Research

Formalizing Refactorings with Graph Transformations

Tom Mens¹,*, Niels Van Eetvelde², Serge Demeyer² and Dirk Janssens²

¹ Service de Génie Logiciel, Université de Mons-Hainaut, Av. du Champ de Mars 6, 7000 Mons, Belgium
² Department of Mathematics and Computer Science, Universiteit Antwerpen, Middelheimlaan 1, 2020 Antwerpen, Belgium

SUMMARY

The widespread interest in refactoring—transforming the source-code of an object-oriented program without changing its external behaviour—has increased the need for a precise definition of refactoring transformations and their properties. In this paper we explore the use of graph rewriting for specifying refactorings and their effect on programs. We introduce a graph representation for programs and show how two representative refactorings can be expressed by graph productions. Then we demonstrate that it is possible to prove that refactorings preserve certain program properties, and that graph rewriting is a suitable formalism for such proofs.

KEY WORDS: refactoring, formal specification, graph transformation, behaviour preservation

1. Introduction

Refactorings are software transformations that restructure an object-oriented program while preserving its behaviour [11, 25, 26, 22]. 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 [11].

Although it is possible to refactor manually, tool support is considered crucial. Tools such as the Refactoring Browser support a semi-automatic approach [29], which has also been adopted by industrial strength software development environments (see http://www.refactoring.com/ for

*Correspondence to: Service de Génie Logiciel, Université de Mons-Hainaut, Av. du Champ de Mars 6, 7000 Mons, Belgium
Contract/grant sponsor: Research carried out in the framework of research project G.0452.03 of the Fund for Scientific Research - Flanders (Belgium).
an overview of refactoring tools. Other researchers demonstrated the feasibility of fully automated tools \[3\], studied ways to make refactoring tools less dependent on the implementation language being used \[33, 18\] and investigated refactoring in the context of UML \[1, 31\]. Despite the existence of such tools, a precise specification of refactorings does not yet exist. Today, refactorings are mainly specified through examples \[11\], although there exist more precise specifications by means of pre- and postconditions expressed in natural language \[25, 33\]. The most precise specification so far is based on asserting pre- and postconditions on the abstract syntax tree of a Smalltalk program \[28\]. Although a good basis, such a specification is too language specific to serve a general tool builder audience.

In order to provide reliable tool support, a formal model defining the effects of refactorings on source code is required. In particular there is a need for (i) mathematical proofs that a given refactoring preserves certain program properties and changes others; (ii) formal analysis of pre- and postconditions to show the soundness and completeness of the specification and to construct composite refactorings out of primitive ones. In view of the wide acceptance of graph-like representations in today’s refactoring tools, it seems natural to use graph rewriting as the basis for the desired formal model. Indeed, most refactoring tools represent the source code by means of an abstract syntax tree augmented with extra links to represent frequently used relations (e.g., inheritance, method invocation and variable access). Since a refactoring is supposed to change that graph, a formal specification for a refactoring would quite naturally correspond with a graph rewriting rule. Among others, our early work on that topic \[20, 35\] and that of Bottini et Al. \[27\] has shown that such an approach is promising. Obviously, graph rewriting is not the only way in which refactorings can be formalised. For example, Lämmel \[18\] represented refactoring transformations in a language-parametric manner using functional strategic programming as a sufficiently expressive and concise specification medium. Also Kniesel used an algebraic medium to express the pre- and postconditions \[15\].

However, in this paper we will restrict ourselves to graph rewriting and we will argue that it is feasible to use graph rewriting for specifying the precise effect of refactorings on source code. The basis of our argument is the specification of the complete list of primitive refactorings which can be combined to form arbitrary chains of composite refactorings.

Nevertheless, it remains a challenge to specify a formal refactoring model which is sufficiently understandable by tool builders. Therefore, the following criteria must be addressed.

(i) Transparency. A formal refactoring model should fit into the natural realm of refactoring tools. Thus, the graph representation should be as close as possible to the typical augmented abstract syntax tree found inside a refactoring tool. Additionally, the graph rewriting rules should resemble the manipulations a refactoring tool performs on that abstract syntax tree.

(ii) Conciseness. To keep the formal model manageable it should remain as small as possible. On the one hand, the rewrite rules themselves should be concise: simple refactorings like a renaming should be specified in one rewrite rule, while more elaborate ones should require no more than a few rules. On the other hand, all constructs that are not necessary for specifying the properties of interest should be omitted from the graph representation. For example, control flow statements may be omitted from the abstract syntax tree, since typical refactoring tools do not manipulate those. Also, constructs that are too specific to one programming language —e.g., inner classes in Java— may be ignored when studying generic refactorings.
(iii) **Elegance.** The notation should be convenient to read, so that tool builders will understand what is specified. Therefore, it is acceptable to define ad hoc notations for frequently used manipulations, as long as a sound formal basis is available.

(iv) **Expressiveness.** The formalism should be able to specify the core concepts present in any class-based object-oriented language—namely classes, methods and variables—and allow us to verify whether the core relationships—inheritance, message sends, variable access—between them are preserved after applying a series of rewriting rules.

This paper presents a *feasibility study* to illustrate whether graph rewriting can be used to formalise refactorings as well as the properties to be preserved by these refactorings. The paper starts with a motivating example of a LAN simulation (section 2), and then proceeds with the different steps in the formal specification, namely (a) a formal graph representation of an abstract syntax tree augmented with variable access and method invocation relations (section 3); (b) rewrite rules for two representative refactorings (namely, *EncapsulateVariable* and *PullUpMethod*) (section 4); (c) preconditions for the two refactorings, which must be verified before the actual rewrite rules are fired (section 5); (d) proofs, showing that a rewrite rule applied on a graph satisfying the necessary precondition actually preserves the necessary properties (section 6); (e) validation, reporting on the experiment we performed converting Java programs into our graph representation and specifying the rewrite rules using two state-of-the-art graph transformation tools (section 7). Subsequently we use the criteria (i) — (iv) to reflect on where the graph rewriting formalism worked well, but also where we encountered problems (section 8). Finally, we conclude the paper with a summary of our main findings (section 9).

### 2. Motivating example

As a motivating example, this paper uses a simulation of a Local Area Network (LAN) [7]. The example has been used successfully in several introductory programming courses to illustrate and teach good object-oriented design and also in advanced software engineering courses on refactoring [6]. 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 suitable to serve as a basis for a feasibility study.

#### 2.1. Local Area Network simulation

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 *WorkStation* 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 code is in Java, other implementation
languages could serve just as well, since we restrict ourselves to core object-oriented concepts
only.

```java
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 }
```

Figure 1. A piece of Java code implementing a Local Area Network Simulation.

The initial version serves as the basis for a rudimentary LAN simulation (see Figure 1). 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
ExtractMethod refactoring ([11], p110) and an EncapsulateVariable refactoring ([11], p206). Then, the PrintServer functionality is enhanced to distinguish between ASCII and PostScript
documents, which introduces complex conditionals and requires an ExtractClass refactoring
([11], p149). The latter is actually a composite refactoring which creates a new intermediate
superclass and then performs several *PullUpVariable* ([11], p320) and *PullUpMethod* ([11], p322) refactorings. Finally, a broadcast packet is added which again introduces complex conditionals, resolved by means of an *ExtractClass*, *ExtractMethod*, *MoveMethod* ([11], p142) and *InlineMethod* ([11], p117).

```java
01 public class ASCIIPrintserver extends Printserver {
02     public void print(Packet p) {
03         System.out.println("Printing packet with contents " + p.getContents()
04             + " on printer " + this.getName());
05     }

06 public class PostscriptPrintserver extends Printserver {
07     public void print(Packet p) {
08         System.out.println("Printing packet with contents " + p.contents
09             + " on printer " + this.getName());
10     }
```

Figure 2. Extracts of the Java code for the LAN simulation before applying *PullUpMethod*.

Figure 2 shows part of the code at an intermediate stage in this sequence. When distinguishing between ASCII and PostScript Documents we have introduced two new classes *ASCIIPrintserver* and *PostscriptPrintserver*. Both these classes define a *print* method with the same signature and an equivalent method body. This is duplicated code which can be easily refactored away with a *Pull Up Method* refactoring.

### 2.2. Selected refactorings

Fowler’s catalogue [11] lists seventy-two refactorings and since then many others have been proposed. Because the list of possible refactorings is infinite, it is impossible to prove that all of them preserve behaviour, even if one would agree on the precise meaning of the phrase “preserve behaviour”. However, refactoring theory and tools assume that there exists a finite set of *primitive refactorings* which can be combined into larger *composite refactorings* [25, 29]. The list of primitive refactorings boils down to:

- **Class**: CreateClass, RenameClass, RemoveClass
- **Parameter**: AddParameter, RemoveParameter
- **Variable** (or Field): CreateVariable, RenameVariable, RemoveVariable, EncapsulateVariable, PullUpVariable, PushDownVariable

Because it is impossible to discuss in detail how all of these refactorings can be formalized, we restrict ourselves to the *EncapsulateVariable* and *Pull Up Method* refactorings which illustrate very well the concepts in this paper. Indeed, the *EncapsulateVariable* refactoring requires many complex changes to the abstract syntax tree, because it must replace all references to a field by a method invocation and pass the appropriate parameter values. The *Pull Up Method* on
the other hand has a very complex precondition, because it involves many checks across the inheritance hierarchy. Moreover, both are good examples of subtle interpretation issues that arise when providing a natural language definition, yet can be defined precisely in a formal specification. The specifications of the other primitive refactorings can be found in the technical reports [34, 21].

**Encapsulate Variable.** Is used 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 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 ancestors and descendants, \( C \) included. Otherwise, the new methods may accidentally override (or be overridden by) an existing method, and then it is possible that the behaviour is not preserved.

**Example.** When applying the EncapsulateVariable refactoring on the field \texttt{name} of the class \texttt{Node} in Figure 1, both occurrences of \texttt{name} in line 07 should be replaced by a call to a getter method, most likely \texttt{getName()}. Note that in the constructor—which is not shown here—there is an assignment which should be replaced by the setter method.

**Potential Ambiguity.** Fowler makes the following note concerning the Encapsulate Variable refactoring: “If the field is an object and the client invokes a modifier on the object that is a use. Only use the setting method to replace an assignment.” This observation is necessary to explain precisely what is meant by “updating” the variable, i.e. for fields containing objects, it is only actual assignments that should be replaced by the setter methods; all others must be replaced by replacing them with “getter” methods. Unfortunately, this only works in a Java context where explicit pointer representations are omitted; in a C++ context issues like const parameters and pointers must be taken into consideration as well. To resolve this issue in C++, the precondition of EncapsulateField should be extended to cope with such language specific constructs.

**Pull Up Method.** Is used to move equivalent methods in subclasses into a common superclass. This refactoring removes code duplication and increases code reuse by inheritance.

**Precondition.** When a set of methods with signature \( m \) is pulled up into a class \( C \), all method definitions corresponding to this signature that are defined in the direct descendants of \( C \) must be removed and replaced by a single method definition now defined in \( C \). However, a tool should verify that this method implementation does not refer to any variables or methods defined in the subclass. Otherwise the pulled-up method would refer to an out-of-scope variable or method and then the transformed code would not compile. Also, no method definition with signature \( m \) may exist in \( C \), because a method signature cannot have more than one definition in the same class.

A strict specification of the precondition should verify whether the two method bodies are equivalent. A more pragmatic approach however, would leave this part to the programmer.
Example. The double print functions in Figure 2 are an ideal case for applying the PullUpMethod refactoring. While both methods have the same signature and functionally equivalent method bodies, it is not so easy for a tool to discern the latter. Indeed, line 03 retrieves the packet contents via a method invocation, while line 08 achieves this via a direct reference. Humans will have no trouble to confirm that the precondition is satisfied.

Potential Ambiguity. Unfortunately, the literature and the refactoring tools disagree on how “equivalent” these method bodies should be. The refactoring browser [29] for instance, requires that the byte code for the method bodies should be identical. Fowler [11] and Tichelaar [33] on the other hand, rely on a manual check by a human and do not verify whether the method bodies of the pulled up methods are identical. For pragmatic reasons, most programmers prefer that the equivalence of the method bodies is not verified, because otherwise the refactoring tool will refuse too many harmless cases.

2.3. Behaviour preservation

Defining the notion of behaviour preservation can be done in many ways, and an in-depth treatment of this topic is outside the scope of this paper, as we are mainly interested in the question whether graph rewriting is a suitable formalism for specifying refactorings. Moreover, all researchers agree that a full guarantee on preservation of behaviour is impossible. Therefore they use a relaxed notion of behaviour preservation, demanding that the program will perform the same actions before and after executing the refactoring. For each refactoring, one may list behaviour-related properties that need to be preserved, and that can be verified statically. For the feasibility study of this paper, we concentrate on three properties that are important and non-trivial for the selected refactorings: verifying whether the same method calls and the same variable accesses and updates occur. Section 6 discusses to which extent the selected refactorings are behaviour preserving in this sense.

- 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. Graph Representation

The obvious first step towards our goal of representing refactorings by graph rewriting is the introduction of a suitable graph representation of object oriented programs: one needs to decide which entities are represented by nodes, which relationships are represented by edges, and which information is represented by labels of nodes and edges. The graph representation
used may be viewed, in broad lines, as an abstract syntax tree augmented by extra edges. This representation is not intended as a proposal competing with the numerous representations that have been defined in the literature, and notably in the context of the UML. Rather it is a lightweight, formal vehicle for exploring the value of graph rewriting when formalizing refactorings. We assume that, if the approach is successful, the model can be incorporated easily into more widely used and practice-oriented formalisms. Recent experiments with the UML meta-model confirm that this assumption holds [36]. The notion of a graph to be used in the paper is the following one.

**Definition 3.1 (Graph)** Let \( \Sigma \) be a set of node labels and \( \Delta \) a set of edge labels. A (labeled) graph over \( \Sigma \) and \( \Delta \) is a 3-tuple \( G = (V_G, E_G, nlab_G) \), where \( V_G \) is the set of nodes, \( nlab_G : V_G \to \Sigma \) is the node labeling function and \( E_G \subseteq V_G \times \Delta \times V_G \) is the set of edges. An edge with label \( l \) from node \( v \) into node \( w \) will be denoted by \( v \xrightarrow{l} w \).

Thus programs are represented by typed, labeled, directed graphs. They are called *program graphs*. In a program graph, 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 \( \texttt{C} \) (i.e., a \( \texttt{C} \)-node). The set \( \Sigma = \{\texttt{C, M, MD, V, VD, P, E}\} \) of all possible node types is clarified in Figure 3. Method definitions (\( \texttt{MD} \)-nodes) have been separated from their method signatures (\( \texttt{M} \)-nodes) because the same method may have many possible definitions due to late binding and dynamic method lookup. For similar reasons, a distinction has been made between variable names (\( \texttt{V} \)-nodes) and their definition (\( \texttt{VD} \)-nodes). \( \texttt{MD} \)-nodes and \( \texttt{P} \)-nodes (method parameters) have an empty name.

Relationships between software entities (such as membership, inheritance, method lookup, variable accesses and method calls) are represented by *edges* between the corresponding nodes. 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 \( \texttt{C} \)-nodes *Workstation* and *Node*. Edge labels consist of an optional number and a type. The number is used to distinguish between edges with the same type and the same source node. This is for example the case with method parameters (\( p \)-edges) and (sub)expressions in a method definition (\( e \)-edges). The set \( \Delta = \{l, i, m, t, p, e, c, a, u\} \) of all possible edge types is clarified in Figure 3. For \( m \)-edges (membership), the type is often omitted in the figures. Note that the set of edges is a set of 3-tuples rather than a separate sort. Equivalently, each edge label can be viewed as a binary predicate. This choice was made because it facilitates the description of the embedding mechanism needed in Section 4.

Using this notation, an entire program can be represented by means of a single program 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 4 only shows the graph representation of the static structure of the LAN simulation.

A method definition is represented by a parse tree-like structure consisting of of \( \texttt{E} \)-nodes (the expressions in the parse tree) connected by \( e \)-edges. Outgoing edges from \( \texttt{E} \)-nodes express information about method calls and variable references (accesses and updates). For example, Figure 5 represents the method definitions in class *Node*. The method definition of \texttt{send} contains a sequence of two expressions, which is denoted by two numbered \( e \)-edges from
the MD-node to two different E-nodes. The second expression \texttt{nextNode.accept(p)} is a subexpression composed of a variable access (represented by an E-node with outgoing a-edge to the V-node with label \texttt{nextNode}) followed by a method call with one parameter (represented by an E-node with outgoing c-edge and e-edge). The actual parameter being used is the local parameter of the \texttt{send} method definition.

We have deliberately kept the graph model very simple to make it as flexible as possible. By attaching specific attributes to the nodes in the graph, one can extend it with language-dependent features. For example, one can attach visibility attributes to nodes of type \texttt{C, MD} and \texttt{VD} to deal with Java modifiers (such as \texttt{static, abstract, protected, final}). One can attach attributes to E-nodes to distinguish between different kinds of parse tree nodes (such as control statements, conditional statements, assignments, calls, exceptions, actual parameters). In a similar way, other Java-specific features can be modelled. For some language-specific constructs (e.g., Java interfaces and exception handling) new types of nodes or edges need to be introduced.

---

### Figure 3

<table>
<thead>
<tr>
<th>node type</th>
<th>description</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Class</td>
<td>\texttt{Node, PrintServer, Packet} accept, send, print</td>
</tr>
<tr>
<td>M</td>
<td>Method signature</td>
<td>{this.send(p);}</td>
</tr>
<tr>
<td>MD</td>
<td>Method Definition</td>
<td>\texttt{name, nextNode, contents}</td>
</tr>
<tr>
<td>V</td>
<td>Variable</td>
<td>\texttt{public Node nextNode} \texttt{p} contents</td>
</tr>
<tr>
<td>VD</td>
<td>Variable Definition</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>Parameter of a method definition</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>Expression in method definition</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>edge type</th>
<th>description</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>l</td>
<td>( M \rightarrow MD )</td>
<td>dynamic method lookup</td>
</tr>
<tr>
<td>i</td>
<td>( C \rightarrow C )</td>
<td>variable lookup</td>
</tr>
<tr>
<td>m</td>
<td>( VD \rightarrow C )</td>
<td>variable membership</td>
</tr>
<tr>
<td>t</td>
<td>( V \rightarrow C )</td>
<td>variable type</td>
</tr>
<tr>
<td>p</td>
<td>( MD \rightarrow P )</td>
<td>method return type</td>
</tr>
<tr>
<td>e</td>
<td>( MD \rightarrow E )</td>
<td>expression in method definition</td>
</tr>
<tr>
<td>c</td>
<td>( E \rightarrow M )</td>
<td>(dynamic) method call</td>
</tr>
<tr>
<td>a</td>
<td>( E \rightarrow { V</td>
<td>P } )</td>
</tr>
<tr>
<td>u</td>
<td>( E \rightarrow { V</td>
<td>P } )</td>
</tr>
</tbody>
</table>

The node type set \( \Sigma = \{ C, M, MD, V, VD, P, E \} \) and edge type set \( \Delta = \{ l, i, m, t, p, e, c, a, u \} \).
Figure 4. Static structure of LAN simulation

Figure 5. Method definitions in class Node

With such a graph representation it is possible to solve a first kind of ambiguities in normal language specifications, namely exactly which nodes are affected by a given refactoring. In the case of the ExtractField refactoring for example, there is a potential ambiguity concerning
setter methods and modifiers. With the choice of the graph representation it is possible to specify exactly which kind of updates are not allowed and include them in the precondition.

3.1. Well-formedness constraints

We need to impose constraints on the graph representation in order to guarantee that the program graphs are well-formed in the sense that they correspond to syntactically correct programs. Formalizing these well-formedness constraints is essential if one wishes to ensure that refactorings preserve syntactical correctness. We use two mechanisms to express these constraints: a type graph and forbidden subgraphs. Both are well-known from the literature about graph transformation [10, 5]. In order to specify the large or even infinite sets of subgraphs that are needed we use graph expressions; These are known from the literature about graph databases and query languages [19, 4].

Definition 3.2 (Type Graph) A type graph is a labelled graph over a set $\Sigma$ of node types and the set $\Delta$ of edge types. Let $G$ be a graph over $\Sigma, \Delta$ and let $TG$ be a typegraph. $G$ is typed over $TG$ if there exists a graph morphism from $G$ into $TG$: a node mapping that preserves sources, targets and labels. For the labels, only the type component is taken into account.

Figure 6 displays the type graph needed for our particular program graph representation. It should be generic enough to fit for a subset of any class-based object-oriented programming language. This type graph expresses a restriction on the program graphs that are allowed: it specifies which types of edges may occur between certain types of nodes. A program graph is well-formed only if there exists a graph morphism into the type graph.

![Figure 6. Type Graph](image)

The numbers 1 associated with the incoming $l$-edge and outgoing $m$-edge of node $MD$ express the additional constraints that, in a program graph, each node of type $MD$ may have at most one incoming $l$-edge and at most one outgoing $m$-edge.

The typing mechanism has an advantage that it is relatively easy to show that typing is preserved by the graph productions modeling the refactorings (as introduced in Section 4): when a correctly typed graph is rewritten using one of these productions then the resulting graph is also correctly typed. On the other hand it should be clear that not all well-formedness constraints can be expressed by the type graph. In particular a type graph is not suitable to
express the fact that certain substructures should not be present. Examples of such constraints are the following.

**WF-1** No two variables with the same name can be defined in a class.

**WF-2** No two methods with the same signature can be implemented in a class.

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

**WF-4** An expression in a method definition cannot refer to the method parameters belonging to another method definition.

Each of these constraints can be expressed by requiring that a set of structures do not occur as subgraphs of a program graph. An additional problem here is that these sets of forbidden subgraphs are in general infinite: their elements represent all conceivable ways in which the constraints can be violated, in all conceivable program graphs. Hence specifying these graphs explicitly is impossible and we need to introduce a notation for denoting such sets: graph expressions. Informally, a graph expression is a graph where edges are labeled by regular expressions over the set of edge labels. It specifies the set of all graphs obtained by replacing the edges by paths taking into account the edge labels. Formally one has the following.

**Definition 3.3 (Graph expression)** (1) A graph expression $GE$ is a graph $(V_{GE}, E_{GE})$ over $\Sigma$ and the set of regular expressions over $\Delta$. (2) Let $G$ be a program graph. An occurrence of $GE$ in $G$ is an injective mapping $oc : V_{GE} \rightarrow V_G$ such that

- for each node $v$ of $V_{GE}$, $nlab_{GE}(v)$ is the type of $oc(v)$
- for each edge $v \xrightarrow{\exp w}$ of $GE$, there is a path $p$ from $oc(v)$ into $oc(w)$ in $G$ such that $\text{word}(p) \in L(\exp)$, where $L(\exp)$ is the language of the regular expression $\exp$ and $\text{word}(p)$ is the sequence of types corresponding to the edges of $p$.

An example of such a graph expression $GE$ is shown in Figure 7. $\xrightarrow{e^*} 2$ is one of its three edges. Its edge label $e^*$ is a regular expression over edge type set $\Delta$. There are two occurrences, $oc_1$ and $oc_2$, of this graph expression $GE$ in the graph $G$ of Figure 5. These occurrences are given by the following mappings from $V_{GE}$ to $V_G$:

<table>
<thead>
<tr>
<th>$v$</th>
<th>$oc_1(v)$</th>
<th>$v$</th>
<th>$oc_2(v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

For $oc_1$, there is a path $p$ from node 1 to node 2 in $G$ consisting of three consecutive $e$-edges. Hence, in this case, $\text{word}(p) = e^3 \in L(e^*)$.

The constraints WF-1 through WF-4 can now be expressed formally by requiring that a program graph may not contain occurrences of the graph expressions of Figure 8.

As an example of the interpretation of these graph expressions, consider the graph expression for **WF-3**. Edge $\xrightarrow{e^*} 2$ denotes a parse tree traversal to an arbitrary subexpression in the method parse tree, $\xrightarrow{a|u}$ 3 denotes an access or update to a variable from within that subexpression, and $\xrightarrow{lmi}$ 4 denotes that the variable is defined in a subclass of the given class.
4. Expressing refactorings by graph productions

Although it seems quite natural to represent refactorings as graph productions to be applied to program graphs, there are a few issues that require attention.

(i) For certain refactorings, the part of the program graph that changes is not of a fixed size, which makes it impossible to describe the refactoring by a single production. This is the case for the PullUpMethod refactoring: if method $M$ is pulled up from class $C_1$ to $C_2$, then all remaining implementations of $M$ in subclasses of $C_2$ must be removed. This problem can be solved by expressing them as a combination of several graph productions. To control the order in which these productions have to be applied, we need the mechanism of controlled graph rewriting (also known as programmed or regulated graph rewriting) that has been studied in, e.g., [2, 17, 30].

(ii) The representation of a refactoring should not refer to concrete names of methods or classes to which it is applied, but rather to a generic pattern that has to be replaced by another one, and that can be instantiated by providing concrete methods or classes. This can be solved by using parameterised graph productions that contain variables for labels. Each parameterised production may be viewed as a specification of an infinite set of productions.
A third perhaps more fundamental problem is that the representation of a refactoring should not depend on the context of that refactoring in a specific program graph. Consider, for example, the EncapsulateVariable refactoring of Figure 9 (only the relevant parts of the graphs are shown): essentially, 2 MD-nodes are added, and accesses to name become calls to getName. The outgoing edges of the gray E-nodes represent accesses and calls from within method bodies that are not directly involved in the refactoring. When using the well-known double pushout approach [8] to graph rewriting, one has to consider the E-nodes as part of the LHS and RHS of the corresponding production because otherwise there would be dangling edges which are not allowed. However, incorporating such nodes into the production would make these more complex and, moreover, one would need an infinite set of productions because the number of E-nodes needed can be arbitrarily large. This problem can be solved by using productions with an embedding mechanism, where dangling edges are allowed. It will be briefly introduced in the next section, using only notions from basic set theory.

Figure 9. The desired graph production for the EncapsulateVariable refactoring on the LAN example

4.1. Graph transformation: definitions

To formalize the refactorings in this paper as graph transformation rules, some elementary definitions about these rules need to be given first. We use embedding-based graph transformation systems. Such systems have been studied for a long time (see e.g. [9]). The approach in this paper is very similar to the one from [14].

Definition 4.1 (production) A production is a 4-tuple \((L, R, \text{Emb}_{\text{in}}, \text{Emb}_{\text{out}})\) where \(L\) and \(R\) are graphs and \(\text{Emb}_{\text{in}}, \text{Emb}_{\text{out}}\) are subsets of \((\Delta \times V_L) \times (\Delta \times V_R)\).

In order to define what it means to apply a production to a graph, one needs the following notions:
1. Let $G$ be a graph and let $W \subseteq V_G$. The full subgraph of $G$ on $W$ is the graph $G_W = (W, E_G \cap (W \times \Delta \times W), nlab)$ where $nlab$ is the restriction of $nlab_G$ to $W$.

2. Let $G$ be a graph, let $W$ be a set and let $f : V_G \rightarrow W$ be an injective function. Then $f(G)$ denotes the graph $(f(V_G), \{(f(v), \delta, f(w))|(v, \delta, w) \in E_G\}, nlab_G \circ f^{-1})$.

3. Let $K$ and $G$ be graphs. An occurrence of $K$ in $G$ is an injective function $oc : V_K \rightarrow V_G$ such that $oc(K)$ is the full subgraph of $G$ on the set $oc(V_K)$. Hence the notion of an occurrence $oc$ of $K$ in $G$ used in this paper is rather restricted: not only are occurrences injective, but it is also required that all edges of $G$ between nodes of $oc(K)$ are also edges of $oc(K)$.

The derivation relation between graphs is now defined as follows:

**Definition 4.2.** Let $G$, $H$ be graphs, let $\pi = (L, R, Emb_{in}, Emb_{out})$ be a production and let $m : L \rightarrow G$ and $n : R \rightarrow H$ be occurrences such that $n(V_R) \cap V_G = \emptyset$. Then $G$ directly derives $H$ using $\pi$ via $m$ and $n$, if

1. $V_H = (V_G - m(V_L)) \cup n(V_R)$
2. $nlab_H(x) = \begin{cases} nlab_G(x), & \text{if } x \in V_G - m(V_L) \\ nlab_{n(R)}(x), & \text{if } x \in n(V_R) \end{cases}$
3. $E_H$ is the set of 3-tuples $(u, \delta, w) \in (V_H \times \Delta \times V_H)$ such that one of the following four conditions holds:

   (i) $(u, \delta, w) \in E_G$
   (ii) $(u, \delta, w) \in E_{n(R)}$
   (iii) there exist nodes $w' \in V_R$ and $v \in V_L$ such that $w = n(w')$, $(u, \gamma, m(v)) \in E_G$ and $((\gamma, v), (\delta, w')) \in Emb_{in}$
   (iv) there exist nodes $w' \in V_R$ and $v \in V_L$ such that $u = n(u')$, $(m(v), \gamma, w) \in E_G$ and $((\gamma, v), (\delta, w')) \in Emb_{out}$

Thus, informally, $m(L)$ is replaced by $n(R)$. An item $((\gamma, v), (\delta, w'))$ of $Emb_{in}$ is used to redirect incoming edges of node $m(v)$ to node $m(w')$, at the same time changing its label from $\gamma$ to $\delta$. The situation is illustrated by Figure 10. The interpretation of $Emb_{out}$ is similar to that of $Emb_{in}$, but for outgoing edges.

Note that it is easy to give a sufficient condition for a production to preserve the correct typing of graphs: let $G$ be a graph, typed over a type graph $TG$, let $\pi = (L, R, Emb_{in}, Emb_{out})$ be a production and assume that $G$ directly derives $H$ using $\pi$. Then $H$ is typed over $TG$ if the following holds:

- $R$ is typed over $TG$
- For each $((\gamma, v), (\delta, w)) \in Emb_{in}$, and for each node label $a$, $(a, \delta, nlab_R(w))$ is an edge of $TG$.
- For each $((\gamma, v), (\delta, w)) \in Emb_{out}$, and for each node label $a$, $(lab_R(w), \delta, a)$ is an edge of $TG$.

It is straightforward to check that the productions mentioned in this chapter satisfy these conditions with respect to the typegraph introduced in Section 3 (Figure 6).
4.2. Encapsulate Variable

Figure 11 shows the parameterised productions needed to express the refactoring `EncapsulateVariable(var, getter, setter)`. Here `var`, `getter`, `setter` are formal parameters, to be
replaced by concrete names in order to obtain concrete productions. Using the mechanism of controlled graph rewriting, the EncapsulateVariable refactoring can be expressed using imperative programming constructs like sequences and iteration: $P_1$ is applied first, and subsequently $P_2$ is applied as often as possible.

In Figure 11, the LHS and RHS are separated by means of an arrow symbol. All nodes are numbered. Nodes that have a number occurring in both the LHS and the RHS are preserved by the rewriting (e.g., node 1). 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 in $P_1$) are newly created.

Figure 11 also specifies the embedding mechanism for both productions $P_1$ and $P_2$. For example, for $P_1$, the item $(t, 1) \rightarrow (t, 1), (1.p, 2), (t, 3)$ means that the return type of method getter and the argument type of method setter are the same as the type of variable var. In $P_2$, $(a, 1) \rightarrow (c, 3)$ means that each access of the variable var (represented by an incoming $a$-edge to node 1) is replaced by a method call to the getter method (represented by an incoming $c$-edge to node 3). $(m, 0) \rightarrow (m, 0), (m, 4), (m, 5)$ means that the method definitions (nodes 4 and 5) that correspond to the getter and setter methods must be implemented in the same class as the one to which the variable definition (node 0) belongs.

Figure 12 shows the concrete production instances for EncapsulateVariable(name, getName, setName) that may be applied to the graph of Figure 5, in the context of the LAN example. For $P_2$, we have also represented (in gray) nodes that do not belong to the production, but that would have to be included in an approach where dangling edges are not allowed. These nodes represent accesses to the variable that is encapsulated. When applying the production, the two gray $E$-nodes in Figure 9 match the gray $E$-nodes of Figure 5. In this particular example, $P_2$ is only applied once, because there is only one definition for variable name. In other situations, however, there can be more than one definition of the same variable in the inheritance chain. If this is the case, $P_2$ will be applied repeatedly, and the refactoring will introduce accessor methods in each class where the encapsulated variable is defined.

### 4.3. Pull Up Method

The second refactoring to be expressed as a set of parameterised graph productions with embedding mechanism is PullUpMethod(parent,child,name). It moves the implementation of a method name in some child class to its parent class, and removes all isomorphic copies of this function in its sibling classes. Again, we need to use controlled graph rewriting to express the transformation as a combination of two parameterised productions $P_1$ and $P_2$ (see Figure 13). $P_1$ moves the definition of method name one level higher in the inheritance hierarchy (i.e., from child to parent), and is followed by $P_2$, which removes the definitions of method name; a consequence of this action is that the parse trees of that method become unreachable from the other parts of the program graph and can be garbage collected. Because a single execution of $P_1$ might break existing behaviour, its execution must immediately be followed by the repetitive execution of $P_2$, until the LHS of this productions no longer occurs in the
program graph. As a consequence, the control flow of this refactoring is identical to that of the Encapsulate Variable refactoring.

![Diagram](image_url)

**Figure 12.** Instantiated graph productions for EncapsulateVariable(name,getName,setName) obtained from the productions of Figure 11

<table>
<thead>
<tr>
<th>production</th>
<th>incoming edges</th>
<th>outgoing edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>$\forall (\tau, n) \in \Delta \times {1, 2, 3, 4}$: $(\tau, n) \rightarrow (\tau, n)$</td>
<td>$\forall (\tau, n) \in (\Delta \times {1, 2, 3, 4}) \setminus {(m, 3)}$: $(\tau, n) \rightarrow (\tau, n)$</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$\forall (\tau, n) \in \Delta \times {1, 3, 4, 5}$: $(\tau, n) \rightarrow (\tau, n)$</td>
<td>$\forall (\tau, n) \in \Delta \times {1, 3, 4, 5}$: $(\tau, n) \rightarrow (\tau, n)$</td>
</tr>
</tbody>
</table>

**Figure 13.** Graph productions $P_1$ and $P_2$ with embedding table for the PullUp-Method(parent,child,name) refactoring
5. Refactoring Preconditions

As explained in Subsection 2.2, each refactoring has to ensure that the well-formedness constraints remain valid, and has to satisfy a number of additional conditions, called refactoring conditions. For example, in the presence of inheritance, a refactoring should avoid accidental method overriding, i.e., a newly introduced method definition or variable definition does not override (resp. is not overridden by) an existing definition in a superclass (resp. subclass). (RC-1)

These constraints can be expressed in a natural way as preconditions or postconditions on the graph production. For efficiency reasons, it is desirable to use preconditions instead of postconditions. 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 [12, 13]. Heckel [13] showed that graph productions with postconditions can be transformed into corresponding graph productions with negative preconditions only. He proved formally that the restriction to rules having negative preconditions with injective satisfaction does not decrease the expressive power of the corresponding graph grammars. First of all, a rule with postconditions can always be transformed into an equivalent (but possible expanded) rule with preconditions (with injective satisfaction) only. Secondly, a rule with an arbitrary precondition can always be replaced by a set of rules having negative preconditions only. Kniesel [15] explored this idea in the context of refactoring, and showed how it can be used to build tools that facilitate the static composition of refactoring transformations.

5.1. Encapsulate Variable

Figure 14 presents the negative preconditions needed in order for EncapsulateVariable to satisfy refactoring constraint RC-1. The conditions specify that no ancestor or descendant of the class containing var define a method with name setter. Two similar negative application conditions are needed for the getter method.

Informally, well-formedness constraint WF-1 is preserved since EncapsulateVariable does not introduce or move any variables or variable definitions. Constraint WF-2 is preserved thanks to the preconditions of Figure 14, in the special case where i* is the empty word: by disallowing a definition of method setter (or getter) in the class of the encapsulated variable, we guarantee that such a method definition can be safely introduced by the transformation without giving rise to multiple method definitions for setter (or getter) in the same class. Constraint WF-3 is preserved because EncapsulateVariable only introduces a new variable access and update to a variable that is defined by the class itself. Hence, it doesn’t affect variables defined in descendant classes. Constraint WF-4 is preserved because the method parameter of the setter method is only referred to from within its own method definition.

5.2. Pull Up Method

Figure 15 presents two negative preconditions for PullUpMethod, or more specifically, for its subproduction P_1. The condition on the left specifies that the method name to be pulled up should not yet be defined in parent, and the condition on the right specifies that the method...
definition to be pulled up should not refer to (i.e., access or update) variables outside the scope of the parent.

Note that we did not specify the precondition that the method bodies of all pulled up methods should be equivalent. In principle, all this requires is stating that there should exist an isomorphism between the two subgraphs representing the method bodies. In practice however, verifying whether this isomorphism is there requires more sophisticated specification mechanisms such as hierarchical graphs [10]. Since this check is usually ignored in practical tool implementations as well, we preferred to keep the model transparent and concise and do not include this isomorphism in the precondition. Of course this part of the specification is essential when one really wants to resolve all potential ambiguities.

To prove that \textit{PullUpMethod} preserves well-formedness constraint \textit{WF-2} it suffices to show that an application of \textit{P}_1 or \textit{P}_2 to a graph containing no occurrences of \textit{WF-2} cannot result in a graph in which \textit{WF-2} does occur. Assume, to the contrary, that a new occurrence of \textit{WF-2} is created in a step using \textit{P}_1. Clearly one has to consider only occurrences of \textit{WF-2} that overlap with the RHS of the production. Taking into account the node labels, and the fact that each \textit{MD} has exactly one incoming \textit{l}-edge and only one outgoing \textit{m}-edge, the overlap must consist of either node 1, node 4, nodes 1 and 4, or nodes 1, 3 and 4. It immediately follows from the form of the embedding relation that all edges incident to nodes 1 and 4 are preserved, and thus the first three possibilities would imply that an occurrence of \textit{WF-2} was present already before the rewriting, contradicting the assumption. The last case, where nodes 1, 3 and 4 would belong to an occurrence of \textit{WF-2}, is excluded by the first negative precondition for \textit{P}_1 (left part of Figure 15), which states that the method to be pulled up should not yet exist in the parent class. To see that no new occurrences of \textit{WF-2} can be created by applying production \textit{P}_2, it suffices to see that the effect of \textit{P}_2 is simply the removal of a node together with all its incident edges.

Well-formedness constraints \textit{WF-1}, \textit{WF-3} and \textit{WF-4} can be proven in a similar way for \textit{PullUpMethod}. Intuitively, well-formedness constraint \textit{WF-1} is preserved since it does not introduce or redirect any variables or variable definitions. Constraint \textit{WF-3} is preserved thanks
to the precondition on the right of Figure 15, and constraint WF-4 is preserved because the refactoring does not introduce or change any method definition.

6. Preservation of behaviour

In this section we explore the use of our formalization for proving that certain properties are preserved by refactorings. In particular the properties of access, update and call preservation are considered. In order to formalize them, one needs a way to express the fact that, for each occurrence of a graph expression before a refactoring takes place, there exists a corresponding occurrence after the refactoring. To this aim the notion of a tracking function $tr$ is introduced: if $G$ directly derives $H$ using a production $p$, then $tr$ is a function from $V_G$ into $V_H$, enabling one to relate occurrences of graph expressions in $G$ and $H$. Part of $tr$ is specified together with each production, whereas the remaining part of $tr$ is simply the identity function on the part of the graph that is not rewritten. Formally one has the following:

**Definition 6.1 (Preservation of a graph expression)** Let $GE$ be a graph expression, let $G$ and $H$ be program graphs and let $tr : V_G \rightarrow V_H$ be a node mapping. Then $tr$ preserves $GE$ if, for each occurrence $oc$ of $GE$ in $G$, $tr \circ oc$ is an occurrence of $GE$ in $H$.

In order to construct the tracking functions for graph rewriting steps, each production is equipped with its own function $tr_p$ mapping the nodes of the LHS into those of the RHS.

**Definition 6.2 (Tracking function)** If $G$ derives $H$ using a production $p = (L, R, Emb_{in}, Emb_{out})$ via $m$ and $n$, then the tracking function $tr : V_G \rightarrow V_H$ is defined by

$$tr(v) = \begin{cases} n \circ tr_p \circ m^{-1}(v), & \text{if } v \text{ is a node of } m(L) \\ v, & \text{otherwise} \end{cases}$$

The production $p$ preserves the graph expression $GE$ if, for each application of $p$ to a graph, the corresponding tracking function preserves $GE$.

For refactoring PullUpMethod of Figure 13, the tracking functions are defined as follows. For production $P_1$, $tr_{P_1}$ is the identity function. For production $P_2$, the tracking function is illustrated in Figure 16. $tr_{P_2}(1) = 1$, $tr_{P_2}(3) = tr_{P_2}(6) = 3$, $tr_{P_2}(4) = 4$, and $tr_{P_2}(5) = 5$.

6.1. Types of behaviour preservation

The types of behaviour preservation informally introduced in Subsection 2.3 can be expressed formally using the definition of graph expressions of Subsection 3.1: the idea is that for each occurrence of a graph expression that is present before a rewriting, there must be a corresponding occurrence after the rewriting.

The graph expression $MD \xrightarrow{(e|c) \cdot a} V \xrightarrow{t} VD$ can be used to express the property of access preservation. It specifies all possible access paths from a method definition ($MD$-node) to a variable ($V$-node) defined in a $VD$-node. Access preservation means that, for each occurrence of $MD \xrightarrow{(e|c) \cdot a} V \xrightarrow{t} VD$ in the initial graph to be rewritten, there is a corresponding
occurrence of this graph expression in the resulting graph. In a similar way, we can express update preservation by means of the graph expression \( MD \xrightarrow{e} M \xrightarrow{c} MD \).

Graph expression \( MD \xrightarrow{e} M \xrightarrow{c} MD \) formalises the property of call preservation. For each method definition (\( MD \)-node) that performs a method call (\( c \)-edge) to some signature (\( M \)-node) that is implemented by some method definition (\( MD \)-node) in the initial graph, there should still be a call to the same method definition in the resulting graph.

### 6.2. Preservation of Behaviour: Pull Up Method

As an example of the way our formalization may be used to prove properties related to the preservation of behaviour, consider call preservation for PullUpMethod. In order to prove this, one has to prove that the productions \( P_1, P_2 \) of Figure 13 preserve the graph expression \( GE = MD \xrightarrow{e} c \xrightarrow{c} M \xrightarrow{c} l \xrightarrow{c} MD \). It is clearly sufficient to prove the preservation of the subexpressions \( GE_1 = MD \xrightarrow{e} c \xrightarrow{c} M \) and \( GE_2 = M \xrightarrow{c} l \xrightarrow{c} MD \). (see Figure 17). Only the preservation of \( GE_1 \) will be considered in detail, since the preservation of \( GE_2 \) can be shown in a similar, but simpler, way. The proof consists essentially in considering the various ways in which an occurrence of \( GE_1 \) can be positioned relative to the parts of \( G \) that are replaced, and inspecting the embedding relation to verify that, in each case, the edges are redirected in an appropriate way. For only one of the cases one needs to take into consideration the refactoring conditions.

(A) First, it is shown that \( P_1 \) preserves \( GE_1 \). Let \( G \) and \( H \) be graphs such that \( G \) directly derives \( H \) using \( P_1 \) via occurrences \( m \) and \( n \), and let \( tr \) be the corresponding tracking function. Let \( oc : V_{GE_1} \rightarrow V_G \) be an occurrence of \( GE_1 \) in \( G \). Then there exists a path \( v_0 \xrightarrow{c} v_1 \xrightarrow{c} v_2 \xrightarrow{c} \ldots \xrightarrow{c} v_{n-1} \xrightarrow{c} v_n \) in \( G \) such that \( n \geq 1 \), and \( v_0, v_n \) are the images under \( oc \) of nodes 1 and 2 of \( GE_1 \), respectively. It remains to be shown that there is a path \( tr(v_0) \xrightarrow{c} tr(v_1) \xrightarrow{c} tr(v_2) \xrightarrow{c} \ldots \xrightarrow{c} tr(v_{n-1}) \xrightarrow{c} tr(v_n) \) in \( H \).
Firstly, consider the nodes $v_1, v_2, \ldots, v_{n-1}$. It follows from Figure 3 that these have label $E$, and hence do not belong to the replaced part $m(L)$ of $G$ (where $L$ is the LHS of $P_1$). Thus $tr$ is the identity on these nodes, and there is a path $tr(v_1) \xrightarrow{e} tr(v_2) \xrightarrow{e} \cdots \xrightarrow{e} tr(v_{n-1})$ in $H$.

Secondly, consider node $v_0$. If $v_0$ does not belong to $m(L)$, then $tr(v_0) = v_0$ and $tr(v_0) \xrightarrow{e} tr(v_1)$ in $H$. If $v_0$ belongs to $m(L)$, then $v_0$ corresponds to node 3 in $L$, and $tr(v_0)$ corresponds to node 3 of the RHS of $P_1$. Since for $P_1$, $((e, 3), (e, 3)) \in Emb_{out}$ one has $tr(v_0) \xrightarrow{e} tr(v_1)$ in $H$.

Finally, consider node $v_n$. If $v_n$ does not belong to $m(L)$, then $tr(v_n) = v_n$ and $tr(v_{n-1}) \xrightarrow{e} tr(v_n)$ in $H$. If $v_n$ is a node of $m(L)$, then $v_n$ corresponds to node 6 of $L$, and $tr(v_n)$ corresponds to node 4 of the RHS of $P_1$. Since in $P_1$, $((c, 4), (c, 4)) \in Emb_{in}$, $tr(v_{n-1}) \xrightarrow{e} tr(v_n)$ in $H$. Thus one concludes that $GE_1$ is preserved by $P_1$.

(B) Now consider $P_2$: assume that $G$ directly derives $H$ using $P_2$ via occurrences $m$ and $n$, and let $tr$ be the corresponding tracking function. Let $oc, v_0 \ldots v_n$ be as in (A). For nodes $v_1, \ldots, v_{n-1}$ one may follow the same reasoning as in (A), and hence there is a path $tr(v_1) \xrightarrow{e} tr(v_2) \xrightarrow{e} \cdots \xrightarrow{e} tr(v_{n-1})$ in $H$.

Now consider $v_0$. If $v_0$ does not belong to $m(L)$, i.e. $v_0$ is not replaced, then again $tr(v_0) \xrightarrow{e} tr(v_1)$. If $v_0$ is a node of $m(L)$, then it corresponds to node 3 or to node 6 of the LHS of $P_2$. If it corresponds to node 3, then it follows from $((e, 3), (e, 3)) \in Emb_{out}$ that $tr(v_0) \xrightarrow{e} tr(v_1)$ in $H$. If it corresponds to node 6, however, then one uses the fact that, a correct application of Pull Up Method can take place only if the syntax trees under nodes 3 and 6 of $P_2$ are isomorphic; this implies that there exists another occurrence $oc'$ of $GE_1$ in $G$ where node 1 of $GE_1$ corresponds to node 3 in the LHS of $P_2$, and since the tracking function of $P_2$ maps nodes 3 and 6 of the LHS to the same node of the RHS, the fact that $oc'$ is preserved implies that $oc$ is preserved. Thus the case where $v_0$ corresponds to node 6 needs not to be considered separately.

Finally, consider node $v_n$. If $v_n$ is not a node of $m(L)$, then $tr(v_{n-1}) \xrightarrow{e} tr(v_n)$. If $v_n$ is a node of $m(L)$, then it corresponds to node 4 of $P_2$, and it follows from $((c, 4), (c, 4)) \in Emb_{in}$ that $tr(v_{n-1}) \xrightarrow{e} tr(v_n)$. We conclude that $P_2$ also preserves $GE_1$.

![Figure 17. Subexpressions $GE_1$ and $GE_2$ of the call preservation property](image-url)
6.3. Preservation of Behaviour: Encapsulate Variable

We only give an informal discussion of update preservation for EncapsulateVariable, because the proof is similar to the one above. It suffices to show that the graph expression $MD \xrightarrow{(e|c)} u V \xrightarrow{t} VD$ is preserved by each method definition $MD$ that updates the variable $var$ that is being encapsulated. It follows from the form of the graph productions $P_1$ and $P_2$ of Figure 11 that this is the case. This is illustrated in Figure 18, that shows how a direct update of $var$ is replaced by a slightly longer path that still preserves the graph expression $MD \xrightarrow{(e|c)} u V$. The graph productions do not change anything to occurrences of graph expression $V \xrightarrow{t} VD$. Access preservation can be shown in a similar way. Call preservation is also trivial since the refactoring does not change any method calls or method definitions. (It does add new method signatures and method definitions, but this does not affect existing method calls.)

![Figure 18](image)

Figure 18. A path from $MD$ to $var$ remains but becomes longer after the Encapsulate Variable refactoring.

7. Tool Validation

In order to validate our approach in practice, three steps are needed: converting source code into a graph, applying graph transformations to this graph, and verifying the preconditions and invariants in the graph representation. The first step, writing a Java to graph parser was implemented by a student during a programming project. For the second and third step, we implemented refactorings in two state-of-the-art graph transformation tools, namely Fujaba and AGG. Fujaba [16, 23] is tightly integrated with Java, and has a programmer-friendly notation of graph productions using UML notation and story diagrams. This requires that the nodes in our typegraph are converted to UML classes, and the edges between nodes converted to UML associations. The graph rewriting operations and preconditions were expressed using Fujaba’s story diagrams. Figure 19 shows how the Pull Up Method refactoring is specified as a story diagram. It turned out to be straightforward to specify the graph productions for the two refactorings in this paper. Most of the preconditions could also be expressed without problems. However, using graph patterns for expressing the equality of two subgraphs in the program graph was not possible. This is for example needed for checking the equivalence of two method bodies for the PullUpMethod refactoring. We circumvented this problem by implementing the required precondition directly in Java code using a statement activity.

AGG [32] is a graph transformation tool implemented in Java. It supports the specification of type graphs with multiplicities and attributes. Node and edge attributes act like ordinary variables to which a value can be assigned. By specifying Java expressions, the graph...
production rules can specify how attribute values need to be updated by the transformation. The graph productions can also contain negative application conditions and extra well-formedness constraints that need to be satisfied when the production rule is applied in the context of an input graph. Figure 20 shows how the Encapsulate Variable refactoring can be specified in AGG. The name of the getter and setter method introduced by the transformation depends on the name of the variable. This constraint can be expressed by means of the Java expressions \texttt{s.equals("set"+v)} and \texttt{g.equals("get"+v)} where \(v\) is the name of the variable, \(s\) the name of the setter method, and \(g\) the name of the getter method.

We did an experiment to map the type graph in section 3 onto the UML metamodel [24]. As such, our approach can be used for supporting UML model refactorings as well. The main difference with UML is that our type graph also takes information about method implementations into account, which was needed for detecting or guaranteeing behaviour preservation (see section 6). To achieve the same effect, Van Gorp et al. [36] provided a set of minimal extensions to the UML metamodel to enable source-consistent refactoring implementations on UML. Additionally, OCL constraints were used to express refactoring preconditions, postconditions and invariants.
8. Lessons Learned

In the previous sections, we provided a stepwise specification of the effects refactorings have on source code. We used graph rewriting as our formal basis, but at the outset it was unclear whether the available formal techniques and tools would provide the necessary scalability. From this experiment, we learned that it was feasible to use graph rewriting as a formal basis for specifying refactorings, and that such a specification may resolve subtle ambiguities that creep in natural language specifications. Also, we have relied on the underlying graph rewriting theory during the formal analysis of such refactorings. For example, in [21] we used the idea of critical pair analysis to reason about parallel and sequential dependencies of refactorings, and we performed some experiments with this in AGG.

Nevertheless we encountered some problems that required special treatment. All of these problems were due to the inevitable trade-offs one has to make between the transparency, conciseness, elegance and expressiveness of a formal specification.

(a) Graph Representation. We specified a formal graph representation of an abstract syntax tree augmented with variable access and method invocation relations (section 3). The specification of the graph structure itself was straightforward, although special care was taken to balance conciseness against transparency. For instance we opted for omitting control-flow structures from the graph itself, but included a generic (sub)expression node type, where one can attach attributes to distinguish between different kinds of language constructs.
More interesting is the way we ensure that the rewriting rules result in syntactically correct programs. We learned that the notion of type graphs, well-known in the graph rewriting community, does not allow for an elegant specification of all the syntactical constraints that are needed. Therefore, we introduced the notion of graph expressions (Definition 3.3), which have their formal origins in regular expressions.

(b) Rewrite Rules. We specified rewrite rules for two representative refactorings (section 4). Here we learned that basic rewrite rules are insufficient to achieve a transparent and concise specification, and hence we adopted a few extra mechanisms. In order to avoid infinite sets of productions, we used parameterized node labels. Since, even then, using the usual rewrite rules would result in an infinite set of productions, we used an embedding mechanism similar to the one of [14], to specify how to redirect incoming and outgoing edges. Finally, we learned that a transparent specification requires the use of controlled graph rewriting (among others studied in [2, 17, 30]). Indeed, to ensure that the rewrite rules resemble the manipulations a refactoring tool performs on an abstract syntax tree, one must be able to express repeated applications of a single rule. Also, while specifying other refactorings than the ones described in this paper, we experienced that normal, embedding-based graph transformation rules pose difficulties for the refactorings that require manipulation of lists and sets, or require a flexible representation of an arbitrary-shaped program component. For example, it is very difficult to express the Push down Method refactoring using basic graph transformation techniques because a complete syntax tree has to be copied to a set of classes. In [35], it is shown how graph transformation rules with variables can be used to enhance the specification power of graph transformation rules.

(c) Preconditions. For the two refactorings of interest we specified preconditions which ensure that firing the rewrite rules does not break the syntactical correctness of a program, as expressed by the well-formedness constraints (section 5). Here we learned that graph rewriting with negative application conditions are a very natural way to express preconditions.

We mention the issue of the isomorphism between two subgraphs. Normally, the precondition for the PullUpMethod, includes the presence of an isomorphism between the parse trees of the method definitions in the subclasses. However, such an isomorphism is difficult to specify completely and would reduce the transparency and elegance of the graph model. Since this check is usually abandoned in practical tool implementations as well, we preferred to sacrifice the expressiveness of our model instead and choose to leave this precondition unverified.

(d) Behaviour Preservation. We used our formalism to prove that a rewrite rule applied to a graph satisfying the necessary precondition preserves properties like preservation of calls, preservation of accesses and preservation of updates (section 6). Here we showed the expressiveness of our approach, and confirmed that the graph rewriting formalism is well suited for specifying the effect of refactorings on source code.

An interesting notion here was the introduction of a Tracking Function (Definition 6.2), which allowed us to relate occurrences of nodes before and after applying a rewrite rule.
(e) Tool Validation. We briefly reported on the experiment we performed converting Java programs into our graph representation and specifying the rewrite rules and the preconditions in two graph transformation tools: Fujaba and AGG. These experiments confirmed to validity of our model, in the sense that the model is expressive enough to represent Java and UML programs and that the specification of the refactorings as rewrite rules is indeed feasible.

(f) Completeness. Given that the set of possible refactorings is infinite, we cannot show that graph rewriting is a sufficient basis for specifying all refactorings. However, refactoring theory and tools assume that there exists a small set of primitive refactorings which can be combined into larger composite refactorings. The current list of primitive refactorings consists of nineteen elements grouped in four categories, namely Class (CreateClass, RenameClass, RemoveClass), Method (AddMethod, RenameMethod, RemoveMethod, PullUpMethod, PushDownMethod, ExtractMethod, InlineMethod, MoveMethod), Parameter (AddParameter, RemoveParameter), Variable or Field (CreateVariable, RenameVariable, RemoveVariable, EncapsulateVariable, PullUpVariable, PushDownVariable). The two refactorings specified in this paper form a representative selection out of this set, we refer the interested reader to other material we wrote on the subject for the complete list [34].

9. Conclusion

This paper investigated the feasibility of using graph rewriting as a formal specification for refactoring. Based on the specification of a number of typical refactorings we conclude that this formalism is indeed suitable for specifying the effect of refactorings, because (i) graphs can be used as a transparent representation of the source code; (ii) graph rewriting rules are a concise and elegant way to specify the source-code transformations implied by a refactoring; (iii) the formalism is expressive enough to prove that refactorings preserve certain kinds of relationships (updates, accesses and invocations) that can be inferred statically from the source code. As a proof-of-concept, we implemented our approach in the state-of-the-art graph rewriting tools Fujaba [23] and AGG [32].

With such a formal specification it is possible to give a precise and operational definition of what each refactoring does with the source code. Moreover, the model allows formal reasoning about certain properties of refactorings, such as the preservation of behavior or what happens when we combine several refactorings into composites.

The larger issue is of course the practical relevance of such a formal specification. We took great care to balance the transparency, conciseness, elegance and expressiveness of our specification, so that it would be understandable by tool builders and point out potential holes in natural language specifications. Nevertheless, the finer details of the specification can only be understood by persons sufficiently fluent in graph rewriting, which is certainly not the case for the majority of tool builders. Yet once these are available, graph rewriting can be used to write a refactoring “standard”. Whether that is desirable is for others to decide; we at least have shown that it is feasible.
REFERENCES


AUTHORS' BIOGRAPHIES

**Tom Mens** obtained the degrees of Licentiate in Mathematics in 1992, Advanced Master in Computer Science in 1993 and PhD in Science in 1999 at the Vrije Universiteit Brussel. At this university, he was a teaching and research assistant, a research councillor for two industrial research projects, and a postdoctoral fellow of the Fund for Scientific Research Flanders (FWO). In October 2003 he became a lecturer at the Université de Mons-Hainaut, where he currently leads a research lab on software engineering. His main research interests are formal methods, software evolution and model-driven software engineering, and he published numerous peer-reviewed articles on these topics. He has been co-organiser, PC member and referee of many international workshops and conferences on software evolution and software engineering. He co-founded and coordinates two international scientific research networks on software evolution, and is involved in several research project on software restructuring.

**Niels Van Eetvelde** is a PhD student in the Department of Mathematics and Computer Science at the University of Antwerp in Belgium. His main research topic is software refactoring. He focuses on the use of graph rewriting as a formal specification mechanism for refactorings.

**Serge Demeyer** is a professor in the Department of Mathematics and Computer Science at the University of Antwerp in Belgium where he leads a research group investigating the theme of Software Reengineering (LORE - Lab On REengineering). His main research interest concerns reengineering (more precisely, reengineering in an object-oriented context) but due to historical reasons he maintains a heavy interest in hypermedia systems as well. He is an active member of the corresponding international research communities, serving in various conference organization and program committees. He has written a book entitled “Object-Oriented Reengineering Pattern”.

**Dirk Janssens** is a professor in the Department of Mathematics and Computer Science at the University of Antwerp in Belgium. His research interest is the development and application of formal methods, in particular graph rewriting, in Software Engineering. Recently his main focus is on software refactoring and evolution.