A final version of this paper will appear in a Special Issue of Fundamenta Informaticae (IOS Press) devoted to Graph Rewriting in 2003

Formalising Refactorings with Graph Transformations*

Tom Mens† C
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
tom.mens@vub.ac.be

Niels Van Eetvelde, Dirk Janssens, Serge Demeyer
Department of Mathematics and Computer Science
Universiteit Antwerpen
Middelheimlaan 1, 2020 Antwerpen, Belgium
{niels.vaneetvelde, dirk.janssens, serge.demeyer}@ua.ac.be

Abstract. 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. This paper introduces a graph representation of those aspects of the source code that should be preserved by refactorings, and graph transformations as a formal specification for the refactorings themselves. To this aim, we use type graphs, forbidden subgraphs, embedding mechanisms, 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.

Keywords: graph transformation, refactoring, behaviour preservation

1. Introduction

Refactorings are software transformations that restructure an object-oriented program while preserving its behaviour [8, 17, 18]. 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 [8].

Although it is possible to refactor manually, tool support is considered crucial. Tools such as the *Refactoring Browser* support a semi-automatic approach [20], which has also been adopted by industrial

*Research carried out in the framework of research project G.0452.03 of the Fund for Scientific Research - Flanders (Belgium).
†Postdoctoral Fellow of the Fund for Scientific Research - Flanders
CCorresponding author

strength software development environments[1]. Other researchers have demonstrated the feasibility of fully automated tools [3], studied ways to make refactoring tools less dependent on the implementation language being used [25] and investigated refactoring in the context of UML casetools [1, 24].

Despite the existence of such tools, the notion of *behaviour preservation* is poorly defined. This is mainly so because most definitions of the behaviour of an object concentrate on the run-time aspects while refactoring tools typically restrict themselves to the static description as specified in the source-code.

Refactoring tools often rely on an abstract syntax tree representation of the source-code and assert pre- and postconditions before and after transforming the tree [19]. The formal model proposed in this paper extends the tree-based representation to a graph-based representation that allows one to make relations (such as method calls and variable accesses) between different program elements explicit as edges, and that provides a more localised naming scheme for classes, variables and methods. The graph representation is *lightweight* because it deliberately omits the details necessary for sophisticated data- and control flow analysis, since these are necessarily dependent on the programming language. Instead it focuses on the core concepts present in any class-based object-oriented language –namely classes, methods and variables– and allows us to verify whether the relationships between them are preserved. Moreover, it yields a transparent yet formal specification of the refactorings, as direct manipulations of the graph representation.

This paper presents a *feasibility study* to see whether graph rewriting and related techniques can be used to formalise refactorings as well as the properties to be preserved when performing a refactoring. Section 2 introduces the concept of refactoring 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 selected refactorings by graph transformations. In Section 4 we use this formalisation to guarantee the preservation of well-formedness and certain types of behaviour. Section 5 discusses tool support issues. Section 6 discusses some open problems. Finally, Section 7 concludes the paper with the lessons we learned from our feasibility study.

2. Motivating example

As a motivating example, this paper uses a simulation of a Local Area Network (LAN) [5]. 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. 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*

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

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.

```
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   }
```

This 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 *ExtractMethod* refactoring ([8], p110) and an *EncapsulateVariable* refactoring ([8], p206). Then, the *PrintServer* functionality is enhanced to distinguish between ASCII- and PostScript documents, which introduces complex conditionals and requires

an *ExtractClass* refactoring ([8], p149). The latter is actually a composite refactoring which creates a new intermediate superclass and then performs several *PullUpVariable* ([8], 320) and *PullUpMethod* ([8], p322) refactorings. Finally, a broadcast packet is added which again introduces complex conditionals, resolved by means of an *ExtractClass*, *ExtractMethod*, *MoveMethod* ([8], p142) and *InlineMethod* ([8], p117).

2.2. Selected refactorings

Fowler's catalogue [8] lists seventy-two refactorings and since then many others have been proposed. Since 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 then freely be combined into *composite refactorings*.

Below we summarise some frequently used primitive refactorings: *RenameVariable*, *RenameMethod*, *EncapsulateVariable*, *PullUpMethod*, *PushDownMethod*, *ExtractMethod* and *RemoveParameter*. The preconditions for these object-oriented refactorings are quite typical, hence they may serve as representatives for the complete set of primitive refactorings.

RenameVariable and **RenameMethod** are used to change the name of a variable or method. This typically is a global operation that affects the source code in many places, since it requires all references to the variable or method to be renamed as well.
Precondition. Before renaming the variable or method, the refactoring tool should ensure that the new name of the variable or method does not collide with existing names in the inheritance hierarchy in any way.

EncapsulateVariable 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 refactoring may accidentally override (or be overridden by) an existing method, and then it is possible that the behaviour is not preserved.

PullUpMethod is used to move similar 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 definition 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 defined in the subclass. Otherwise the pulled-up method would refer to an out-of-scope variable 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.

PushDownMethod is the opposite of *PullUpMethod*. It is used to move a method from a superclass to a selected number of its subclasses, either because it only makes sense there, or to provide a different implementation of this method in each subclass.
Precondition. (1) The method should not already be implemented in the subclasses to where it is pushed down to; (2) The method should not refer to attributes that are also defined in one or more of the target subclasses; (3) The method should not refer to private methods or variables of the containing superclass; (4) None of the direct subclasses may invoke the method via a super call; (5) The method should not refer to any variables also defined in the superclass; (6) The method must not be invoked in or through its defining class.

ExtractMethod is used to extract a fragment of a method implementation into a separate method. It makes overly long methods smaller, and increases the possibilities for reuse in later iterations. Therefore *ExtractMethod* is often the start of a sequence of other refactorings, such as *PullUpMethod*.
Precondition. (1) The method to be extracted should not already be defined in the inheritance hierarchy; (2) All variables accessed by the extracted method and which have local scope should be passed as a parameter; (3) When the last statement of the method definition is an assignment, the programmer may choose to define the method as a function. In that case, the function returns the right-hand of the last assignment and the assignment itself is omitted from the method definition.

RemoveParameter is used to remove method parameters that are no longer referenced inside the method definition.
Precondition. None of the implementations of this method in the class hierarchy should use this parameter. Otherwise, the method blocks will not correspond to the same method signatures anymore.

Because it is impossible to discuss in detail how all of the above refactorings can be formalised using graph transformations, we will restrict ourselves to only two of these refactorings in the remainder of the paper: *EncapsulateVariable* and *PullUpMethod*.

2.3. Behaviour preservation

Since we take a lightweight approach to source code refactoring, we only consider notions of behaviour preservation that can be detected statically and do not rely on sophisticated data- and control-flow analysis techniques. The general idea is that, for each considered refactoring, one may catalog the behaviour-related properties that need to be preserved. For the feasibility study of this paper, we concentrate on three types of behaviour that are important and non-trivial for the selected refactorings. Section 4 discusses to which extent the selected refactorings preserve these notions of behaviour:

- 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.

Obviously, other useful notions of behaviour preservation can be defined (e.g., type preservation), but their definition and treatment is outside the scope of this paper.

3. Formalising refactoring by graph transformation

3.1. Program graphs

Definition 3.1. (Graph)
Let L_v be a set of node labels and L_e a set of edge labels. A *(labeled) graph over* L_v *and* L_e is a 4-tuple $G = (V_G, E_G, nlab_G, elab_G)$, where $nlab_G : V_G \rightarrow L_v$ is the node labeling function and $elab_G : E_G \rightarrow L_e$ is the edge labeling function. An edge with label l from node v into node w will be denoted by $v \underset{l}{\rightarrow} w$.

Programs are represented by typed, labeled, directed graphs, 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 C (i.e., a C-node). The set $\Sigma = \{C, M, MD, V, VD, P, E\}$ of all possible node types is clarified in Table 1. Method definitions (MD-nodes) have been separated from their method signatures (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 (V-nodes) and their definition (VD-nodes). MD-nodes and 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 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 Table 1. For m-edges (membership), the type is often omitted in the figures.

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.[2] For example, Figure 1 only shows the graph representation of the static structure of the LAN simulation.

A method definition is represented by a parsetree-like structure consisting of of E-nodes (the expressions in the parse tree) connected by e-edges. Outgoing edges from E-nodes express information about method calls and variable references (accesses and updates). For example, Figure 2 represents the method definitions in class *Node*. The method definition of *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 `nextNode.accept(p)` is a subexpression composed of a variable access (represented by an

[2] In [26], an hierarchical graph approach is proposed to address the problem of visualising large graphs.

node type	**description**	**examples**
C	**C**lass	*Node*, *Workstation*, *PrintServer*, *Packet*
M	**M**ethod signature	*accept*, *send*, *print*
MD	**M**ethod **D**efinition	`System.out.println(p.contents)`
V	**V**ariable	*name*, *nextNode*, *contents*, *originator*
VD	**V**ariable **D**efinition	`public Node nextNode`
P	**P**arameter of a method definition	*p*
E	(sub)**E**xpression in method definition	`p.contents`
edge type	**description**	**examples**
$l: \quad M \rightarrow MD$	dynamic method **l**ookup	`accept(Packet p)` has 3 possible method definitions
$V \rightarrow VD$	variable **l**ookup	...
$i: \quad C \rightarrow C$	**i**nheritance	`class PrintServer` **extends** `Node`
$m: \quad VD \rightarrow C$	variable **m**embership	variable *name* is defined in *Node*
$MD \rightarrow C$	method **m**embership	method *send* is defined in *Node*
$t: \quad V \rightarrow C$	variable **t**ype	**String** `name`
$M \rightarrow C$	method return **t**ype	**String** `getName()`
$p: \quad MD \rightarrow P$	**p**arameter definition	`send(Packet` **p**`)`
$M \rightarrow C$	**p**arameter type	`send(`**Packet** `p)`
$e: \quad MD \rightarrow E$	**e**xpression in method definition	`System.out.println(p.contents)`
$E \rightarrow E$	sub**e**xpression in method definition	`p.contents`
$c: \quad E \rightarrow S$	(dynamic) method **c**all	`this.send(p)`
$a: \quad E \rightarrow \{V \mid P\}$	variable or parameter **a**ccess	`p.contents`
$u: \quad E \rightarrow \{V \mid P\}$	variable or parameter **u**pdate	`p.`**originator** = `this`

Table 1. Node type set $\Sigma = \{C, M, MD, V, VD, P, E\}$ and edge type set $\Delta = \{l, i, m, t, p, e, c, a, u\}$.

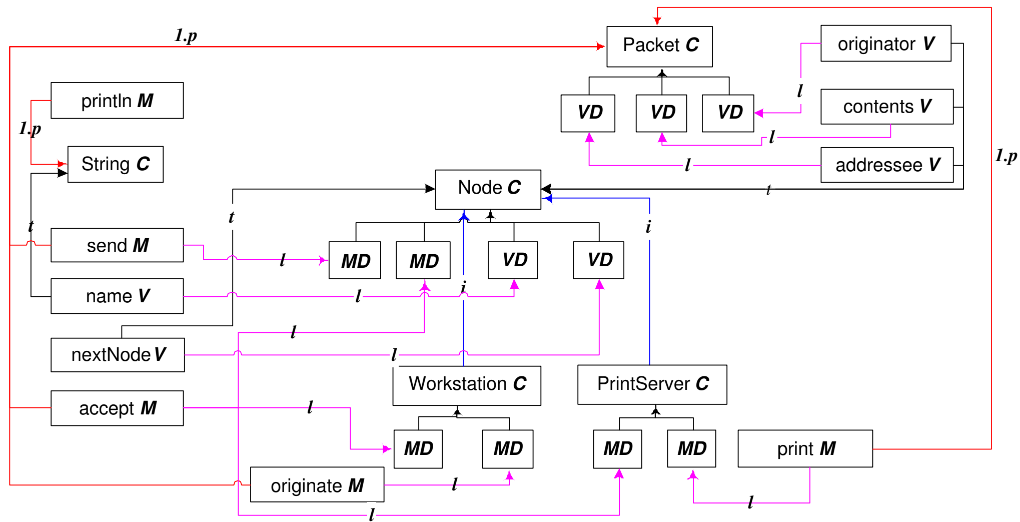


Figure 1. Static structure of LAN simulation

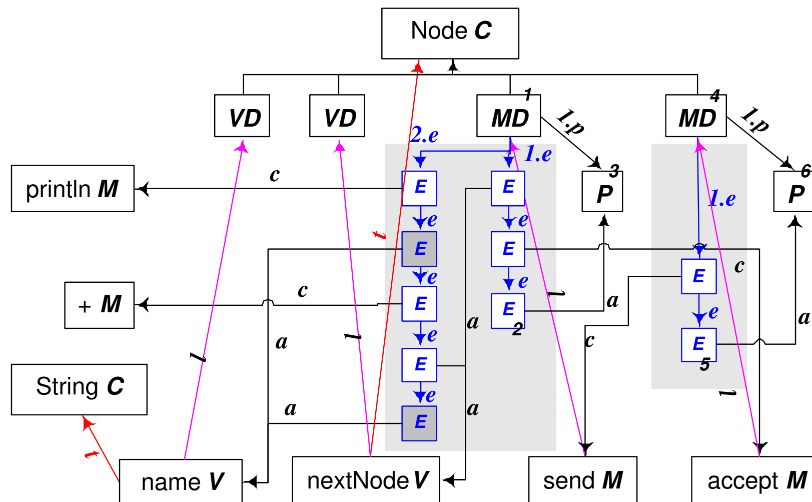


Figure 2. Method definitions in class *Node*

E-node with outgoing a-edge to the V-node with label *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 `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 C, MD and VD to deal with Java modifiers (such as `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 state-

ments, 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.

3.2. 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. These *well-formedness constraints* are essential to fine-tune our graph notation to a particular programming language (in this case Java). We use two mechanisms to express these constraints: a *type graph* and *graph expressions*.

Definition 3.2. (Type Graph)
The notion of a *type graph* (or graph schema) is formally defined in [4, 7]. Informally, a type graph is a labelled graph over node type set Σ and edge type set Δ (see Table 1). This type graph expresses restrictions on the program graphs that are allowed. A program graph is well-formed only if there exists a graph morphism into the type graph: a node mapping and edge mapping that preserves sources, targets and labels. For the labels, only the type component is taken into account. Figure 3 displays the type graph needed for our particular program graph representation.

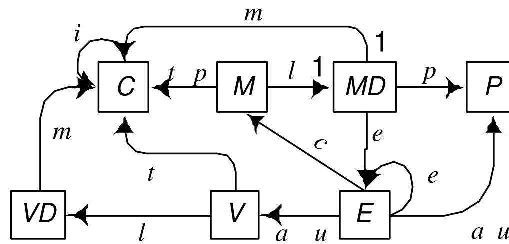


Figure 3. Type Graph

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.

In addition to the type graph, we need a mechanism to express constraints that exclude illegal configurations in a graph. Some typical examples of such well-formedness constraints are given below:

WF-1 No two variables with the same signature can be defined 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.

These well-formedness constraints can be expressed by *graph expressions*. Formally, graph expressions and their occurrences are defined as follows:

Definition 3.3. (Graph expression)

1. A *graph expression* GE is a graph (V_{GE}, E_{GE}) over Σ and all regular expressions over Δ.
2. Let G be a program graph. An *occurrence* of GE in G is a mapping $oc : V_{GE} \to V_G$ such that

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

An example of such a graph expression GE is shown in Figure 4. $1 \underset{e^*}{\longrightarrow} 2$ is one of its three edges. Its edge label e^* is a regular expression over edge type set Δ. There are two occurrences, oc_1 and oc_2, of this graph expression GE in the graph G of Figure 2. These occurrences are given by the following mappings from V_{GE} to V_G:

v	$oc_1(v)$	v	$oc_2(v)$
1	1	1	4
2	2	2	5
3	3	3	6

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

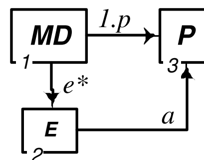


Figure 4. An example graph expression

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 5. For each of these constraints, one only has to exclude *injective* occurrences. Note that this use of graph expressions can be viewed as a shorthand for the mechanism of *forbidden subgraphs*: one may associate with a graph expression the set of all graphs in which that graph expression does not occur, and then consider those graphs as forbidden subgraphs.

The graph expression for **WF-3** contains edges that represent regular expressions over Δ. Edge $1 \xrightarrow[e*]{} 2$ denotes a parsetree traversal to an arbitrary subexpression in the method parsetree, $2 \xrightarrow[a|u]{} 3$ denotes an access or update to a variable from within that subexpression, and $3 \xrightarrow[lmi^+]{} 4$ denotes that the variable is defined in a subclass of the given class.

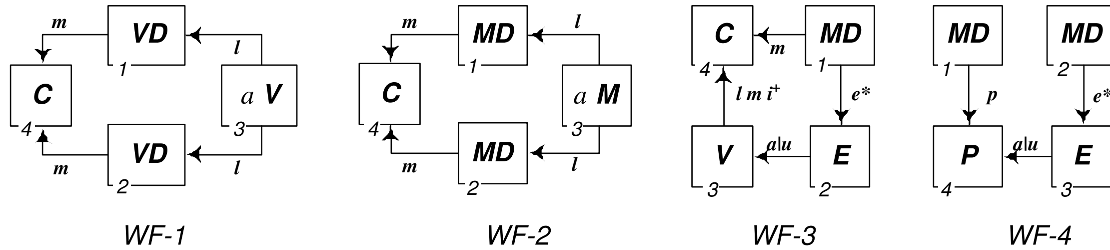


Figure 5. Well-formedness constraints expressed as graph expressions representing forbidden subgraphs

3.3. Expressing refactorings by graph productions

Programs are represented by program graphs, and thus refactorings are transformations of program graphs. In the theory of graph rewriting, such transformations result from the application of predefined rules, called *graph productions*. 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.

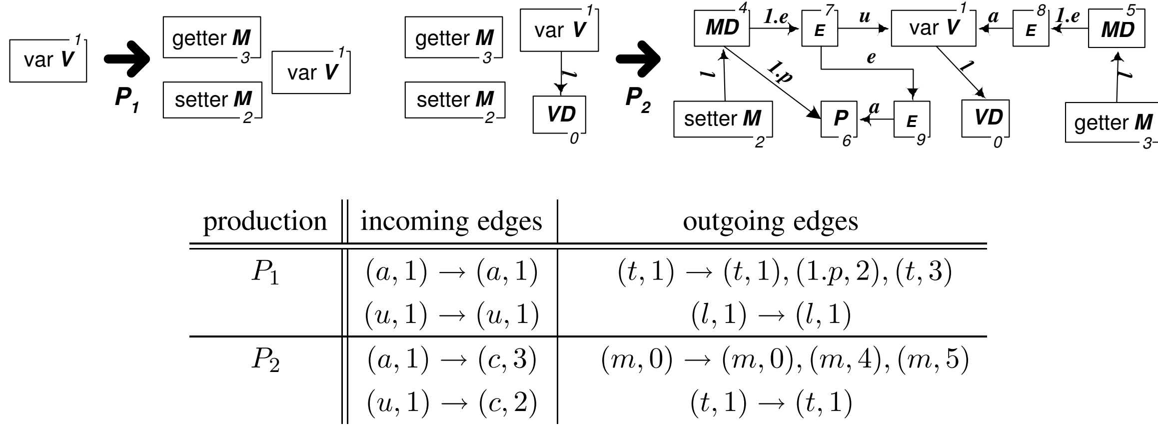
Often, a graph production can be applied to different parts of a graph, leading to different occurrences (or matches) of the graph production's LHS. In this paper, we use *parameterised graph productions* that contain variables for labels. Each parameterised production may be viewed as a specification of an infinite set of productions in the algebraic approach to graph rewriting [6, 14]. In order to take into account the context in which a production is applied, the production is equipped with an *embedding mechanism* similar to the one of [11]. This embedding mechanism specifies how incoming and outgoing edges are redirected. [3] A concrete instance of this production can be obtained by filling in the variables of the parameterised graph production with concrete values and extending the LHS and RHS of the production with a concrete context.

For most refactorings, it does not suffice to apply a single graph production. Instead, these refactorings need to be expressed 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, 13, 22].

EncapsulateVariable

Figure 6 shows the parameterised productions needed to express the refactoring *EncapsulateVariable(var,getter,setter)*. Here *var, getter, setter* are formal parameters, to be replaced

[3] A similar but more visual mechanism is available in the graph rewriting tools PROGRES [23] and Fujaba [15].



production	incoming edges	outgoing edges
P_1	$(a,1) \rightarrow (a,1)$	$(t,1) \rightarrow (t,1),(1.p,2),(t,3)$
	$(u,1) \rightarrow (u,1)$	$(l,1) \rightarrow (l,1)$
P_2	$(a,1) \rightarrow (c,3)$	$(m,0) \rightarrow (m,0),(m,4),(m,5)$
	$(u,1) \rightarrow (c,2)$	$(t,1) \rightarrow (t,1)$

Figure 6. Graph productions with embedding table for refactoring *EncapsulateVariable(var,getter,setter)*

by concrete names in order to obtain concrete productions. Using the mechanism of controlled graph rewriting, the *EncapsulateVariable* refactoring is expressed by the graph productions P_1, P_2 of Figure 6, where the application of those productions is controlled by the state-transition diagram of Figure 7: first P_1 must be applied, and then P_2 must be applied repeatedly for each of the possible occurrences of its LHS.

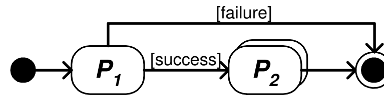


Figure 7. Specifying the order of productions P_1 and P_2.

In Figure 6, 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 6 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 8 shows the concrete production instance *EncapsulateVariable(name,getName,setName)* that may be applied to the graph of Figure 2, in the context of the LAN example. When applying the production, the two gray E-nodes in Figure 8 match the gray E-nodes of Figure 2. 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.

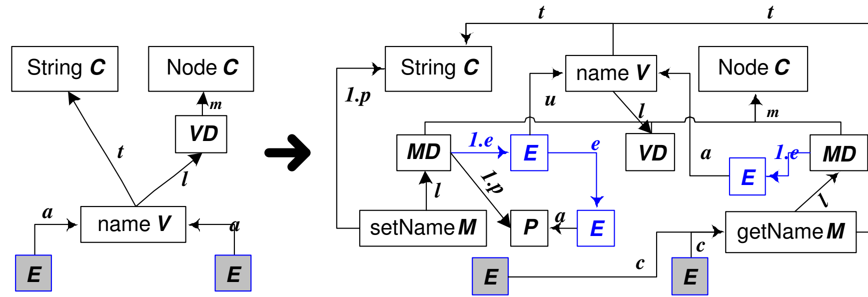


Figure 8. Graph production instance *EncapsulateVariable(name,getName,setName)* obtained from the parameterised production of Figure 6

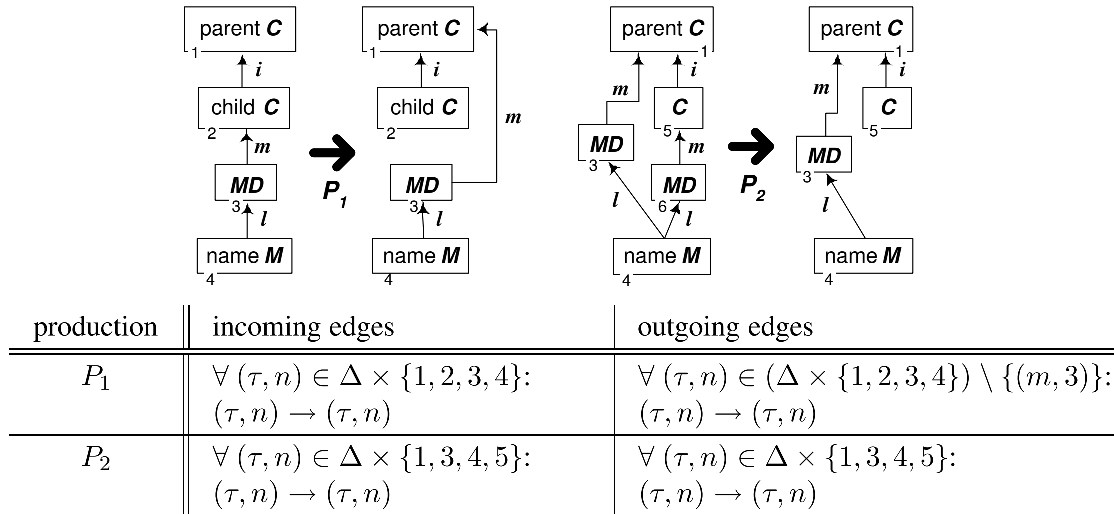
PullUpMethod

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 the definitions of the method $name$ in all other children of $parent$. 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 9). P_1 moves the definition of 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 definition of method $name$ from another subclass of $parent$, and has to be applied for each possible occurrence of its LHS. Therefore, the application order of P_1 and P_2 can also be specified by the state-transition diagram in Figure 7.

3.4. 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**)

All 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 [9, 10]. It has



production	incoming edges	outgoing edges
P_1	$\forall\,(\tau, n) \in \Delta \times \{1,2,3,4\}$: $(\tau, n) \rightarrow (\tau, n)$	$\forall\,(\tau, n) \in (\Delta \times \{1,2,3,4\}) \setminus \{(m,3)\}$: $(\tau, n) \rightarrow (\tau, n)$
P_2	$\forall\,(\tau, n) \in \Delta \times \{1,3,4,5\}$: $(\tau, n) \rightarrow (\tau, n)$	$\forall\,(\tau, n) \in \Delta \times \{1,3,4,5\}$: $(\tau, n) \rightarrow (\tau, n)$

Figure 9. Graph productions P_1 and P_2 with embedding table for refactoring *PullUpMethod(parent,child,name)*. In P_2, a method definition (node 6) is removed, which implies that its contained parse tree can be garbage collected.

also been shown in [10] that postconditions can be transformed into equivalent preconditions for a graph production.

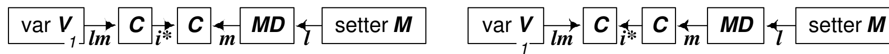


Figure 10. Negative preconditions for production P_1 of *EncapsulateVariable* refactoring

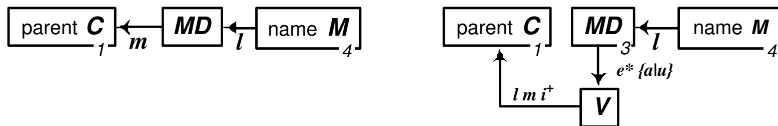


Figure 11. Negative preconditions for production P_1 of *PullUpMethod* refactoring

PullUpMethod

Figure 11 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*.

To prove that *PullUpMethod* preserves well-formedness constraint **WF-2** it suffices to show that an application of P_1 or P_2 to a graph containing no occurrences of **WF-2** cannot result in a graph in which

WF-2 does occur. Assume, to the contrary, that a new occurrence of **WF-2** is created in a step using P_1. Clearly one has to consider only occurrences of **WF-2** that overlap with the RHS of the production. Taking into account the node labels, and the fact that each MD has exactly one incoming l-edge and only one outgoing 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 **WF-2** was present already before he rewriting, contradicting the assumption. The last case, where nodes 1, 3 and 4 would belong to an occurrence of **WF-2**, is excluded by the first negative precondition for P_1 (left part of Figure 11), which states that the method to be pulled up should not yet exist in the parent class. To see that no new occurrences of **WF-2** can be created by applying production P_2, it suffices to see that the effect of P_2 is simply the removal of a node together with all its incident edges.

Well-formedness constraints **WF-1**, **WF-3** and **WF-4** can be proven in a similar way for *PullUp-Method*. Intuitively, well-formedness constraint **WF-1** is preserved since it does not introduce or redirect any variables or variable definitions. Constraint **WF-3** is preserved thanks to the precondition on the right of Figure 11, and constraint **WF-4** is preserved because the refactoring does not introduce or change any method definition.

EncapsulateVariable

Figure 10 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.

Without giving a formal proof, well-formedness constraint **WF-1** is preserved since *Encapsulate-Variable* does not introduce or move any variables or variable definitions. Constraint **WF-2** is preserved thanks to the preconditions of Figure 10, 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 *Encapsulate-Variable* 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.

4. Preservation of behaviour

In this section we combine the formalisation of refactorings of Subsection 3.3 with a technique to express the fact that certain occurrences of graph expressions are preserved. In particular, we consider certain types of behaviour preservation that can be detected statically.

4.1. 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 3.2: 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 $MD \underset{?*a}{\longrightarrow} V \underset{l}{\longrightarrow} VD$ 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 $MD \underset{?*a}{\longrightarrow} V \underset{l}{\longrightarrow} VD$ 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 $MD \underset{?*u}{\longrightarrow} V \underset{l}{\longrightarrow} VD$.

Graph expression $MD \underset{e*c}{\longrightarrow} M \underset{l}{\longrightarrow} MD$ 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.

4.2. Preserving occurrences of a graph expression

In order to relate occurrences before and after the rewriting, one needs to introduce a *tracking function* tr. 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 4.1. (Tracking function)
Let GE be a graph expression, let G and H be program graphs and let $tr : V_G \to V_H$ 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. In a rewriting $G \to H$, using production p, the tracking function $tr : V_G \to V_H$ is defined by

$$tr(v) = \begin{cases} v, \text{ if } v \text{ is a node of the part of } V_G \text{ that is not rewritten.} \\ tr_p(v), \text{ if } v \text{ belongs to the part of } V_G \text{ that is rewritten.} \end{cases}$$

For refactoring *PullUpMethod* of Figure 9, 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 12. $tr_{P_2}(1) = 1$, $tr_{P_2}(3) = tr_{P_2}(6) = 3$, $tr_{P_2}(4) = 4$, and $tr_{P_2}(5) = 5$.

4.3. Behaviour preserving refactorings

PullUpMethod

To show *call preservation* for *PullUpMethod*, one has to prove that the productions P_1, P_2 of Figure 9 preserve the graph expression $GE = MD \underset{e*c}{\longrightarrow} M \underset{l}{\longrightarrow} MD$. It is clearly sufficient to prove the preservation of the subexpressions $GE_1 = MD \underset{e*c}{\longrightarrow} M$ and $GE_2 = M \underset{l}{\longrightarrow} MD$. Moreover, the construction of the tracking function implies that one needs to consider only occurrences of GE_1 and GE_2 separately. Also, it implies that one must only consider those occurrences GE_1 and GE_2 that overlap with the LHS of the productions, because only for the nodes in that overlap the tracking function may differ from the identity.

First consider GE_1. For both P_1 and P_2, one has to consider a number of cases for the possible overlap of an occurrence of GE_1 and the LHS. In each of them, a simple inspection of the embedding relation

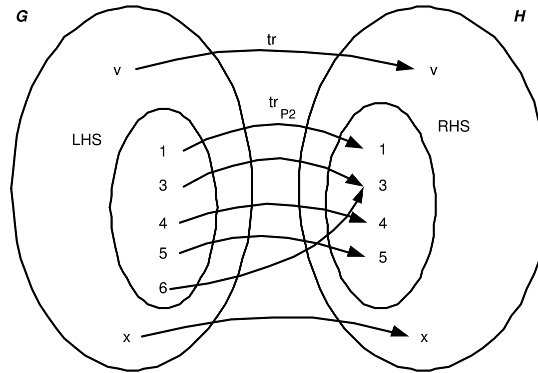


Figure 12. Construction of the tracking function for P_2 of *PullUpMethod*

suffices to see that occurrences of GE_1 are preserved. Note that all nodes on the path corresponding to the edge of P_1 have label E, and hence do not overlap with the LHS of P_1 or P_2.

P_1-a Node 1 of GE_1 corresponds to node 3 of P_1, and node 2 of GE_1 does not correspond to node 4 of P_1. Since the embedding of P_1 simply preserves outgoing e-edges of node 3, it follows that the occurrence is preserved.

P_1-b Nodes 1 and 2 of GE_1 correspond to nodes 3 and 4 of P_1. The embedding of P_1 preserves outgoing e-edges of node 3 and incoming c-edges of node 4. It follows that the occurrence is preserved.

P_1-c Node 1 of GE_1 does not correspond to a node of P_1, but node 2 of GE_1 corresponds to node 4 of P_1. This time it is sufficient to see that the embedding preserves incoming c-edges of node 4.

P_2-a Node 1 of GE_1 corresponds to node 3 of P_2, and node 2 of GE_1 does not correspond to node 4 of P_2. This case is analogous to P_1-a.

P_2-b Nodes 1 and 2 of GE_1 correspond to nodes 3 and 4 of P_2. This case is analogous to P_1-b.

P_2-c Node 1 of GE_1 corresponds to node 6 of P_2, and node 2 GE_1 does not correspond to node 4 of P_2. Now one needs the fact that a correct application of *PullUpMethod* can take place only if the syntax trees under nodes 3 and 6 of P_2 are isomorphic: if there exists an occurrence of GE_1 where node 1 corresponds to node 6 of P_2, then there exists also an occurrence of GE_1 where node 1 corresponds to node 3 of P_2. Thus, since the tracking function maps node 6 to node 3, and since the embedding mechanism preserves outgoing nodes of node 3, the occurrence is preserved. (Note, that one might be less restrictive for *PullUpMethod* by allowing a weaker form of equivalence for the syntaxtrees under node 3 and 6. However, this would not change the reasoning.)

P_2-d Nodes 1 and 2 of GE_1 correspond to nodes 6 and 4 of P_2. One may use the same reasoning as in P_2-c, and the fact that the embedding preserves incoming c-edges of node 4.

P_2-e Node 1 of GE_1 does not correspond to a node of P_2, but node 2 of GE_1 corresponds to node 4 of P_2. This case is analogous to P_1-c.

As a second step of our proof, we have to consider all possible cases for GE_2.

P$_1$-a' If node 1 of GE_2 corresponds to node 4 of P_1 and node 2 of GE_2 does not correspond to a node of P_1, then it suffices to see that the embedding of P_1 preserves outgoing l-edges of node 4.

P$_1$-b' If node 2 of GE_2 corresponds to node 3 of P_1, then it follows from the the fact that each MD-node has at most one incoming l-edge that node 1 of GE_2 must correspond to node 4 of P_1. It follows from the form of the RHS of P_1 that the occurrence is preserved.

P$_2$ For the overlap of an occurrence of GE_2 and the LHS of P_2, there are four cases to consider, all of which are analogous to P$_1$-a' or P$_1$-b'.

Since both P_1 and P_2 preserve GE, one may conclude that *PullUpMethod* is call preserving. In a similar way, we can show access preserving and update preserving. Note that, in case P$_2$-c of the above proof, we explicitly relied on the fact that all method definitions that are affected in the subclasses must be isomorphic. If this is not the case, the refactoring will not be behaviour preserving.

Figure 13. Update preservation for *EncapsulateVariable(var,getter,setter)*

EncapsulateVariable

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 $MD \xrightarrow[?^*u]{} V \xrightarrow[l]{} VD$ 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 6 that this is the case. This is illustrated in Figure 13, that shows how a direct update of *var* is replaced by a slightly longer path that still preserves the graph expression $MD \xrightarrow[?^*u]{} V$. The graph productions do not change anything to occurrences of graph expression $V \xrightarrow[l]{} VD$. *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.)

5. Tool support

In order to validate our results in practice, we need to implement at least three aspects: converting Java code into a graph, applying refactoring transformations to this graph, and verifying preconditions and invariants in the graph representation.

A converter from Java source code into the graph representation of Section 3.1 has been implemented in Java by Jessie Dedecker during a programming project.

To specify and execute refactorings as graph productions we identified two graph transformation tools that seemed to satisfy most of our needs: *PROGRES* [21, 23] and *Fujaba* [15, 12]. Because the latter

tool is tightly integrated with Java, and its user interface is easier to learn than the one of PROGRES, we used Fujaba to implement our initial ideas.

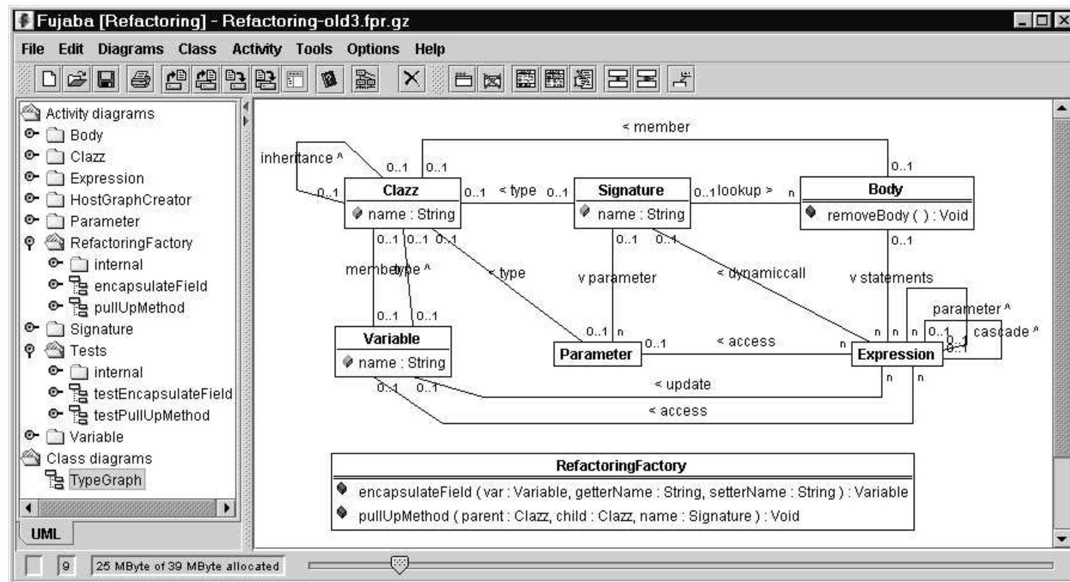


Figure 14. Screenshot of the Fujaba tool

A screenshot of the Fujaba tool is given in Figure 14. It shows how to express the type graph as a UML class diagram. By expressing type edges as UML associations, we can easily express cardinality constraints on them. By expressing type nodes as UML classes, we can exploit the subclassing mechanism to avoid redundancy. The two graph transformations *EncapsulateField* and *PullUpMethod* were implemented in a way that is very similar to the specification in Section 3.3. Only the embedding mechanism used in this paper was specified in a more graphical way, by using optional nodes.

The well-formedness constraints and refactoring preconditions of this paper were more difficult (though not impossible) to express in Fujaba. An alternative would be to express and validate these constraints with OCL, the standard object constraint language that comes with UML [16]. OCL constraints can be used to express invariants, pre- and postconditions. However, due to the limitations of the UML metamodel, not all constraints that we need can be expressed in this way, especially if we want to deal with constraints that require information that is only visible in the method parse trees.

6. Open Problems

When we tried to specify the refactorings *ExtractMethod* and *PushDownMethod*, we encountered a number of problems related to the parse trees contained in a method definition. To guarantee behaviour preservation of *PullUpMethod*, we needed to assume a notion of isomorphism between the parse trees of the method definitions in the subclasses. Also in *PullUpMethod*, we saw the need to remove method

definitions together with its entire containing parse tree. For *PushDownMethod* (not expressed in this paper), we need to make several copies of a method definition, including the entire method parse tree. Thus, there is a need for more sophisticated techniques (such as hierarchical graphs [7] and automatic garbage collection) that tackle the inevitable complexity of large graphs.

An open question concerns the expressiveness of our proposed formalism. The refactorings as well as the types of behaviour preservation we studied, are realistic and well documented. It remains to be seen whether the notion of type graphs, graph expressions, embedding mechanisms, and controlled parameterised graph transformations are sufficient to specify all possible refactorings as well as more sophisticated notions of behaviour preservation.

A central topic in future work will be the implementation of tools that automate a variety of tasks related to refactoring:

- Detect, for a given (behaviour preservation) property and graph transformation, whether or not the property is preserved by the transformation. If not, provide additional feedback on why this is the case and how this may be resolved.

- Check, for a given graph transformation, whether its preconditions "subsume" the well-formedness constraints. If this is the case, the well-formedness constraints don't have to be checked again after applying the transformation. A further step would involve an automatic translation of the well-formedness constraints into specific preconditions for each refactoring.

- Compare the preconditions of refactorings to determine whether they are parallel independent [10]. If they are, they can be serialised in any order. This is a useful property for the composition of primitive refactorings into a sequence [19], because it allows us to change the order in the sequence without affecting the overall end result. As such, the transformation sequence may be optimized, in much the same way as database tools perform query optimisations. This is crucial for the performance and usability of next generation refactoring tools.

Similar to what has been described in [25], we will also study the impact of language specific features on the proposed approach. The current type graph is sufficiently language independent to deal with refactorings of simple Java and Smalltalk programs. Because Smalltalk is dynamically typed, t-edges are not needed there. On the other hand, the type graph is too simple to express some language-specific constructs and their associated refactorings. For example, Java interfaces require the introduction of a new node type in the type graph, and Java exception handling requires the introduction of a new edge type in the type graph. Additionally, new well-formedness constraints and refactoring transformations may be needed to cope with these language-specific constructs.

7. Conclusion

This paper investigated the feasibility of using graph transformations 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 language-independent representation of the source code; (ii) graph transformation rules are a concise and precise way to specify the source-code transformations implied by a refactoring; (iii) the formalism allows us to

prove that refactorings preserve certain kinds of behaviour that can be inferred statically from the source code.

In order to achieve our goal, we had to combine a number of existing graph rewriting mechanisms and techniques. Type graphs and graph expressions 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 are provided by state-of-the-art graph rewriting tools such as PROGRES [23] and Fujaba [15].

References

[1] Boger, M., Sturm, T., Fragemann, P.: Refactoring Browser for UML, *Proc. 3rd Int'l Conf. on eXtreme Programming and Flexible Processes in Software Engineering*, 2002, Alghero, Sardinia, Italy.

[2] Bunke, H.: Programmed graph grammars, in: *Graph Grammars and Their Application to Computer Science and Biology* (V. Claus, H. Ehrig, G. Rozenberg, Eds.), vol. 73 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979, 155–166.

[3] Casais, E.: Automatic reorganization of object-oriented hierarchies: a case study, *Object Oriented Systems*, **1**, 1994, 95–115.

[4] Corradini, A., Ehrig, H., Löwe, M., Montanari, U., Padberg, J.: 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*, vol. 1073 of *Lecture Notes in Computer Science*, Springer-Verlag, 1996, 56–74.

[5] Demeyer, S., Janssens, D., Mens, T.: Simulation of a LAN, *Electronic Notes in Theoretical Computer Science*, **72**(4), 2002.

[6] Ehrig, H.: Introduction to the algebraic theory of graph grammars, in: *Graph Grammars and Their Application to Computer Science and Biology* (V. Claus, H. Ehrig, G. Rozenberg, Eds.), vol. 73 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979, 1–69.

[7] Engels, G., Schürr, A.: Encapsulated Hierarchical Graphs, Graph Types and Meta Types, *Electronic Notes in Theoretical Computer Science*, **2**, 1995.

[8] Fowler, M.: *Refactoring: Improving the Design of Existing Programs*, Addison-Wesley, 1999.

[9] Habel, A., Heckel, R., Taentzer, G.: Graph Grammars with Negative Application Conditions, *Fundamenta Informaticae*, **26**(3,4), June 1996, 287–313.

[10] Heckel, R.: *Algebraic Graph Transformations with Application Conditions*, Master Thesis, TU Berlin, 1995.

[11] Janssens, D., Mens, T.: Abstract semantics for ESM systems, *Fundamenta Informaticae*, **26**(3 and 4), June 1996, 315–339.

[12] Köhler, H., Nickel, U., Niere, J., Zündorf, A.: Integrating UML Diagrams for Production Control Systems, in: *Proc. Int. Conf. Software Engineering*, ACM Press, 2000, 241–251.

[13] Kreowski, H.-J., Kuske, S.: Graph Transformation Units and Modules, in: *Handbook of Graph Grammars and Computing by Graph Transformation*, vol. 2, World Scientific, 1999, 607–638.

[14] Löwe, M.: Algebraic approach to single-pushout graph transformation, *Theoretical Computer Science*, **109**, 1993, 181–224.

[15] Niere, J., Zündorf, A.: Using Fujaba for the development of production control systems, in: *Proc. Int. Workshop Agtive 99* (M. Nagl, A. Schürr, M. Münch, Eds.), vol. 1779 of *Lecture Notes in Computer Science*, Springer-Verlag, 2000, 181–191.

[16] Object Management Group: Unified Modeling Language Specification version 1.4, formal/2001-09-67, September 2001.

[17] Opdyke, W.: *Refactoring Object-Oriented Frameworks*, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1992.

[18] Opdyke, W., Johnson, R.: Creating abstract superclasses by refactoring, in: *Proc. ACM Computer Science Conference*, ACM Press, 1993, 66–73.

[19] Roberts, D.: *Practical Analysis for Refactoring*, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1999.

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

[21] Schürr, A.: PROGRES - A VHL-language Based on Graph Grammars, *Proc. Int. Workshop on Graph Grammars and their Application to Computer Science* (H. Ehrig, H.-J. Kreowski, G. Rozenberg, Eds.), 532, Springer-Verlag, 1991.

[22] Schürr, A.: Logic based programmed structure rewriting systems, *Fundamenta Informaticae*, **26**(3 and 4), June 1996, 363–385.

[23] Schürr, A., Winter, A. J., Zündorf, A.: Graph grammar engineering with PROGRES, in: *Proc. European Conf. Software Engineering* (W. Schäfer, P. Botella, Eds.), vol. 989 of *Lecture Notes in Computer Science*, Springer-Verlag, 1995, 219–234.

[24] Sunyé, G., Pollet, D., LeTraon, Y., Jézéquel, J.-M.: Refactoring UML models, in: *Proc. UML 2001*, vol. 2185 of *Lecture Notes in Computer Science*, Springer-Verlag, 2001, 134–138.

[25] Tichelaar, S.: *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*, Ph.D. Thesis, University of Bern, 2001.

[26] Van Eetvelde, N., Janssens, D.: A Hierarchical Program Representation for Refactoring, *Proc. of UniGra'03 Workshop*, 2003.