal{T}$ denotes the class of *graphs* that are labeled with terms $\mathcal{T}$.

An *assignment* $\alpha\colon X \to \mathcal{A}$ is a family of mappings $\alpha_s\colon X_s \to \mathcal{A}_s$ that extends to terms by defining $\alpha(op\,(t_1 \cdots t_k)) = op_{\mathcal{A}}(\alpha(t_1) \cdots \alpha(t_k))$ for all $op \in \Omega_{s_1 \cdots s_k, s}$ and $t_i \in \mathcal{T}_{s_i}$ for $1 \leqslant i \leqslant k$.

The *evaluation* of a graph $G \in \mathcal{G}_\mathcal{T}$ according to an assignment α is the graph $G^\alpha = \langle \dot{G}, \bar{G}, s_G, t_G, \alpha \circ \dot{\ell}_G, \alpha \circ \bar{\ell}_G \rangle$ that is obtained by applying α to all terms labeling the items of G.

Example 4 (Attribute Evaluation). Fig. 2 shows how a term-labeled graph from the *Encapsulate-Field* refactoring [32] is evaluated. The term-labeled graph on

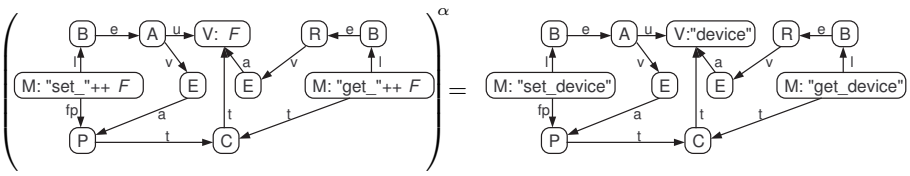

Fig. 2. A graph and its evaluation

the left hand side contains a string variable F and string terms that concatenate the names of getter and setter methods. This graph is evaluated according to an assignment α mapping the variable $\mathsf{F} \in X_{\mathsf{String}}$ onto the string "device".

[7] The *interpretation* $t_\mathcal{A}$ of a term t without variables is defined by $(op\ t_1 \cdots t_k)_\mathcal{A} = op_\mathcal{A}(t_{1,\mathcal{A}}, \ldots, t_{k,\mathcal{A}})$.

[8] **Bert**: *The description of the labels used in the running example has to be completed.*

3.2 Expansion

Graphs are expanded by replacing variable nodes and their incident edges by graphs. Variable nodes are labeled with graph variables X_n of distinguished *nonterminal sorts* $n \in \mathcal{N} \subseteq S$ that has no operation symbols (i.e., $\Omega_{w,n} = \emptyset$ for all $w \in S^*$) so that $\mathcal{T}_n = X_n$.[9] The boxes of variables nodes are drawn with sharp corners whereas those of constant nodes are rounded.

The expansion of variable nodes is specified by star replacement, a generalization of node replacement [11] and hyperedge replacement [14, 4].

Definition 5 (Star Replacement). In a graph $H \in \mathcal{G}_\mathcal{T}$, a node $v \in \dot{H}$ is a *variable node* if $\ell_H(v) = x \in X_n$ with $n \in \mathcal{N}$. The subgraph $G \subseteq H$ that contains such a variable node v as a *center*, all edges incident with v as *arms*, and all nodes adjacent to v as *border nodes*, is called an *x-star*, or just a *variable star*. (We assume that all arms have their source in the center v and do not loop.)

A rule $r = L/R$ specifies a *replacement* for a graph variable $x \in X_n$ if L is an x-star, and the interface $I = \dot{L} \cap \dot{R}$ consists of the border nodes of L, which have the same labels and cardinalities in L and in R.

If there is a transformation step $G \Rightarrow_r H$ for graphs $G, H \in \mathcal{G}_\mathcal{T}$ via r, we say that r *expands* G to H, and write $G \Rightarrow_{v,r} H$ if $v \in \dot{G}$ is the variable node removed by the transformation step.

Star replacement steps of different variable nodes are parallel-independent, as defined in [8] so that they can be performed in any order.

Lemma 1 (Star Replacement Commutes [5]). *For $G, H, H' \in \mathcal{G}_\mathcal{T}$ and star rules r and r':*

$$\textit{If } H' \underset{v',r'}{\Longleftarrow} G \underset{v,r}{\Longrightarrow} H \textit{ with } v \neq v', \textit{ then } H' \underset{r}{\Longrightarrow} K \underset{r'}{\Longleftarrow} H \textit{ for some } K \in \mathcal{G}_\mathcal{T}$$

Example 5 (Uniform Star Replacement). The star rule r in Fig. 3 expands the variable nodes a and b in Fig. 4 in two steps. The variable nodes and the star rule determine the result of the expansion steps uniquely (up to isomorphism). However, this does not hold for every star replacement step: If both arms of the variable nodes carried the same label, say 1 (both in the rule and in the graph), the rule has two different matches. Then star replacement could yield the same result as before, but also a graph with counter-parallel instead of parallel edges. To avoid such a situation, we require that the arms of variable nodes carry different labels.

Expansion shall do star replacement in a uniform way: all nodes labeled with the same variable $x \in X_\circledast$ shall be transformed via the same star rule. In general, these nodes (unlike a and b in Fig. 4) can have different numbers of incident edges, also with different labels. We must thus restrict the form of x-stars so that a single star rule applies to all of them.

[9] The carrier set for $\circledast$ is never used since graph variables will never be evaluated like attributes, but replaced, together with their nodes, before a rule is evaluated.

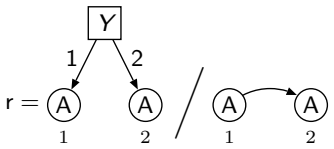

Fig. 3. An expansion rule

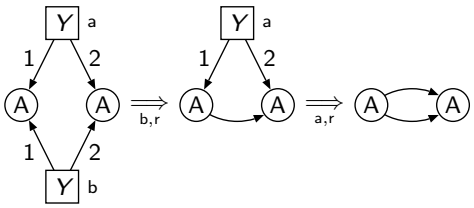

Fig. 4. Expansion steps of a cloned variable

Definition 6 (Untangledness, Uniformity). A star G is *untangled* if its edge labeling function $\bar{\ell}_G$ is injective. A pair of untangled x-stars G, H (where $x \in X_n$ and $n \in \mathcal{N}$) is *uniform* if there is a bijection $\bar{m}\colon \bar{G} \to \bar{H}$ such that $\bar{\ell}_H \circ \bar{m} = \bar{\ell}_G$ and $\dot{\ell}_H \circ s_H \circ \bar{m} = \dot{\ell}_G \circ s_G$.[10]

A graph G is *untangled* if it contains only untangled stars; it is *uniform* if all pairs of x-stars in G are uniform, for all variables $x \in X_n$ with $n \in \mathcal{N}$. In the following, we will silently assume that all graphs are untangled and uniform, and denote the class of such graphs as $\mathcal{G}^{\circledast}_{\mathcal{I}}$.

Fact 1. *For untangled graphs $G \in \mathcal{G}^{\circledast}_{\mathcal{I}}$, the node v and star rule r used in star replacements $G \Rightarrow_{v,r} H$ determine H uniquely up to isomorphism.*

Proof. Let $G \Rightarrow_{v,r} H$ for some star rule $r = L/R$. Then there must be a morphism $m\colon L \to G$ mapping the center of r's left hand side, say $\bar{v}$, to v because the border nodes in L are constant, and m preserves labels. Since $\bar{v}$ is deleted by the replacement step, every edge in G with source v must be in $\bar{m}(\bar{L})$; otherwise v would violate the dangling condition stated in Def. 2. Since v is untangled, all edges incident with v have different labels. Since $\bar{m}$ preserves labels, $\bar{v}$ is untangled as well so that $\bar{m}$ is uniquely defined. Then $\dot{m}$ is uniquely defined for the border nodes of L since the targets of all arms in L are distinct. □

Substitutions are sets of star rules that should apply to the variable nodes occurring in a graph.

Definition 7 (Substitution). A set γ of variable replacement rules is a *substitution* if it contains at most one rule for every $x \in X_n$ with $n \in \mathcal{N}$, and if no right hand side in γ contains variable nodes. A substitution γ *fits* a graph $G \in \mathcal{G}^{\circledast}_{\mathcal{I}}$ if every star in G is uniform with the left hand side of a rule in γ.

Expansion of all graph variables applies the rules defined in a substitution in arbitrary order.

Definition 8 (Expansion). Consider a graph $G \in \mathcal{G}^{\circledast}_{\mathcal{I}}$ and a substitution γ fitting G.

[10] **Niels**: *The border nodes may have different labels.*

The graph H is a γ-*expansion* of G if there is a sequence of expansion steps

$$G = G_0 \Rightarrow_{\tilde{r}_1} \cdots \Rightarrow_{\tilde{r}_n} G_n \cong H$$

where $\tilde{r}_i = L_i^{\alpha_i}/R_i^{\alpha_i}$ with $r_i = L_i/R_i \in Q(\gamma)$, and the assignments $\alpha_i \colon X \to \mathcal{T}$ assign terms only to the variables in L_i, for $1 \leqslant i \leqslant n$ so that no expansion rule in γ matches H.

Fact 2. *For some $G \in \mathcal{G}_{\mathcal{T}}^{\circledast}$ and some substitution γ fitting G:*

1. *The γ-expansion H is determined uniquely up to isomorphism.*
2. *H does not contain variable nodes.*

Proof. 1. This follows from Facts 1 and Lemma 1.

2. If γ fits G, every star in G can be transformed by a star rule in γ. Since the right hand sides of γ do not contain graph variables, G^γ does contain no graph variable any more. □

From now on, for a fitting substitution γ, G^γ shall denote the unique γ-expansion of a graph $G \in \mathcal{G}_{\mathcal{T}}^{\circledast}$.

The substitutions defined so far allow variable nodes to be expanded by graphs $G \in \mathcal{G}_{\mathcal{T}}^{\circledast}$ of arbitrary shape. Often, the substitution of a variable node shall be of a particular shape that is specified in a certain way, for instance by a graph grammar.

Definition 9 (Shaped Expansion). let Γ be some grammar that induces a *derivation relation* $\Rightarrow_\Gamma \subseteq \mathcal{G}_{\mathcal{T}}^{\circledast} \times \mathcal{G}_{\mathcal{T}}^{\circledast}$, with a reflexive-transitive closure denoted by $\Rightarrow_\Gamma^*$.

A substitution γ is *shaped* (according to Γ) if all star rules $L/R \in \gamma$ satisfy $L \Rightarrow_\Gamma^* R$.

The expansion G^γ of some graph $G \in \mathcal{G}_{\mathcal{T}}^{\circledast}$ according to a shaped substation γ is called a *shaped expansion*.

Star grammars [5] are suited to define shaped substitutions in the following way: Given a function sort$\colon \mathcal{A} \to S$ that replaces values $a \in \mathcal{A}_s$ by their least sort $s \in S$, i.e., such that there is no other sort $\bar{s} \in S$ with $\bar{s} \lhd s$ and $a \in \mathcal{A}_{\bar{s}}$. The *sort graph* sort$(G)$ is obtained from a graph G over $\mathcal{A}$ or $\mathcal{T}$ by applying sort to all its labels.

A star grammar Γ^* is defined over graphs $\mathcal{G}_S$, with star rules where the left hand side is a nonterminal $n \in \mathcal{N}$.[11] A rule L/R in a substitution γ is *shaped* according to Γ^* if sort$(G) \Rightarrow_{\Gamma^*}^*$ sort(H). The variables $\mathsf{A_i}$ used in Figure 15 are shaped according to the nonterminal *Body* of the star grammar defining program graphs that is described in [30].

[11] The symbols labeling the graphs are denoted by Σ instead of S in [5].

3.3 Cloning

Often, a certain subgraph may appear repeatedly in a graph, or may be missing altogether. In PROGRES [28], for instance, it can be specified that a node may appear repeatedly, by drawing its circle or box with a "shade". A so-called "set node" v is a placeholder for $n \geqslant 0$ nodes $v_1, \ldots, v_n$ that have the same label as v, and the same edges to the adjacent nodes of v, see Fig. 5.[12]

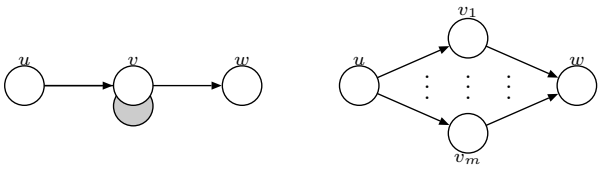

Fig. 5. Multiple node

We call v a multiple node, and $v_1, \ldots, v_n$ its clones. Note that the cloning of nodes implies that their incident edges are cloned as well. In contrast to PROGRES, we allow that two multiple nodes u and v are adjacent and have m and n clones, resp., $m \cdot n$ copies of the edges will be made between their clones, as shown in Fig. 6.

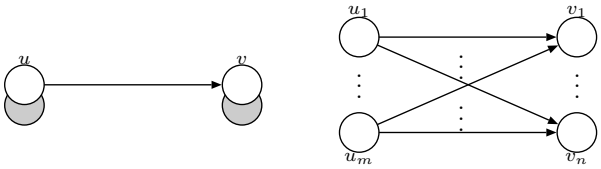

Fig. 6. Adjacent multiple nodes

Furthermore, we allow to specify that an entire subgraph can occur repeatedly. We introduce cardinality variables that are placeholders for arbitrary cardinality numbers $n \geqslant 0$, and write these variables into the shade of set nodes. A multiple node annotated with the cardinality variable x is then called "x-fold". Multiple nodes that are annotated with the same cardinality variable x belong to the same multiple subgraph, or "x-fold subgraph", which is the graph induced by all multiple nodes annotated with x and by their adjacent nodes, which are called border nodes. Fig. 7 shows a multiple subgraph induced by the

[12] In PROGRES, nodes may also be optional, i.e., have 0 or 1 clone, or may be required to have $n \geqslant 1$ clones.

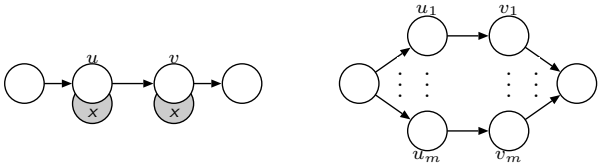

Fig. 7. Multiple subgraphs

x-fold nodes u and v. As with single multiple nodes (which correspond to multiple nodes annotated with mutually different cardinality variables), the incident edges of multiple nodes have as many clones as the nodes themselves.

Items of graphs can be labeled with attribute terms. If attribute variable names occur in labels of multiple nodes, like U and V, and z in the graph of Fig. 8, simple cloning just copies these variable names so that they are placeholders for equal attribute values in the cloned graph. This is not the only way in which

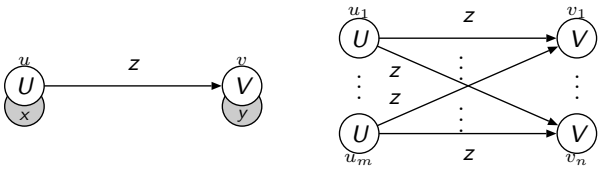

Fig. 8. Multiple attribute variables

one may want to clone variables: There are situations where the clones shall be different variable names (of the same sort) so that they may be placeholders for different attribute values. To specify this, each of the $n \geqslant 0$ clones of an x-fold subgraph gets an actual clone number $0 \leqslant i \leqslant n$ that is denoted by an actual clone variable of the form $!x$. (The cardinality variable x itself is a placeholder for the total number n of clones.) By indexing attribute variables with these actual clone variables, we may obtain clones with individual variable names. In Fig. 9, $\mathsf{U}_{!x}$ and $\mathsf{V}_{!y}$ denote individual attribute variables of the x-fold node u and the y-fold node v, respectively. Their clones are labeled with indexed variables U_i for $1 \leqslant i \leqslant m$, and V_j for $1 \leqslant j \leqslant n$. In order to "individualize" the variable z on the edge between these multiple nodes, we must index it with the actual clone variables $!\mathsf{x}$ and $!\mathsf{y}$ of both nodes, in order to distinguish all the $m \cdot n$ clones of the attribute variable. If we used the individual attribute variables $\mathsf{z}_{!x}$ or $\mathsf{z}_{!y}$ as an edge label, either all outgoing edges of the nodes u_i would carry the same label z_i $(1 \leqslant i \leqslant m)$, or all ingoing edges of the nodes v_j would carry the same label z_j $(1 \leqslant j \leqslant n)$.

We may also want to clone the border nodes of variable nodes as in Fig. 10. If the edge incident with the arm of such a border node is labeled by some

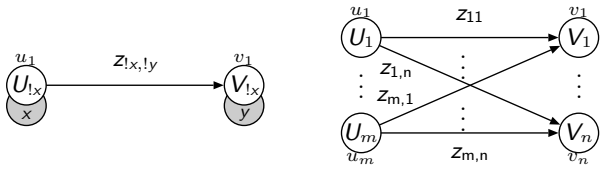

Fig. 9. Individual multiple attribute variables

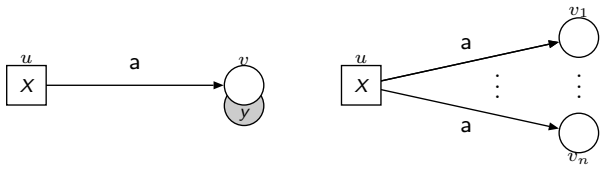

Fig. 10. Variable nodes with multiple arms

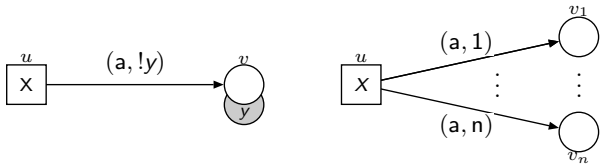

Fig. 11. Variable nodes with untangled multiple arms

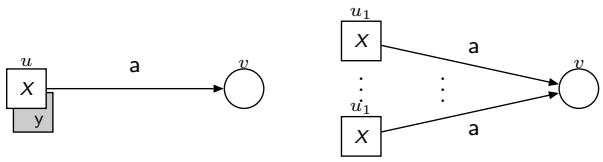

Fig. 12. Multiple variable nodes

constant, the variable node will be tangled in the cloned graph as soon as more than one clone is made. Again, we may use actual clone variables in order to preserve untangledness, as shown in Fig. 11. Here we label the arm of the multiple border node with a pair consisting of the "immutable" label a and the actual clone variable !x. Then the clones of the edges carry different labels (a, i) for $1 \leqslant i \leqslant n$, and the variable node is untangled as it was before.

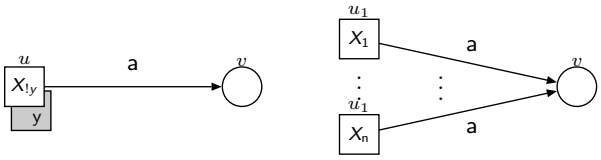

Fig. 13. Individual multiple variable nodes

Finally, variable nodes may themselves be multiple, as shown in Fig. 12. Simple cloning as in this figure yields clones with uniform graph variable names that are placeholders for isomorphic subgraphs, to be determined by expansion. As in the case of attribute variables, on may want to have different variable names in the clones on the multiple variable node. This can again be achieved by using the actual clone variable !x to index the name of the variable node, see Fig. 13. Multiple variables with multiple arms are also possible (and useful!), as can be seen in Fig. 14. [13]

We now define cloning precisely. We assume that the set S of sorts contains a sort Nat with the carrier set $\mathcal{A}_{\mathsf{Nat}} = \mathbb{N}$ and number terms $t^n \in \mathcal{T}_{\mathsf{Nat}}$ denoting the numbers $t^n_{\mathcal{A}} = n \in \mathbb{N}$, for $n \geqslant 0$.[14] The variables of sort Nat will be used to annotate multiple nodes.

Definition 10 (Cardinalities). A pair $\langle G, \otimes \rangle$, consisting of a graph $G \in \mathcal{G}^{\circledast}_{\mathcal{T}}$ and a partial *cardinality function* $\otimes\colon \dot{G} \dashrightarrow X_{\mathsf{N}at}$ is called a *template*. We shall often denote such a template just by its graph component G, and refer to its cardinality function as $\otimes_G$. A node $v \in \dot{G}$ with $\otimes_G(v) = x \in X_{\mathsf{N}at}$ is called *x-fold*, or just *multiple*; otherwise, if $\otimes_G(v)$ is undefined, v is called *singular*. In figures, an x-fold node v is drawn by inscribing x to a circle or box that is a "shade" of the circle or box depicting v. We denote the class of templates as $\mathcal{G}^{\circledast\otimes}_{\mathcal{T}}$. (Note that the templates with totally undefined cardinality functions coincide with $\mathcal{G}^{\circledast}_{\mathcal{T}}$.)

[13] **All**: *Long live the power user of graph transformation! Only he will be patient enough to read this completely!*

[14] In examples, we denote t^n just by n, i.e., the number, written in sans serif font. In finite operator domains, t^a is usually represented by a term of the form "$\mathsf{s}^n\ \mathsf{0}$", where the constant $\mathsf{0} \in \Omega_{\varepsilon,\mathsf{Nat}}$ represents the number zero, and the unary function $\mathsf{s} \in \Omega_{\mathsf{Nat},\mathsf{Nat}}$ is the *successor operation*.

Definition 11 (Multiple Subgraph). Consider a template $\langle G, \otimes \rangle \in \mathcal{G}_{\mathcal{T}}^{\circledast\otimes}$ with a cardinality variable $x \in \otimes(\dot{G})$.

The *x-fold subgraph* $G^x \subseteq G$ is induced by the x-fold nodes in $\dot{G}$ and their adjacent nodes. The nodes of $v \in \dot{G}^x$ with $\otimes(v) \neq x$ are called *border nodes* of G^x; all other items of G^x are called *internal*.

Assumption 2 (Variables in Templates and Graphs). In the graph G underlying $\langle G, \otimes \rangle$, a cardinality variable $y \in \otimes(\dot{G})$ may occur in two forms: Its plain name y is a placeholder for the *overall cardinality*, i.e., the natural number of clones that will be made of G^y; it may occur in every term used as a label in G. In the terms labeling internal items of a multiple subgraph G^y, an *actual clone variable* of the form $!y \in X_{\mathsf{Nat}}$ may occur, as a placeholder for the number of the actual copy of G^y. Furthermore, the set X_s shall contain, besides plain names like x to be used as *uniform variables*, names of the form $x_{!y}$ and $x_{!y,!z}$ as *individual variables* for some actual clone variables $!y, !z \in X_{\mathsf{N}at}$ and for all sorts s in S. Like the actual clone variables themselves, individual variables may only be used in the terms labeling internal nodes or edges of multiple subgraphs G^y or G^z. In particular, individual variables of the form $x_{!y,!z}$ may only occur in terms labeling edges that connect y-fold and z-fold nodes, as illustrated in Fig. 9, because such edges are the only internal items that may occur in the intersection of two multiple subgraphs. In a graph $G \in \mathcal{G}_{\mathcal{T}}^{\circledast}$, actual clone variables will be substituted by terms representing numbers, so that the individual variables are substituted as well, yielding *indexed variables* of the form x_{t^i} or x_{t^i,t^j} for cardinality terms $t^i, t^j \in \mathcal{T}_{\mathsf{Nat}}$. (Fig. 8 and Fig. 10–12 show uniform attribute variables U, V, z and graph variables X, respectively. Fig. 9 and Fig. 13 show individual variables in the templates on the left hand side, and indexed variables in the graphs on the right hand side, respectively.)

To avoid confusion by the use of different forms of variable names in a graph G, we assume that a plain name x either occurs only as a uniform variable, or only as the base name of an individual variable, or only in indexed variables.

Definition 12 (Clone). For some $i \geqslant 0$, the *i-th clone* $G^{x,i}$ of a multiple subgraph G^x is obtained by replacing all occurrences of $!x$ in all labels of G^x by $t^i \in \mathcal{T}_{\mathsf{Nat}}$.

The *k-fold x-clone* $G\frac{x}{k}$ of G is obtained by gluing G with $k \geqslant 0$ clones $G^{x,1}$ to $G^{x,k}$ in the corresponding border nodes of G^x, removing the inner items of G^x, and by substituting all occurrences of the cardinality variable x by the term $t^k \in \mathcal{T}_{\mathsf{Nat}}$, in all terms labeling the resulting graph.

Lemma 2 (Cloning is Commutative [20]). *For some $G \in \mathcal{G}_{\mathcal{T}}^{\circledast\otimes}$ containing cardinality variables $x, y \in X_{\otimes}$ with $x \neq y$, $G\frac{x}{k}\frac{y}{m} \cong G\frac{y}{m}\frac{x}{k}$ for arbitrary numbers $k, m \geqslant 0$.*

Sketch of Proof. If G^x and G^y overlap in border nodes (at most), their cloning, seen as graph transformation steps, are parallel-independent so that there are commuting cloning steps joining $G\frac{x}{k}$ and $G\frac{y}{m}$ to a common template $(G_x^k)_y^m = (G_y^m)_x^k$.

Otherwise, G^x and G^y overlap in edges between x-fold and y-fold nodes. ... □

Lemma 2 allows to define a cloning operation.

Definition 13 (Cloning). Consider a graph $G \in \mathcal{G}_{\mathcal{I}}^{\circledast\otimes}$ containing cardinality variables $\{x_1, \ldots, x_n\} \subseteq X_\otimes$, and a *multiplicity assignment* $\mu\colon X_\otimes \to \mathbb{N}$.

Then the *μ-clone* of G is the graph G^μ obtained by a cloning sequence

$$G\tfrac{x_1}{\mu(x_1)} \cdots \tfrac{x_n}{\mu(x_n)}$$

Definition 14 (Boundness). All occurrences of an actual clone variable $!x \in X_\otimes$ in labels of the x-fold subgraph G^x of a graph G are *bound*.

Fact 3. *If all occurrences of individual variables in a graph $G \in \mathcal{G}_{\mathcal{I}}^{\circledast\otimes}$ are bound, then every clone G^μ contains no individual variables.*

Proof. Every occurrence of a cardinality index variable $!x$ within an x-fold subgraph G^x of G is replaced by in index. Since all individual graph or attribute variables of the form $y_{!x}$ are required to occur in G^x, all of them are replaced by indexed variables of the form y_{i}. □

4 Adaptive Graph Transformation

Now, knowing how graphs can be adapted, we define the transformation of graphs by rules that are derived by adaptation, and describe how transformations can be constructed in practice.

4.1 Rule Derivation

Adaptive rules shall be cloned, then expanded, and finally be evaluated. They must thus satisfy conditions guaranteeing that all variables can really be instantiated.

Definition 15 (Adaptive Rule). A rule $r = L/R$ with $L, R \in \mathcal{G}_{\mathcal{I}}^{\circledast\otimes}$ is *adaptive* if the following conditions hold:

1. Every interface node $v \in I$ has the same cardinality in L and R.
2. No interface node is a variable node in L or R.
3. All clones L^μ and R^μ of L and R under some multiplicity function μ are untangled.

Condition 3 is necessary since cloning does not always preserve untangledness, as can be seen in Figs. 10, 11.

Definition 16 (Rule Derivation). Let $r = L/R$ be an adaptive rule, $\mu\colon X_\otimes \to \mathbb{N}$ a multiplicity function, γ a substitution fitting L^μ and R^μ, and $\alpha\colon X \to \mathcal{A}$ an assignment. Then $r' = ((L^\mu)^\gamma)^\alpha / ((R^\mu)^\gamma)^\alpha$ is a *derived rule* of r the interface I' of which contains the nodes of I and their clones. $\mathcal{D}_r$ denotes the set of derived rules of r.

We define a partial order in order to capture the property of maximal cloning and expansion in transformations.

Definition 17 (Saturated Transformation). We define a partial order $\sqsubseteq$ on multiplicity functions, expansions, graphs, rules, and matches as follows:

1. For two multiplicity functions $\mu, \tilde{\mu}$, $\mu \sqsubseteq \tilde{\mu}$ if $\mu(x) \leqslant \tilde{\mu}(x)$ for all $x \in X_{\otimes}$.
2. Two star rules $r, \tilde{r}$ satisfy $G \sqsubseteq \tilde{G}$ if the rule graph of r is included in that of $\tilde{r}$.
3. For two substitutions $\gamma, \tilde{\gamma}$, $\gamma \sqsubseteq \tilde{\gamma}$ if $\gamma(x) \sqsubseteq \tilde{\gamma}(x)$ for all $x \in X_{\circledast}$.
4. For two matches $m, \tilde{m} \colon L \to G$, $m \sqsubseteq \tilde{m}$ if $L \sqsubseteq \tilde{L}$ and m is the restriction of $\tilde{m}$ to L.

Let r be an adapted rule as above and let $G \in \mathcal{G}_{\mathcal{A}}$. Then r transforms G to a graph H, written $G \Rightarrow_{m,r,\mu,\gamma,\alpha} H$ if $G \Rightarrow_{m,r^{\mu\gamma\alpha}} H$, and there is no bigger match $\tilde{m} \sqsupseteq m$ and no bigger multiplicity $\tilde{\mu} \sqsupseteq \mu$ or substitution $\tilde{\gamma} \sqsupseteq \gamma$ so that $G \Rightarrow_{\tilde{m},r^{\tilde{\mu},\tilde{\gamma},\alpha}} H$.

Derivations of adaptive rules replace all variables, and yield rules.

Lemma 3. *Every derived rule $r' \in \mathcal{D}_r$ of an adaptive rule r is a rule according to Def. 2.*

Proof. Let r, μ, γ, and α be given as in Def. 15.

By definition of $\mathcal{G}_{\mathcal{T}}^{\circledast\otimes}$ and Fact 3, cloning with μ removes all multiple nodes from L and R, and replaces all individual variables by indexed variables. Def. 15.3 assures that L^{μ} and R^{μ} are untangled. Then expansion of L^{μ} and R^{μ} removes all graph variables (by Fact 2), and the evaluation of $(L^{\mu})^{\gamma}$ and $(R^{\mu})^{\gamma}$ according to α as in Def. 4 yields graphs in $\mathcal{G}_{\mathcal{A}}$.

Finally, $((L^{\mu})^{\gamma})^{\alpha}$ and $((R^{\mu})^{\gamma})^{\alpha}$ can be chosen so that they have common interface nodes as required, since the interface nodes I in r have the same cardinalities in L and R by Def. 15.1. □

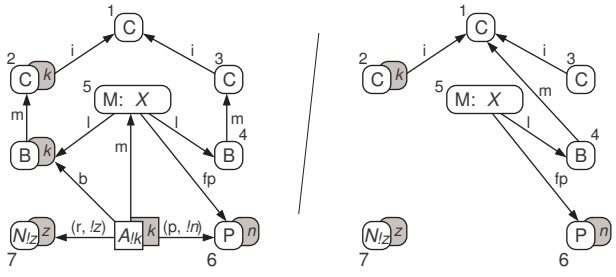

Fig. 14. The adaptive rule for the *Pull-Up-Method* refactoring

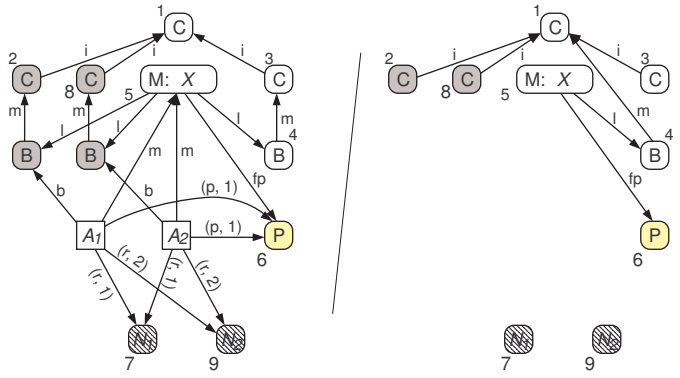

Fig. 15. The adaptive rule in Fig. 14 after cloning

Example 6 (An Adaptive Rule for Refactoring). The general *Pull-Up-Method* rule can now be described as in Figure 14. The rule applies to a class (3) with its superclass (1), and k other subclasses (2); the method signature (5) has n parameters (6), and is implemented by bodies that may refer to z names (7).

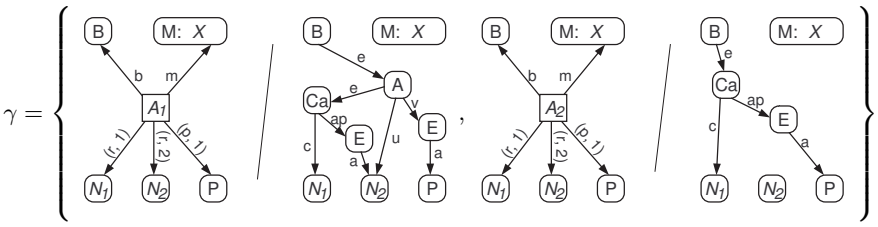

Fig. 16. A substitution γ fitting the cloned rule in Fig. 15

The k-fold variable node (labeled with $A_{!k}$) represents the bodies to be removed. Note that the number of (possibly different) bodies equals the number of removed B-nodes and the number of classes they are removed from; thus these multiple nodes have the same cardinality variable k. The node (4) is the root of the method body that will be moved to its superclass (1). No variable is needed for the body itself, because only its membership (the m-edge) is changed.[15]

The method body in the sibling classes (2) is represented by the individual graph variable $A_{!k}$ as the method bodies may be different; the variable node has arms to the signature (5), the parameters (6), and referred names (7), because these nodes may be accessed in the syntax trees of the bodies. The individual attribute variable $N_{!z}$ stand for different types of the entities referenced in the

[15] **Bert**: *Justify that this rule is complete.*

bodies. Note that the labels r and p of two arms must be paired with a clone number ($!z$) in order to preserve untangledness under cloning.

In the right hand side of the refactoring rule, the graph variables, and thus the complete syntax trees are deleted together with their root node labeled B. The other nodes remain untouched, because they are not part of the actual syntax tree and can all be used by other entities in the program.

A cloned rule of the *Pull-Up-Method* refactoring in Fig. 14 is shown in Fig. 15; it uses the multiplicity assignment μ with $\mu(\mathsf{k}) = \mu(\mathsf{z}) = 2$ and $\mu(\mathsf{n}) = 1$. Thereby the individual graph variable is replaced by two indexed variables A_1 and A_2, and the clones of the z-fold individual attribute variable $N_{!z}$ are two indexed variables N_1 and N_2.

The set γ of expansion rules in Fig. 16 fits the cloned rule in Fig. 15. Together with the assignment α with $\alpha(X) = \mathsf{get}$, $\alpha(N_1) = \mathsf{V}$, and $\alpha(N_2) = (\mathsf{M}, \text{"m2"})$, it expands this rule to the rule in Fig. 1.

4.2 Adaptation

Def. 16 does not tell how the transformation of a graph $G \in \mathcal{G}_{\mathcal{A}}$ with an adaptive rule $r = L/R$ can actually be performed. We cannot just derive $\mathcal{D}_r$, and then choose one $r' \in \mathcal{D}_r$ to match and transform G, because $\mathcal{D}_r$ is infinite in general. Rather, a goal-oriented process must construct a derivation of L that matches G, by finding the constant items, and substitutions for the variables of L in an incremental and interleaved fashion. If this succeeds, the substitution found for L is used to adapt R, and G is transformed accordingly. We discuss this process for our running example first.

Example 7 (Applying the Pull-Up-Method *Refactoring Rule).* The left hand side L f the rule shown in Fig. 14 can be matched by binding its variables to values, and constructing a morphism to a host graph G as follows:

1. Match the B-node 4. (This node will probably be indicated in G by the user of the refactoring operation.)
2. Match the outgoing m-edge of node 4. (Due to shape rules, this cannot fail.)
3. Match the target of that m-edge, the C-node 3.
4. Match the outgoing i-edge of node 3. This may fail if this node has no superclass node; then the match of L fails as well. (We may try with other matches of node 4, however.)
5. Match the target of that i-edge, the C-node 1.
6. Match the ingoing l-edge of node 4. (Due to shape rules, this cannot fail.)
7. Match the source of that edge, the node 5.
8. Bind the attribute variable X in the label of node 5 to the string constant "get".
9. Determine the set $\ddot{l} \subseteq \dot{G}$ of outgoing l-edges of node $\dot{m}(5)$.
10. Determine the set $\ddot{n} \subseteq \dot{G}$ of all nodes that match a variable of type N.[16] Define $\mu(z) = |\ddot{n}|$ and bind the indexed variables N_i to the attribute values of the corresponding node in $\ddot{n}$.

[16] **Bert**: *Clarify the meaning of the* abstract sort N.

11. If there is an i-edge with target node $\dot{m}(1)$ in G then do the following steps:
    (a) Increase $\mu(k)$ by 1 and let $j = \mu(k)$.
    (b) Glue the j-th clone $L^{k,j}$ to L.
    (c) Match the i-edge in $L^{k,j}$ to the i-edge with target 2_j in $\dot{m}(1)$ in G.
    (d) Match the source node 2_j in $L^{k,j}$.
    (e) Determine the set $\ddot{m} \subseteq \dot{G}$ of ingoing m-edges of node $\dot{m}(2_j)$.
    (f) If the node sets $\ddot{m}$ and $\ddot{l}$ have a node in common, say $b \in \dot{G}$, match the B-node to b; if the intersection is empty, the match of the clone $L^{k,j}$ fails; undo the actions 11a to 11e; goto step 12.
    (The shape rules do not allow that two method bodies with the same method signature exist within a single class so that the intersection may not contain more than one node.)
    (g) Determine the set $\ddot{p} \subseteq \dot{G}$ of outgoing fp-edges of node $\dot{m}(5)$.
    (h) Clone L^n $|\ddot{p}|$ times.
    (i) If $|\bar{p}| > 0$, match the clones 6_1 to $6_{|\bar{p}|}$ with the target nodes in $\bar{p}$.
    (j) Find a subgraph G^i of G that may have b, the nodes in $\bar{p}$, and $\ddot{n}$ as border nodes, i.e., as nodes that are incident with edges outside G^i. We include as many items in G^i as possible.
    (k) Go back to step 11.
12. Delete L^k.
13. Then, all variables in L are bound to values, and the match $m\colon L^{\mu\gamma\alpha} \to G$ is complete.
14. Remove all nodes which are derived by cloning or expanding from nodes that are not in the interface, to obtain the context graph C.
15. Construct the instance $R^{\mu\gamma\alpha}$.
16. Glue the instance $R^{\mu\gamma\alpha}$ to C, yielding the transformed graph H.

In order to derive the left hand side of the rule in Fig. 1, step 11 succeeds twice, and expands the variable nodes labeled A_1 and A_2 by the respective bodies of the methods, the inner items of which are shaded in Fig. 1.

When constructing an application for the example rule, we used several nice properties of this particular rule that made the construction easier:

1. The left hand side is not a star. If this had been the case, the rule would could be expanded to an arbitrary graph. Furthermore, if then the right hand side contained the graph variable again, applications of this rule would never terminate.
2. All graph and attribute variables on the rule's left hand side have different names (X, $A_{!k}$, and $N_{!z}$). Since all multiple nodes are labeled with individual variables this holds for every clone of the left hand side as well (not only for the clone shown in Fig. 15). Otherwise, the attribute values of different occurrences of a variable name had to be compared to each other. This property is known as left-linearity; it is not required for the cardinality variables.
3. After cloning, every graph variable A_i is adjacent to a single clone of the k-fold B-node that is already matched in the host graph. We call this node the *anchor* of these graph variables. Expansion of the variables can then

start from the anchor's matching node in the host graph. If a graph variable would have only multiple border nodes (with a multiplicity different to the variable node itself), it would be hard to find the node where the expansion should start.
4. In the cloned left hand side of the rule, the anchors of graph variables are mutually distinct. This property is called *apartness* in [20]. If two graph variables would share their anchor node, the items connected (directly or indirectly) to the match of this node could be expansions of either variable. This would lead to a combinatorial explosion of the possibilities to define the expansion of these variables.
5. The terms used as labels on the left hand side of the adaptive rule are either constant, like C, or variable, like N_z and $\mathsf{M}\ X$. An attribute variable matches every attribute value in the host graph (binding the variable to that value), while a constant label can just be compared to that value. For a more general term, like $\mathsf{M}\ X +\!\!+\ Y$, there may be many attribute assignments making the term match an actual value like M "get": Any split of "get" into two substrings gives a matching assignment for X and Y.
6. All variables of the rule's right hand side appear already on its left hand side. This property (often called *closedness* in the literature) makes the instantiation of the right hand side in step 15 unique. Otherwise, some attribute or graph or cardinality variables would not be bound by the match of the left hand side, and their values had to be "guessed" in order to construct the instance.

We require that adaptive rules satisfy the following conditions supporting the rule application process.

Definition 18 (Well-Formedness). An adaptive rule $r = L/R$ is *well-formed* if each of the following conditions are satisfied:

1. The left hand side of r is not a star.
2. The left hand side L is *linear*, i.e., every attribute or graph variable $x \in X \setminus \{X_\otimes\}$ occurs at most once in L, and in the terms labeling the multiple nodes $v \in \dot{L}$, all variables are individual (and bound, by the requirements of adaptiveness in Def. 15).
3. Every variable node v has a distinguished *anchor arm* to a border node b, called its *anchor node*, so that v and b have the same cardinality (i.e., are either both single, or have the same cardinality variable attached).
4. On the left hand side of r, the anchor nodes of all variable nodes are distinct from each other.
5. All labels occurring in the left hand side L are either variables, or constant terms; if they are tuples, each of their components is either a variable or constant.
6. The right hand side R is *closed*, i.e., it contains only variable names that occur already on the left hand side L.[17]

[17] **All**: *Should we add connectedness of the left hand side as a further well-formedness criteria?*

The adaptive rule for *Pull-Up-Method* in Fig. 14 is well-formed.
We define an algorithm for matching an adaptive rule in a host graph.

Algorithm 1 (Matching of Well-formed Adaptive Rules).

***Input*:** 1. A totally labeled *host graph* $G \in \mathcal{G}_\mathcal{A}$ over attribute values $\mathcal{A}$.
2. A well-formed and adaptive rule $r = L/R$.

***Result*:** 1. A multiplicity assignment $\mu\colon X_\otimes \to \mathbb{N}$.
2. A graph substitution γ fitting the cloned adaptive rule r^μ.
3. A family of assignments $\alpha_s\colon X_s \to \mathcal{A}$, for $s \in S \setminus \{\otimes, \circledast\}$.
4. A *match* of r in G, i.e., a morphism $m\colon L^{\mu\gamma\alpha} \to G$ that satisfies the dangling condition wrt. the derived interface inclusion $I^{\mu\gamma\alpha} \to L^{\mu\gamma\alpha}$.

(We define μ, γ, and α_s just for the variable names occurring in L, and L^μ, respectively.)

The algorithm starts with a multiplicity μ_0, a graph substitution γ_0, an assignment α_0, and a pair $m_0 = \langle \dot{m}, \bar{m} \rangle$ of mappings $\dot{m}_0\colon \dot{L} \to \dot{G}$ and $\bar{m}_0\colon \bar{L} \to \bar{G}$ that are all totally undefined; it performs the following steps:

1. The node v is a y-fold node with label l: search for a set of nodes $v_1, v_2, \ldots, v_k$ in G with the same label l and connected to n with the same edge type as in L. Then s is cloned according to the multiplicity k for y, and for every $v_1, v_2, \ldots, v_k$ the corresponding nodes in L and the connecting edges to nodes that are already in $\underline{s}$ are added to $\underline{s}$ and $\underline{m}$.
2. The node n is labeled with a graph variable x; the algorithm analyses G to determine the part of it that is the expanded version of this variable. Again the variable is expanded in s by the resulting subgraph and the items are added to $\underline{s}$ and $\underline{m}$.
3. The node is a single node n and the edges connecting it to already existing nodes in $\underline{G}$ are added to $\underline{s}$ and $\underline{m}$.
4. ...
5. ...[18]

Ideas for the algorithm:

1. The pattern (left hand side) L is a gluing of following parts:
   (a) A subgraph induced by the single and constant nodes, given as a set of connected components $K_1, \ldots, K_m$. A component K_i is *simple* if it consists of a singleton node. (For the rule in Fig. 14, this is the graph induced by the nodes 1, 3, 4, 5 consisting of a single connected component.)
   (b) A set of multiple subgraphs $L^{x_1}, \ldots, L^{x_n}$ for all cardinality variables $x_i \in X_\otimes$ occurring in L. A multiple subgraph is called *star-like* if it has a single multiple node. (In rule in Fig. 14, L^k is not star-like, and connected while L^z and L^n are star-like.)

[18] **Mark**: *We should come up with a rather sketchy algorithm that can be shown to be correct, without being optimized towards efficiency? I think this needs more time and thoughts than I want to spend on this issue at the moment.*

   (c) A set of single variable stars $S_1, \ldots, S_k$. (The rule in Fig. 14 contains no single variable star. However, every new clone L^i of L^k inserts a single constant subgraph induced by the clone of node 2 and its adjacent B-node, plus a single variable star.)
2. We may assume that a set of nodes is designated as the start points of matching. (This may be the nodes indicated by a user invoking a rule interactively, or nodes adjacent to a predicate node in a "program".)
3. The algorithm performs one of the following steps:
   (a) Match a constant single subgraph.
   (b) Try to match the expansion of a single variable node according to some star rule with lazy cloning [5]. If the star rules for the variable are tried in descending size of their right hand sides, this gives maximal expansions.
   (c) Make another clone for some multiple subgraph. Every new clone contains further constant single subgraphs or single variable nodes that can be matched according to 3a or to 3b, respectively.
4. We may be able to drop some of the WF conditions above, and also may wish to add new ones (forever, or just for convenience), like:
   (a) L is connected.
   (b) Every multiple subgraph L^i is connected.
   (c) ...
5. The order in which these steps are taken can be determined heuristically so that failure is detected as soon as possible, and that nondeterministic matching actions (yielding a set of possible matches) are done as late as possible (for instance).
6. Left linearity can be dropped if we modify matching as follows: After binding the first occurrence of a variable, all other occurrences of that variable are substituted by this binding.

Theorem 1 (Correctness of Algorithm 1).

1. *Algorithm 1 terminates.*
2. *Algorithm 1 is correct: Every result defines a saturated transformation* $G \Rightarrow_{m,r,\mu,\gamma,\alpha} H$.

Proof. ... □

- Nondeterminism and failability of adaptation [7].
- Restricting the use of variables to make matching easier [7].

5 Conclusions

The derivation of adaptive rules by cloning, expansion, and attribute evaluation makes graph transformation more expressive. This is indispensable to specify transformations in real-life applications, like a refactoring operation, with a single rule. Such a specification is still *declarative*; this is not the case if simple rules are combined using control structures.

Numerous authors have used rules with placeholders in order to make rule systems more concise. Often, this is done on the "meta level" when talking about transformation rules. Our definition of attribution follows [25]. Considering attributes as labels is much simpler than the model proposed in [9], where every graph is burdened with an infinite set of nodes that represents all values of the algebra $\mathcal{A}$, and cluttered up with edges that point from nodes to their actual attribute values. Our version of cloning extends embedding instructions of node replacement [11]; the notation is similar to the set nodes in PROGRES [27]. Other graph transformation languages, like FUJABA [12] and GREAT [21], to mention just a few, support an UML-like way of expressing multiplicities of nodes. In all these cases, only single nodes may be cloned. In the model transformation language GMORPH [29], a more general notion of cloning is provided that uses nested *collection containers* that correspond to the multiple subgraph of a cloning variable. Graph variables have been first devised for substitutive transformation [24]. Here, as in [20], expansion is more general as it allows to replace nodes with a varying number of arms.

This work can be extended in several directions. Typing according to a type graph [9] or graph schema [27] can be added to our concepts in an orthogonal way, including subtyping and cardinality constraints. Then cloning can also be specified to produce l to u copies of a subgraph, for $u \geqslant l \geqslant 0$. Some refactoring operations call for nested cloning, where subgraphs of a cloned subgraph are cloned again. Attribution could be extended to allow nondeterministic computations. Then graphs could be attribute domains, and graph transformation rules could specify operations, as in hierarchical graph transformation [18]. For practical reasons, the expansions of graph variables should be restricted to graph languages. The star grammars defined in a companion paper [5] are suitable for such a "shaped expansion". Last but not least, control is needed even for most expressive rules, if only for efficiency. We shall use transformation predicates [18] for this purpose since they do not only control *which* rule is applied, but also *where* it is applied.

We are convinced that the transformation concepts described in this paper are not only useful to *specify* refactorings, but will lead to our ultimate goal: to implement them in Dia**plan**. A restricted implementation of graph expansion exists already [6].

References

1. A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors. *1st Int'l Conference on Graph Transformation (ICGT'02)*, number 2505 in Lecture Notes in Computer Science. Springer, 2002.
2. A. Corradini, H. Ehrig, U. Montanari, and J. Padberg. The category of typed graph grammars and its adjunction with categories of derivations. In Cuny et al. [3], pages 56–74.
3. J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors. *Graph Grammars and Their Application to Computer Science*, number 1073 in Lecture Notes in Computer Science. Springer, 1996.

4. F. Drewes, A. Habel, and H.-J. Kreowski. Hyperedge replacement graph grammars. In Rozenberg [26], chapter 2, pages 95–162.
5. F. Drewes, B. Hoffmann, D. Janssens, M. Minas, and N. Van Eetvelde. Adaptive star grammars. In A. Corradini, H. Ehrig, G. Engels, L. Ribeiro, and G. Rozenberg, editors, *3rd Int'l Conference on Graph Transformation (ICGT'06)*, Lecture Notes in Computer Science. Springer, 2006. To appear.
6. F. Drewes, B. Hoffmann, R. Klein, and M. Minas. Rule-based programming with diaplan. *Electronic Notes in Theoretical Computer Science*, 127(1):15–26, 2005.
7. F. Drewes, B. Hoffmann, and M. Minas. Constructing shapely nested graph transformations. In H.-J. Kreowski and P. Knirsch, editors, *Proc. Int'l Workshop on Applied Graph Transformation (AGT'02)*, 2002. 107–118.
8. H. Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science and Biology*, number 73 in Lecture Notes in Computer Science, pages 1–69. Springer, 1979.
9. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 2006.
10. H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors. *2nd Int'l Conference on Graph Transformation (ICGT'04)*, number 3256 in Lecture Notes in Computer Science. Springer, 2004.
11. J. Engelfriet and G. Rozenberg. Node replacement graph grammars. In Rozenberg [26], chapter 1, pages 1–94.
12. T. Fischer, J. Niere, L. Turunski, and A. Zündorf. Story diagrams: A new graph grammar language based on the Unified Modelling Language and Java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Theory and Application of Graph Transformation (TAGT'98), Selected Papers*, number 1764 in Lecture Notes in Computer Science, pages 296–309. Springer, 2000.
13. M. Fowler. *Refactoring—Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, Reading, MA, 1999.
14. A. Habel. *Hyperedge Replacement: Grammars and Languages*. Number 643 in Lecture Notes in Computer Science. Springer, 1992.
15. A. Habel, J. Müller, and D. Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
16. A. Habel and D. Plump. Relabelling in graph transformation. In Corradini et al. [1], pages 135–147.
17. B. Hoffmann. Shapely hierarchical graph transformation. In *Proc. IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 30–37. IEEE Computer Press, 2001.
18. B. Hoffmann. Abstraction and control for shapely nested graph transformation. *Fundamenta Informaticae*, 58(1):39–56, 2003.
19. B. Hoffmann. Graph transformation with variables. In H.-J. Kreowski et al., editors, *Formal Methods in Software and System Modeling*, volume 3393 of *Lecture Notes in Computer Science*, pages 101–115. Springer, 2005.
20. B. Hoffmann, D. Janssens, and N. Van Eetvelde. Cloning and expanding graph transformation rules for refactoring. *Electronic Notes in Theoretical Computer Science*, 2006. Proc. Graph and Model Transformation Workshop (GraMoT'05), to appear.
21. G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the use of graph transformation in the formal specification of model interpreters. *Journal of Universal Computer Science*, 9(11):1296–1321, Nov. 2003.

22. T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour-preserving transformation. In Corradini et al. [1], pages 286–301.
23. M. Minas and B. Hoffmann. An example of cloning graph transformation rules for programming. *Electronic Notes in Theoretical Computer Science*, 2006. To appear.
24. D. Plump and A. Habel. Graph unification and matching. In Cuny et al. [3], pages 75–89.
25. D. Plump and S. Steinert. Towards graph programs for graph algorithms. In Ehrig et al. [10], pages 128–143.
26. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations*. World Scientific, Singapore, 1997.
27. A. Schürr. Programmed graph replacement systems. In Rozenberg [26], chapter 7, pages 479–546.
28. A. Schürr, A. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In G. Engels, H. Ehrig, H.-J. ¡ Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. II: Applications, Languages, and Tools*, chapter 13, pages 487–550. World Scientific, Singapore, 1999.
29. S. Sendall. Combining generative and graph transformation techniques for model transformation: An effective alliance? In *Proc. OOPSLA'03-Workshop on Generative Techniques in the Context of MDA*, 2003. URL: `www.softmetaware.com/oopsla2003/mda-workshop.html`.
30. N. Van Eetvelde and B. Hoffmann. A graph grammar for program graphs. Report, Universeiteit Antwerpen, 2006.
31. N. Van Eetvelde and D. Janssens. Extending graph rewriting for refactoring. In Ehrig et al. [10], pages 399–415.
32. N. Van Eetvelde and D. Janssens. Refactorings as graph transformations. Technical report, University of Antwerp, February 2005. UA WIS/INF 2005/04.

A Weaknesses of the Paper

This appendix discusses major weak points of the paper. These are parts that are not quite right (to avoid the word "*wrong*"), some are technically too complicated, to restricted, or hard to motivate on the one hand, or missing on the other hand so that the usefulness of the concepts is limited.

A.1 Problems

1. *Pull-Up-Method rule (Ex. 3 on page 4).*
   The transformation rule does not quite implement the refactoring described in Example 1: Since the M: "get"- node and all C-nodes belong to the interface, the rule may remove some implementations of the method from some of the sibling classes, leaving other sibling classes (with or without an implementation of "get") as they are.
   Niels proposes to apply *maximal cloning* in rule applications. To avoid maximal cloning, we could remove the M-node 5 from the interface if we add a *y*-fold R-node to the interface with a c-edge to node 5 that represents all "external" calls to *X*. However, both attempts would only make sure that all

other subclasses with a body for the method are matched; it is still allowed that some subclasses exist that have no body for the method. (I think we need a kind of negative application condition for this situation.)
And, if we use maximal cloning, we should use *maximal expansion* for graph variables as well.
Done, see Def. 17 on page 16.
By the way: When playing with this examples, I did not understand why there is an m-edge between the k-fold individual variable $A_{!k}$ and the signature node labeled with M. This edge is used to allow recursive calls to the method from its body, isn't it? Then it should be labeled with r? (Or with $(\mathsf{r}, !z+1)$, for untangledness?) Or, node 5 is already contained the the set of N-nodes, and the m-edge could be omitted. This would have the additional advantage that the rule becomes *apart* (see condition 18.4).

2. *Star replacement on term-labeled graphs (Def. 5 on page 7).*
   We have to clarify a point in star replacement:
   (a) Star rules should have border nodes that are labeled with mutually distinct variables.
   (b) Star replacement instantiates star rules by substituting the variable labels of border nodes by terms as they occur in the expanded graph.
   (c) Every occurrence of a graph variable may be replaced according to a different ("individual") substitution.
   (d) If we consider shaped expansion of graph variables, where every variable is expanded by a graph derived according to a star grammar, the non-border nodes on the right hand sides of the rules of that grammar are also instantiated according to some substitution. (This modifies star replacement according to [5]. However, this may also also help to remove hook 1 below, if we consider *order-sorted terms* instead of many-sorted terms, i.e., allow subtyping on the attributes.)

   Done.

A.2 Hooks

1. *DPO with relabeling (Def. 2 on page 3).*
   Do we actually need rules that relabel a node without removing it? It is *not* needed in Example 14 on page 16. What about the other refactorings Niels *et al.* have investigated so far?
   Removing relabeling has the following consequences
   ⊖ Less practical expressiveness for rules
   ⊕ Purely standard DPO transformation for rule instances.
   Niels believes that this is needed for renaming refactorings.
   (If these are just global renamings, it probably not worth it, however.)
   We keep relabeling.
2. *DPO with injective matches (Def. 2 on page 3).*
   The definition of identification (with quotients) is not very elegant.
   Do we need non-injective matches as the standard case? It seems that it does not matter for our Example 14 on page 16, because there may be no

identification of nodes, due to the the shape of program graphs.
Or do we have situations where we want to exclude certain identifications? In the latter case, we should stay with injective matches.

3. *Adaption of labels to more special labels (Def. 5 on page 7).*
In the expansion of graph variables, it should be possible to have different labels on corresponding border nodes of x-stars, even if this is not needed in our example.
Star rules could either be allowed to have unlabeled border nodes (they come for free if we still use DPO with relabeling), or introduce another (minor) adaption of rules: The sorts S are equipped with a *subtype relation* $\leqslant$ (a partial order) so that $s' \leqslant s$ implies $\mathcal{T}_{s'} \subseteq \mathcal{T}_s$. A graph G (and a rule r) can be *specialized* by replacing a term $t \in \mathcal{T}_s$ with a term $t' \in \mathcal{T}_{s'}$ provided that $s' \leqslant s$.
Pros and cons:
   - $\oplus$ Increases expressiveness.
   - $\ominus$ Requires some extra definitions (and space); adds yet another concept.
   - $\ominus$ May be in conflict with shaped expansion, as star grammars derive graphs with particular labels that *cannot* be specialized.

   Niels finds that this matter is not that urgent. Order-sorted term algebras do the job, if we define an "abstract sort" like N as the sum of concrete types C, $\mathsf{M}\ \mathsf{string}$ etc. See Assumption 1 on page 6.
4. *Complicated definition of terms.*
Many forms of terms exist in rules, finds Niels. I rather think that their explication is too short and bad.
   - According to Def. 4 on page 6, the graphs after attribution are labeled with values from a many-sorted algebra, and those before attribution (and after expansion) are labeled with many-sorted terms.
     (a) Labels without proper attribute values, like B, are represented by a sort of that name (B) with a unique constant operator of the same name and $X_{\mathsf{B}} = \emptyset$, so that $\mathcal{T}_{\mathsf{B}} = \{\mathsf{B}\}$ and $\mathcal{A}_{\mathsf{B}} = \{\mathsf{B}\}$: then attribute evaluation leaves them as they are.
     (b) Labels with attribute values (of type String in our examples), like V, are represented by a sort of that name (V) with a unique "constructor operation" $\mathsf{V}\colon \mathsf{String} \to \mathsf{V}$ with $X_{\mathsf{V}} = \emptyset$ so that $\mathcal{T}_{\mathsf{V}} = \{\mathsf{V}s \mid s \in \mathcal{T}_{\mathsf{String}}\}$ and $\mathcal{A}_{\mathsf{V}} = \{\mathsf{V}\ s \mid a \in \mathcal{A}_{\mathsf{String}}\}$.

     This explains the notation in Ex. 4 on page 6 (if we remove the ":" in the attributed labels like "$\mathsf{M} : \text{"}\mathsf{get}\text{"} +\!\!+ F$").
   - According to the first paragraph of Subsection 3.2 on page 7, the graph variables occurring in the graphs and rules after cloning (and before expansion) are terms of the "graph variable sort" $\circledast \in S$ without any operations so that $\mathcal{T}_{\circledast} = X_{\circledast}$. The carrier set $\mathcal{A}_{\circledast}$ is irrelevant since graph variables do not survive the expansion.
     (If we use shaped expansion, we have a family $(\circledast_n)_{n \in N}$ of graph variable sorts for all nonterminals of the star grammar.)
   - According to Def. 10 on page 13, labels of multiple nodes are pairs (t, x) where t is the "base label" and $x \in X_{\otimes}$ a cardinality variable. This fits

into the model if we say that we have a *multiple sort* $s^{\otimes}$ for every base sort s so that the terms of multiple sorts are pairs: $\mathcal{T}_{s^{\otimes}} = \{(t, x) \mid t \in \mathcal{T}_s, x \in X_{\otimes}\}$. (Maybe we should have denoted these terms by something like "$t{\otimes}x$".) Again, the carrier sets $\mathcal{A}_{s^{\otimes}}$ are useless because the terms $\mathcal{T}_{s^{\otimes}}$ do not survive cloning.

Terms of sort $\otimes$ *do* survive cloning, however, since index variables $!x$ of that sort may be used to individualize variables (which then do not survive expansion or attribute evaluation) or to untangle edge labels. That is why the carrier sort $\mathcal{A}_{\otimes} = \mathbb{N}$ is needed.

With a better explanation, and slightly different notation in figures, this should be not too complicated.

Niels proposes to model multiple nodes with a partial cardinality function on nodes, as in the Tallinn paper.

Bert has done this; cardinality variables are now of sort Nat so that they (or rather: their substitutions determined by cloning) can be used in attribute computations.

5. *Untangledness is difficult (Def. 6 on page 8).*
   However, I find expansion of tangled graphs difficult to understand (if the reader is able to see that expansion is no longer unique, that is). I propose to keep this notion, even if it requires a motivating example (Ex. 5 on page 7) and sophisticated use of indexed cardinality variables to untangle edge labels during cloning.
   Niels agrees to keep untangledness.
6. *The form of variables is difficult to understand (Ass. 2 on page 14).*
   Unfortunately I see no way to avoid these different kinds of variables:
   - We certainly need individually cloned variable names, if seldom for graphs (like in our example), so more often for attribute variables (as in our example).
   - If we need untangledness, we also need indexed cardinality variables of the form "$!x$" to preserve untangledness under cloning.
   - The idea was that these variables are "just" attribute variables that can be used to compute with their values. (This was mentioned explicitly in an early draft, but canceled later because it is not used in our example, and also not easily explained.)

   Niels sees no simpler way to define variables – all of them are needed.
   Bert has introduced a two page discussion with 9 figures that should motivate individual and indexed variables and the actual clone variables.
7. *Boundness is badly explained (Def. 14 on page 15).*
   I think is now better placed, directly before the fact stating its purpose. Boundness, as well as untangledness und uniformity, are conditions guaranteeing that adaption works as intended.
   We might try to find better explanations for it.
   Niels thinks it is OK now.
8. *Well-formedness conditions for adaptive rules are not well motivated (Def. 15 on page 15).*
   Conditions 18.1, 18.2, and 18.6 correspond to conditions on term rewrite

rules. Reference to that fact should be motivation enough. Conditions 18.3, 18.4, and 18.5 should be made consistent with the algorithm and its correctness proof.
In progress. See Alg. 1 on page 21 and Thm. 1 on page 22.

A.3 Holes

The following concepts are missing; some of them have been removed to reduce space and/or technical complexity:

1. *Proofs of facts.*
   I will prepare proof sketches for the facts. If they seem to be useful for every reader, they go into the paper; otherwise they go into the appendix (to convince the reviewers).
   Done (up to lemma 2 on page 14).
2. *Construction of transformation steps (before Def. 18 on page 20).*
   This has to be added. I propose to sketch the "primitive actions" of such a construction:
   - Match a constant node and the edges connecting it to constant nodes.
   - Match one more clone of a multiple subgraph (or of its constant subgraph).
   - Expand one graph variable according to a star rule.
   - Match an attribute variable to an attribute value.

   Then I would talk informally about strategies how to order primitive steps. I hope this is sufficient to convince the reader that this gives an effective procedure.
   Maybe We could also discuss the degree of nondeterminism involved in the procedure.
3. *Type morphisms (Def. 1 on page 2).*
   In the present version, types occur only implicitly, by the many-sortedness of the labels (terms and attribute values, resp.). Typed graphs as in [2] are "standard".
   Type morphisms have the following pros and cons:
   - ⊕ A standard definition.
   - ⊖ Require some extra definitions (and space).
   - ⊖ Are not really the typing one wants: no overloaded labels (nodes/edges in the type graph).
   - ⊖ Are not really needed for this paper.
   - ⊖ When considering shaped graphs (star grammar languages), a more flexible typing can be induced by the star grammar, by the way how edges and node labels are used in the rules.

   Niels would like to have type morphisms, but agrees that they are not urgently needed.
   Bert proposes to assume a function type: $S \to 2^{S \times S}$ and to require that for every edge e with $\bar{\ell}(e) \in \mathcal{T}_s$, $(\dot{\ell}(s(e)), \dot{\ell}(t(e)) \in \text{type}(s)$. (This also allows "overloaded" edge sorts.) The type function can be deduced from the shape rules, however.

4. *Shaped expansion (Def. 5 on page 7).*
   If we refine expansion to replace variables by graphs derived from a star grammar, we clarify the relation of this paper to the ICGT-paper. This also helps to construct transformation steps (see next point). However, it requires some more space, and introduces some more notions (star grammars, their language, ...
   One could even define the transformation of languages à la [17] (where CF hypergraph languages were used): The graphs being transformed belong to a star language, and the graphs L and R of a shaped rule $r = L/R$ are are sentential forms of a nonterminal (after replacing graph variables by the nonterminal defining their possible expansions). Then one can easily show that this kind of transformations preserves language membership.
   However, this makes only sense if all refactorings you (Niels) have considered are of such a form, or can be brought into such a form.

A.4 Tasks for Niels

1. Give a stable definition of the program graph grammar (PGG).
2. Give a stable definition of the refactoring rules, as adaptive graph transformation rules.
3. verify that the star language of program graphs is closed under the refactoring rules.
4. Check the following questions for the refactoring rules and the PGG:
   (a) Can all refactorings be defined with adaptive rules?
   (b) Are all refactoring rules correct if one assumes *maximal cloning and expansion*? (I.e., "saturated transformation"?)
   (c) Is there a refactoring rule needing relabeling of interface nodes?
   (d) Which refactoring needs "subtyping" of labels?
   (e) Are all expansions of graph variables shaped according to Def. 9 on page 9?
   (f) Are the let and right hand sides of refactoring rules sentential forms of the PGG?
   (g) Are all nonterminals of the PGG anchored? Are all star rules of the PGG anchored? See Def. 18 on page 20.

A.5 Tasks for Bert

1. Prove lemma 2 on page 14.
2. Define "shapely adaptive rules".
3. Complete definition of Algorithm 1 on page 21 and prove Theorem. 1 on page 22.

(Version 1.09 of July 6, 2006)