# Formalising Behaviour Preserving Program Transformations

**Draft version — April 7, 2002 — Submitted to ICGT 2002**

Tom Mens[1], Serge Demeyer[2], and Dirk Janssens[2]

[1] Programming Technology Lab, Vrije Universiteit Brussel,
Pleinlaan 2, B-1050 Brussel, Belgium
`tom.mens@vub.ac.be`

[2] Department of Mathematics and Computer Science, Universiteit Antwerpen,
Universiteitsplein 1, B-2610 Antwerpen, Belgium
`{serge.demeyer, dirk.janssens}@ua.ac.be`

**Abstract.** The notion of refactoring —transforming the source-code of an object-oriented program without changing its external behaviour— has increased the need for a precise definition of refactorings and their properties. This paper introduces a graph representation of those aspects of the source code that should be preserved by a refactoring, and graph rewriting rules as a formal specification for the refactoring transformations themselves. To this aim, we use type graphs, forbidden subgraphs, embedding mechansims, negative application conditions and controlled graph rewriting. We show that it is feasible to reason about the effect of refactorings on object-oriented programs independently of the programming language being used. This is crucial for the next generation of refactoring tools.

## 1 Introduction

Refactorings are software transformations that restructure an object-oriented program while preserving its behaviour [1–3]. 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 [1].

Although it is possible to refactor manually, tool support is considered crucial. Tools such as the *Refactoring Browser* support a semi-automatic approach [4], which is recently being adopted by industrial strength software development environments (e.g., VisualWorks, TogetherJ, JBuilder, Eclipse[3]). Other researchers demonstrated the feasibility of fully automated tools [5]; studied ways to make refactoring tools less dependent on the implementation language being used [6] and investigated refactoring in the context of a UML case-tool [7].

Despite the existence of such tools, the notion of *behaviour preservation* is poorly defined. This is mainly because most definitions of the behaviour of an object concentrate on the run-time aspects (e.g., pre- and postconditions [8], observable input-output

[3] see `http://www.refactoring.com/` for an overview of refactoring tools

[9]) while refactoring tools must necessarily restrict themselves to the static description as specified in the source-code. Therefore, refactoring tools typically rely on an abstract syntax tree as a representation of the source-code and assert pre- and postconditions before and after transforming the tree [10]. Unfortunately, this representation contains details about the control flow of a program, which are largely irrelevant when specifying the effects of a refactoring on the program structure. Moreover, an abstract syntax tree is necessarily dependent on the programming language being used, while refactorings should be defined independently of the programming language [6].

For these reasons, we conclude that a lightweight graph representation of the source code is more appropriate for studying refactorings. Such a representation should not bother with the details necessary for sophisticated data- and control flow analysis or type inferencing techniques, since these are necessarily dependent on the programming language. Instead it should focus on the core concepts present in any class-based object-oriented language –namely classes, methods and variables– and allow us to verify whether the relationships between them are preserved. Moreover, it should allow for a transparant yet formal specification of the refactorings, as direct manipulations of the graph representation.

Therefore, this paper presents a *feasibility study* to see whether graph rewriting can be used to formalise what exactly is preserved when performing a refactoring. Section 2 introduces the concept of refactorings by means of a small motivating example and presents several types of behaviour that should be preserved. In section 3 we introduce the typed graph representation of the source code and formalise two selected refactorings (“encapsulate field” and “pull up method”) by graph rewriting productions. Section 4 shows how this formalisation can be used to guarantee the preservation of well-formedness and certain types of behaviour. Section 5 concludes the paper with the lessons learned regarding the feasibility of graph rewriting as a formal basis for the behaviour preserved during refactoring.

## 2 Motivating example: A Local Area Network simulation

As a motivating example, this paper uses a simulation of a Local Area Network (LAN). The example has been used successfully by the Programming Technology Lab of the Vrije Universiteit Brussel and the Software Composition Group of the University of Berne to illustrate and teach good object-oriented design. The example is sufficiently simple for illustrative purposes, yet covers most of the interesting constructs of the object-oriented programming paradigm (inheritance, late binding, super calls, method overriding). It has been implemented in Java as well as Smalltalk. Moreover, the example follows an incremental development style and as such includes several typical refactorings. Thus, the example is sufficiently representative to serve as a basis for a feasability study.

### 2.1 Initial version

In the initial version there are 4 classes: *Packet*, *Node* and two subclasses *Workstation* and *PrintServer*. The idea is that all *Node* objects are linked to each other in a token ring network (via the *nextNode* variable), and that they can *send* or *accept* a *Packet* object. *PrintServer* and *Workstation* refine the behaviour of *accept* (and perform a super call) to achieve specific behaviour for printing the *Packet* (lines 18–20) and avoiding endless cycling of the *Packet* (lines 26–28). A *Packet* object can only *originate* from a *Work-Station* object, and sequentially visits every *Node* object in the network until it reaches its *addressee* that accepts the *Packet*, or until it returns to its *originator* workstation (indicating that the *Packet* cannot be delivered).

Below is some sample Java code of the initial version where all constructor methods have been omitted due to space considerations. Although the sample code is in Java, other implementation languages could serve just as well, since we restrict ourselves to core object-oriented concepts only.

```
01 public class Node {
02   public String name;
03   public Node nextNode;
04   public void accept(Packet p) {
05     this.send(p); }
06   protected void send(Packet p) {
07     System.out.println(name + nextNode.name);
08     this.nextNode.accept(p); }
09   }

10 public class Packet {
11   public String contents;
12   public Node originator;
13   public Node addressee;
14   }

15 public class PrintServer extends Node {
16   public void print(Packet p) {
17     System.out.println(p.contents); }
18   public void accept(Packet p) {
19     if(p.addressee == this) this.print(p);
20     else super.accept(p); }
21   }

22 public class Workstation extends Node {
23   public void originate(Packet p) {
24     p.originator = this;
25     this.send(p); }
26   public void accept(Packet p) {
27     if(p.originator == this) System.err.println("no destination");
28     else super.accept(p); }
29   }
```

## 2.2 Subsequent versions

The initial version serves as the basis for a rudimentary LAN simulation. In subsequent versions, new functionality is incorporated incrementally and the object-oriented structure is refactored accordingly. First, logging behaviour is added which results in an “extract method” refactoring ([1], p110) and an “encapsulate field” refactoring ([1], p206).

Second, the *PrintServer* functionality is enhanced to distinguish between ASCII- and PostScript documents, which introduces complex conditionals and requires an “extract class” refactoring ([1], p149). The latter is actually a composite refactoring which creates a new intermediate superclass and then performs several “pull up field” ([1], 320) and “pull up method” ([1], p322) refactorings. Finally, a broadcast packet is added which again introduces complex conditionals, resolved by means of an “extract class”, “extract method”, “move method” ([1], p142) and “inline method” ([1], p117).

## 2.3 Selected refactorings

Fowler’s catalogue [1] lists seventy-two refactorings and since then many others have been discovered. Since the list of possible refactorings is infinite, it is impossible to prove that all of them preserve behaviour. However, refactoring theory and tools assume that there exist a finite set of *primitive refactorings*, which can then freely be combined into *composite refactorings*.

For this feasability study, we restrict ourselves to two frequently used primitive refactorings, namely “encapsulate field” and “pull up method”. The preconditions for these two object-oriented refactorings are quite typical, hence they may serve as representatives for the complete set of primitive refactorings.

**EncapsulateField.** Fowler [1] introduces the refactoring *EncapsulateField* as a way to encapsulate public variables by making them private and providing accessors. In other words, for each public variable a method is introduced for accessing (“getting”) and updating (“setting”) its value, and all direct references to the variable are replaced by dynamic calls (`this` sends) to these methods.

*Precondition.* Before creating the new accessing and updating methods on a class *C*, a refactoring tool should verify that no method with the same signature exists in any of *C*’s subclasses and superclasses, *C* included. Otherwise, the refactoring may accidentally override (or be overridden by) an existing method, and then it is possible that the behaviour is not preserved.

**PullUpMethod.** Fowler [1] introduces the refactoring *PullUpMethod* as a way to move similar methods in subclasses into a common superclass. This refactoring removes code duplication and increases code reuse by inheritance.

*Precondition.* When a method *m* with signature *s* is pulled up into a class *C*, it implies that all methods with signature *s* defined on the direct descendants of *C* are removed and replaced by a single occurrence of *m* now defined on *C*. However, a tool should verify whether the method *m* does not refer to any variables defined in the subclass, because otherwise the pulled-up method would refer to an out-of-scope variable and then the transformed code would not compile. Also, no method with signature *s* may exist on *C*, because otherwise a method is overwritten accidentally, which may break existing behaviour.

### 2.4 Behaviour preservation

Since we focus on refactoring of source code, we only look at notions of behaviour preservation that can be detected statically and do not rely on sophisticated data- and control-flow analysis or type inferencing techniques. This restriction to the static structure of a program is important because source-code is all that refactoring tools may operate on.

The general idea is that, for each considered refactoring, one may catalog the types of behaviour that need to be preserved. For the feasibility study of this paper, we consider three types of behaviour preservation, based on the fact that they are important and non-trivial for the two selected refactorings. Section 4 discusses to which extent the selected refactorings satisfy these preservation properties:

A refactoring is *access preserving* if each method implementation accesses at least the same variables after the refactoring as it did before the refactoring. These variable accesses may occur transitively, by first calling a method that (directly or indirectly) accesses the variable. A refactoring is *update preserving* if each method implementation performs at least the same variable updates after the refactoring as it did before the refactoring. A refactoring is *call preserving* if each method implementation still performs at least the same method calls after the refactoring as it did before the refactoring.

## 3 Formalising refactoring by graph rewriting

This section introduces a graph rewriting formalism to express refactorings in an intuitive and generic way. To this extent, we first explain the graph notation used to represent source code.

### 3.1 Graph notation

The graph representation of the source code is rather straightforward. Software entities (such as classes, variables, methods and method parameters) are represented by *nodes* whose label is a pair consisting of a name and a node type. For example, the class *Packet* is represented by a node with name *Packet* and type $C$ (i.e., a $C$-node). The set $\Sigma = \{C, B, V, S, P, E\}$ of all possible node types is clarified in Table 1. Method bodies ($B$-nodes) have been separated from their signatures ($S$-nodes) to make it easier to model late binding and dynamic method lookup, where the same signature can have many possible implementations. $B$-nodes (method bodies) and $P$-nodes (formal parameters) have an empty name.

Relationships between software entities (such as containment, inheritance, method lookup, variable accesses and method calls) are represented by *edges* between the corresponding nodes. The label of an edge is simply the edge type. For example, the inheritance relationship between the classes *Workstation* and *Node* is represented by an edge with type $i$ (i.e., an $i$-edge) between the $C$-nodes *Workstation* and *Node*. The set $\Delta = \{l, i, m, t, p, e, \bullet, d, a, u\}$ of all possible edge types is clarified in Table 2. For $m$-edges (membership) the label is often omitted in the figures.

| **Type** | **Description** | **Examples** |
|---|---|---|
| $C$ | **C**lass | *Node*, *Workstation*, *PrintServer*, *Packet* |
| $B$ | method **B**ody (or implementation) | `System.out.println(p.contents)` |
| $V$ | **V**ariable | *name*, *nextNode*, *contents*, *originator* |
| $S$ | method **S**ignature in lookup table | *accept*, *send*, *print* |
| $P$ | formal **P**arameter of a message | *p* |
| $E$ | (sub)**E**xpression in method body | `p.contents` |

**Table 1.** Node type set $\Sigma = \{C, B, V, S, P, E\}$

| **Type** | **Description** | **Examples** |
|---|---|---|
| $l: \quad S \to B$ | dynamic method **l**ookup | `accept(Packet p)` has 3 possible method bodies |
| $i: \quad C \to C$ | **i**nheritance | `class PrintServer` **`extends`** `Node` |
| $m: V \to C$ | variable **m**embership | variable *name* belongs to *Node* |
| $B \to C$ | method **m**embership | method *send* is implemented in *Node* |
| $t: \quad P \to C$ | message parameter **t**ype | `print(`**`Packet`**` p)` |
| $V \to C$ | variable **t**ype | **`String`**` name` |
| $S \to C$ | signature return **t**ype | **`String`**` getName()` |
| $p: \quad S \to P$ | formal **p**arameter | `send(Packet `**`p`**`)` |
| $E \to E$ | actual **p**arameter | `System.out.println(`**`nextNode.name`**`)` |
| $e: \quad B \to E$ | **e**xpression in method body | `if (p.addressee==this) this.print(p);` |
|  |  | `else super.accept(p);` |
| $\bullet: \quad E \to E$ | cascaded expression | `nextNode.accept(p)` |
| $d: \quad E \to S$ | **d**ynamic method call | `this.send(p)` |
| $a: \quad E \to P$ | parameter **a**ccess | **`p`**`.originator` |
| $E \to V$ | variable **a**ccess | `p.`**`originator`** |
| $u: \quad E \to V$ | variable **u**pdate | `p.`**`originator =`**` this` |

**Table 2.** Edge type set $\Delta = \{l, i, m, t, p, e, \bullet, d, a, u\}$

Using this notation, an entire program can be represented by means of a single typed graph. Because the graph representation can become very large, we only display those parts of the graph that are relevant for the discussion. For example, Figure 1 only shows the graph representation of the static structure of the LAN simulation. (For $B$-nodes, a name has been put between parentheses to make the graph more readable.)

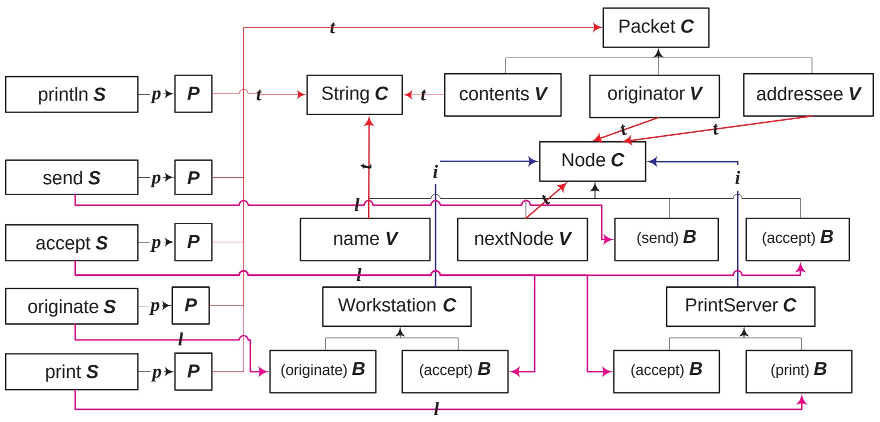

**Fig. 1.** Graph representation of the static structure of the LAN simulation

A method body is represented by a structure consisting of $E$-labeled nodes connected by edges that express information about dynamic method calls and variable invocations (accesses and updates). For example, Figure 2 represents the method bodies in class *Node*. The method body of *send* contains a sequence of two subexpressions, which is denoted by two $e$-edges from the $B$-node to two different $E$-nodes. The second subexpression `nextNode.accept(p)` is a cascaded method call (represented by a $\bullet$-edge) consisting of a variable access (represented by an $a$-edge to the $V$-node labelled *nextNode*) followed by a dynamic method call with one parameter (represented by a $d$-edge and corresponding $p$-edge originating from the same $E$-node).

We have deliberately kept the graph model very simple to make it as language independent as possible. It does not model Java-specific implementation features such as: Java interfaces; explicit references to `this`; constructor methods; control statements (such as `if`, `for`, etc.); Java modifiers (such as `abstract`, `protected`, `final`); inner classes; threads; exceptions.

## 3.2 Well-formedness constraints

On top of the above graph representation, we need to impose constraints to guarantee that a graph is well-formed in the sense that it corresponds to a syntactically correct program. These so-called *well-formedness constraints* are essential to fine-tune our graph

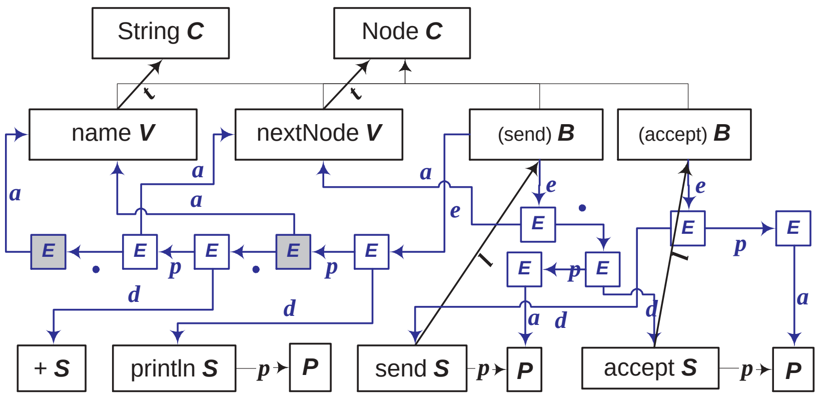

**Fig. 2.** Graph representation of the behaviour of class *Node*

notation to a particular programming language (in this case Java). In this paper we use two mechanisms to express these constraints: a *type graph* and *forbidden subgraphs*.

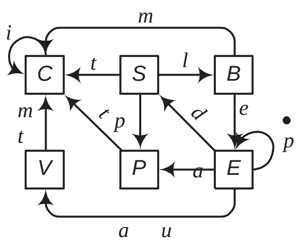

**Fig. 3.** Type Graph

The notion of a *type graph* is formally presented in [11–13]. Intuitively, a type graph is a meta-graph expressing restrictions on the graphs that are allowed. Formally, a graph is allowed only if there exists a graph morphism into the type graph: a node mapping and edge mapping that preserves sources, targets and labels. For node labels, only the second component, as introduced in Table 1, is taken into account. Figure 3 displays the type graph needed for our particular graph representation.

Because type graphs alone are insufficient to express all constraints that we are interested in, we use a second mechanism called *forbidden subgraphs*. A graph $G$ satisfies the constraint expressed by a forbidden subgraph $F$ if there does not exist an injective graph morphism from $F$ into $G$.

To specify forbidden subgraphs we use graph expressions of the form $a \xrightarrow[exp]{} b$, where $a$ and $b$ belong to the node type set $\Sigma$ of Table 1, and $exp$ is a regular expression over the edge type set $\Delta$ of Table 2. This graph expression denotes the set of graphs of the form depicted in Figure 4, where the word $l_1 \dots l_n$ belongs to the language of $exp$. If $a = b$ and $n = 0$ (i.e., $l_1 \dots l_n$ is the empty word) then the graph of Figure 4 consists of one node. A graph that contains more than one graph expression denotes the set of all graphs obtained by substituting these edges in the obvious way.

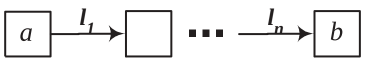

**Fig. 4.** The graph represented by the word $l_1 \dots l_n$ belonging to the language of $exp$

Some typical examples of well-formedness constraints, needed to guarantee that a refactorings does not lead to an ill-formed graph, are given below:

**WF-1** A variable cannot be defined in a class if there is already a variable with the same name in the inheritance hierarchy of this class (i.e., in the class itself or in an ancestor class or descendant class).

**WF-2** A method with the same signature cannot be implemented twice in the same class.

**WF-3** A method in a class cannot refer to variables that are defined in its descendant classes.

These constraints can be expressed by the forbidden subgraphs of Figure 5. The forbidden subgraph for **WF-3** uses two graph expressions. For example, expression $B \overrightarrow{_{?^*\{a|u\}}} V$ denotes the set of all nonempty edge paths from a $B$-node to a $V$-node, where the last edge must have either type $a$ or type $u$.

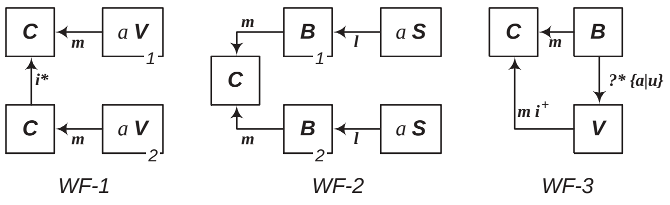

**Fig. 5.** Well-formedness constraints expressed as forbidden subgraphs

## 3.3 Graph rewriting productions

A *graph rewriting* is a transformation that takes an initial graph as input and transforms it into a result graph. This transformation occurs according to some predetermined rules that are specified in a so-called *graph production*. Such a graph production is specified by means of a *left-hand side* (LHS) and a *right-hand side* (RHS). The LHS is used to specify which parts of the initial graph should be transformed, while the RHS specifies the result after the transformation. Often, a graph production can be applied to different parts of a graph, leading to different occurrences (or matches) of the graph production's LHS. In this paper, we use *parameterised graph productions*, so that a production contains variables for labels. Such parameterised productions can be instantiated by choosing concrete values for the parameters. Figure 6 shows a production where *var*, *accessor* and *updater*are such parameters.

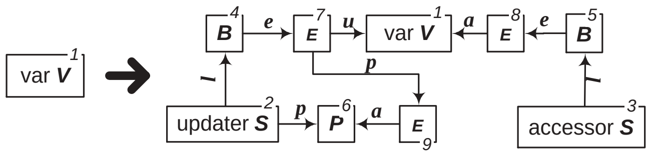

**Fig. 6.** Parameterised graph production $EncapsulateField(var, accessor, updater)$

The production of Figure 6 represents the refactoring *EncapsulateField*. LHS and RHS are separated by means of an arrow symbol. In this production, all nodes are numbered. Nodes that have a number occurring in both the LHS and the RHS are preserved by the rewriting (as is the case with node 1 in the example). Nodes with numbers that only occur in the LHS are removed, and nodes with numbers that only occur in the RHS (e.g., nodes 2 and 3) are newly created.

In order to take into account the "context" in which the production is applied, consisting of the (sub)expressions referring to the variable that is encapsulated, the production is equipped with an *embedding mechanism* similar to the one investigated in [14]. This embedding mechanism specifies how edges are redirected. *Incoming edges*, i.e., edges that have their target node in the LHS but not their source node, are redirected according to the incoming edges specification in Table 3. For example, $(u, 1) \to (d, 2)$ means that each update of the variable *var* (represented by an incoming $u$-edge to node 1) is replaced by a dynamic method call to the updater method (represented by an incoming $d$-edge to node 2). *Outgoing edges* are treated similarly, using the outgoing edges specification in Table 3. For example, $(m, 1) \to (m, 1), (m, 4), (m, 5)$ means that the method bodies (nodes $4$ and $5$) that correspond to the *accessor* and *updater* signature must be implemented in the same class as the one in which the variable *var* (node 1) was defined. Similarly, $(t, 1) \to (t, 1), (t, 3), (t, 6)$ means that the return type of the *accessor* method and the parameter type of the *updater* method must be the same as the type of the variable *var*. The type graph of Figure 3 guarantees that embedding table 3 is *complete*, i.e., all possible types of incoming and outgoing edges are covered. Indeed, according to the type graph, a $V$-node can only have incoming $a$-edges and $u$-edges, and outgoing $m$-edges and $t$-edges.

| incoming edges | outgoing edges |
|---|---|
| $(u,1) \to (d,2)$ | $(m,1) \to (m,1), (m,4), (m,5)$ |
| $(a,1) \to (d,3)$ | $(t,1) \to (t,1), (t,3), (t,6)$ |

**Table 3.** Embedding Table

The embedding-based parameterised production of Figure 6 may be viewed as a specification of an infinite set of productions in the algebraic approach to graph rewriting [15–17]. A concrete production can be obtained from the embedding-based one by the following two steps: (1) fill in the parameters of the parameterised graph production with concrete values; (2) extend the LHS and RHS of the embedding-based production with a concrete context. Figure 7 shows the production *EncapsulateField(name,getName,setName)* that is applied in the context of the LAN example of Figure 2. The two gray $E$-nodes in Figure 7 are matched with the gray $E$-nodes of Figure 2.

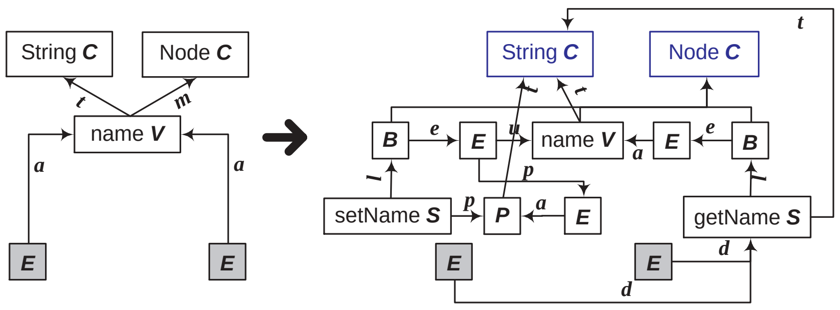

**Fig. 7.** Single-pushout graph production $EncapsulateField(name, getName, setName)$ obtained from the parameterised production of Figure 6

The second refactoring that we want to express by graph rewriting is $PullUpMethod(parent, child, name)$, which moves the implementation of a method $name$ in some $child$ class to its $parent$ class, and removes the implementations of the method $name$ in all other children of $parent$. Expressing this refactoring by a single production –even a parameterised one with embedding mechanism– is problematic: changes may have to be made in *all* subclasses of $parent$, and the number of such subclasses is not a priori bounded. A way to cope with the problem is to control the order in which productions are applied. Mechanisms for *controlled graph rewriting* have been studied in, e.g., [18–20]. Using these mechanisms, *PullUpMethod* can be expressed by two parameterised productions $P_1$ and $P_2$. $P_1$ moves the method $name$ one level higher in the inheritance hierarchy (i.e., from $child$ to $parent$), and can only be applied if it is immediately followed by an application of $P_2$. This second production removes the implementation of method $name$ from another subclass of $parent$, and has to be applied until there are no more occurrences of its left-hand side present. Both productions $P_1$ and $P_2$ are equipped with an identity embedding, i.e., all incoming and outgoing edges are preserved.

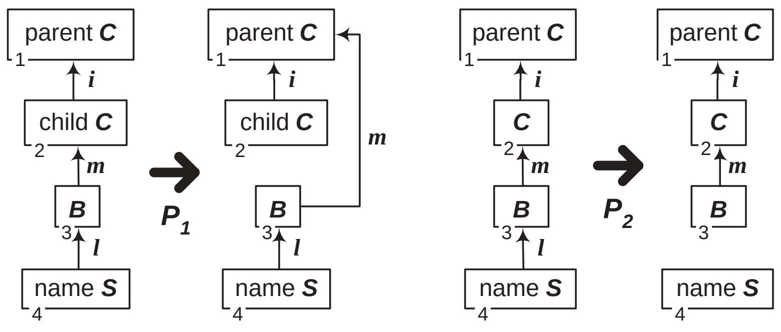

**Fig. 8.** Productions $P_1$ and $P_2$ for controlled rewriting $PullUpMethod(parent, child, name)$

## 4 Preservation of refactoring properties

In this section we combine the formalisation of refactorings of subsection 3.3 with another graph rewriting technique, negative application conditions, to guarantee certain properties of the graphs that are derived. In particular, we consider certain types of behaviour preservation, refactoring preconditions and well-formedness constraints.

### 4.1 Preserving behaviour

The types of behaviour preservation discussed in section 2.4 can be expressed formally using the graph expression notation introduced in section 3.2.

The graph expression $B \xrightarrow[?^*a]{} V$ can be used to express the property of *access preservation*. It specifies all possible access paths from a method body ($B$-node) to a variable ($V$-node). The constraints imposed by the type graph of Figure 3 guarantee that there is only one $a$-edge on such a path, and this edge is the last one in the path. Access preservation means that, for each occurrence of $B \xrightarrow[?^*a]{} V$ in the initial graph to be rewritten, there is a corresponding occurrence of this graph expression in the resulting graph, connecting the same $B$-node and $V$-node. Thus, the nodes that match $B$ and $V$ should not be removed or added by the graph production. In a similar way, we can express *update preservation* by means of the expression $B \xrightarrow[?^*u]{} V$.

Graph expression $B \xrightarrow[?^*d]{} S \xrightarrow[lm]{} C \xleftarrow[i^*]{} C$ formalises the property of *call preservation*. Subexpression $B \xrightarrow[?^*d]{} S$ specifies that for each method body ($B$-node) that performs a dynamic method call ($d$-edge) to some signature ($S$-node) in the initial graph, there should still be a dynamic method call from the same method body to the same signature in the resulting graph. Subexpression $S \xrightarrow[lm]{} C \xleftarrow[i^*]{} C$ reflects that each signature that is implemented in some class is understood by the class itself and all its descendant classes.

**EncapsulateField.** To show *update preservation* for *EncapsulateField*, it suffices to show that the preservation property expressed by $B \xrightarrow[?^*u]{} V$ is satisfied for each method body $B$ that updates the variable *var* that is being encapsulated. It follows from the form of the graph production of Figure 6 that this is the case. This is illustrated in Figure 9, that shows how a direct update of *var* is replaced by a slightly longer path that still satisfies the property $B \xrightarrow[?^*u]{} V$. *Access preservation* can be shown in a similar way. *Call preservation* is trivial since the refactoring does not change any dynamic method calls or method bodies. (It does add new method signatures and method bodies, but this does not affect existing method calls.)

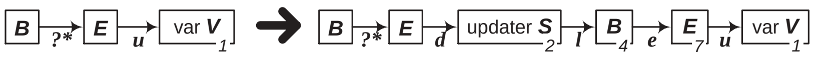

**Fig. 9.** Update preservation property of $EncapsulateField(var, accessor, updater)$

**PullUpMethod.** To show *call preservation* for *PullUpMethod*, we have to check whether the property $B \xrightarrow[?^*d]{} S \xrightarrow[lm]{} C \xleftarrow[i^*]{} C$ is preserved by the refactoring. Subexpression $B \xrightarrow[?^*d]{} S$ is trivially fulfilled. Subexpression $S \xrightarrow[lm]{} C \xleftarrow[i^*]{} C$ is illustrated in Figure 10: all implementations of the signature *name* in some child class of *parent* are replaced by the implementation of signature *name* in *parent* itself. This means that the path $S \xrightarrow[lm]{} C$ is replaced by a path $S \xrightarrow[lm]{} C \xleftarrow[i]{} C$ and both paths belong to the graph expression $S \xrightarrow[lm]{} C \xleftarrow[i^*]{} C$. *Access preservation* and *update preservation* are trivial, except in the case where an implementation of the signature *name* in some child class of *parent* accesses or updates a variable. Since this implementation is removed (pulled up) by the refactoring, it is possible that variable accesses or updates in this method implementation are not preserved. Hence, the *PullUpMethod* refactoring is not necessarily access preserving or update preserving!

## 4.2 Preserving constraints

In general, the graph obtained by the application of a refactoring production has to satisfy several constraints. On the one hand, it has to satisfy well-formedness constraints, and on the other hand, refactorings are often subject to more specific constraints. For example, *EncapsulateField* may not cause *accidental method overriding*, i.e.,

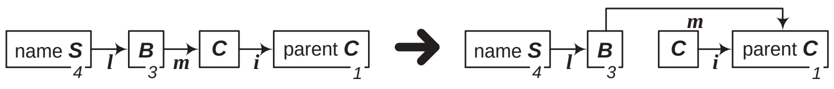

**Fig. 10.** Call preservation property of $PullUpMethod(parent, child, name)$

the refactoring may not introduce new methods in a class if these methods are already defined in one of its subclasses (**RC-1**)

All these constraints can be expressed in a natural way as postconditions for the graph rewriting. However, for efficiency reasons, it is desirable to transform these post-conditions into *preconditions*. This avoids having to undo the refactoring if it turns out that the constraints are not met. Formally, preconditions can be defined by using graph rewriting with negative application conditions [21–23].

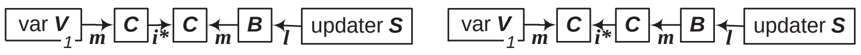

**Fig. 11.** Negative preconditions for the *EncapsulateField* refactoring

**EncapsulateField.** Figure 11 presents the negative preconditions needed in order for *EncapsulateField* to satisfy refactoring constraint **RC-1**. The conditions specify that no ancestor or descendant class of the class containing the variable *var* should have implemented a method with signature *updater*. Two similar negative application conditions are needed for the *accessor* method. Well-formedness constraint **WF-1** is satisfied since *EncapsulateField* does not introduce or move any variables, or change anything to the class hierarchy. Constraint **WF-2** is satisfied thanks to the preconditions of Figure 11, in the special case where $i^*$ is the empty word. Constraint **WF-3** is satisfied because *EncapsulateField* only introduces a new variable access and update to a variable that is defined by the class itself.

**PullUpMethod.** Figure 12 presents two negative preconditions for *PullUpMethod*, or more specifically, for subproduction $P_1$ of Figure 8. The condition on the left specifies that the method *name* to be pulled up should not yet be implemented in *parent*, and the condition on the right specifies that the implementation of the method to be pulled up should not refer to (i.e., access or update) variables outside the scope of the *parent*.

*PullUpMethod* satisfies well-formedness constraint **WF-1** since it does not introduce or redirect any variables, or change anything to the class hierarchy. Constraint **WF-2** is satisfied thanks to the precondition on the left of Figure 12. Constraint **WF-3** is satisfied thanks to the precondition on the right of Figure 12.

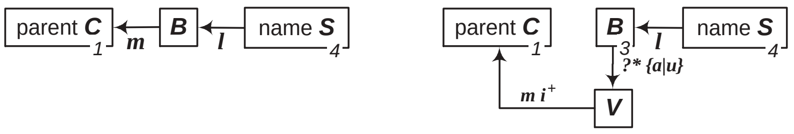

**Fig. 12.** Negative preconditions for the *PullUpMethod* refactoring

## 5 Conclusion and Future Work

This paper presented a feasibility study concerning the use of graph rewriting as a formal specification for refactoring. Based on the specification of two refactorings ("encapsulate field" and "pull up method") we conclude that graph rewriting is a suitable formalism for specifying the effect of refactorings, because (i) graphs can be used as a language-independent representation of the source code; (ii) rewriting rules are a natural yet precise way to specify the source-code transformations implied by a refactoring; (iii) the formalism allows us to prove that refactorings indeed preserve the behaviour that can be inferred directly from the source code.

In order to achieve our goal, we had to enrich the basic graph rewriting mechanism by a number of additional techniques. Typing and forbidden subgraphs made it possible to express well-formedness constraints in a natural way. The specification of infinite sets of productions was facilitated by using parameterisation and an embedding mechanism. The application of graph productions was restricted by using negative application conditions and controlled graph rewriting. All these techniques have been investigated in the literature, but a better integration is necessary to make the approach usable for people less acquainted with graph rewriting.

The two refactorings as well as the types of behaviour preservation we studied, are realistic and well documented. Because there are many other types of refactorings and behaviour preservation, further research is needed in order to find out whether the used graph representation needs to be modified, or whether other graph rewriting techniques should be used. For example, initial attempts to specify refactorings such as "move method" and "push down method" showed that it is difficult to manipulate nested structures in method bodies.

A central topic in future work will also be the investigation of methods to detect, for a given graph property and graph transformation, whether or not the property is preserved by the transformation. This requires further research into formalisms to express such properties, as well as the syntax that could be used as input for an automated refactoring tool.

Similar to what has been described in [6], we will also study the impact of language specific features (e.g., Java interfaces) to verify whether it is possible to express refactorings independently of the programming language being used.

In the longer run, we want to investigate combinations of refactorings. Roberts [10] has argued that primitive refactorings can be chained in sequences, where the preconditions of one refactoring are guaranteed by the postconditions of the previous ones.

Moreover, in some refactoring sequences it is possible to change the order without changing the global effect. Such properties can be expressed using graph rewriting formalisms like the one in [22]. Refactoring tools may exploit these properties to optimise the number of program transformations, in much the same way as database tools perform query optimisations. This reduces the amount of analysis that must be performed by a tool, which is crucial for the performance and usability of refactoring tools.

## References

1. Fowler, M.: Refactoring: Improving the Design of Existing Programs. Addison-Wesley (1999)
2. Opdyke, W.: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign (1992)
3. Opdyke, W., Johnson, R.: Creating abstract superclasses by refactoring. In: Proc. ACM Computer Science Conference, ACM Press (1993) 66–73
4. Roberts, D., Brant, J., Johnson, R.: A refactoring tool for Smalltalk. Theory and Practice of Object Systems **3** (1997) 253–263
5. Casais, E.: Automatic reorganization of object-oriented hierarchies: a case study. Object Oriented Systems **1** (1994) 95–115
6. Tichelaar, S.: Modeling Object-Oriented Software for Reverse Engineering and Refactoring. PhD thesis, University of Bern (2001)
7. Sunyé, G., Pollet, D., LeTraon, Y., Jézéquel, J.M.: Refactoring UML models. In: Proc. UML 2001. Volume 2185 of Lecture Notes in Computer Science., Springer-Verlag (2001) 134–138
8. Meyer, B.: Object-Oriented Software Construction (2nd edition). Prentice Hall (1979)
9. Bergstein, P.L.: Maintenance of object-oriented systems during structural evolution. Theory and Practice of Object Systems **3** (1991) 185–212
10. Roberts, D.: Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana-Champaign (1999)
11. Corradini, A., Montanari, U., Rossi, F.: Graph processes. Fundamenta Informaticae **26** (1996) 241–265
12. Corradini, A., Ehrig, H., Löwe, M., Montanari, U., Padberg, K.: The category of typed graph grammars and their adjunction with categories of derivations. In: Proceedings 5th International Workshop on Graph Grammars and their Application to Computer Science. Lecture Notes in Computer Science, Springer-Verlag (1996)
13. Engels, G., Schürr, A.: Encapsulated hierarchical graphs, graph types and meta types. Electronic Notes in Theoretical Computer Science **2** (1995)
14. Janssens, D., Mens, T.: Abstract semantics for ESM systems. Fundamenta Informaticae **26** (1996) 315–339
15. Ehrig, H.: Introduction to the algebraic theory of graph grammars. In Claus, V., Ehrig, H., Rozenberg, G., eds.: Graph Grammars and Their Application to Computer Science and Biology. Volume 73 of Lecture Notes in Computer Science., Springer-Verlag (1979) 1–69
16. Ehrig, H., Löwe, M.: Parallel and distributed derivations in the single-pushout approach. Theoretical Computer Science **109** (1993) 123–143
17. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theoretical Computer Science **109** (1993) 181–224
18. Bunke, H.: Programmed graph grammars. In Claus, V., Ehrig, H., Rozenberg, G., eds.: Graph Grammars and Their Application to Computer Science and Biology. Volume 73 of Lecture Notes in Computer Science., Springer-Verlag (1979) 155–166

19. Kreowski, H.J., Kuske, S.: Graph transformation units and modules. Handbook of Graph Grammars and Computing by Graph Transformation **2** (1999) 607–638
20. Schürr, A.: Logic based programmed structure rewriting systems. Fundamenta Informaticae **26** (1996) 363–385
21. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae **26** (1996) 287–313
22. Heckel, R.: Algebraic graph transformations with application conditions. Master's thesis, TU Berlin (1995)
23. Heckel, R., Wagner, A.: Ensuring consistency of conditional graph grammars: A constructive approach. Lecture Notes in Theoretical Computer Science **1** (1995)