Shaped Generic Graph Transformation

Frank Drewes[1], Berthold Hoffmann[2], Dirk Janssens[3],
Mark Minas[4], and Niels Van Eetvelde[3]

[1] Umeå universitet, Sweden
[2] Universität Bremen, Germany
[3] Universiteit Antwerpen, Belgium
[4] Universität der Bundeswehr München, Germany

Abstract. Since the systematic evolution of graph-like program models has become important in software engineering, graph transformation has gained much attention in this area. For specifying model evolution concisely, graph transformation rules should be as expressive as possible. The generic rules proposed in this paper may contain placeholders for graphs of varying number and shape. Expansion of these placeholders by graphs yields the actual transformation rules to be applied. Even rather complex transformations occurring in real-life applications, such as the *Pull-Up-Method* refactoring operation, can be specified by a single generic rule.

1 Introduction

The systematic transformation of models and programs has become an important issue in the world of software engineering. On the one hand, the general idea of model-driven engineering has attracted a lot of attention from both the academic and the industrial communities, and on the other hand the need for better support of software evolution has become clear. In the model-driven approach, a software system is seen as a cluster of models, on various levels of abstraction and with various characteristics. Each of these models captures certain features or aspects of the systems, allows its own kind of analysis, and has its own tools available. In this way one may apply the many sophisticated tools and theories that have been developed for particular models by the research community. It is clear, however, that this will not work unless one develops powerful tools for integrating the various models, transforming them into one another, generating code from them, and keeping them consistent. Thus model transformation is a key issue here. In the area of software evolution, a lot of attention has been devoted to refactoring: the stepwise modification of programs, aimed at improving their internal organization, but preserving their behavior. The list of refactoring operations published by Fowler [12] is a well-known example. In order to get to a precise and manageable definition of what constitutes a model (or program) transformation, it is quite natural to view a model or program as a graph, and to describe large transformation processes as being compositions of "small" transformations – and thus, to describe model transformations by graph transformation systems.

Unfortunately, the rules of classical graph transformation formalisms are rather restricted. E.g., double pushout (DPO) rules [10] 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 using control structures. In this paper, which continues [17], we pursue another idea: we make rules *generic* by introducing *(i)* multiple nodes that represent sets of nodes and *(ii)* placeholders for subgraphs of various shapes. A *shape* is a set of graphs that may be assigned to a given placeholder. A generic transformation rule abstracts from a (possibly infinite) set of ordinary graph transformation rules, one for every assignment of node sets and subgraphs to its multiple nodes and placeholders, respectively. Thus graph transformation is a two-level process: it first instantiates a generic rule, and then applies the resulting ordinary rule to the host graph afterwards.

To define the shapes that may replace the placeholders in generic rules, we use *adaptive star grammars*. These have been introduced in [9], partly motivated by earlier research on modeling and refactoring of object-oriented programs [21]. A first issue to be addressed was the specification of the set of graphs representing programs. Being context-free devices with nice computational properties, hyperedge and node replacement grammars [14,8,11] have proven particularly useful for defining graph languages. Unfortunately, these types of graph grammars turned out to be too weak to generate program graphs in a reasonable way. Therefore, we have proposed the adaptive star grammar as an extension which is able to generate languages of this type. The rules of an adaptive star grammar have a context-free flavor: each of them replaces a so-called star (a nonterminal node and its incident edges) with another graph.

The remainder of this paper is structured as follows. In the next section, we recall basic notions regarding graphs and graph transformation, and discuss how refactoring can be modeled. It turns out that we need a grammatical mechanism to specify the shape of graph models, and generic rules to specify their transformation in a concise way. In Section 3, we define adaptive star grammars and show as an example how they can be used to define the shape of method bodies, a part of program graphs. Section 4 constitutes the main part of this paper. Here we introduce generic rules, and define how placeholders are expanded by shaped graphs, before the resulting rule is applied. We also sketch how expansion and cloning can be done by incrementally matching of a generic rule to a host graph. Section 5 discusses related work. Finally, we summarize our results, and indicate future work in Section 6.

2 Graph Transformation

In this section, we recall standard notions of graphs and graph transformation, and check how useful they are for model transformation, by discussing a case study on refactoring.

Graphs. Graph-like diagrams have become very popular for representing artefacts that describe software in all its development phases, especially after the Unified Modeling Language (Uml) emerged. We recall a general notion of graphs, and show how it is used for a language-independent representation of object-oriented programs, called *program graphs*.

Throughout the paper, we let $\mathbf{S}$ be our universe of symbols to be used as *labels*. It is the union of two disjoint infinite sets $\dot{\mathbf{S}}$ and $\bar{\mathbf{S}}$ of node and edge labels, resp. For $S \subseteq \mathbf{S}$, we let $\dot{S} = S \cap \dot{\mathbf{S}}$ and $\bar{S} = S \cap \bar{\mathbf{S}}$.

Definition (Graph). A *graph* $G = \langle \dot{G}, \bar{G}, s_G, t_G, \dot{\ell}_G, \bar{\ell}_G \rangle$ 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 $\dot{\ell}_G \colon \dot{G} \to \dot{\mathbf{S}}$ and $\bar{\ell}_G \colon \bar{G} \to \bar{\mathbf{S}}$ that assign labels to its nodes and edges.

If all labels of nodes and edges in G are in $S \subseteq \mathbf{S}$, then G is a *graph over* S. Let $\mathcal{G}_S$ denote the set of all graphs over S.

We use common terminology regarding graphs. For instance, an edge is said to be incident with its source and target nodes, and makes these nodes adjacent to each other. $G \subseteq H$ denotes that G is a subgraph of H, and $G \uplus H$ is the disjoint union of G and H. If a graph G contains a node y, the subgraph $G(y)$ consisting of y, all its incident edges, and all its adjacent nodes is called the neighborhood of y. Finally, $G \setminus \{y\}$ denotes G without the node y, and without the edges of $G(y)$. A pair $g = \langle \dot{g}, \bar{g} \rangle$ of bijective functions $\dot{g} \colon \dot{G} \to \dot{H}$ and $\bar{g} \colon \bar{G} \to \bar{H}$ that preserve sources, targets and labels is called an isomorphism; it makes the graphs G and H isomorphic, written $G \cong_g H$.

Example 1 (Program Graphs). In the case study [21] of refactoring, *program graphs* have been proposed as a language-independent representation of object-oriented programs. Fig. 1 shows two subgraphs of a class of program graphs.

The labels $\{\mathsf{B}, \mathsf{C}, \mathsf{E}, \mathsf{M}, \mathsf{V}\}$ classify nodes as program entities: *bodies* of methods, *classes*, *expressions*, *method signatures*, and *variables*, respectively. The labels $\{\mathsf{a}, \mathsf{ap}, \mathsf{c}, \mathsf{e}, \mathsf{fp}, \mathsf{i}, \mathsf{l}, \mathsf{m}, \mathsf{u}, \mathsf{val}\}$ represent relations between entities: *access*, *actual parameter*, *call*, *element*, *formal parameter*, *inheritance*, *lookup*, *membership*, *update*, and *value*.

A graph must satisfy certain constraints in order to be a valid program graph. The following are typical examples of constraints:

- *Incidence*: An i-edge (modeling inheritance) must be incident with C-nodes (representing classes) only.
- *Cardinality*: An E-node (representing an expression) may have at most one outgoing edge labeled a or u (modeling access resp. update).
- *Structure*: The i-edges must induce a partial order on classes.
- *Context*: An E-node may access a variable (via an a-edge) only if that variable is visible in the context to which the E-node belongs.

In Section 3, we propose graph grammars for specifying the *shape* of graphs, which comprises structural and contextual constraints, e.g., of program graphs, in an intuitive way. Incidence or cardinality constraints can be inferred automatically from the definition of such a grammar. Note also that such constraints can be specified by meta-models (like UML class diagrams or type graphs [4]) along with certain well-formedness constraints (expressed, e.g., by OCL formulas), too. However, we prefer graph grammars for the following reasons:

- Using graph grammars is not only elegant, but also provides a sound foundation for parsing and analysis, as witnessed by the well-developed theory of graph transformation.
- We aim at graph languages like the language of program graphs. Graph grammars are particularly well suited for specifying such recursive (graph) languages which are not that easily specified by meta-models with constraints.

Graph transformation. Since software models can be represented as graphs, graph transformation is a natural candidate for specifying the evolution of models. We use a simple form of DPO graph transformation with injective occurrences. [10].

Definition (Graph Transformation). A (*graph transformation*) *rule* $r = L/R$ consists of two graphs L and R so that the nodes $\dot{I} = \dot{L} \cap \dot{R}$ define a discrete *interface graph* $L \supseteq I \subseteq R$.

Consider a graph G and a rule $r = L/R$. A subgraph $O \subseteq G$ is an *occurrence* of r in G if $O \cong_g L$ for some isomorphism g so that no node in $\dot{O} \setminus g(\dot{I})$ is incident with an edge in $\bar{G} \setminus \bar{O}$. Then r *transforms* G (via the isomorphism g) to a graph that is denoted as $G[L /_g R]$ and is obtained from the disjoint union $G \uplus R$ by *(i)* removing $\bar{O}$ and $\dot{O} \setminus \dot{g}(\dot{I})$ from G, and *(ii)* identifying every interface node $x \in \dot{I}$ with $\dot{g}(x) \in \dot{G}$.

Example 2 (Pull-Up-Method). Pull-Up-Method is a refactoring used when each subclass of a class A defines a method with the same signature and behavior. These methods are then removed from each subclass of A and replaced by a single, equivalent method in A. In the following, we assume that equivalence of different methods has been checked before *Pull-Up-Method* is applied.

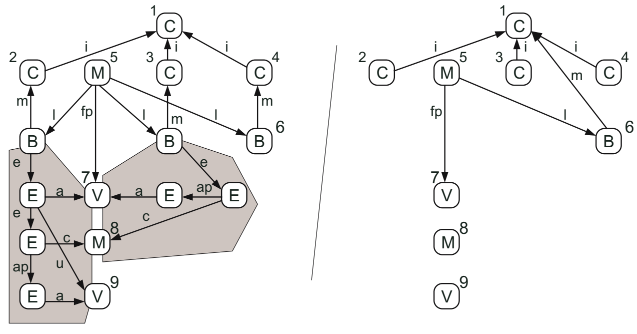

Fig. 1. A concrete rule for *Pull-Up-Method*

```
void m5(v7) {
    v9 := v7;
    m8(v9);
}

void m5(v7) {
    m8(v7);
}
```

Fig. 2. Pseudo code for the method bodies in Fig. 1

Fig. 1 shows a rule implementing a specific case of a *Pull-Up-Method* refactoring for 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 (4) with its superclass (1) and two sibling classes (2, 3). A method signature (5) with one parameter (7) has overloaded bodies (B-nodes) in the sibling classes (2, 3); both implementations make a call to another method (8). The one of class (3) uses the formal parameter (7) as its actual parameter, whereas the other assigns this parameter to a variable (9) first, and calls method (8) with this variable afterward (cf. Fig. 2). Obviously, these implementations have the same semantics but differ syntactically.

The implementations of method (5) in the sibling classes (2, 3), which are emphasized in gray, are removed on the right-hand side of the rule, and its body (6) is moved to the superclass (1). The expressions defining the body (6) 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 (4) has two sibling classes (2, 3); the method (5) has one parameter, and its bodies in the sibling classes use two and three visible names, respectively. In general, there can be any number of sibling classes, parameters, and names.
– The syntactic structure of the method bodies in this example is fixed, but a general rule should be applicable to bodies of different forms. However, the graph of a method body is not just an arbitrary graph, but must have the shape of a method body.

Note that several transformation rules would be needed in order to express the general *Pull-Up-Method* refactoring in the usual graph transformation systems: some for checking that the method is implemented in all sibling classes, others for removing all but one of its implementations, and finally a rule pulling up the remaining implementation. The applications of these rules would have to be controlled in a non-trivial way, and it might not be easy to see that they do what they should, let alone to prove it. As an alternative, we propose to define this refactoring by a single generic rule that is expanded w.r.t. the form of certain subgraphs. Section 4 describes this generic graph transformation approach.

Example 3. As a running example – besides *Pull-Up-Method* in Example 2 – let us consider a graph transformation as shown in Fig. 3: An S-node is connected to several M-nodes that point to linear lists of Q-nodes being connected by next-edges. Each Q-node is connected to each V-node by a var-edge. The transformation removes one of the M-nodes and its list of Q-nodes, and "bends" the m-edge to the remaining M-node. Fig. 3 shows the case where a list of two Q-nodes is removed. The following sections introduce the concepts of shaped graphs and generic rules which allow to specify this transformation for Q-node lists of arbitrary length.

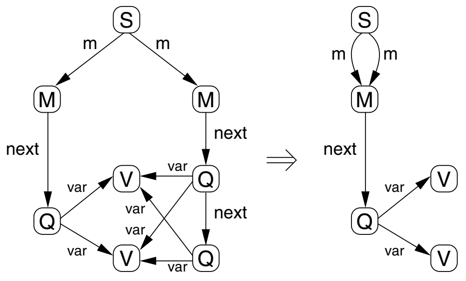

Fig. 3. A sample transformation

3 Shapes

The shape of (a class of) graphs, i.e., their structural and contextual constraints can be specified by graph grammars, like Chomsky grammars specify languages of strings. For describing software models like the program graphs of Example 1, we propose adaptive star grammars [9], which combine star replacement, a very simple way of graph transformation, with an operation called cloning.

Star Replacement. A star replacement replaces a node with its incident edges and adjacent nodes by a graph. Later on, the replaced nodes will be considered to be nonterminals.

A *star* X is a graph that consists of a *center node* y, $n \geqslant 0$ *border nodes*, and n edges making y adjacent to all border nodes.

A rule X/P is a *star rule* if X is a star and the interface graph I consists of the border nodes of X; star rules are denoted as $X ::= P$ to emphasize that they are used to generate languages, like context-free Chomsky rules.

According to the definition of graph transformation, a star $\tilde{X} \subseteq G$ is an *occurrence* of a star rule $X ::= P$ if $\tilde{X} \cong_g X$ and $\tilde{X}$ is the neighborhood of its center node in G. Then star replacement via g yields the graph $G[X /_g P]$.

Cloning. Star replacement is closely related to hyperedge replacement [14,8]. For grammars based on star replacement, this implies certain limitations. For instance, the maximal number of border nodes in the left-hand sides of a grammar restricts the connectivity of the generated graphs, so that star replacement cannot generate the class of all graphs, or the class of all complete graphs, over any set S of labels (provided that $\dot{S} \neq \emptyset \neq \bar{S}$).

To overcome these limitations, we introduce *multiple nodes* that are placeholders for any number of nodes. The latter are called *clones* because each of them has the same incident edges, and is adjacent to the same nodes as the multiple node.[1]

[1] Note that cloning is not "deep copying" of subgraphs; it just copies a single node with its incident edges. Deep copying can be achieved by cloning placeholders of subgraphs (see Section 4).

We designate multiple nodes by a special set of *multiple node labels* $\ddot{S} \subset \dot{S}$. The remaining node labels $\dot{S} \setminus \ddot{S}$ are called *singular.* We further assume that there is a bijection $\ddot{}: \dot{S} \setminus \ddot{S} \to \ddot{S}$ that maps every singular label s to its multiple counterpart $\ddot{s}$. A node is called singular or multiple depending on its label. The set of multiple nodes in a graph G is denoted by $\ddot{G}$. In figures, we draw multiple nodes as circles or boxes with a "shadow", e.g., the V-nodes in Fig. 4.

The cloning operation turns a multiple node into any number of singular and multiple clones: we define $G\frac{x}{(m,k)}$ to be obtained from G by replacing the multiple node x with $m+k$ clones whereof m are multiple, and k are singular.

Formally, for a graph G with a multiple node $x \in \ddot{G}$, and $m, k \geq 0$, the graph $G\frac{x}{(m,k)}$ is constructed as follows. Let $G'(x)$ be obtained from the neighborhood $G(x)$ by replacing the label $\ddot{s}$ of x by the singular label s. Then take the disjoint union of the graph $G \setminus \{x\}$ with m copies of $G(x)$ and k copies of $G'(x)$. Finally, identify the $m+k+1$ copies of each node in $\dot{G}(x) \setminus \{x\}$ with each other.

As an example, consider Fig. 10 with its multiple V-node in the left-hand side G. The left-hand side of the rule shown in Fig. 11 is $G\frac{3}{(0,2)}$, i.e., the multiple node 3 is turned into two singular nodes and no multiple node.

Obviously, $G\frac{x}{(m,k)}$ is defined only up to isomorphism. Note that cloning is closely related to node replacement. It cannot be specified by finitely many graph transformation rules in the sense of Definition 2, because a multiple node x may have a neighborhood $G(x)$ of arbitrary size.

Although distinct multiple nodes may be adjacent to each other, cloning is commutative: For a graph with distinct multiple nodes x, x', and numbers $m, k, m', k' \geqslant 0$, $\left(G\frac{x}{(m,k)}\right)\frac{x'}{(m',k')} \cong \left(G\frac{x'}{(m',k')}\right)\frac{x}{(m,k)}$. We can thus define an operation that clones all multiple nodes in a graph G. The number of desired clones is indicated by a so-called *multiplicity function* $\mu\colon \ddot{G} \to \mathbb{N}^2$. If $\ddot{G}$ contains n multiple nodes $x_1, \ldots, x_k$, the *μ-clone of* G is defined as $G^\mu = \left(\cdots\left(G\frac{x_1}{\mu(x_1)}\right)\cdots\frac{x_k}{\mu(x_k)}\right)$.

Adaptive Star Replacement. Star replacement is made adaptive by cloning the star rule and the graph to be transformed before performing the replacement. Let G be a graph containing a star $\tilde{X}$, and consider a star rule $X ::= P$. We assume without loss of generality that the nodes of G and P are disjoint.

A multiplicity function $\mu\colon \ddot{G} \cup \ddot{P} \to \mathbb{N}^2$ is *an adapter of X/P and $\tilde{X}$* if $X^\mu \cong_g \tilde{X}^\mu$ for some isomorphism g. Then, the *adaptive replacement* of $\tilde{X}$ by P using μ is defined as $G[X\,{}^{\mu}\!/_{g}\,P] = G^\mu[X^\mu\,/_{g}\,P^\mu]$.

It is straightforward to show that adaptive star replacement is commutative and associative. We note this result, but leave out the proof:

Lemma 1 (Commutativity and Associativity). *If $H = G[X\,{}^{\mu}\!/_{g}\,P][\tilde{X}\,{}^{\tilde{\mu}}\!/_{\tilde{g}}\,\tilde{P}]$ for some graph G, star rules X/P, $\tilde{X}/\tilde{P}$, adapters μ, $\tilde{\mu}$, and isomorphisms g, $\tilde{g}$, then, for suitable adapters μ', $\tilde{\mu}'$ and isomorphisms g', $\tilde{g}'$,*

1. *$H = G[\tilde{X}\,{}^{\tilde{\mu}'}\!/_{\tilde{g}'}\,\tilde{P}][X\,{}^{\mu'}\!/_{g'}\,P]$ if the center of the occurrence $\tilde{g}(\tilde{X}^{\tilde{\mu}})$ is in G (commutativity), and*

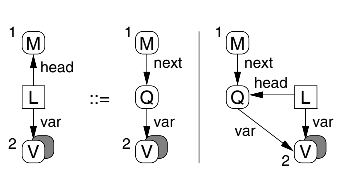

Fig. 4. Adaptive star rules generating linear Q-node lists of arbitrary length

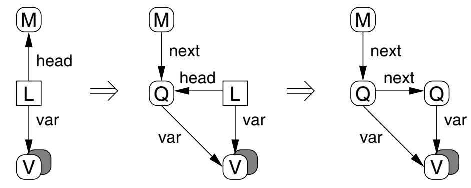

Fig. 5. A derivation of a list of two Q nodes using the adaptive star rules in Fig. 4

2. $H = G[X\, {}^{\tilde\mu'}\!/_{\tilde g'}\, P[\tilde X\, {}^{\mu'}\!/_{g'}\, \tilde P]]$ *if the center of the occurrence* $\tilde g(\tilde X^{\tilde\mu})$ *is in* P *(associativity).*

We can now define adaptive star grammars and the graph languages they generate. We write $G \Rightarrow_{\mathcal{P}} H$ if $H \cong G[X\, {}^{\mu}\!/_{g}\, P]$ for some adapter μ, isomorphism g, and rule $p = X ::= P$ from a set $\mathcal{P}$ of star rules, and $G \Rightarrow^{*}_{\mathcal{P}} H$ if $G \cong G_0 \Rightarrow_{\mathcal{P}} \cdots \Rightarrow_{\mathcal{P}} G_n \cong H$ for $n \geqslant 0$; thus $\Rightarrow^{*}_{\mathcal{P}}$ is the transitive-reflexive closure of $\Rightarrow_{\mathcal{P}}$.

Definition. An *adaptive star grammar* is a tuple $\Gamma = \langle S, N, \mathcal{P}, Z\rangle$ consisting of a finite set $S \subseteq \mathbf{S}$ of *terminal labels*, a finite $N \subseteq \dot{\mathbf{S}} \setminus \dot{S}$ of singular *nonterminal labels*, a finite set $\mathcal{P}$ of star rules $X ::= P$, where X and P are graphs over $S \cup N$, and an *initial star* Z over $S \cup N$.

For Z as well as the left- and right-hand sides of rules in $\mathcal{P}$, we require that the neighborhoods of all nonterminal nodes are stars with terminal border nodes (where a node is called terminal or nonterminal according to its label). Moreover, the center nodes of Z and all left-hand sides are required to be nonterminal. Stars of this kind are called N-stars.

The *language generated by* Γ is defined as

$$\mathcal{L}(\Gamma) = \{G \in \mathcal{G}_{S\setminus\ddot{S}} \mid Z \overset{*}{\underset{\mathcal{P}}{\Rightarrow}} G\}.$$

Note that, in an adaptive star grammar (and in the graphs they generate), nonterminal nodes cannot be adjacent to each other.

As an example, consider the language introduced in Example 3. Fig. 4 shows the adaptive star rules of the adaptive star grammar that generates this language. L is the only nonterminal label (note that nonterminal nodes are drawn as rectangles whereas terminal nodes are drawn with round corners). The common left-hand side of both rules is the initial star Z. Fig. 5 shows a derivation of a graph consisting of a Q-node list of length two. Note that the derived graph does not belong to the generated language, because it still contains a multiple V-node that has to be turned into an arbitrary number of singular V-nodes.

Adaptive star grammars generate languages that cannot be generated by node replacement [11], like the class of all graphs, or classes of graphs defined by contextual constraints such as the program graphs from Example 1.

It should be mentioned that the variant of adaptive star grammars originally introduced in [9] is more general than the one considered here, because stars with parallel edges (being incident with the same border node) and rule application using non-injective occurrences are considered. In [9], the resulting type of adaptive star grammar is shown to generate all recursively enumerable string languages (represented as chain graphs), whereas the one considered in the present paper is shown to have a decidable membership problem.

Example 4 (Adaptive Star Rules for Method Bodies). The rules in Fig. 6 generate simple method bodies for the program graphs discussed in Example 1 if the left-hand side of the rule for the nonterminal ST* is the initial star. A method body has a root labeled B pointing to E-nodes representing the assignments and calls in the body; the right-hand sides of assignments, and the actual parameters of calls may again be calls. All stars in these rules have a def-edge to a singular node representing the subgraph generated by the star, and vis-edges to multiple or singular nodes representing the methods (labeled M) and variables (labeled V) that are visible in these subgraphs. A call to a method, or an access or update of a variable within an expression is represented as an edge to one of these nodes. The rules for ASS, CALL, and ACC introducing these edges apply only if corresponding nodes are visible. Thus every entity used in the body is a clone of the multiple border nodes of the initial star. This expresses the contextual constraint that every used entity should have a declaration. In the complete program graph grammar given in [27], these entities are generated as members of the class hierarchy that are visible in the context of the method body. There, method bodies may also contain control structures and local declarations.

In the rules for ST* and CALL, we introduce a useful shorthand for star rules, somewhat similar to the use of the Kleene star in the right-hand side of a context-free Chomsky rule. The shaded subgraphs on the right-hand sides of these rules are called *iterated subgraphs*. As this name suggests, an iterated subgraph may be copied any number of times, the copies sharing the nodes on its border. To emphasize this, we draw the nodes to be copied similarly to multiple nodes and annotate their "shades" with a common index (k and n, resp.). Iteration can

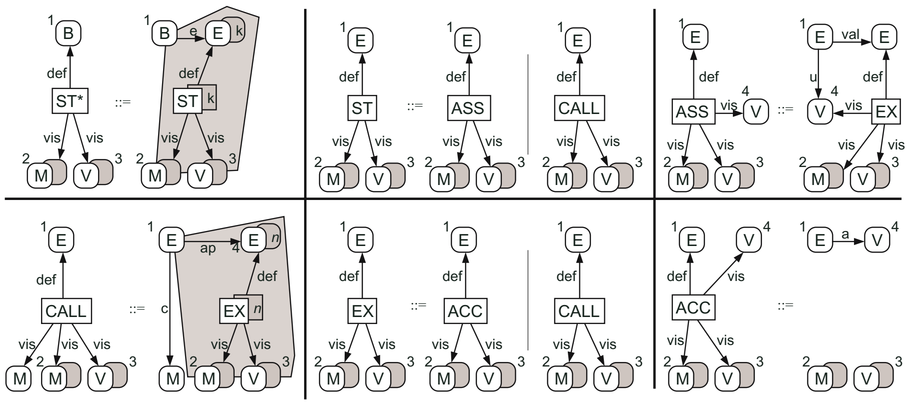

Fig. 6. Adaptive star rules defining the structure of method bodies

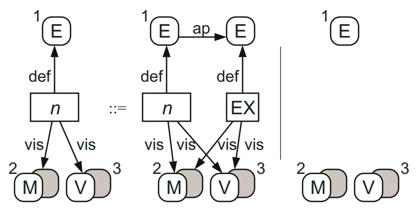

Fig. 7. Adaptive rules for Subgraph Iteration

obviously be implemented by adding a star rule that differs from the given one in that its right-hand side contains an additional star, isomorphic to the left-hand side and connected to the nodes on the border of the shaded part. The star rules generating the iterated subgraph in the rule for CALL in Fig. 6 is shown in Fig. 7.

4 Generic Transformation Rules

This section contains the main contribution of the paper. We extend the transformation rules of Section 2 so that they become generic: their graphs may contain multiple nodes and nonterminal nodes. Multiple nodes are cloned, as in adaptive star grammars, and nonterminal nodes are expanded to graphs before a generic rule is applied.

Shaped Expansion. Shaped expansion allows for graphs (in transformation rules) that contain N-stars as placeholders. These can be expanded to graphs generated by an adaptive star grammar, where isomorphic stars are expanded to isomorphic graphs. For this, and throughout the rest of this paper, let $\Gamma = \langle S, N, \mathcal{P}, Z \rangle$ be an adaptive star grammar. In the following, we will only consider graphs over $S \cup N$.

A set σ of star rules is a *substitution* if *(i)* the left-hand sides of rules in σ are pairwise nonisomorphic N-stars, *(ii)* the right-hand sides of rules in σ are terminal, and *(iii)* each rule $X/P \in \sigma$ satisfies $X \Rightarrow^*_{\mathcal{P}} P$. A graph G is *covered* by a substitution σ if, for every N-star $G(x)$ in G, there is a star rule $X/R \in \sigma$ with $G(x) \cong X$.

Intuitively, expanding a graph G means to apply the rules of a substitution σ to all N-stars in G. To make this precise, consider a graph G whose (pairwise distinct) N-stars are $G(x_1), \ldots, G(x_n)$, and let σ be a substitution that covers G. A *σ-expansion* G^σ of G is a graph of the form

$$G[X_1 /_{g_1} P_1] \cdots [X_n /_{g_n} P_n] \text{ where } X_i/P_i \in \sigma \text{ and } G(x_i) \cong_{g_i} X_i, \text{ for } 1 \leqslant i \leqslant n.$$

Since star replacement is commutative, the order of the replacement steps is irrelevant. However, as the isomorphisms $g_i \colon X_i \to \tilde{X}_i$ need not be uniquely determined, there may be several σ-expansions of G.

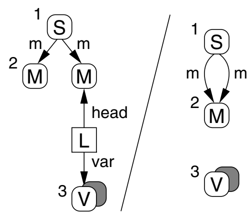

Fig. 8. The generic rule r used for the transformation in Fig. 3

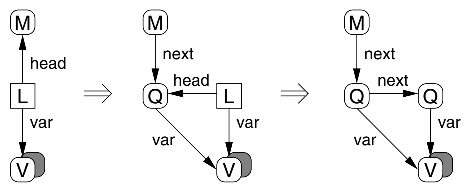

Fig. 9. The derivation of Fig. 5, using the adaptive star rules in Fig. 4 for specifying a substitution σ

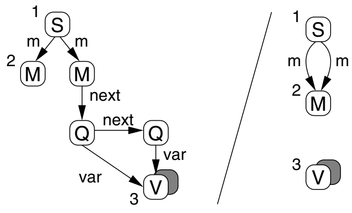

Fig. 10. σ-expansion L^σ/R^σ of r in Fig. 8 using substitution σ in Fig. 9

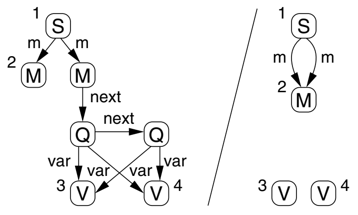

Fig. 11. The ordinary transformation rule $(L^\sigma)^\mu \mathbin{/\!\!/_g} (R^\sigma)^\mu$ obtained from L^σ/R^σ in Fig. 10 by multiplicity function $\mu : 3 \mapsto (0, 2)$

Generic Transformation. Generic graph transformation is plain transformation with transformation rules that have been expanded and cloned. More precisely, let us call a transformation rule $r = L/R$ *generic* if its interface nodes are terminal. A multiplicity function $\mu\colon \ddot{G} \to \mathbb{N}^2$ is *singular* if $\mu(x) = (0, k)$ with $k \geqslant 0$ for every $x \in \ddot{G}$.

Now, let G, H be graphs, $r = L/R$ a generic rule, and σ a substitution covering $L \cup R$. Consider a σ-expansion L^σ/R^σ of r, consisting of σ-expansions L^σ and R^σ of L and R, resp. Then r *transforms* G into H, written $G \Longrightarrow_{r,\sigma,\mu} H$, if $H = G[(L^\sigma)^\mu \mathbin{/\!\!/_g} (R^\sigma)^\mu]$ for some singular multiplicity function $\mu\colon \ddot{L^\sigma} \cup \ddot{R^\sigma} \to \mathbb{N}^2$, and an isomorphism g.

Fig. 8 shows the generic rule $r = L/R$ that is used for the transformation shown in Fig. 3, i.e., that removes an M-node together with its list of Q-nodes. The contained L-star is the placeholder for an arbitrary list of Q-nodes pointing to all V-nodes. This graph language is specified by the adaptive star rules in Fig. 4. The transformation shown in Fig. 3 uses the ordinary transformation rule shown in Fig. 11 that is generated by first substituting the L-node by substitution σ specified in Fig. 9, yielding L^σ/R^σ (Fig. 10), and then choosing a multiplicity function $\mu : 3 \mapsto (0, 2)$ for turning the multiple V-node into two singular V-nodes, yielding $(L^\sigma)^\mu \mathbin{/\!\!/_g} (R^\sigma)^\mu$ (Fig. 11).

Example 5 (A Generic Rule for Refactoring). The general *Pull-Up-Method* rule is specified in Figure 12. The rule applies to a class (3) with its superclass (1), and a set of other subclasses (2); the method signature (5) has parameters (6), and is implemented by bodies that may refer to variables (7) and methods (8).

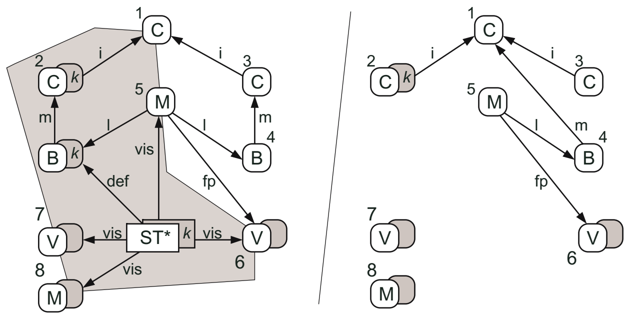

Fig. 12. The generic rule for the *Pull-Up-Method* refactoring

The sibling classes (2) are represented by an iterated subgraph (designated by the nodes with index k). The nonterminal ST* in the iterated subgraph is a placeholder for the method bodies for the signature (5). These bodies are removed by the transformation rule since they do not appear on its right-hand side. 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.

The ST*-star is a placeholder for method bodies. Thus, the expansions of these stars are shaped according to the method body grammar. Recall that the iterated subgraph is a shorthand for a star which can be turned into any number of copies of the given subgraph, using iteration rules added to Γ, as described in Example 4. Here, a minor technical complication is caused by the fact that one of the nodes (2) of the iterated subgraph is an interface node. The (intuitively obvious) meaning of this is that all copies of this node are intended to belong to the interface as well.

In a generic rule, all occurrences of a nonterminal n are expanded to isomorphic subgraphs; having several occurrences of n on the left-hand side thus allows to check equality of subgraphs of the host graph, whereas having several occurrences of n on the right-hand side allows one to make so-called deep copies of the expansions.

Goal-Oriented Matching. The definition of generic transformation is not operational: In order to transform a graph with a generic rule, we cannot generate all its expansions, and choose one of them for application, because generic rules usually have infinitely many expansions.

However, the instantiation of a rule (i.e. expansion and cloning) can be done in a more *goal-oriented fashion.* In order to apply a generic rule $r = L/R$ to a graph G, one may proceed as follows:

- *Find* a *kernel occurrence* $\underline{O}$ of the constant subgraph $\underline{L}$ of L in G.
- *Match* the stars and multiple nodes in r one after another, by expanding and cloning them, respectively, so that $\underline{O}$ is gradually extended to a complete occurrence O of the instantiated left-hand side L.

– *Instantiate* the right-hand side R according to the substitution and multiplicity function found in the matching process, and *insert* the instantiated right-hand side for O. If, for every star X in R, there is an isomorphic star X' in L, the instantiation is uniquely determined.

Moreover, the matching of a star can be done *incrementally*, applying one of the star rules defining the shape of a star at a time.

Since adaptive star grammars are parseable, it is decidable whether an expansion exists. Parsing may be complex in general. However, for grammars occurring in practice, like those for method bodies, and for program graphs as a whole, we expect parsing to be reasonably efficient. Experiments with an implementation of a star grammar parser suggest the parsing time for such grammars is polynomial [22].

For the intended application area of software refactoring (and certainly many other application areas as well), it must be pointed out that the matching process sketched above should be coupled with user interaction to resolve the inherent nondeterminism. Obviously, there may be many generic rules that can be applied, at many different places in the host graph, and with many different expansions. Thus, a reasonable implementation must present the different possibilities to the user, and let her choose the one that reflects her refactoring intentions.

5 Related Work

Generic rules have been proposed quite early for string languages, e.g., *Van Wijngaarden grammars* [28]. A precursor of the generic graph transformation rules described in this paper has been investigated in [23], where the placeholders are stars with a fixed number of adjacent nodes (called hyperedges). Substitutions shaped according to hyperedge replacement grammars have been proposed in [16]. Path expressions specifying implicit edges, as known in programmed graph transformation [26], can be considered as a special case of substitutions shaped according to the path expression. The set nodes in that work have been the model for our multiple nodes. In fact, cloning concepts have become quite popular. Apparently, sesqui-pushout rewriting [6] and Kahl's approach [18] support cloning as well. In a recent paper, Lindqvist *et al.* have proposed the *star operator* that is motivated by the *Kleene star* [20]. Patterns are generated from generic patterns by deep copying and chaining of so-called star regions.The graph transformation language GReAT used for model transformation also allows to specify patterns containing multiple objects that can be single nodes or compound patterns containing subgraphs [1]. Several graph transformation tools have been further extended by "set" operators: Viatra2 allows to match graph patterns recursively, which allows for dealing with set-valued patterns [29]. A *grouping operator* has been introduced to GReAT [2]. This operator allows to simultaneously operate on the set of all isomorphic matches of a single pattern. And Progres has been extended by two such operators: A new language construct has been introduced to specify and operate on *successively connected repetitive subgraphs* [19], and the extension for *set-valued transformations* [13] is very similar to [17].

Finally, amalgamated graph transformations (e.g., [3]) are related to set nodes. This approach does not introduce multiple objects, but it provides a formalism to generate ordinary transformation rules from rule templates by applying these templates in parallel. This allows to specify the cloning of set nodes presented in this paper or in previous papers [9,17].

However, apart from our previous work [17], we are not aware of any kind of graph transformation that combines cloning with expansion, i.e., with the instantiation of placeholders by subgraphs that are shaped according to graph grammars.

6 Conclusions

Being a formalism that allows a direct manipulation of the diagrammatic representations of programs, graph transformation is a natural candidate to be used as the formal foundation for tools supporting program transformations. Such transformations are at the heart of the model-driven approach to software development, and also of so-called refactoring techniques, where the structure of existing software is improved through the application of certain precisely specified operations. Modeling such operations by graph transformation rules requires, however, that these rules are sufficiently expressive, so that they can be considered to be at the same level of abstraction as the operations one wants to model. If the rules lack expressive power, one is forced to govern their application by more complicated control programs, and the result may be that much of the inherent complexity of the operations to be modeled is reflected in this control structure rather than in the graph rewriting.

In order to improve the expressive power of graph rewriting rules so that the complexity of control programs is reduced, we have proposed generic graph transformation rules wherein placeholders are expanded to graphs, and multiple nodes are cloned as often as necessary. Expansions of placeholders are shaped, i.e., the placeholders are nonterminal stars whose possible expansions are defined by an adaptive star grammar. This allows for structural and contextual constraints on graphs to be described. The concept makes it possible to specify complex transformations, e.g., the *Pull-Up-Method* refactoring [12], by a single generic rule in an intuitive manner. The parsing algorithm for adaptive star grammars opens the door to a goal-oriented matching algorithm that will be an essential part of a forthcoming implementation of generic rules.

The work on shaped generic graph transformation rules and their properties is not finished. As a first step toward extending the results for DPO graph transformation to generic rules, a *parallel independence theorem* has been shown in [15], for generic rules wherein stars have a fixed rank, unshaped substitutions, and are not cloned. This work shall be extended to the study of critical pairs, for the generic rules defined here.

For practical use, we need graphs with attribute values, and rules that specify attribute evaluation. For instance, signature nodes in program graphs could have an attribute counting its parameters, and transformation rules would update this value when necessary. In [24], attribute values are (additional) labels, and rules are labeled with expressions specifying computations on these values. This fits well with the variable concept in generic rules. The values and expressions could be taken from some host language, but they could also be defined by (nested) graphs and transformations, as in [16].

Adaptive star grammars fail to describe some contextual constraints of program graphs, like the correspondence of formal to actual parameters of a method. However, these properties can be specified with pre- or post-conditions of the star rules, sacrificing neither commutativity nor associativity. For practical applications, like the definition of software models, one should focus on grammars generating connected, or tree-like graphs with "cross-links" (like the program graphs). This will not only make parsing more efficient, but is also supposed to be useful in order to establish a static type discipline as in [16]: If the rules, and the contexts of their application are "shaped" like the substitutions, it can be shown that transformations preserve the shape of the graphs being transformed. In other words: such transformation rules can be guaranteed to preserve the integrity of a model.

References

1. Agrawal, A., Karsai, G., Neema, S., Shi, F., Vizhanyo, A.: The design of a language for model transformations. J. Software and System Modeling 5(3), 261–288 (2006)
2. Balasubramanian, D., Narayanan, A., Neema, S., Ness, B., Shi, F., Thibodeaux, R., Karsai, G.: Applying a grouping operator in model transformations. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088. Springer, Heidelberg (2008)
3. Boehm, P., Fonio, H.-R., Habel, A.: Amalgamation of graph transformations: A synchronization mechanism. J. Computer and System Sciences 34, 377–408 (1987)
4. Corradini, A., Ehrig, H., Montanari, U., Padberg, J.: The category of typed graph grammars and its adjunction with categories of derivations. In: [7], pp. 56–74
5. Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.): 3rd Int. Conf. on Graph Transformation (ICGT 2006). LNCS, vol. 4178. Springer, Heidelberg (2006)
6. Corradini, A., Heindel, T., Hermann, F., König, B.: Sesqui-pushout rewriting. In: [5], pp. 30–45
7. Cuny, J.E., Ehrig, H., Engels, G., Rozenberg, G. (eds.): Graph Grammars 1994. LNCS, vol. 1073. Springer, Heidelberg (1996)
8. Drewes, F., Habel, A., Kreowski, H.-J.: Hyperedge replacement graph grammars. In: [25], pp. 95–162
9. Drewes, F., Hoffmann, B., Janssens, D., Minas, M., Van Eetvelde, N.: Adaptive star grammars. In: [5], pp. 77–91
10. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. In: EATCS Monographs on Theoretical Computer Science. Springer, Heidelberg (2006)
11. Engelfriet, J., Rozenberg, G.: Node replacement graph grammars. In: [25], ch. 1, pp. 1–94
12. Fowler, M.: Refactoring—Improving the Design of Existing Code. Object Technology Series. Addison-Wesley, Reading (1999)

13. Fuss, C., Tuttlies, V.E.: Simulating set-valued transformations with algorithmic graph transformation languages. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088. Springer, Heidelberg (2008)
14. Habel, A.: Hyperedge Replacement: Grammars and Languages. LNCS, vol. 643. Springer, Heidelberg (1992)
15. Habel, A., Hoffmann, B.: Parallel independence in hierarchical graph transformation. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 178–193. Springer, Heidelberg (2004)
16. Hoffmann, B.: Shapely hierarchical graph transformation. In: Proc. IEEE Symposia on Human-Centric Computing Languages and Environments, pp. 30–37 (2001)
17. Hoffmann, B., Janssens, D., Van Eetvelde, N.: Cloning and expanding graph transformation rules for refactoring. Electronic Notes in Theoretical Computer Science 152(4), 53–67 (2006); Proc. GraMoT 2005
18. Kahl, W.: A relation-algebraic approach to graph structure transformation, 2001. Habil. Thesis, Fak.für Informatik, Univ. der Bundeswehr München, TR 2002-03
19. Körtgen, A.-T.: Modeling successively connected repetitive subgraphs. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088. Springer, Heidelberg (2008)
20. Lindqvist, J., Lundkvist, T., Porres, I.: A query language with the star operator. In: Proc. 6th Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007). Electronic Comm. of the EASST, vol. 6 (2007)
21. Mens, T., Demeyer, S., Janssens, D.: Formalising behaviour-preserving transformation. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 286–301. Springer, Heidelberg (2002)
22. Minas, M.: Parsing of adaptive star grammars. In: Proc. GraMoT 2006. Electronic Comm. of the EASST, vol. 4 (2006)
23. Plump, D., Habel, A.: Graph unification and matching. In: [7], pp. 75–89
24. Plump, D., Steinert, S.: Towards graph programs for graph algorithms. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 128–143. Springer, Heidelberg (2004)
25. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation. Foundations, vol. I. World Scientific, Singapore (1997)
26. Schürr, A.: Introduction to the specification language PROGRES. In: Nagl, M. (ed.) IPSEN 1996. LNCS, vol. 1170, pp. 248–279. Springer, Heidelberg (1996)
27. Van Eetvelde, N.: A Graph Transformation Approach to Refactoring. Doctoral thesis, Universiteit Antwerpen (May 2007)
28. van Wijngaarden, A., Mailloux, B.J., Peck, J.E.L., Koster, C.H.A., Sintzoff, M., Lindsey, C.H., Meertens, L.G.L.T., Fisker, R.G.: Revised report on the algorithmic language ALGOL 68. Acta Informatica 5, 1–236 (1975)
29. Varró, G., Horváth, A., Varró, D.: Recursive graph pattern matching with magic sets and global search plans. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088. Springer, Heidelberg (2008)