Cloning and Expanding Graph Transformation Rules for Refactoring

Berthold Hoffmann\textsuperscript{1,4}

Technologiezentrum Informatik
Universität Bremen
D-28334 Bremen, Germany

Dirk Janssens\textsuperscript{2,4} Niels Van Eetvelde\textsuperscript{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.

Keywords: object-oriented programming, refactoring, graph transformation, variables

\textsuperscript{1} Email: hof@tzi.de
\textsuperscript{2} Email: Dirk.Janssens@ua.ac.be
\textsuperscript{3} Email: Niels.VanEetvelde@ua.ac.be
\textsuperscript{4} The authors are supported by the ESPRIT Working Group \textit{Syntactic and Semantic Integration of Visual Modeling Techniques (SeGRAVis)}. 

1571-0661/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.entcs.2006.01.014
1 Introduction

Although graph transformation provides an obvious way to formalize the manipulation of discrete structures, their application to concrete problems often requires 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 [7,11,14]. 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 [7]. Although it is possible to refactor manually, tool support is considered crucial. Tools such as the Refactoring Browser support a semi-automatic approach [15], 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 [7]. Such descriptions are however ambiguous. In [10] and [1] graph transformation was proposed as a formalism to express refactorings and to reason about their 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 [18] 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 [19]. However, they use a complex yet powerful mechanism where both nodes and edges and their connections can be substituted with graphs. Hoffmann [9] 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. Section 3 illustrates why this is not sufficient to specify refactoring. In Section 4 we define the cloning and expansion mechanism, and show another refactoring. We conclude with some indications of related and future work in Section 5.

2 Basic Graph Transformation

This section briefly recalls double pushout graph transformation [5]. 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 \( s_G, t_G : \bar{G} \to \dot{G} \). A morphism \( m : G \to H \) between directed graphs \( G \) and \( H \) consists of two functions \( \dot{m} : \dot{G} \to \dot{H} \) and \( \bar{m} : \bar{G} \to \bar{H} \) 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 \( \Sigma = \langle \dot{\Sigma}, \bar{\Sigma}, s_\Sigma, t_\Sigma \rangle \) be a fixed type graph, specifying node types, edge types, source types, and target types, respectively. Then a graph \( G \), together with a morphism \( \ell_G : G \to \Sigma \) is a labeled graph (over \( \Sigma \)).\(^5\) A morphism \( m : G \to H \) 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

\[
\begin{array}{ccc}
L & \leftarrow & I \\
\downarrow & & \downarrow \\
G & \leftarrow & C \\
\downarrow & & \downarrow \\
& & H
\end{array}
\]

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.

\(^5\) Such graphs are often called typed [2].
Provided we find a match of \( t \) in \( G \), (i.e., a morphism \( m : L \rightarrow G \) 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 [5] for details.

## 3 The Push-Down-Method Refactoring

In this section we consider a rule that specifies a concrete refactoring that shall be used as a running example. The example is based on program graphs, a representation for object-oriented programs that has been developed for the LAN simulation discussed in [10].

### 3.1 Program Graphs

Programs are represented by program graphs. In a program graph, software entities (such as classes, variables, methods and method parameters) are represented by typed nodes. The node types \( \Sigma = \{ \text{C, M, B, V, P, E} \} \) represent the basic kinds of program entities Class, Method signature, Block structure, Variable, Parameter and Expression. The possible relations between these entities are listed in \( \Sigma = \{ l, i, m, t, e, \text{ap, fp, •, c, a, u, val} \} \): method lookup, inheritance, membership, (sub)type, expression, actual parameter, formal parameter, cascaded expression (\( \bullet \)), call, variable access and update, update value. These node types and relations can be visualized in a type graph, as depicted in Figure 1. Program graphs are typed over this graph. Method bodies are represented as simplified syntax trees with a root (B-node), connected to different E-nodes that can represent calls, assignments and accessed variables and parameters.

![Fig. 1. The typegraph for program graphs](image)

### 3.2 Expressing Refactorings as Graph Transformation

Expressing software programs as graphs makes it possible to model program transformations (e.g. refactorings) as graph transformations. If the trans-
formation is local, i.e. it has an effect on only a small part of the program graph, then it makes sense to express it as a rule. This is typically the case for refactorings. A concrete instance of the push-down-method refactoring is shown in Fig. 2. The body of method originate is copied from its containing class (Node) to its subclasses (Workstation and PrintServer). The method body, containing a call to a method send, 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.

When applied, the left hand side of this rule is matched against the program graph, and then the rule removes the syntax tree in the left hand side (the shaded part, which is not in the interface) and adds two copies of it as part of the two subclasses.

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 and an expansion concept to replace it by an arbitrary method body.

The rule of Fig. 2, 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. Applying a rule now becomes a two-stage process. For the push-down-method refactoring, this implies that after matching the Node class and the originate method, it is determined that Node has two subclasses (by cloning) and that the syntax tree contains a call to the method send (by expansion).

4 Graph Transformation for Refactoring

Some extra notations are required to define the concepts of the previous section. First we extend the type graph with 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 in detail, starting from the notion of a rule scheme.

**Definition 4.1 [Extended Type Graph]** We extend the type graph Σ to a type graph with variables Σ(\(X\)) by adding a set \(\hat{X}\) of graph variables and a set \(\hat{X}\) of tentacle types that connect graph variables of \(X\) to node types of Σ. Furthermore we fix an alphabet \(Y\) of cardinality variables, disjoint from Σ(\(X\)).

For this paper, we assume that \(\hat{X}\) contains the graph variable \(\beta\), \(\hat{X}\) the tentacle types calls, names, root, types, and \(Y\) the cardinality variables \(u\), \(w\), \(x\), \(y\), and \(z\). The extended typegraph is depicted in Fig. 3.

**Definition 4.2 [Pattern]** A pattern is a graph \(G\) labeled over Σ(\(X\)), together with a partial cardinality function \(#_G: \hat{G} \rightarrow Y\). A labeled morphism \(m: G \rightarrow H\) between patterns \(G\) and \(H\) is a pattern morphism if for all nodes \(n \in \text{Dom}(\#_G), \hat{m}(n) \in \text{Dom}(\#_H)\) and \(#_G(n) = #_H(\hat{m}(n))\).

**Definition 4.3 [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 4.4 [A Rule Scheme for Refactoring] The *rule scheme pdm* in Fig. 4 defines the *push-down-method* refactoring in full generality. Here the graph variable $\beta$ represents an arbitrary method body, and its annotation $x$ on the right hand side indicates that this body will 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 $\beta$-labeled node. When the rule is applied to a concrete program graph, the graph variable $\beta$ and the cardinality variables $x, y, u, w, z$ are bound (to a method body and a set of natural numbers, respectively). The former determines how the nodes with label $\beta$ are to be expanded, and the latter determines for each cardinality variable how many copies (clones) are needed of the part of the rule that is designated to it (e.g. the value for $x$ determines the number of subclasses). In Fig. 4 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 $\beta$, 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 $\mu$.

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

(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 4.1 to 4.3 below.

![Fig. 3. The extended typegraph](image-url)
4.1 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 4.5** [Clone] Let \( G \) be a pattern. The master graph of a cardinality variable \( y \in Y \) is the subgraph \( G_y \) of \( G \) induced by the \( y \)-fold nodes in \( \hat{G} \), their incident edges, and their adjacent nodes so that the cardinality function is defined for \( n \in \hat{G}_y \) with \( \#_{G_y}(n) = \bot \) if \( \#_G(n) = y \), and \( \#_{G_y}(n) = \#_G(n) \) otherwise. The border graph \( G_0^y \) is the discrete subgraph of \( G_y \) that contains all but the \( y \)-fold nodes of \( G \).

For some \( k \geq 0 \), 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 \( k \geq 0 \) disjoint copies of \( G_y \) to the corresponding border nodes in \( G \).

**Lemma 4.6 (Cloning is Commutative)** \( (G_x^k)_y^m = (G_y^m)_x^k \).

Lemma 4.6 allows to define a cloning operation.

**Definition 4.7** [Cloning] Consider a pattern \( G \) containing the set \( \{y_1, \ldots, y_n\} \) of cardinality variables, and let \( \mu: Y \to \mathbb{N} \) be a multiplicity function with \( \mu(y_i) = k_i \) for \( 1 \leq i \leq n \).

Then the \( \mu \)-clone of a pattern \( G \) is the cloned pattern \( G^\mu \) that is obtained by a cloning sequence

\[
G^\mu = \cdots (G_{y_1}^{k_1})^{k_2} \cdots y_n^{k_n}
\]

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

---

[6] Here, as in Def. 4.16 below, equality is only relevant “up to isomorphism.”
Example 4.8 [Cloning] A clone $\text{pdm}^\mu$ of the push-down-method refactoring $\text{pdm}$ in Fig. 4, using the multiplicity function $\mu$ with $\mu(x) = 2$, $\mu(y) = \mu(u) = 0$ and $\mu(z) = \mu(w) = 1$ is shown in Fig. 5.

4.2 Expansion

Expansion expands the variable nodes in cloned patterns by graphs. Nodes are qualified as variable if they are labeled by $\hat{X}$. In particular, $\hat{G}_x$ denotes the subset of variable nodes in a pattern $G$ that is labeled with $x \in \hat{X}$. After expansion, cloned patterns contain no variables, and are ordinary graphs over $\Sigma$.

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

Definition 4.9 [Handles] The set $\mathcal{H}_x$ of handles of a graph variable $x \in X$ contains the cloned patterns $H$ with nodes $\tilde{H} = \{n_0, n_1, \ldots n_k\}$ and tentacles $\bar{H} = \{e_1, \ldots, e_k\}$ such that one node, say $n_0$, is labeled with $x$, and connected to every other node $n_i$ by exactly one tentacle, for $k \geq 0$. The node $n_0$ is called the center node. The discrete subgraph of a handle $H$ containing all the nodes in $\tilde{H}$ except the center node is called the border graph of $H$, and denoted by $H^\circ$.

Definition 4.10 [Handle Replacement] A rule $r = (L \leftarrow I \rightarrow R)$ over cloned patterns is a handle replacement rule for a graph variable $x \in \hat{X}$ 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.

Example 4.11 The rule in Figure 6 shows a handle replacement rule for the syntax tree variable $\beta$. When matched on a pattern graph, it replaces a variable node $\beta$ by the syntax tree shown in the right hand side of the rule.

Fig. 5. A cloned scheme $\text{pdm}^\mu$ of the rule scheme $\text{pdm}$ in Fig. 4
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.

Fig. 6. A substitution suiting the cloned scheme $\text{pdm}^{\mu}$ in Fig. 5

**Definition 4.12** [Homogeneous Pattern] A variable $x$ is *homogeneous* in a pattern $G$ if all variable nodes $v \in \tilde{G}_x$ 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. 4 with a multiplicity $\mu'(z) = 2$ yields a cloned scheme similar to that in Fig. 5 where the variable $\beta$ has two tentacles labeled calls. A handle replacement rule for $\beta$ 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 4.13** [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 cardinalities as well. The cloning operation of Def. 4.5 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 \( G^k_y \) correspond to each other.
- The other correspondences in \( G_y \) are transferred to every copy of \( G_y \) in \( G^k_y \).

**Definition 4.14** [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 \( G \xrightarrow{r} x H \), if there is a sequence of expansion steps applying \( r \) to all nodes \( v \in \hat{G}_x \) so that their matches respect \( \sim_G \), i.e., whenever morphisms \( m, m': L \rightarrow G \) map a tentacle \( \tilde{e} \in \bar{L} \) onto distinct tentacles \( e \) and \( e' \) in \( \hat{G} \), 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 4.15** [Substitution] A substitution is a function \( \gamma \) that maps the graph variables \( X \) onto one of their expansion rules. A substitution \( \gamma \) 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 4.16** [Expansion] Let \( \gamma \) be a substitution suitting a cloned pattern \( G \). The \( \gamma \)-expansion of \( G \) is the graph \( G^\gamma \) obtained by the expansion steps

\[
G = G_0 \xrightarrow{\gamma(x_1)} G_1 \xrightarrow{\gamma(x_2)} \cdots \xrightarrow{\gamma(x_n)} G_n = G^\gamma
\]

where \( \{x_1, x_2, \ldots, x_n\} \subseteq \hat{X} \) 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 \( \gamma \), and associating every node and edge that is inserted in \( (L \cup R)^\gamma \) to that part of the rule \( t^\gamma \) to which the variable inserting it belongs.

**Example 4.17** [Expansion] The handle replacement rule in Fig. 6 defines \( \gamma(\beta) \) for the running example. When applying this rule three times to \( \text{pdm}^\mu \), the resulting expanded rule \( (\text{pdm}^\mu)^\gamma \) is the rule in Fig. 2.

4.3 Graph Transformation with Instantiation

We now can define graph transformation with variables.
Definition 4.18 [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 \( G \Rightarrow_s H \), if there is a multiplicity function \( \mu \), and a substitution \( \gamma \) suiting \( s^\mu \) so that \( G \Rightarrow_t H \) for the rule \( t = (s^\mu)^\gamma \).

Def. 4.18 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 4.19 [Applying pdm] The rule scheme \( \text{pdm} \) shown in Fig. 4 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 accessed by leaf E-nodes; 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, which can be described by a single re scheme as well.

Example 4.20 [The Inline Method refactoring] The rule scheme \( \text{im} \) in Fig. 7 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 dangling condition assures that no other calls to the method must exist). 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 \( \beta \) has the same meaning as in push-down-method.

The substitution \( \gamma \) in Fig. 6 is suited for a cloned scheme \( \text{im}' \), but only if the
Fig. 7. rule scheme $\text{im}$ for the *inline-method* refactoring

multiplicity function $\nu$ is defined with $\nu(u) = 0$ and $\nu(w) = \nu(y) + \nu(x) = 1$; otherwise, the occurrence of $\beta$ in $\text{im}''$ has too few or too many tentacles.

5 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 [18], and cloning is more tightly integrated with expansion than in [9]. And, the instantiation now yields rules in the double-pushout approach [5], a standard way of graph transformation with a rich theory.

The set nodes of PROGRES [16] and Fujaba correspond to the cloning of single nodes. In the model transformation language GMORPH [17], a more general notion of cloning is provided that uses nested *collection containers* that correspond to the master graph of a cloning variable. Both alternatives, however, lack the possibility to be combined with graph variables such that refactoring operations like *push-down-method* and *inline-method*, where graph variables with a varying number of tentacles are required, can be expressed. In our case study [19], a representative set of elementary refactoring operations from the list of [7] could be described with rule schemes using these concepts and only minimal control flow. In [12], hyperedges (nodes with a fixed number of tentacles) have been used as graph variables for the first time. 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 [10]; 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 [13]. It is also necessary to clearly define the set of possible substitutions for a variable in a refactoring rule. Shape grammars [4] are a promising candidate for this. Finally, 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 Diplan [8] shall provide all these concepts.

References


