Program *Graph* Transformation

Berthold Hoffmann

Universität Bremen
hof@informatik.uni-bremen.de

Abstract. Graph transformation, a branch of theoretical computer science, is about the definition of graph languages by grammars, and the study of computations on graphs by rewrite rules. In this paper, we sketch a software engineering problem – the refactoring of object-oriented software – and indicate how graph grammars and graph rewrite rules can be extended for specifying and implementing refactoring operations on a high level of abstraction.

1 Introduction

Graph transformation generalizes two areas of theoretical computer science. Chomsky grammars and Lindenmeyer systems [28, chap. 4, 5] generate sets, or *formal languages*, of words. Graph grammars allow more general languages to be generated: data structures like double-linked lists, the admissible states of some system, or the growth and habitus of plants [26]. Term rewriting and narrowing define *rule-based computational models* on terms, or trees [1]. With graph rewriting, it is possible to study rule-based computations on more general structures: term graph rewriting [25], or petri nets [12], to mention just a few.

Graphs are often drawn as diagrams, and diagrams are abstractly represented as graphs. This makes graph transformation a natural choice for specifying and implementing systems that generate and modify diagrams. Diagram languages like UML are now widely used for the model-driven design of object-oriented software. Graph transformation has a high potential for specifying and implementing visual software models and their transformation, witnessed by a series of international workshops on *graph transformation and visual modeling techniques* [16]. In this paper, we use a particular scenario from this area to illustrate the potential of graph transformation.

Refactoring of Object-Oriented Code. Refactoring aims at improving the structure of object-oriented software for maintenence and adaptation, without changing its behavior. Martin Fowler's book [15] gives a catalog of basic refactoring operations, which, skillfully combined, shall attain these goals. A typical operation in this catalog, *Pull-up Method*, is summarized as follows:

- All subclasses of a class c contain equivalent implementations of a method m.
- Pull one of these implementations up to c, and remove all other implementations of m in the subclasses of c.

An informal description of the "mechanics" then prescribes a series of steps by which a programmer should interactively change the code, compile, and test it, in order to make sure that the consistency of the code, and the behavior of the program is preserved. Nowadays, many CASE tools do support refactoring. However, these implementation have no formal basis so that is remains unclear whether the application of a refactoring will always preserves code consistency, let alone program behavior.

The extensive case study [20, 21] by D. Janssens, T. Mens et al. has shown that graph transformation has a high potential for formalizing refactoring operations. Many advantages are equally true for software models and model transformation in general.

Graph grammars allow languages of graphs and diagrams to be described in a general way. In Section 2 we show how the shape of program graphs, a language-independent representation of object-oriented code, can be be defined with adaptive star grammars. Refactoring operations can be defined by graph rewriting rules. In Section 3, the Pull-up Method refactoring is specified by a single generic graph rewrite rule. These rules allow to specify expressive patterns, including sub-patterns of variable number and shape. In Section 4, we study how it can be checked whether rewriting steps preserve the shape of graphs. In particular, we discuss under which conditions generic rules do preserve shape automatically. Only if all these concepts are supported by an efficient tool, they can be used for developing refactoring tools. In Section 5 we outline how such a tool could be developed. We conclude with some remarks on related and future work in Section 6.

2 Models and Graph Grammars

We describe program graphs informally, discuss why their properties cannot be captured easily by meta-models like class diagrams, and define them precisely, by adaptive star grammars with application conditions.

Program Graphs. The case study [20] has proposed a language-independent representation of object-oriented programs as *program graphs*, which can be used for refactoring code written in many languages. In a program graph, nodes represent syntactic entities, and edges establish relations between entities. Relations represent a hierarchical structure representing abstract syntax, and context links between uses and definitions of variables and methods that result from the analysis of declarations. Thus the names of variables or methods are not relevant in this representation.

Fig. 1 below shows two program graphs G and H. (We prefer the classical way to draw graphs. A denotation as UML object diagrams would be less concise.) In G, the subgraph P (underlaid in light grey) represents a class with a method signature and two sub-classes (the nodes labeled C and M). Every subclass contains bodies for the method (the subgraphs underlaid in darker grey), containing body, expression, and variable nodes, which are labeled B, E, and V.

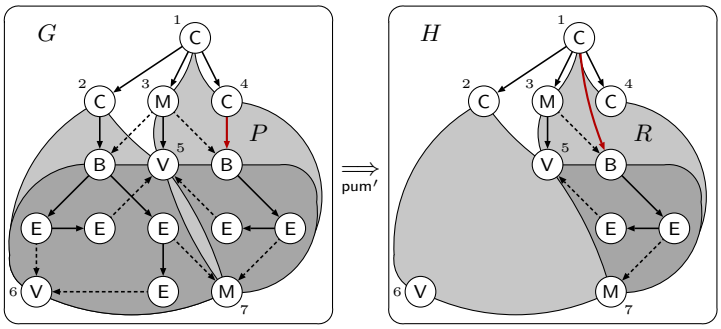

Fig. 1. Two program graphs

In H, one of the method bodies in P has been deleted, and the other one pulled up to the superclass (by altering the source of the red edge). Actually, this is what a *Pull-up Method* refactoring should do to any pattern P of a program graph G.

In model-driven design, the shape of models is usually specified by meta-modeling. In UML, for instance, a class diagram specifies the shape of the object diagrams that can be generated by a system. A class diagram usually specifies *subclass relations* on nodes, *incidence constraints* for edges (i.e., which type of node may be the source or target of some type of edge), and *cardinality constraints* (i.e., how many edges of some type may be the source or target of a certain type of node). More involved constraints concerning, e.g., the existence of paths in a graph, can be specified by logical formulas, as in the UML's sub-language OCL.

Program graphs are trees representing the nesting of the abstract syntax (by solid arrows), with contextual links from the uses of names to their declaration (by dashed arrows). A class diagram can capture the syntactic structure by subclasses. But, the definition-to-use links of program trees can only be specified by OCL formulas.

Therefore we propose to use graph grammars for specifying program graphs, rather than meta-models. A graph grammar generates a language of graphs (the valid models) by inductive rules.

Star Grammars. We start from a class $\mathcal{G}$ of graphs that are labeled and directed, and may have loops and parallel edges (with the same label, connecting the same nodes).

Star grammars are based on variable substitution. In a graph, a *variable* is a star-like subgraph X with a center node that is labeled with a variable symbol, and has edges connecting the center node to border nodes that are labeled with terminal symbols. The graph in Figure 7 is a star.

A *star rule* is written $Y ::= S$, where Y is a variable and S a graph so that Y and S share the border nodes of Y. Figure 2 shows a star rule for the body of a method. The box annotated e that surrounds the variable named **Exp** (with

center node x and border node v) indicates a *multiple subgraph*: this box may have $n > 0$ clones, where each of them is connected to the rest of the graph in the same way (by an edge from the B-node to each clone of the node v in this case). Multiple subgraphs transfer the notations for repetitions in the EBNF of context-free grammars to graphs.

We draw star rules by "blowing up" the center node of Y and drawing the inserted nodes and edges of S inside, as can be seen in Figure 4. (The second rule in the lower row of is that of Figure 2.).

A variable X in a graph G can be substituted with a star rule $Y ::= S$ if there is an isomorphism $m\colon Y \to X$. Then a *variable substitution* yields the graph denoted as $G[X/_m S]$, which is constructed from the disjoint union of G and S by identifying the border nodes of X and Y according to m and removing their edges and center nodes. Let Σ be a finite set of star rules. Then we write $G \Rightarrow_\Sigma H$ if $H = G[X/_m S]$ for some $y ::= S \in \Sigma$, and denote the reflexive-transitive closure of this relation by $\Rightarrow^*_\Sigma$. Let $\mathcal{Y}$ be the set of all variables occurring in Σ. We call them *syntactic variables*, like in Chomsky grammars, and denote the graphs with syntactical variables by $\mathcal{G}(\mathcal{Y})$. Then $\Gamma = \langle \mathcal{G}(\mathcal{Y}), \mathcal{Y}, \Sigma, Z \rangle$ is a *star grammar* with a *start variable* $Z \in \mathcal{Y}$. The *language* of Γ is obtained by exhaustive substitution of its rules to the start variable:

$$\mathcal{L}(\Gamma) = \{G \in \mathcal{G} \mid Z \Rightarrow^*_\Sigma G\}$$

Figure 4 shows the star rules generating the program trees that underlie program graphs, and Figure 3 shows the generation of a method body tree.

Adaptive Star Grammars. The dashed edges in the rules use, assign, and call point to new nodes generated by these rules. In program graphs, dashed edges are *contextual links* that shall point to a visible declaration of an attribute, a variable, or a method, respectively. The contextual links of program graphs cannot be generated with star grammars. Because, in a finite star grammar, the number of border nodes in stars is bounded. For the insertion of a contextual link, say to a method in a call rule, all potential targets of the link had to be border nodes of the left-hand side of the call rule. Since the number of visible

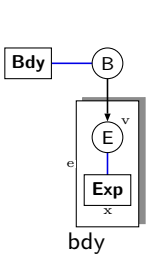

Fig. 2. A star rule

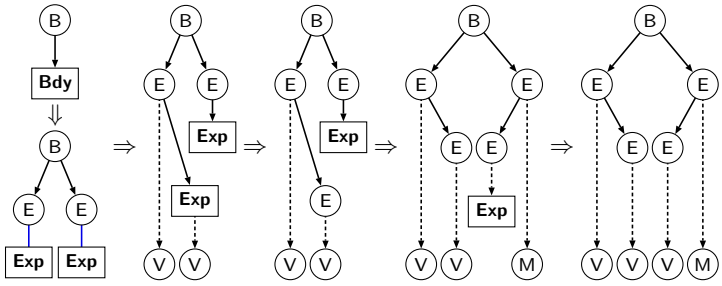

Fig. 3. Variable substitutions generating a method body

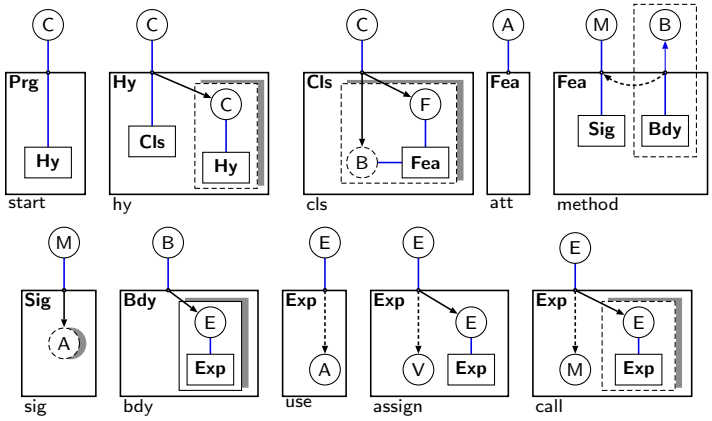

Fig. 4. Star rules generating program trees

methods in a program graph is not bounded, but grows with the size of the graph, this is not possible.

This observation, however, gives a clue how this limitation can be overcome. We make stars, star rules, and variable substitutions *adaptive* wrt. the numbers of border nodes in stars, as proposed in [7]. In Figure 5, the star rule bdy on the left is adaptive: The node d is a *multiple node* (which is drawn with a shade, like the multiple subgraph in Figure 2), representing a set of clone nodes. The clones of d are the declarations visible in the body[1] A clone of a graph X is denoted as $X\frac{v}{k}$, where v is a multiple node or subgraph, and k the number of its clones. On the right, a schematic view of the clones $\mathsf{bdy}\frac{\mathrm{e}}{k}\frac{\mathrm{d}}{n}$ of the adaptive star rule bdy is given, for $k > 0$, $n \geqslant 0$. When all multiple nodes and subgraphs in a rule are cloned, the result is an ordinary star rule, but every adaptive rule may have an infinite number of clones.

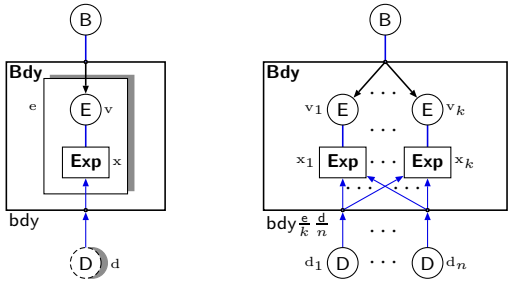

Fig. 5. An adaptive star rule for bodies with clones

[1] The type D has subtypes K, V, and M representing declarations of constants, variables, and methods. We silently assume that types in rules may be replaced by subtypes.

Let $\ddot{\Sigma}$ denote the sets of all possible clones a set Σ of adaptive star rules. Then, an *adaptive star grammar* $\Gamma = \langle \mathcal{G}(\mathcal{Y}), \mathcal{Y}, \Sigma, Z \rangle$ has the same components as a star grammar, where the syntactic variables in $\mathcal{Y}$ and the rules in Σ are adaptive, but the start variable Z is not. The adaptive star grammar Γ generates the language

$$\ddot{\mathcal{L}}(\Gamma) = \{G \in \mathcal{G} \mid Z \Rightarrow^*_{\ddot{\Sigma}} G\}$$

The set of cloned star rules $\ddot{\Sigma}$ is infinite. It has been shown in [7] that this gives adaptive star grammars greater generative power than grammars based on hyperedge [17] or node replacement [13], but are still parseable. The adaptive

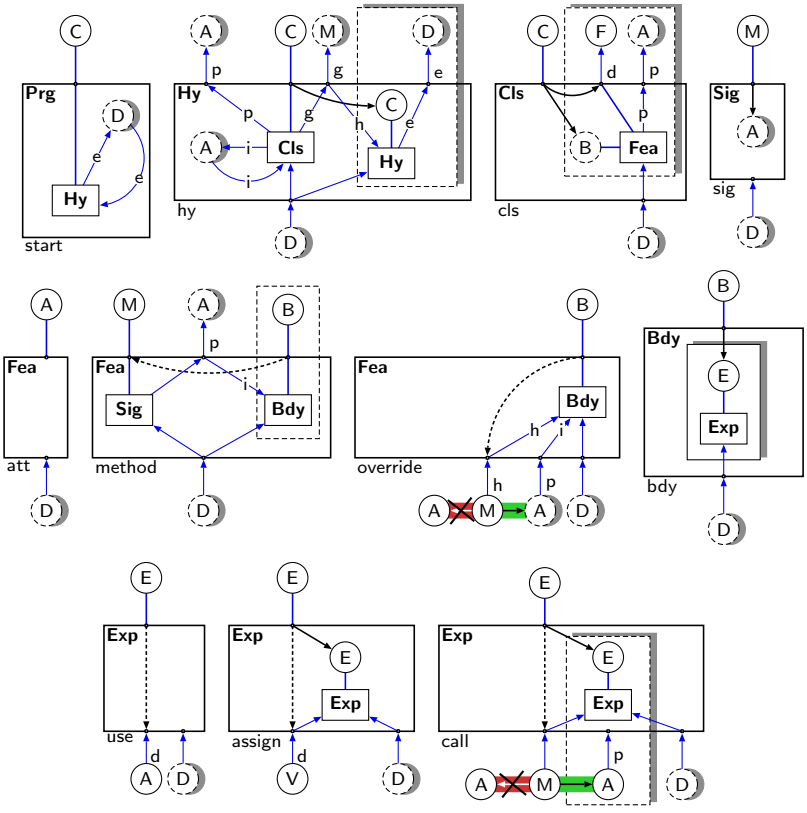

Fig. 6. The set PG of adaptive star rules generating program graphs

rules in Figure 6 generate all program graphs. Since space does not permit to explain all rules, we refer the reader to [8] for a thorough explanation. Roughly, the rules start, hy. and cls generate the class hierarchy of a program, collecting the declarations contained in the classes, and distributing them according to the visibility rules: If a declaration is global, it is visible in the entire hierarchy; if it is heritable, it is visible in the class and all its subclasses, if it is internal, it

is only visible inside the class itself. The features of a class are either attributes (variables or constants), or new methods, or bodies overriding some inherited method. Expressions just represent data flow: the use of some visible attribute, or the assignment of the value of an expression to a variable, or the call of a method with expressions as actual parameters. The full grammar for program graphs, with control flow in method bodies, typing, and more sophisticated visibility rules, appeared in Niels Van Eetvelde's PhD thesis [31].

Application Conditions. Adaptive star rules do not suffice to capture all properties of program graphs. In a call, e.g., the number of actual parameters must match the formal parameters of the method being called. If a program contained $m \geqslant 0$ methods, this would require to collect m different sets $F_1, \ldots, F_m$ of formal parameters in distinct multiple border nodes of a variable like **Exp**. However, the number m is not bounded, whereas there are only finitely many edge types. So adaptive stars do not suffice to distinguish all parameter sets $F_1, \ldots, F_m$.

Our solution is to extend adaptive star rules with *positive* and *negative application conditions* as in [11]. A *conditional adaptive star* rule has the form $\langle A, N\rangle \to Y ::= S$ where the graphs $A, N \in \mathcal{G}$ may share nodes with Y. For some clone $\langle \ddot{A}, \ddot{N}\rangle \to \ddot{Y} ::= \ddot{S}$ of this rule, a variable substitution $G[X/_m \ddot{S}]$ is only defined if the isomorphism $m\colon \ddot{Y} \to X$ can be extended to the positive application condition $\ddot{A}$, and cannot be extended to the negative application condition $\ddot{N}$.

In PG, application conditions are used as follows: The syntactic variables are attached, not only to all visible declarations, but also to all formal parameters of all methods. In rule call, the edge from M to A underlaid in green is a positive condition, collecting formal parameters of M, and the edge from M to the other A-node underlaid in red is a negative condition, forbidding the existence of further formal parameters of M. Then the multiple subgraph makes sure that an actual parameter will be generated for every formal parameter of the method being called.[2]

The adaptive star grammar in Figure 6 is harder, both to develop, and to understand, than a class diagram for program graph.

However, the definition of a graph language by adaptive star grammars is inductive. For instance, the rule $\mathsf{bdy}\frac{\mathrm{e}}{k}\frac{\mathrm{d}}{n}$ in Figure 5 expresses the following inductive property: If $E_1, \ldots, E_k$ are expression graphs, where every graph E_i has a root v_i, and uses the same set $\{d_1, \ldots, d_n\}$ of declarations, adding a B-node as a root, with edges to all nodes $v_1, \ldots, v_k$ yields a body graph using the same declarations. This allows to prove properties of adaptive star languages by induction over the rules, and to define operations on these languages by structural recursion over the rules.

Adaptive star rules have further properties that will be useful in the next section.

[2] In rule override, a similar application condition makes sure that all formal parameters of a method are visible as variables within its body.

Lemma 1 (Commutativity and Substitutivity [7]). *Consider two consecutive variable substitutions $G[X/_m S][X'/_{m'} S']$. Then the following holds:*

1. *Commutativity: If X' is in G, then $G[X/_m S][X'/_{m'} S'] = G[X'/_{m'} S'][X/_m S]$.*
2. *Substitutivity: If X' is in R, then $G[X/_m S][X'/_{m'} S'] = G[X/_m S[X'/_{m'} S']]$.*

Commutativity and substitutivity are typical for context-free grammars on words or graphs, as discussed by B. Courcelle in [5]. However, adaptive star grammars are not context-free, because they fail to be *universal*: this would mean that every variable X in a graph may be substituted by every rule $Y ::= S$ if the variable symbols at their center nodes are the same. Obviously, this is not always the case for adaptive star rules. For instance, consider a variable X with center labeled **Exp** and a single E-labeled border node, i.e., no declaration node is "visible" to X. Then X (like all graphs containing X) is a blind alley. None of the rules use, assign, call applies to X, because each of these rules requires that an attribute, a variable, or a method is visible to X.

3 Generic Graph Rewriting

When applying a graph rewrite rule to a host graph G, a match $m(P)$ of a graph pattern P in G shall be identified in order to insert a replacement graph R for it. The connection between G and R is established by letting P and R share an interface at which the match $m(P)$ and R shall be glued.

Gluing Rules. A *gluing rule* consists of a pair of star rules $Y ::= P$, $Y ::= R$ using the same variable Y as their *interface*; it is denoted as $t = P \dashrightarrow R : Y$, and applies to a graph G if there exists a context graph C containing a star X with an isomorphism $m\colon Y \to X$ so that $G = C[X/_m P]$. Then, the application of t yields the graph $H = C[X/_m R]$; such a transformation step is denoted as $G \Rightarrow_t H$.[3]

In Figure 1, the grey parts of graphs G and H form pattern and replacement of a gluing rule $\mathsf{pum}' = P \dashrightarrow R : Y$; the interface Y is shown in Figure 7. When drawing rules, we omit the interface, and indicate common border nodes of Y in P and R by attaching the same numbers to them, as done in Figure 1. The figure thus shows a rewrite step $G \Rightarrow_{\mathsf{pum}'} H$, for a schematic context as shown in Figure 8.

Rule pum' captures just some of the situations where the *Pull-up Method* refactoring shall be performed. A general rule for *Pull-up Method* should apply to a class with any number of subclasses (not just two), to methods with any number of parameters (not just one), and to bodies of arbitrary shape. Therefore we extend gluing rules with sub-patterns of arbitrary number or shape.

[3] Rewriting with gluing rules corresponds to double pushout graph transformation [10] with discrete interfaces and injective occurrences.

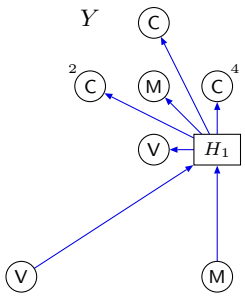

Fig. 7. An interface star

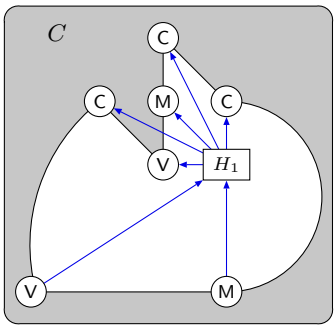

Fig. 8. A context graph

Cloning and Expansion. Adaptive star rules do already provide multiple nodes and subgraphs that may have a variable number of clones. It is easy to allow them in gluing rules as well. According to the other requirement, rule pum′ should apply to method bodies of arbitrary shape, not just to the particular bodies in the example (which are underlaid in dark grey).

We therefore allow that in a *generic rewrite rule* $t : P \dashrightarrow R : Y$, the graphs P and R may not only contain multiple nodes and multiple subgraphs, but also *graph variables* from a set $\mathcal{X}$. Graph variable are expanded to varying subgraphs before applying a rule.

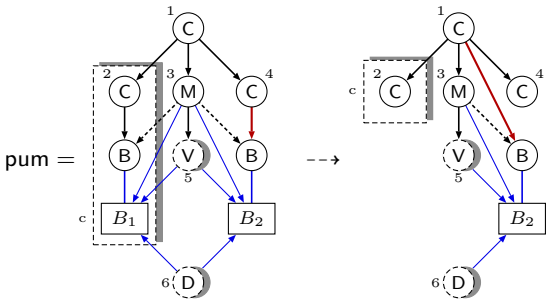

Fig. 9. The generic Pull-up Method rule

Figure 9 shows a generic rule pum for *Pull-up Method*. The rule contains a multiple subgraph c, two multiple nodes 5 and 6, and two graph variables B_1 and B_2. In order to apply a generic rule, its multiple nodes and subgraphs must be cloned, and its variables must be expanded by graphs. For simplicity, we assume that cloning is done before expansion. (In practice, cloning and expansion should be incremental, and interleaved. This is useful for implementing matching of generic rules.)

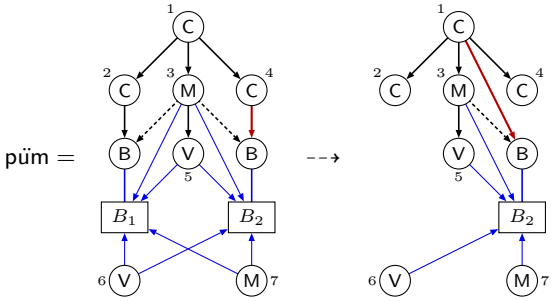

Fig. 10. A clone of the generic Pull-up Method rule

The rule püm in Figure 10 is the clone $\mathsf{pum}\frac{c}{1}\frac{5}{1}\frac{6}{2}$ of the generic rule pum in Figure 9.

A *substitution* σ is a set of star rules with mutually distinct graph variables on their left-hand sides. Since σ actually defines a partial function from variables to graphs, we write $\mathrm{Dom}(\sigma)$ for the set of the left-hand sides in σ, and $\sigma(Y) = S$ for every $Y \in \mathrm{Dom}(\sigma)$ if $Y ::= S$ is a star rule in σ. The *expansion* of a graph G applies σ to each graph variable in G, and is denoted by G^σ. Due to Lemma 1.1, variables can be expanded in any order, even in parallel.[4]

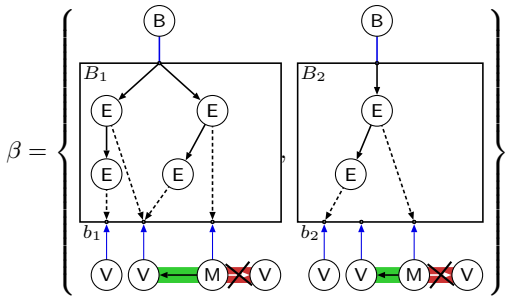

Fig. 11. A shaped substitution for method bodies

The gluing rule pum′ applied in Figure 1 is the expansion $\mathsf{püm}^\beta$ of the clone püm in Figure 10, where β is the substitution defined in Figure 11.

A generic rule $t : P \dashrightarrow R : Y$ applies to a host graph $G = C[X/_m \ddot{P}^\sigma]$, where $\ddot{t} : \ddot{P} \dashrightarrow \ddot{R} : \ddot{Y}$ is a clone of t and σ is a substitution. The application of t yields the graph $H = C[X/_m \ddot{R}^\sigma]$, and we write $G \underset{t,\sigma}{\Longrightarrow} H$.

[4] In this notation, we omit the morphisms from the variables of σ to those of G by which the expansion becomes unique.

Shaped Expansion. Expansions insert arbitrary graphs into a generic rule. This is not only hard to implement, because the substitution has to be found while applying the generic rule, but it is also too general. In our example, rule pum shall be applied to program graphs. Then the graph variables B_1 and B_2 are placeholders for method bodies. So we require that substitutions are shaped according to adaptive star rules.

Let $\Gamma = \langle \mathcal{G}(\mathcal{Y}), \mathcal{Y}, \Sigma, Z \rangle$ be an adaptive star grammar with application conditions. We assume that every graph variable $X \in \mathcal{X}$ refers to a syntactic variable $\lceil X \rceil \in \mathcal{Y}$, which is called the *shape* of X. For some graph $G \in \mathcal{G}(\mathcal{X})$, replacing graph variables by the corresponding syntactic variables yields the shape graph $\lceil G \rceil \in \mathcal{G}(\mathcal{Y})$. A graph $G \in \mathcal{G}(\mathcal{X})$ is *shaped* according to Σ if $Y \Rightarrow^*_\Sigma \lceil G \rceil$ for some $Y \in \mathcal{Y}$; this is written as $\Sigma \vdash G : Y$.

A substitution σ is *shaped* (according to Σ) if $\Sigma \vdash \sigma(Y) : \lceil Y \rceil$ for every graph variable $Y \in \mathrm{Dom}(\sigma)$.

Let the graph variables B_1 and B_2 in rule püm have the shape **Bdy**. Then the substitution β is shaped.

4 Shape Preservation

For our example, it is relevant to ask whether the refactoring of a program graph will always yield a valid program graph again. A simple solution to this problem would be to check this property after every rewrite step. Since the parsing for adaptive star grammar is decidable, this can be done automatically, but is not efficient. It would be better if the rules and the way they are applied could be restricted so that shapes are preserved in any case.

A generic rule $t = P \dashrightarrow R : Y$ is called *shaped* if $\Sigma \vdash \lceil P \rceil : \lceil Y \rceil$ and $\Sigma \vdash \lceil R \rceil : \lceil Y \rceil$. A shaped generic rule $t = P \dashrightarrow R : Y$ applies to a graph G if there exists a shaped context graph C containing a star X with an isomorphism $m \colon Y \to X$ so that $G = C[X/_m P^\sigma]$, for some substitution σ. Then, the application of t yields the graph $H = C[X/_m R^\sigma]$; a shaped rewrite step is denoted as $G \overset{\Sigma}{\Longrightarrow}_t H$.

Theorem 1 (Shape-Correctness). *For any graph G and any shaped generic rule t:*

$$\Sigma \vdash G : Z \wedge G \overset{\Sigma}{\Longrightarrow}_T H \Longrightarrow \Sigma \vdash H : Z$$

Proof Sketch. Assume w.l.o.g. that $t = P \dashrightarrow R : Y$ and $H = C[X/_m \ddot{R}^\sigma]$, for some clone $\ddot{t} = \ddot{P} \dashrightarrow \ddot{R} : \ddot{Y}$ of t, some shaped substitution σ and some isomorphism $m \colon \ddot{Y} \to X$.

The following fact is easily shown by induction over the number of star rule applications, using Lemma 1.2. (*) If $Y \Rightarrow^*_\Sigma S$ and $Y' \Rightarrow^*_\Sigma S'$ with an isomorphism $m \colon Y' \to X'$ for some syntactic variable X' in S, then $Y \Rightarrow^*_\Sigma S[X'/_m S']$.

Since t is shaped, we have $\lceil Y \rceil \Rightarrow^*_\Sigma \lceil R \rceil$. This carries over to the clone, so $\lceil Y \rceil \Rightarrow^*_\Sigma \lceil \ddot{R} \rceil$. Consider those variables $X_1, \ldots, X_n$ in $\ddot{R}$ where every X_i has a $Y_i \in \mathrm{Dom}(\sigma)$ with an isomorphism $m_i \colon Y_i \to X_i$, for $1 \leqslant i \leqslant n$. Then fact (*) yields that $\lceil Y \rceil \Rightarrow^*_\Sigma \left\lceil \ddot{R}[X_1/_{m_1} \lceil \sigma(Y_1) \rceil] \cdots [X_n/_{m_n} \lceil \sigma(Y_n) \rceil] \right\rceil$ and hence $\lceil Y \rceil \Rightarrow^*_\Sigma \lceil \ddot{R}^\sigma \rceil$ by Theorem 1.1.

Again by fact (*), $Z \Rightarrow^*_\Sigma \lceil C \rceil$ and $\lceil Y \rceil \Rightarrow^*_\Sigma \lceil \ddot{R}^\sigma \rceil$ with the isomorphism $m\colon \lceil Y \rceil \to \lceil X \rceil$ for some variable X in C implies $Z \Rightarrow_\Sigma \lceil C[X/_m \ddot{R}^\sigma] \rceil$. Hence $Z \Rightarrow^*_\Sigma \lceil H \rceil$, and H is shaped. □

Let $\mathsf{pum}_{\mathsf{PG}}$ be the shaped generic rule obtained from rule pum in Figure 9 by removing nodes 2 and 4 from the interface. This rule is shaped according to the syntactic variable with center Hy as it appears in the adaptive star rule hy in Figure 6.

Note however that $\mathsf{pum}_{\mathsf{PG}}$ applies only to (shaped) graphs where the superclass 1 has no further features, and the subclasses 2 and 4 have no other features or subclasses either. In other words, the rule is still not general enough if its applied in a shape-preserving fashion.

We can extend rule $\mathsf{pum}_{\mathsf{PG}}$ by further graph variables that are placeholders for further features and subclasses to change this. Unfortunately, the resulting rule is rather complicated.

5 Towards an Implementation

The concepts for shapes and rules outlined in Sections 2 to 4 shall be incorporated into a diagram programming language, dia**plan**. The structure for an implementation of this language is outlined in Figure 12. The basic layer of the dia**plan** system shall be obtained by extending two kernel components of the *Graph rewrite generator* GrGen developed at Universität Karlsruhe [4]. The GrGen *rule compiler* translates sets of rewrite rules into abstract machine code of a *rewrite engine*, using highly sophisticated optimization techniques, making GrGen the fastest graph rewriting tool available.[5] Since the system already

[5] Admittedly, Fujaba [24] is close in efficiency, and even better on some benchmarks, due to a simpler specification language.

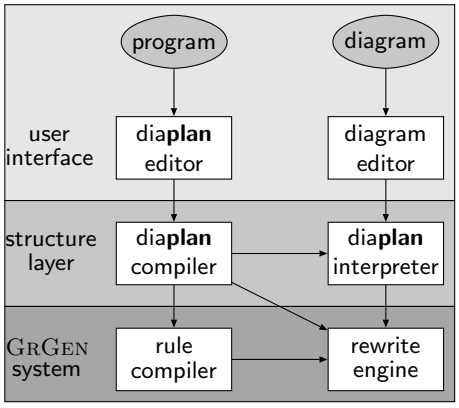

Fig. 12. Outline of the dia**plan** programming system

supports shaped graph variables discussed in Section 4, in form of recursively defined sub-patterns and sub-rules [18], it has "just" to be extended to handle multiple nodes, multiple subgraphs, cloning, and shapes defined by adaptive rather than plain star grammars.

The structure layer of the system shall consist of a compiler that translates classes with shape and predicate definitions into rewrite units. The **diaplan** interpreter shall link these units, call rewrite code recursively, and initiate backtracking for nondeterministic predicates if necessary.

The user interface consists of a diagram editors for **diaplan** programs, and a diagram editor for constructing input graphs, invoking the interpreter, and displaying its results. The editors can be generated with the generator DiaGen for diagram editors developed M. Minas [22, 23].

6 Conclusions

Refactoring shows that graph transformation can be useful for specifying software models and model transformations. Graph grammars, in particular adaptive star grammars with application conditions, can capture the shape of models precisely. Thanks to multiple nodes, multiple subgraphs, and graph variables, generic graph rewrite rules allow to specify refactorings like Pull-up Method refactoring by a single rule. It is even possible to define criteria under which applications of a generic rule will always stay within the language of the host graph, even if these rules may become quite complicated.

Some work related to the concepts shown in this paper shall be mentioned here. *Shape analysis* is about specification and verification of invariants for pointer structures in imperative programming languages, e.g., whether a data structure is a leaf-connected tree. Often, logical formalisms are used for this purpose [29]. When specifying models by meta-models, e.g., in Uml, object constraint logic (ocl) is used for properties that are beyond the descriptive power of class diagrams. Plump et al. have proposed *graph reduction grammars* for shape specification [2]. There is a close relationship between graph languages and (monadic second-order) logic [6], which has not yet been studied for reduction graph grammars or adaptive star grammars, however. The concept of multiple nodes in graph rewrite rules dates back to Progres [30], and is used in several other systems. We have extended it to multiple subgraphs. Graph variables have been implemented, under the name of recursive sub-patterns and sub-rules, in the extension [18] of the GrGen system [4]. Viatra2 [3] seems to be the only other system implementing recursive sub-patterns.

Several questions and challenges remain for future research. Fundamental properties of generic and shaped generic rewrite rules have still to be investigated, such as parallel and sequential independence of rewrite steps, and critical pairs of rewrite rules, which are prerequisites to study confluence of (shaped) generic graph rewrite. Another topic is termination.[6] For our example, analy-

[6] Generic graph transformation could be defined as an instance of adhesive high-level replacement, a categorical axiomatization of graph transformation [9]. Then criteria

sis of these properties would allow to indicate which refactorings could become applicable after a particular refactoring has been performed. And—last but not least—the system outlined in Section 5 has to be implemented.

Acknowledgment. This paper describes the outcome of my research while I am working with Bernd Krieg-Brückner at Universität Bremen. This work was only possible since Bernd allowed my to continue my original line of research – next to my "daily work", the management of an ESPRIT project [19], teaching (programming languages and compiler construction), and administration of his group at Universität Bremen. I have been lucky – and are still happy – to get this freedom and generous support. *Danke schön, Bernd!*

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, Cambridge, UK, 1998.
2. A. Bakewell, D. Plump, and C. Runciman. Specifying pointer structures by graph reduction. *Mathematical Structures in Computer Science*, 2009. Accespted for publication.
3. A. Balogh and D. Varró. Pattern composition in graph transformation rules. In *European Workshop on Composition of Model Transformations*, Bilbao, Spain, July 2006. See also http://viatra.inf.mit.bme.hu/update/R2.
4. J. Blomer and R. Geiß. GrGen.net: A generative system for graph-rewriting, user manual. www.grgen.net, 2008.
5. B. Courcelle. An axiomatic definition of context-free rewriting and its application to NLC rewriting. *Theoretical Computer Science*, 55:141–181, 1987.
6. B. Courcelle. The expression of graph properties and graph transformations in monadic second order logic. In Rozenberg [27], chapter 5, pages 313–400.
7. F. Drewes, B. Hoffmann, D. Janssens, and M. Minas. Adaptive star grammars and their languages. Technical Report 2008-01, Departement Wiskunde-Informatica, Universiteit Antwerpen, 2008.
8. F. Drewes, B. Hoffmann, and M. Minas. Adaptive star grammars for graph models. In H. Ehrig, R. Heckel, G. Rozenberg, and G. Taentzer, editors, *4th International Conference on Graph Transformation (ICGT'08)*, number 5214 in Lecture Notes in Computer Science, pages 201–216. Springer, 2008.
9. H. Ehrig and K. Ehrig. An overview of formal concepts for model transformations based on typed attributes graph transformation. *Electronic Notes in Theoretical Computer Science*, 152(4), 2006. Proc. Graph and Model Transformation Workshop (GraMoT'05).
10. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs on Theoretical Computer Science. Springer, 2006.
11. H. Ehrig and A. Habel. Graph grammars with application conditions. In G. Rozenberg and A. Salomaa, editors, *The Book of L*, pages 87–100. Springer, Berlin, 1986.

for these properties come "for free". However, such an instance would be too "coarse" to provide good results for confluence.

12. H. Ehrig and J. Padberg. Graph grammars and petri net transformations. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 496–536. Springer, 2003.
13. J. Engelfriet and G. Rozenberg. Node replacement graph grammars. In Rozenberg [27], chapter 1, pages 1–94.
14. 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*. World Scientific, Singapore, 1999.
15. M. Fowler. *Refactoring—Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, Reading, MA, 1999.
16. GT-VMT: International workshops on *Graph Transformation and Visual Modelling Techniques*. http://www.cs.le.ac.uk/events/gtvmt09/, 2000–2009.
17. A. Habel. *Hyperedge Replacement: Grammars and Languages*. Number 643 in Lecture Notes in Computer Science. Springer, 1992.
18. B. Hoffmann, E. Jakumeit, and R. Geiß. Graph rewrite rules with structural recursion. In M. Mosbah and A. Habel, editors, *2nd Intl. Workshop on Graph Computational Models (GCM 2008)*, pages 5–16, 2008.
19. B. Hoffmann and B. Krieg-Brückner, editors. *Program Development by Specification and Transformation*, number 680 in Lecture Notes in Computer Science. Springer, 1993.
20. T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour-preserving transformation. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *First International Conference on Graph Transformation (ICGT'02)*, number 2505 in Lecture Notes in Computer Science, pages 286–301. Springer, 2002.
21. T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal on Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.
22. M. Minas. *Specifying and Generating Diagram Editors*. Habilitationsschrift, Universität Erlangen, 2001. [In German].
23. M. Minas. DiaGen/DiaMeta: A diagram editor generator. www.unibw.de/inf2/DiaGen, 2008.
24. J. Niere and A. Zündorf. Using Fujaba for the development of production control systems. In M. Nagl, A. Schürr, and M. Münch, editors, *Int'l Workshop on Applications of Graph Transformations with Industrial Relevance (*Agtive*'99), Selected Papers*, number 1779 in Lecture Notes in Computer Science, pages 181–191. Springer, 2000.
25. D. Plump. Term graph rewriting. In Engels et al. [14], chapter 1, pages 3–102.
26. P. Prusinkiewicz, M. Hammel, J. Hanan, and R. Měch. Visual models of plant development. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3: Beyond Words, chapter 9, pages 535–597. Springer, 1999.
27. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations*. World Scientific, Singapore, 1997.
28. G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, volume 1: Word, Language, Grammar. Springer, 1997.
29. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, 1998.
30. A. Schürr, A. Winter, and A. Zündorf. The Progres approach: Language and environment. In Engels et al. [14], chapter 13, pages 487–550.

31. N. Van Eetvelde. *A Graph Transformation Approach to Refactoring.* Doctoral thesis, Universiteit Antwerpen, May 2007.