Cloning and Expanding Graph Transformation Rules for Refactoring

Berthold Hoffmann [1,4]

Technologiezentrum Informatik
Universität Bremen
D-28334 Bremen, Germany

Dirk Janssens [2,4] Niels Van Eetvelde [3]

Departement Wiskunde & Informatica
Universiteit Antwerpen
B-2020 Antwerpen, Belgium

Abstract

Refactoring is a software engineering technique that aims at enhancing the structure of object-oriented software while preserving its behavior. Several authors have studied how graph transformation can be used to specify refactoring, because such specifications are more precise and can thus, in principle, easier be verified to preserve a program's behavior. It has turned out that "standard" ways of graph transformation do not suffice to define refactoring: their expressive power must be increased if they shall be useful in this application area. Two mechanisms have been proposed so far: one for cloning, and one for expanding nodes by graphs. However, the mechanisms and notations needed are rather complex. In this paper we provide, in the context of double pushout graph transformation, a more elegant and intuitive description. It is based on a notion of rule instantiation, where the instantiation transforms rule schemes into rule instances by cloning and expansion. The power of the technique is demonstrated by an application to two well-known refactoring operations.

Key words: object-oriented programming, refactoring, graph transformation, variables

[1] Email: hof@tzi.de
[2] Email: Dirk.Janssens@ua.ac.be
[3] Email: Niels.VanEetvelde@ua.ac.be
[4] The authors are supported by the ESPRIT Working Group *Syntactic and Semantic Integration of Visual Modeling Techniques* (SeGraVis).

1 Introduction

Although graph transformation provides an obvious way to study the manipulation of discrete structures, their application to concrete problems often require the specification of large sets of similar transformation rules, and hence there is a need for specific mechanisms to support such specifications. The aim of this paper is to present two such mechanisms, cloning and expansion, using the problem of formally describing refactoring operations as a test case. Both mechanism have been introduced before, but in a technically complex and unsatisfactory way.

Refactorings are software transformations that restructure object-oriented programs while preserving their behavior [6,10,13]. The key idea is to redistribute instance variables and methods across the class hierarchy in order to prepare the software for future extensions. If applied well, refactorings improve the design of software, make software easier to understand, help to find bugs, and help to program faster [6]. Although it is possible to refactor manually, tool support is considered crucial. Tools such as the *Refactoring Browser* support a semi-automatic approach [14], which has also been adopted by industrial strength software development environments. These tools rely on a straightforward implementation of each refactoring based on a natural language description like the ones in [6]. Such descriptions are however ambiguous. In [9] and [1] graph transformation was proposed as a formalism to express refactorings and prove that they have properties.

An important potential advantage of graph transformation is that rules may yield a concise visual representation of complex transformations. Unfortunately, traditional graph transformation rules lack the expressiveness needed to specify a refactoring using a single graph transformation rule. This could be resolved by using controlled graph transformation, but the danger is that one ends up with a description where most of the complexity is in the control structure, i.e. one that resembles a traditional imperative program instead of a declarative specification. Van Eetvelde and Janssens [16] proposed to solve this problem by adding graph variables and a cloning mechanism to the rules. This allowed the specification of a complete set of basic refactorings using only one or a few graph transformation rules [17]. However, they use a complex yet powerful mechanism where both nodes and edges and their connections can be substituted with graphs. Hoffmann [8] distinguished three different types of variables in graph transformation rules: graph variables, attribute variables and cloning variables. He also provided a mechanism that instantiates a rule containing one kind of these variables. Most refactorings, however, require the use of several kinds of variables in a single rule. In this paper we develop a formalism that allows to define *rule schemes*, which may contain any kind of variables, and *instantiate* them to concrete rules.

The paper is structured as follows: In the next section we briefly summarize basic graph transformation, and illustrate why this is not sufficient to specify

refactoring. In Sections 3 we define the cloning and expansion mechanism, and show another refactoring. We conclude with some indications of related and future work in Section 4.

2 Basic Graph Transformation

This section briefly recalls double pushout graph transformation [4]. We use this approach as a basis because it is widely used and has a rich theory.

Definition 2.1 [Graph] A *graph* $G = \langle \dot{G}, \bar{G}, s_G, t_G \rangle$ consists of disjoint finite sets $\dot{G}$ of *nodes* and $\bar{G}$ of *edges*, and of *source* and *target functions* $s_G, t_G \colon \bar{G} \to \dot{N}$. A *morphism* $m \colon G \to H$ between directed graphs G and 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 and targets, i.e., $s_H \circ \bar{m} = \dot{m} \circ s_G$ and $t_H \circ \bar{m} = \dot{m} \circ t_G$.

Let $\Sigma = \langle \dot{\Sigma}, \bar{\Sigma}, \mathrm{s}_\Sigma, \mathrm{t}_\Sigma \rangle$ be a fixed *type graph*, specifying *node types*, *edge types*, *source types*, and *target types*, respectively. Then a graph G, together with a morphism $\ell_G \colon G \to \Sigma$ is a *labeled graph (over Σ)*.[5] A morphism $m \colon G \to H$ between labeled graphs is *labeled* if $\ell_H \circ m = \ell_G$.

All graphs and morphisms used henceforth are silently assumed to be labeled. An edge in a graph is said to be *incident* with its source and target nodes, and makes these nodes *adjacent* to each other.

Definition 2.2 [Graph Transformation] A (*graph transformation*) *rule* $t = (L \leftarrow I \rightarrow R)$ consists of two injective morphisms $I \to L$ and $I \to R$.

We say that t *transforms* a graph G to a graph H, written $G \Rightarrow_t H$, if there is a morphism $I \to C$ with two pushouts

$$\begin{array}{ccccc} L & \longleftarrow & I & \longrightarrow & R \\ {\scriptstyle m}\downarrow & & \downarrow & & \downarrow \\ G & \longleftarrow & C & \longrightarrow & H \end{array}$$

in the category of labeled graphs and labeled graph morphisms.

In a rule t as above, we assume that its interface I is a subgraph of its left hand side L and of its right hand side R, and that the morphisms are inclusions. This allows t to be represented as a *rule graph* $L \cup R$ that can itself be subject to graph transformation.

Provided we find a *match* of t in G, (i.e., a morphism $m \colon L \to G$ satisfying the *gluing condition*), a transformation $G \Rightarrow_t H$ can be constructed by replacing the nodes and edges in $m(L \setminus I)$ by fresh copies of those in $R \setminus I$. See [4] for details.

In the rest of this section we consider a rule that specifies a concrete refactoring (*push-down-method*). The example is based on *program graphs*,

[5] Such graphs are often called *typed* [2].

a representation for object-oriented programs that has been developed for the LAN simulation discussed in [9].

Example 2.3 [Program Graphs] In a program graph, software entities (such as classes, variables, methods and method parameters) are represented by *typed nodes*. The node types $\dot{\Sigma} = \{\mathbf{C}, \mathbf{M}, \mathbf{B}, \mathbf{V}, \mathbf{P}, \mathbf{E}\}$ represents the basic kinds of program entities **C**lass, **M**ethod signature, **B**lock structure, **V**ariable, **P**arameter and **E**xpression. The possible relations between these entities are listed in $\bar{\Sigma} = \{\mathsf{l}, \mathsf{i}, \mathsf{m}, \mathsf{t}, \mathsf{e}, \mathsf{ap}, \mathsf{fp}, \bullet, \mathsf{c}, \mathsf{a}, \mathsf{u}, \mathsf{val}\}$: method lookup, inheritance, membership, type, expression, actual parameter, formal parameter, cascaded expression ($\bullet$), call, variable access and update, update value. Method definitions are represented as simplified syntax trees.

Example 2.4 [A Concrete Refactoring Rule] A concrete instance of the *push-down-method* refactoring is shown in Fig. 1. It moves the body of a method (originate) belonging to a class (Node) down to its subclasses (Workstation and PrintServer). The method body is represented by a simplified syntax tree on the left-hand side of the rule, consisting of the node of type **B** and two nodes of type **E**. This subgraph occurs twice on the right-hand side. The concrete class and method names identify the nodes in the rule's interface. In the rest of the paper, we use numbers for indicating the interface.

An obvious drawback of this rule is that it fits only one specific program situation: it cannot be reused for an other method body, and hence, to describe all possible occurrences of *push-down-method* one would need a new rule for each new method body. In order to obtain a precise and concise representation, it would be desirable to have a more abstract rule, that can be instantiated to yield the required concrete rule once it has been decided where (in which class and to which method) *push-down-method* is to be applied (this can be done by providing a class name and a method name as parameters to the abstract rule). As a way to obtain such more abstract rules we propose the introduction of *graph variables*. A node labeled by a graph variable serves as a placeholder for a set of graphs; in the case of our example, we introduce a graph variable β that can be expanded into an arbitrary method body.

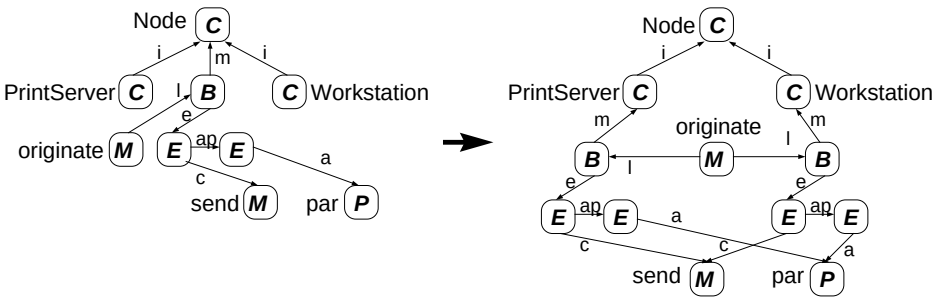

Fig. 1. Rule for a concrete *push-down-method* refactoring

The rule of Fig. 1, however, has a second weakness: even if the structure of the method body would not be fixed, its context is: the two **E**-nodes are only connected to one **M**-node and one **P**-node. In general, a method body can access more method signatures, variables and parameters, and it may also contain any number of local variables that have a type-edge to the corresponding classes. Moreover, it can only copy the method body to exactly two subclasses. If there would be an extra subclass FileServer of Node in the program, it should receive a copy of the method as well. Thus a rule representing the refactoring should provide an additional mechanism where the number of involved subclasses, as well as their adjacent nodes, can vary: again a more abstract rule is needed that can be instantiated into a concrete version that has the required number of subclasses and creates the corresponding number of copies of the method body. Therefore we introduce a second mechanism: *cloning*, which allows one to duplicate a part of the rule the desired number of times.

These concepts are defined in the following section.

3 Graph Transformation for Refactoring

We first extend the type graph by variables, and then define the major concepts proposed in this paper, patterns and rule schemes. In the next two subsections, cloning and expansion of graph variables are considered, starting from the notion of a rule scheme.

Definition 3.1 [Extended Type Graph] We extend the type graph Σ to a *type graph with variables* $\Sigma(X)$ by adding a set $\dot{X}$ of *graph variables* and a set $\bar{X}$ of *tentacle types* that connect graph variables of X to node types of Σ. Furthermore we fix an alphabet Y of *cardinality variables*, disjoint to $\Sigma(X)$.

For this paper, we assume that $\dot{X}$ contains the graph variable β, $\bar{X}$ the tentacle types calls, names, root, types, and Y the cardinality variables u, w, x, y, and z.

Definition 3.2 [Pattern] A *pattern* is a graph G labeled over $\Sigma(X)$, together with a partial *cardinality function* $\#_G \colon \dot{G} \dashrightarrow Y$. A labeled morphism $m \colon G \to H$ between patterns G and H is a *pattern morphism* if for all nodes $n \in \mathrm{Dom}(\#_G)$, $\dot{m}(n) \in \mathrm{Dom}(\#_H)$ and $\#_G(n) = \#_H(\dot{m}(n))$.

Definition 3.3 [Rule Scheme] A *rule scheme* is a rule $s = (L \leftarrow I \rightarrow R)$ where L, I, and R are patterns, and the morphisms are pattern morphisms. We require a rule scheme to be *closed*; this means that every variable from $X \cup Y$ occurring in R must occur in L as well.

Example 3.4 [A Rule Scheme for Refactoring] The *rule scheme* pdm in Fig. 2 defines the *push-down-method* refactoring in full generality. Here the graph variable β represents an arbitrary method body, and its annotation x on the

right hand side indicates the nodes that may be cloned. Relationships between the method body and other entities (e.g. the fact that there is a call to a method or that the method refers to certain names) are represented by tentacles of the β-labeled node. When the rule is applied to a concrete program graph, β and x are bound (to a method body and a natural number, respectively). The former determines how the nodes with label β are to be expanded, and the latter determines how many copies (clones) are needed of the part of the rule that is designated by x. In Fig. 2 the cardinality variables u, w, y, z occur only once (the corresponding nodes belong to the interface, so their occurrences in the left-hand side are identified with those in the right-hand side). Finally, note that cloning is viewed as an operation on a more abstract level than expansion: occurrences of graph variables, such as β, may be cloned, but expanded versions of graph variables do not contain clonable nodes.

A rule scheme s is applied in three steps:

(i) Nodes in the scheme with associated cardinality are *cloned* according to a multiplicity function μ.

(ii) The graph variables in the cloned scheme are *expanded* to graphs according to a substitution γ.

(iii) The so obtained *rule instance* $t = (s^{\mu})^{\gamma}$ is an ordinary rule that is applied, by transformations $G \Rightarrow_t H$ according to Def. 2.2.

Cloning, expansion, and application of rule schemes is defined in Subsections 3.1 to 3.3 below.

3.1 Cloning

Cloning multiplies the nodes with a defined cardinality according to some multiplicity assigned to the cardinality variables. After cloning, the cardinality function of a pattern is entirely undefined, and the pattern is called *cloned*. Nodes whose cardinality is defined, and equals $y \in Y$ are called *y-fold*.

Definition 3.5 [Clone] Let G be a pattern. The *master graph* of a cardinality

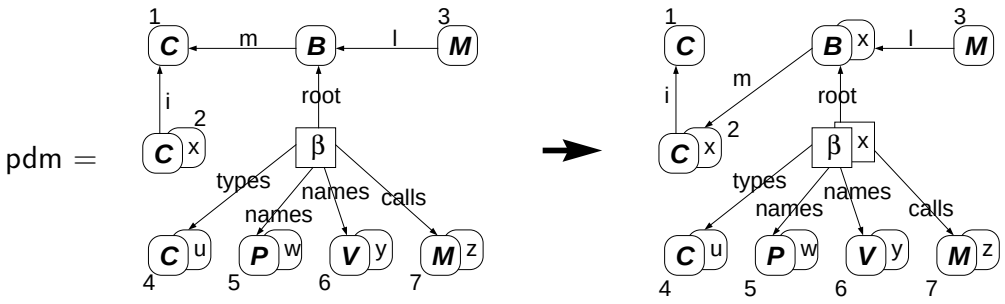

Fig. 2. Rule scheme pdm for the *push-down-method* refactoring

variable $y \in Y$ is the subgraph G_y of G induced by the y-fold nodes in $\dot{G}$, their incident edges, and their adjacent nodes so that the cardinality function is defined for $n \in \dot{G}_y$ with $\#_{G_y}(n) = \bot$ if $\#_G(n) = y$, and $\#_{G_y}(n) = \#_G(n)$ otherwise. The *border graph* G_y^0 is the discrete subgraph of G_y that contains all but the y-fold nodes of G.

For some $k \geqslant 0$, the *k-fold y-clone* G_y^k of a pattern G is obtained by removing the y-fold nodes and their incident edges, and gluing $k \geqslant 0$ disjoint copies of G_y to the corresponding border nodes in G.

Lemma 3.6 (Cloning is Commutative) $(G_x^k)_y^m = (G_y^m)_x^k$. [6]

Lemma 3.6 allows to define a cloning operation.

Definition 3.7 [Cloning] Consider a pattern G containing the set $\{y_1, \ldots, y_n\}$ of cardinality variables, and let $\mu\colon Y \to \mathbb{N}$ be a multiplicity function with $\mu(y_i) = k_i$ for $1 \leqslant i \leqslant n$.

Then the *μ-clone* of a pattern G is the cloned pattern G^μ that is obtained by a cloning sequence

$$G^\mu = (\cdots(G_{y_1}^{k_1})_{y_2}^{k_2}\cdots)_{y_n}^{k_n}$$

The μ-clone s^μ of a rule scheme $s = (L \leftarrow I \rightarrow R)$ is obtained by cloning the rule graph $L \cup R$ and associating every cloned edge and node to that part of s^μ to which its original in s belongs.

Example 3.8 [Cloning] A clone pdm^μ of the *push-down-method* refactoring pdm in Fig. 2, using the multiplicity function μ with $\mu(x) = 2$, $\mu(y) = \mu(u) = 0$ and $\mu(z) = \mu(w) = 1$ is shown in Fig. 3.

3.2 Expansion

Expansion expands the variable nodes in cloned patterns by graphs. Nodes are qualified as *variable* if they are labeled by $\dot{X}$. In particular, $\dot{G}_x$ denotes the subset of variable nodes in a pattern G that is labeled with $x \in \dot{X}$.

[6] Here, as in Def. 3.15 below, equality is only relevant "up to isomorphism".

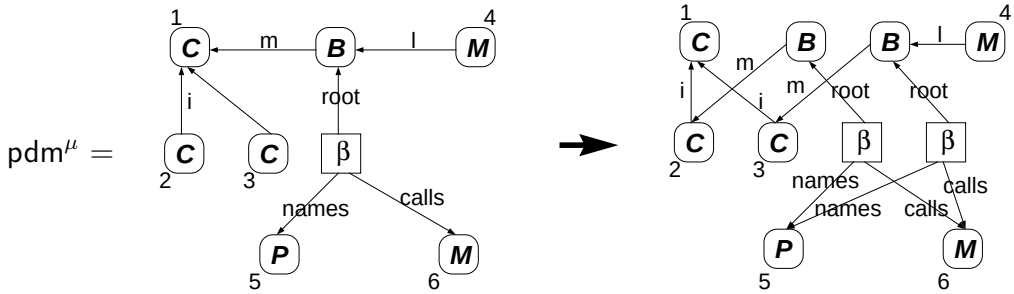

Fig. 3. A cloned scheme pdm^μ of the rule scheme pdm in Fig. 3

After expansion, cloned pattern contain no variables, and are ordinary graphs over Σ.

Expansion of graph variables is based on a simple form of graph transformation, called handle replacement.

Definition 3.9 [Handles] The set $\mathcal{H}_x$ of *handles* of a graph variable $x \in X$ contains the cloned patterns H such that $\dot{H}$ contains one *center node* n labeled with x, and the set of nodes for which each node in this set is connected to n with exactly one tentacle. The discrete subgraph of a handle H containing all the nodes in $\dot{H}$ except the center n is called the *border graph* of H, and denoted by H°.

Definition 3.10 [Handle Replacement] A rule $r = (L \leftarrow I \rightarrow R)$ over cloned patterns is a *handle replacement rule* for a graph variable $x \in X$ if $L \in \mathcal{H}_x$, $I = L^\circ$, and if R is expanded.

We say that a transformation step $G \Rightarrow_r H$ via a handle replacement rule r performs a *handle replacement*.

Handle replacement shall be used to apply the same rule to all variable nodes with the same label so that the result is uniquely determined. In order to achieve this, the occurrences of these variable nodes have to be homogeneous so that a single rule applies to all of them.

Definition 3.11 [Homogeneous Pattern] A variable x is *homogeneous* in a pattern G if all variable nodes $v \in \dot{G}_x$ occur as centers of handles with equal number and type of tentacles.

A pattern is *homogeneous* if all its variables are homogeneous.

Handle replacement is not confluent, as the tentacles of variables may have the same label. E.g., cloning the rule scheme in Fig. 2 with a multiplicity $\mu'(z) = 2$ yields a cloned scheme similar to that in Fig. 3 where the variable β has two tentacles labeled calls. A handle replacement rule for β could then be applied with two different matches, yielding different results in general. Therefore we equip patterns with a correspondence relation that makes sure that the tentacles of different variables of the same name have a one-to-one correspondence to each other.

Definition 3.12 [Straight Pattern] A *straight pattern* consists of a homogeneous pattern G, and an equivalence relation $\sim_G$ on the tentacles in G, called *correspondence*, such that the following holds:

(i) Different tentacles of a variable node belong to different equivalence classes of $\sim_G$.

(ii) For different variable nodes v and v' with the same label, every tentacle e of v satisfies $e \sim_G e'$ for exactly one tentacle e' of v' so that e' has the same label, and its target node has the same label and cardinality as e.

The notions of homogeneity and straightness apply to patterns with car-

dinalities as well. The cloning operation of Def. 3.5 can be extended so that it keeps track of the correspondence relations and inserts new ones if variables or their adjacent nodes are cloned:

- For all y-fold variable nodes in a master graph G_y, the clones of their tentacles in G_y^k correspond to each other.
- The other correspondences in G_y are transferred to every copy of G_y in G_y^k.

Definition 3.13 [Expansion Step] Let G be a straight pattern, and let $r = (L \leftarrow I \rightarrow R)$ be an expansion rule. Then H is the *expansion step* of x by r in G, written $G \rightrightarrows_x^r H$, if there is a sequence of expansion steps applying r to all nodes $v \in \dot{G}_x$ so that their matches respect $\sim_G$, i.e., whenever morphisms $m, m' \colon L \rightarrow G$ map a tentacle $\tilde{e} \in \bar{L}$ onto distinct tentacles e and e' in $\bar{G}$, respectively, then $e \sim_G e'$.

Substitutions map variables onto handle replacement rules that can be applied to the variables occurring in a cloned pattern.

Definition 3.14 [Substitution] A *substitution* is a function γ that maps the graph variables X onto one of their expansion rules. A substitution γ *suits* a graph G if the rules $\gamma(x)$ match every variable node named x in G, for all $x \in X$.

The handles of variables overlap only in their border nodes. This makes expansion steps parallel-independent of each other so that these steps yield a unique result, independent of the order in which they are applied.

Definition 3.15 [Expansion] Let γ be a substitution suiting a cloned pattern G. The γ-*expansion* of G is the graph G^γ obtained by the expansion steps

$$G = G_0 \rightrightarrows_{x_1}^{\gamma(x_1)} G_1 \rightrightarrows_{x_2}^{\gamma(y_2)} \cdots \rightrightarrows_{x_n}^{\gamma(y_n)} G_n = G^\gamma$$

where $\{x_1, x_2, \ldots, x_n\} \subseteq \dot{X}$ is the set of graph variables occurring in G.

Expansion can be lifted to cloned rules $t = (L \leftarrow I \rightarrow R)$ by expanding the rule graph $L \cup R$ according to γ, and associating every node and edge that is inserted in $(L \cup R)^\gamma$ to that part of the rule t^γ to which the variable inserting it belongs.

Example 3.16 [Expansion] Fig. 4 below shows a substitution for the variable β that suits the cloned scheme pdm^μ in Fig. 3. Expanding pdm^μ with γ yields the rule in Fig. 1.

3.3 Graph Transformation with Instantiation

We now can define graph transformation with variables.

Definition 3.17 [Graph Transformation with Variables] Let G be a graph, and $s = (L \leftarrow I \rightarrow R)$ be a straight rule scheme. Then s *transforms* G into a graph H, written $G \Rrightarrow_s H$, if there is a multiplicity function μ, and a substitution γ suiting s^μ so that $G \Rightarrow_t H$ for the rule $t = (s^\mu)^\gamma$.

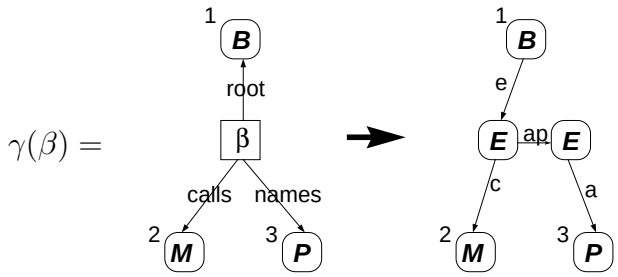

Fig. 4. A substitution suiting the cloned scheme pdm$^{\mu}$ in Fig. 3

Def. 3.17 is not operational. We cannot generate clones and expansions of a rule scheme s until we find an instance $t = (s^{\mu})^{\gamma}$ that applies to the graph G that shall be transformed. For, every rule scheme has infinitely many clones and expansions. Instead, transformation has to start from a match of the *kernel* of s (without any multiplied and variable nodes), and determine the suitable multiplicities and substitutions incrementally, by matching the multiple nodes and graph variables with the graph.

Example 3.18 [Applying pdm] The rule scheme pdm shown in Fig. 2 can be matched as follows:

- Match the kernel nodes 1, 2, and the **B**-node in between.
- Match all subclasses of node 1, defining $\mu(x)$ and the clones of node 2.
- Match the method body by following all edges starting from the **B**-node, and stopping at the nodes representing the types, names and methods used in; this defines $\gamma(\beta)$, and the multiplicities for the variables u, w, y, and z.
- Clone the expansion $\gamma(\beta)$ and the **B**-node $\mu(x)$ times.

Matching is deterministic once the class and the method to be pushed (nodes 1 and 2) have been chosen.

Cloning and expansion allow another refactoring, *inline-method*, to be described by a single rule scheme.

Example 3.19 [The Inline Method refactoring] The rule scheme im in Fig. 5 specifies the *inline-method* refactoring that replaces a method call by a copy of its body. Every parameter of the method is turned into a local variable, that gets assigned the value of the actual parameter of the call. The assignment is represented by an **E**-node connected by an u edge to a node of type **V** and by a val edge to the node representing the r-value of the assignment. Finally, the method body is copied and the references to the formal parameters are replaced by references to the variables. The original method is then deleted (the refactoring precondition assures that no other call to the method exists). One cardinality variable x is needed for the number of parameters. Three others (u, w, and y) stand for the methods, variables and types used in the method body. Note that the expansion for β has the same meaning as in *push-down-method*.

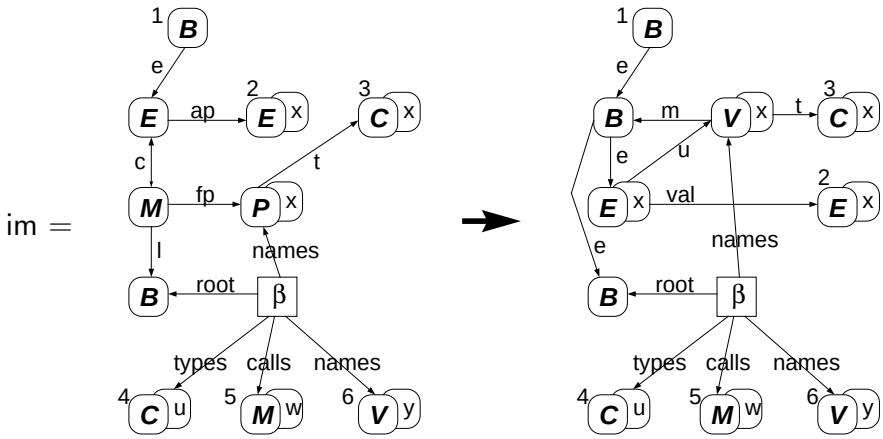

Fig. 5. rule scheme im for the *inline-method* refactoring

The substitution γ in Fig. 4 is suited for a cloned scheme im^ν, but only if the multiplicity function ν is defined with $\nu(u) = 0$ and $\nu(w) = \nu(y) + \nu(x) = 1$; otherwise, the occurrence of β in im^ν has too few or too many tentacles.

4 Conclusions

The instantiation of rule schemes by cloning and expansion makes graph transformation more expressive. This is indispensable to describe refactoring operations in a declarative way. In this paper the authors have joined their earlier work: The rule schemes presented here are much simpler than those proposed in [16], and cloning is more tightly integrated with expansion than in [8]. And, the instantiation now yields rules in the double-pushout approach [4], a standard way of graph transformation with a rich theory.

The set nodes of PROGRES [15] correspond to the cloning of single nodes. In [11], hyperedges (nodes with a fixed number of tentacles) have been used as graph variables for the first time. For refactoring operations like *push-down-method* and *inline-method*, however, we need to clone subgraphs, not just nodes, and graph variables with a varying number of tentacles. In the case study [17], a representative set of elementary refactoring operations from the list of [6] could be described with rules schemes using these concepts and only minimal control flow. In this paper we did not consider all graph transformation concepts that are relevant to refactoring. *Forbidden subgraphs* and *negative application conditions* have already been discussed in [9]; some other concepts should still be added to the formalism. *Attributes* may be used to determine values like numbers or strings, by expressions with (attribute) variables that are evaluated according to some given data types [12]. It is necessary to clearly define the set of possible substitutions for a variable in a refactoring rule. Shape grammars [**?**] are a promising candidate for this. For describing complex refactoring strategies, one needs to *name* and *parameterize*

refactoring operations, and to *control* the way they are applied. The diagram programming language **Diaplan** [7] shall provide all these concepts.

References

[1] P. Bottoni, F. Parisi-Presicce, and G. Taentzer. Specifying integrated refactoring with distributed graph transformations. In M. Nagl et al., editors, *Applications of Graph Transformation with Industrial Relevance (*AGTIVE*'03)*, number 3062 in Lecture Notes in Computer Science, pages 220–242. Springer, 2004.

[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. Cuny et al., editors. *Graph Grammars and Their Application to Computer Science*, number 1073 in Lecture Notes in Computer Science. Springer, 1996.

[4] H. Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus et al., editors, *Graph Grammars and Their Application to Computer Science and Biology*, number 73 in Lecture Notes in Computer Science, pages 1–69. Springer, 1979.

[5] H. Ehrig et al., editors. *2nd Int'l Conference on Graph Transformation (ICGT'04)*, number 3256 in Lecture Notes in Computer Science. Springer, 2004.

[6] M. Fowler. *Refactoring—Improving the Design of Existing Code.* Object Technology Series. Addison-Wesley, Reading, MA, 1999.

[7] B. Hoffmann. Abstraction and control for shapely nested graph transformation. *Fundamenta Informaticae*, 58(1):39–56, 2003.

[8] 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.

[9] T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour-preserving transformation. In A. Corradini et al., editors, *First International Conference on Graph Transformation (ICGT'02)*, number 2505 in Lecture Notes in Computer Science, pages 286–301. Springer, 2002.

[10] W. F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1992.

[11] D. Plump and A. Habel. Graph unification and matching. In Cuny et al. [3], pages 75–89.

[12] D. Plump and S. Steinert. Towards graph programs for graph algorithms. In Ehrig et al. [5], pages 128–143.

[13] D. Roberts. *Practical Analysis for Refactoring.* PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[14] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[15] A. Schürr, A. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In G. Engels et al., 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.

[16] N. van Eetvelde and D. Janssens. Extending graph rewriting for refactoring. In Ehrig et al. [5], pages 399–415.

[17] N. Van Eetvelde and D. Janssens. Refactorings as graph transformations. Technical report, University of Antwerp, February 2005. UA WIS/INF 2005/04.