Object-Oriented Refactoring Using Graph Rewriting

Technical Report vub-prog-tr-02-01

Last modification on 2001-12-19

Tom Mens*	Serge Demeyer	Dirk Janssens
tom.mens@vub.ac.be	serge.demeyer@ua.ac.be	dirk.janssens@ua.ac.be
Vrije Universiteit Brussel	Universiteit Antwerpen	Universiteit Antwerpen

***Abstract**. This paper advocates need for a formal foundation for refactoring object-oriented software. More specifically, it investigates the potential of graph rewriting as candidate formalism. Using a small Java application as a case study, it shows how a range of different notions of behaviour preservation can be expressed in a formal, yet intuitive, way. A number of other open research topics that can benefit from a graph rewriting formalism are discussed as well.*

Keywords. *object-oriented software evolution, behaviour-preserving refactoring, graph rewriting, program transformation*

1. Introduction

Refactorings are software transformations that restructure a software system while preserving its behaviour. For object-oriented software, an important part of the evolution is accomplished by means of refactorings [8] [22] [23]. 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 [25] while others demonstrated the feasibility of fully automated tools [3] [21]. Research has also been conducted into making refactoring tools less dependent on the implementation language being used [31] and several researchers are investigating refactoring at a higher level, e.g., in the context of UML [30].

While the research on refactoring published so far has mainly concentrated on demonstrating the feasibility of the technology, there are still many important questions that remain unresolved.

- *Which program properties should be preserved by refactorings?* Refactoring implies that "behaviour" is preserved, but a precise definition of behaviour is rarely included. Even if it is, there is no commonly accepted definition of behaviour. It is for instance accepted that refactoring will change the execution time of certain operations, although this is an essential aspect of real-time behaviour. Hence, at the moment it is unclear which aspects of behaviour can be safely ignored under which circumstances.
- *What is the complexity of a refactoring?* An analysis of the complexity of refactorings is necessary in order to determine whether it is feasible to apply a given refactoring. We make a distinction between (a) the complexity of determining whether a certain refactoring is applicable; (b) the complexity of applying the refactoring itself. E.g., some refactorings will only make localised changes, while other transformations are more global, and hence require more work or have a higher change impact.
- *How do refactorings affect quality factors*? Some refactoring increase the complexity of the class structure, yet proponents claim that this extra complexity improves understandability. A classification of the refactorings in terms of the quality factors they aim to improve is therefore essential.
- *How can refactorings be composed and decomposed*? Current refactoring approaches combine primitive refactorings into more complex refactorings in an ad-hoc manner. Vice versa, given a program A and a refactored version B, it is possible to extract a sequence of refactorings that transforms A into B?
- *How do refactorings interact*? It has been observed that the parallel application of refactorings may cause potential consistency problems. Such observations must be modelled precisely because with large-scale

* Tom Mens is a Postdoctoral Fellow of the Fund for Scientific Research – Flanders (Belgium)

systems different programmers will refactor parts of the system independently, and then the refactoring tools must be able to determine the amount of overall coordination necessary.

These and many other questions can only be addressed adequately if we resort to a *formal model* of refactoring. This model should be sufficiently transparent to serve as a conceptual tool guiding the refactoring process. Moreover, the model should provide a framework for the development of software tools supporting refactoring.

One of the basic decisions to make when developing a formal model for refactoring is the representation of the object-oriented programs to be refactored. We propose a graph-based representation for this purpose. The use of such representations has a long history in the research about OO systems, and more recently class diagrams, use-case diagrams and sequence diagrams have become widely accepted parts of the UML. The fact that a graph-based representation is based on sets of anonymous vertices, rather than on names, allows one to avoid redundancies and to obtain an intuitive notion of locality. For example, *renaming* can have a global impact in the source code, but can be defined as a local operation when a graph-based representation is used. A concrete example of this will be illustrated later in the paper.

Given that a program is represented by a graph, a refactoring can be seen as a graph rewriting step that transforms that graph. This paper uses the formalism of *conditional* graph rewriting [5] [11] [12], where each graph rewriting can be defined in terms of application preconditions and postconditions. This gives rise to a direct link to existing work on refactoring. For example, [26] used pre- and postconditions to express refactoring transformations. Using this formalism, checking the applicability of a given refactoring simply boils down to pattern matching, by checking whether all preconditions required by the refactoring are satisfied in the graph.

This paper motivates why graph rewriting is a useful formalism for expressing object-oriented refactoring, and mainly focuses on the first question above: *Which program properties should be preserved by refactorings?* We provide an answer to this question by proposing an underlying graph representation for Java programs, by defining refactoring transformations as graph rewritings, and by defining various kinds of behaviour preservation as graph invariants that are preserved by these graph rewritings. This gives us a natural and intuitive means for expressing and dealing with many different notions of behaviour preservation.

2. Running example: a Local Area Network

As a running example, this paper uses the Java implementation of a Local Area Network simulation (LAN). The example has been used successfully by the Programming Technology Lab of the Vrije Universiteit Brussel and the Software Composition Group of the University of Berne to illustrate and teach good object-oriented design to students. The example is sufficiently simple for illustrative purposes, yet covers most of the interesting constructs of the object-oriented programming paradigm.

The LAN simulation starts with a simplistic model that gets refactored as requirements are added. In its most basic version, of which the UML class diagram is given in Figure 1, 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, and that they can *send* or *accept* a *Packet* object. Such 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 *accept*s the *Packet*, or until it returns to its *originator* workstation (indicating that the *Packet* cannot be delivered).

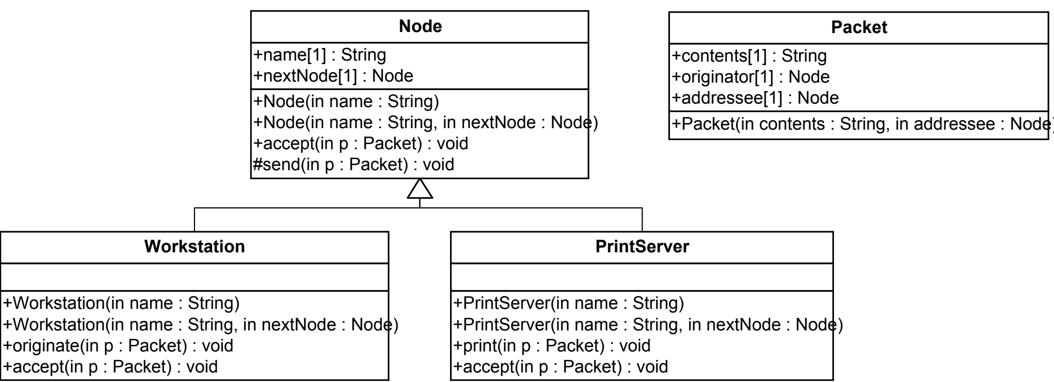


Figure 1: Class diagram of the initial LAN design

The Java code corresponding to this class diagram is presented below. Due to space considerations, constructor methods are omitted from this paper.

```java
public class Node {
  public String name;
  public Node nextNode;
  public void accept(Packet p) {
    this.send(p); }
  protected void send(Packet p) {
    System.out.println(
      this.name + nextNode.name);
    this.nextNode.accept(p); }
  }
```

```java
public class Packet {
  public String contents;
  public Node originator;
  public Node addressee;
  }
```

```java
public class PrintServer extends Node {
  public void print(Packet p) {
    System.out.println(p.contents); }
  public void accept(Packet p) {
    if(p.addressee == this) this.print(p);
    else super.accept(p); }
  }
```

```java
public class Workstation extends Node {
  public void originate(Packet p) {
    p.originator = this;
    this.send(p); }
  public void accept(Packet p) {
    if(p.originator == this)
      System.err.println("no destination");
    else super.accept(p); }
  }
```

Figure 2: Java code of the initial LAN simulation

This LAN simulation can be extended gradually to incorporate features that are more complex. During this extension, refactorings are often used to restructure the framework. Section 4 will present a number of these refactorings. But first we will show how the running example can be expressed using a graph representation in section 3.

3. The graph rewriting formalism

3.1 Graph notation

There is a more or less direct mapping from the Java code in Figure 2 to its graph representation. Classes, attributes, methods and method parameters are mapped to nodes of the corresponding type and with the corresponding name. For example, the class *Packet* is represented by a node with type ***C*** and name Packet. Relationships between software entities (such as containment, inheritance, typing, method lookup, attribute accesses and method calls) are represented by edges with the corresponding type between the corresponding nodes. For example, the inheritance relationship between the classes *Workstation* and *Node* is represented by an edge with type ***I*** between the nodes with type ***C*** and name Workstation and Node. Even the implementation of method bodies is represented by means of edges and nodes (see later).

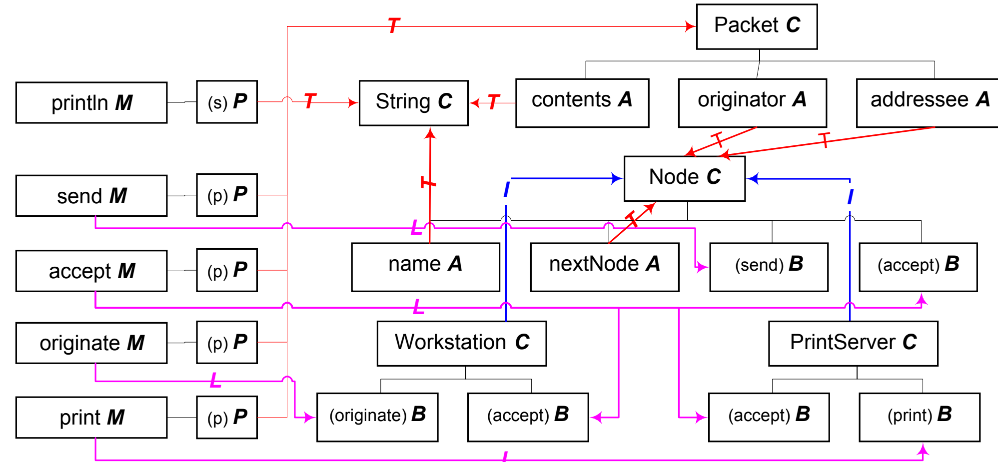


Figure 3: Graph representation of the static structure of the LAN simulation

In this way, the LAN example can be represented by means of a single typed graph. However, because the graph representation of the entire LAN example is large, throughout this paper we will always display those parts of the graph that are relevant for the discussion only. For example, Figure 3 only shows the graph representation of the

static structure of the LAN simulation. It contains essentially the same information as can be found in the UML class diagram of Figure 1.

As can be seen in Figure 3, nodes are represented by rectangles containing a pair of a node name and node type. All possible node types are listed in Table 1. Nodes of type ***B*** (method body) and type ***P*** (formal parameter) do not require a name, but a name between parentheses has been put in the figure to make the graph more readable. Edges are represented by arrows between nodes. Edges have a required type but no name. All possible edge types are listed in Table 2. ***M***-edges (membership) are shown in the figure as arrowless lines. If necessary, multiple edges of the same type that have the same source node may be numbered. For example, an ***M***-node (method signature) can contain multiple edges to ***P***-nodes (formal parameters of the method), and the order of these edges is important.

Node type	Description	Examples
C	**c**lass	*Node*, *Workstation*, *PrintServer*, *Packet*
B	method **b**ody (or implementation) of a method	`System.out.println(p.contents)`
A	**a**ttribute (of variable or field)	*name*, *nextNode*, *contents*, *originator*, …
M	**m**ethod signature (or **m**ethod selector) in the lookup table	*accept*, *send*, *print*
P	formal **p**arameter (or argument) of a message	*p*
E	a (sub)**e**xpression in the method body	`p.contents`

Table 1: Node types

Edge type	Description	Examples
M: A → C	attribute **m**embership	attribute *name* belongs to *Node*
M: B → C	method **m**embership	*send* is defined in *Node*
M: P → M	parameter **m**embership	`send(Packet p)` : *p* is a parameter of *send*
T: P → C	message parameter **t**ype	`print(Packet p)` : parameter *p* has type *Packet*
T: A → C	attribute **t**ype	`String name` : attribute *name* has type *String*
T: M → C	message return **t**ype	`String getName()` : method *getName* has return type *String*
T: E → C	expression result type	`p.contents` has result type *String*
L: M → B	method **l**ookup	`void accept(Packet p)` has three possible method bodies
I: C → C	**i**nheritance (or subclassing)	`class PrintServer extends Node`
E: B|E → E	**e**xpression contained in a method body or in another expression	
S: E → B	**s**tatic call	super call or invocation of static (class) method
D: E → M	**d**ynamic call (or late binding)	`this.send(p)` `nextNode.accept(p)`
P: E → P|E|A	actual **p**arameter	`this.send(p)` `System.out.println(nextNode.name)`
A: E → P|A	**a**ccess of a parameter or attribute	`p.originator == this`
U: E → A	**u**pdate of an attribute	`p.originator = this`
this: E → C	explicit **this** reference	`p.originator == this` `p.originator = this`

Table 2: Edge types

In addition, Figure 4 and Figure 5 represent the (partial) implementation of the method bodies defined in classes *Node* and *Workstation*, respectively, including information about method calls (static calls, dynamic calls and super calls) and attribute invocations (accesses and updates). For example, the implementation of the method body of *send* in the class *Node* is a sequence of two subexpressions, which is denoted by two ordered ***E***-edges from the ***B***-node to two different ***E***-nodes. The second subexpression `this.nextNode.accept(p)` is a cascaded method call consisting of an attribute access (represented by an ***A***-edge originating from the first gray ***E***-node)

followed by a dynamic method call with one parameter (represented by a ***D***-edge and ***P***-edge originating from the second gray ***E***-node).

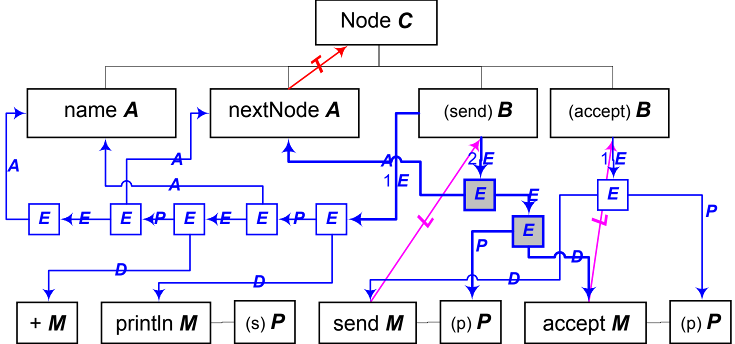


Figure 4: Graph representation of the behaviour of class *Node*

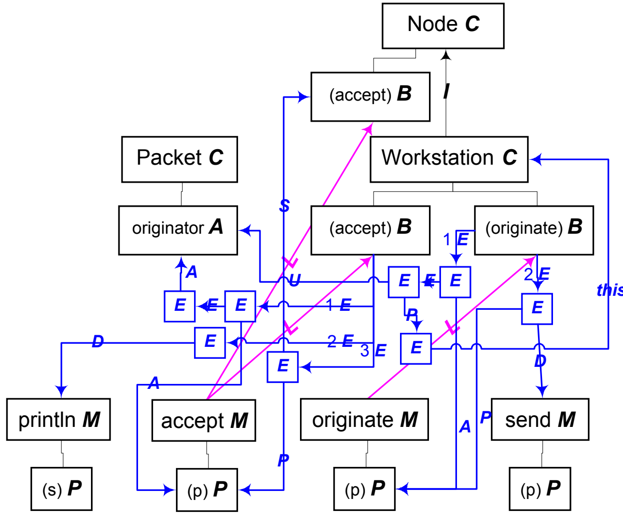


Figure 5: Graph representation of the behaviour of class *Workstation*

3.2 Discussion

This subsection provides a discussion about various issues related to the particular graph representation of the previous subsection. A number of decisions regarding the graph presentation are due to particularities of the Java programming language, while others are due to space considerations.

Type issues

- We only use classes as possible types. In Java, interfaces can also be used as types, but this is not dealt with in this paper due to space limitations.
- Since Java assumes no-variance typing, the return type of a method should be the same in all subclasses. Because of this, we can safely attach the return type to the method signature (***M***-node) rather than the method body (***B***-node).
- Because the graph representation can be generated automatically from the source code, it is type correct, and we assume that it remains type correct after performing a refactoring. In general, however, guaranteeing type correctness requires significant type checking and type inference.

Method parameters

- Because method overriding is only allowed for methods that have the same signature, i.e., the same message name, and the same number and types of parameters, the parameters of a method are attached to the method signature (***M***-node) rather than the method body (***B***-node). Note that the names of parameters can be different in subclasses, but we will ignore this information in the graph representation.

- Since *actual* parameters can only occur in presence of a (static or dynamic) method call, a ***P***-edge must always be accompanied by an ***S***-edge or ***D***-edge originating from the same source node. Conversely, each ***S***-edge or ***D***-edge must be accompanied by a (possibly empty) ordered set of ***P***-edges from the same source node. All these ***P***-edges represent the actual parameters of the call. The number of actual parameters must correspond to the number of formal parameters.

Method calls

- To represent *dynamic method lookup*, we need to make a distinction between the method signature (***M***-node) and the method-bodies (***B***-nodes). They are connected by means of ***L***-edges. Each ***B***-node must be the target of exactly one ***L***-edge. Conversely, an ***M***-node can have many outgoing ***L***-edges.
- *Cascaded method calls* (such as `this.nextNode.accept(p)`)are expressed by means of an ***E***-edge between two ***E***-node (the expression and its subexpression). Cascading is not possible when an ***E***-node has an outgoing ***U***-edge since, for syntactic reasons, attribute update cannot be cascaded with method calls in Java.

Constructor methods. Due to space considerations, we do not treat constructor methods in this paper. Adding them to the formalism is a straightforward matter. Constructor methods can be represented in a similar way as ordinary methods. They also have late binding and super calls. The main difference is that they do not have a return type.

Sequence of statements. A ***B***-node or ***E***-node can have an ordered set of outgoing ***E***-edges to express all subexpressions. In this way, we currently represent sequential statements (`;`) as well as conditional statements (`if (...) A; else B;`).

Attribute update. Since an attribute update specifies the new value of the attribute, an ***U***-edge can be accompanied by at most one ***P***-edge, denoting the expression that is being assigned to the attribute.

Java modifiers. In the current graph representation we ignored visibility information such as abstract, public, private, protected and final. It is a straightforward extension of our graph representation to take these into account.

3.3 Behaviour preservation invariants

There is no commonly accepted definition of what it means to preserve the behaviour of a program. Using an input-output semantics, behaviour preservation means that, for all observable program inputs, the program produces exactly the same output before and after the refactoring. A weaker notion of language preservation is suggested by Bergstein [2]: the set of all acceptable program inputs must be the same before and after the refactoring. When developing real-time systems, yet another notion of behaviour preservation is needed, since it is crucial that the program is executed within the specified time constraints. When developing embedded systems, there might be stringent constraints on the use of computer memory. These observations clearly show the need for a whole range of different kinds of behaviour preservation, each focussing on a different aspect of the program behaviour.

Notation	Interpretation
C —?→ C	There is an edge of undetermined type between two nodes of type ***C*** (class).
? —?*A→ A	There is a nonempty edge path, of which the last edge has type ***A*** (access). The target node has type ***A*** (attribute), and the source node type is undetermined.
C —I+→ C	There is a nonempty edge path of edges of type ***I*** (inheritance) between two nodes of type ***C*** (class)
E —A|U→ A	There is an edge of which the type is either ***A*** (access) or ***U*** (update) between a source node of type ***E*** (expression) and a target node of type ***A*** (attribute).
P|A|M —T→ C	There is an edge of type ***T*** between a source node of type ***P***, ***A*** or ***M*** and a target node of type ***C***.

Table 3: Graph pattern notation

For the purpose of this paper, we have pragmatically chosen to look at only those notions of behaviour preservation that can be expressed and detected in an efficient and intuitive way using our graph representation. More specifically, in our formalism different kinds of behaviour preservation can be expressed by means of

graph invariants. These invariants are occurrences of graph patterns that remain unaltered by the graph rewriting. To express these graph patterns concisely, we use a notation similar to regular expressions as illustrated in Table 3. ***?*** denotes that the type is irrelevant, **|** denotes logical disjunction of types, ***** denotes a path of zero or more edges and **+** denotes a nonempty edge path.

Below we illustrate some typical kinds of behaviour preservation that we would like to detect, and show how they can be expressed in terms of graph invariants.

- **Access preservation**

  Intuitively, a refactoring is *access preserving* if (the implementation of) each method performs at least the same attribute accesses after the refactoring as it did before the refactoring. Note that these attribute accesses may occur indirectly, by first calling another method that (directly or indirectly) accesses the attribute. In our graph formalism, this can be expressed elegantly using the following graph invariant:

  E —*?*A*→ **A**

  It states that, if a (sub-)expression in some method body accesses a certain attribute before the refactoring, it must still access this attribute after the refactoring.

- **Update preservation**

  Similarly to access preservation, a refactoring is *update preserving* if (the implementation of) each method performs at least the same attribute updates after the refactoring as it did before the refactoring. This can be expressed elegantly using the following graph invariant:

  E —*?*U*→ **A**

  Because an attribute update requires one actual parameter that specifies the new value of the attribute to be updated, we can imagine a stronger variant of update preservation that requires the values of all attribute updates to be preserved as well. Since this is more difficult to detect statically we will not deal with it in this paper.

- **Statement preservation**

  A refactoring is *statement preserving* if (the implementation of) each method still performs at least the same actions as it did before the refactoring. Note that these actions may be implemented somewhere else, e.g., in another method that is being called. Again, this can be expressed by means of a graph invariant:

  M —*?*E*→ **E**

- **Type preservation**

  A refactoring is *type preserving* if each statement in each method still has the same result type or return type as it did before the refactoring. This is captured in the following graph invariant:

  E —*?*T*→ **C**

  Note that this is a very weak notion of type preservation. More sophisticated forms can be defined, but this always requires a certain amount of type inference.

Obviously, the above list of behaviour-preservation invariants is by no means complete. Another useful invariant would be *call preservation*, which states that each method still performs the same method calls as it did before the refactoring.

Many kinds of behaviour preservation cannot be expressed in our current graph representation. Examples of this are time preservation and memory preservation, since these would require information about timing constraints or memory constraints to be included in the graph formalism.

3.4 Graph rewriting notation

A *graph rewriting* is nothing more than a transformation that takes an initial graph as input and transforms it into a result graph. This transformation occurs according to some predetermined rules that are specified in a so-called *graph production*. Such a graph production is specified by means of a *left-hand side* and a *right-hand side*. The left-hand side is used to specify which parts of the initial graph should be transformed, while the right-hand side specifies the result after the transformation. In the formalism that we use, the left-hand side can contain *negative*

application preconditions that are used to restrict the applicability of the graph production [5] [11] [12] [13]. Often, a graph production can be applied to different parts of the graph, leading to different *occurrences* (or *matches*) of the graph production's left-hand side. In this paper, we use *parameterised graph productions*, so that different occurrences of a graph production can be distinguished by attaching different parameters to the production.

Figure 6 shows a typical example of a graph production. Left-hand side and right-hand side are separated by means of a ::= symbol. Nodes that are preserved by the graph production are identified by numbers. Nodes on the right-hand side that do not have a number attached are newly created by the graph production. The negative preconditions are denoted by a dashed striked-through oval on the left-hand side. In this case, the negative conditions specify that the transformation is only allowed if the class named *type* does not contain any implementation (i.e., method body) for the method *accessor* with return type *type* and the method *updater* with one parameter of type *type*.

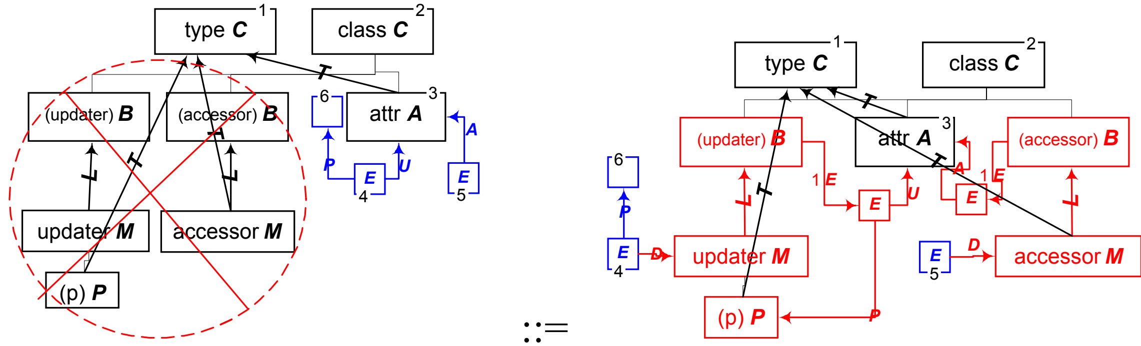


Figure 6: The conditional graph production EncapsulateField**(class,attr,type,accessor,updater)**

4. Graph refactoring productions

In this section, we discuss a number of typical refactorings that we have identified in the LAN example, and we show how they can be expressed using our graph rewriting formalism. Each refactoring in this section follows the same template:

- Motivation
- Example with Java source code
- Example in graph format
- Graph rewriting
- Behaviour preservation invariants

4.1 Rename method

Motivation

On page 273 of [8], the refactoring **RenameMethod** is motivated as follows:

The name of a method does not reveal its purpose.

Change the name of the method.

In Java, as well as most other languages, this refactoring is a *global* operation. The transformation requires a change to all definitions of the method to be renamed (not only in the class where it is defined, but also in all descendant classes where it is overridden) as well as to all static calls, dynamic calls and super calls to this method.

Example with Java source code

In the evolution of the LAN example, suppose the method `accept` is renamed into `receive`. This requires a change to three method definitions of `accept`, as well as to three calls of `accept` (one dynamic call and two super calls). The Java source code before and after this refactoring is given below. The parts of the code that have been modified by the refactoring are shown in *italics*.

```
// *** BEFORE the refactoring RenameMethod ***
public class Node {
  ...
  public void accept(Packet p) {
    this.send(p); }
  protected void send(Packet p) {
    System.out.println(
      name + "sends to" + nextNode.name);
    nextNode.accept(p); }
  }
public class PrintServer extends Node {
  ...
  public void accept(Packet p) {
    if(p.addressee == this) this.print(p);
    else super.accept(p); }
  }
public class Workstation extends Node {
  ...
  public void accept(Packet p) {
    if(p.originator == this)
      System.err.println("no destination");
    else super.accept(p); }
  }
```

```
// *** AFTER the refactoring RenameMethod ***
public class Node {
  ...
  public void receive(Packet p) {
    this.send(p); }
  protected void send(Packet p) {
    System.out.println(
      name + "sends to" + nextNode.name);
    nextNode.receive(p); }
  }
public class PrintServer extends Node {
  ...
  public void receive(Packet p) {
    if(p.addressee == this) this.print(p);
    else super.receive(p); }
  }
public class Workstation extends Node {
  ...
  public void receive(Packet p) {
    if(p.originator == this)
      System.err.println("no destination");
    else super.receive(p); }
  }
```

Example in graph format

The graph representation before the refactoring is depicted in Figure 7. The graph representation after the refactoring is the same, except that the ***M***-node with name *accept* has been renamed into *receive*. This means that renaming can be expressed as a very localised operation in our graph representation. This is because all method implementations and all method calls do not have to be renamed because they do not rely on the name of the method.

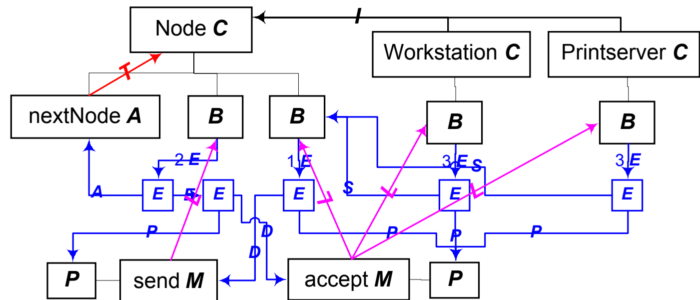


Figure 7: Graph representation before the refactoring RenameMethod

Graph rewriting

The above refactoring can be expressed by the occurrence **RenameMethod**(*Node,accept,receive*) of the parameterised graph production **RenameMethod**(class,old,new) that is formally defined as follows:

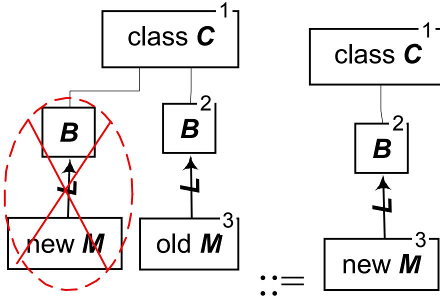


Figure 8: Graph production RenameMethod(class,old,new)

Behaviour preservation invariants

All behaviour preservation invariants of section 3.3 remain satisfied by this graph rewriting, since nothing is changed to the structure of the graph.

4.2 Encapsulate field

Motivation

On page 206 of [8], the refactoring **EncapsulateField** is motivated as follows:

There is a public field.

Make it private and provide accessors.

In other words, this refactoring aims to introduce methods for accessing ("getting") and updating ("setting") the value of an attribute. All direct invocations of an attribute that is defined in a certain class are replaced by calls to these methods. The accessing method that retrieves the attribute's value has no arguments and has a return type that is the same as the type of the attribute. The updating method that updates the attribute's value has no return type, but has one argument whose type is the same as the type of the attribute.

Example with Java source code

In the evolution of the LAN example, we found several occurrences of this particular refactoring. The Java source code before and after one particular instance of this refactoring is given below:

```
// *** BEFORE the refactoring EncapsulateField ***
public class Node {
  public String name;
  public Node nextNode;
  public void accept(Packet p) {
    this.send(p); }
  protected void send(Packet p) {
    System.out.println(
      name +
      "sends to" +
      nextNode.name);
    nextNode.accept(p); }
  }
```

```
// *** AFTER the refactoring EncapsulateField ***
public class Node {
  private String name;
  private Node nextNode;
  public String getName() {
    return this.name; }
  public void setName(String s) {
    this.name = s; }
  public Node getNextNode() {
    return this.nextNode; }
  public void setNextNode(Node n) {
    this.nextNode = n; }
  public void accept(Packet p) {
    this.send(p); }
  protected void send(Packet p) {
    System.out.println(
      this.getName() +
      "sends to" +
      this.getNextNode().getName());
    this.getNextNode().accept(p); }
  }
```

Example in graph format

The picture of the graph representation before the refactoring is shown in Figure 4. The picture of the graph representation after the refactoring is given in Figure 9. Note that, due to space limitations, we only show the accessing methods *getName* and *getNextNode* in Figure 9. The updating methods *setName* and *setNextNode* have been omitted from the figure.

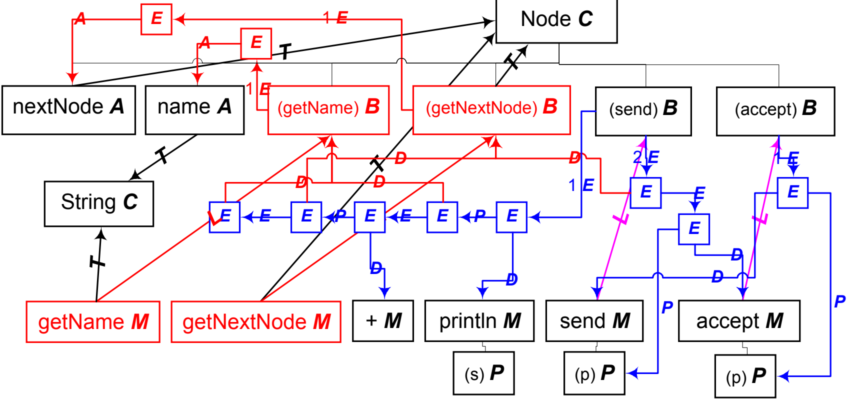


Figure 9: Introducing accessor methods for class *Node*

Graph rewriting

The above refactoring can be formally expressed by two occurrences of the parameterised graph production of Figure 6:

- **EncapsulateField**(*Node*,*name*,*String*,*getName*,*setName*)
- **EncapsulateField**(*Node*,*nextNode*,*Node*,*getNextNode*,*setNextNode*)

Behaviour preservation invariants

It is easy to check that the behaviour preservation invariants of section 3.3 remain satisfied by this graph rewriting:

- The *access preservation* invariant is satisfied since the direct ***A***-edge from ***E***-node *5* to ***A***-node *3* is replaced by a path (***D***-***L***-***E***-***A***) of four edges from ***E***-node *5* to ***A***-node *3*, and the last edge in this path is still an *A*-edge.
- The *update preservation* invariant is satisfied since the direct ***U***-edge from ***E***-node *4* to ***A***-node *3* is replaced by a path (***D***-***L***-***E***-***U***) of four edges from ***E***-node *4* to ***A***-node *3*, and the last edge in this path is still an ***U***-edge.
- The *statement preservation* invariant is automatically satisfied since the graph production **EncapsulateField** has no influence on this graph invariant.
- The *type preservation* invariant is also satisfied, but this requires some type inferencing. On the left-hand side of the graph production, since ***A***-node *3* has a ***T***-edge to ***C***-node *1*, we can infer that ***E***-node *5* must have the same type (since the result type of an attribute access is the same as the type of the attribute itself). On the right-hand side of the graph production, since the new ***M***-node with label *accessor* has a ***T***-edge to ***C***-node *1*, we can infer that ***E***-node *5* must have the same type (since the result type of a dynamic method call is the same as the return type of the method being called). Hence the (inferred) type of ***E***-node *5* on the left-hand side is preserved on the right-hand side.

4.3 Extract method

Motivation

On page 110 of [8], the refactoring **ExtractMethod** is motivated as follows:

You have a code fragment that can be grouped together.

Turn the fragment into a method whose name explains the purpose of the method.

In other words, if we have a method whose implementation is becoming too complex, we can refactor part of its behaviour into a new auxiliary operation.

Example with Java source code

In the LAN example, we extract the printing functionality of the *send* method into a separate *log* method.

```
// *** BEFORE the refactoring ExtractMethod ***
public class Node {
  ...
  public void accept(Packet p) {
    this.send(p); }
  protected void send(Packet p) {
    System.out.println(
      this.getName() +
      "sends to" +
      this.getNextNode().getName());
    this.getNextNode().accept(p); }
  }
```

```
// *** AFTER the refactoring ExtractMethod ***
public class Node {
  ...
  public void accept(Packet p) {
    this.send(p); }
  protected void send(Packet p) {
    this.log(p);
    this.getNextNode().accept(p); }
  protected void log(Packet p) {
    System.out.println(
      this.getName() +
      "sends to" +
      this.getNextNode().getName()); }
  }
```

Example in graph format

The picture of the graph representation before the refactoring is shown in Figure 10. The picture of the graph representation after the refactoring is shown in Figure 11.

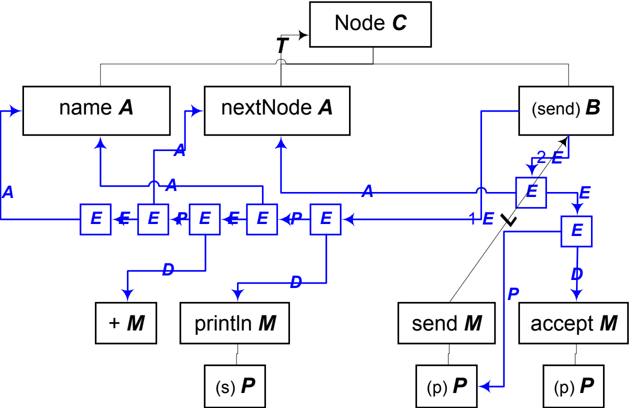


Figure 10: Before extracting logging behaviour from the *send* method

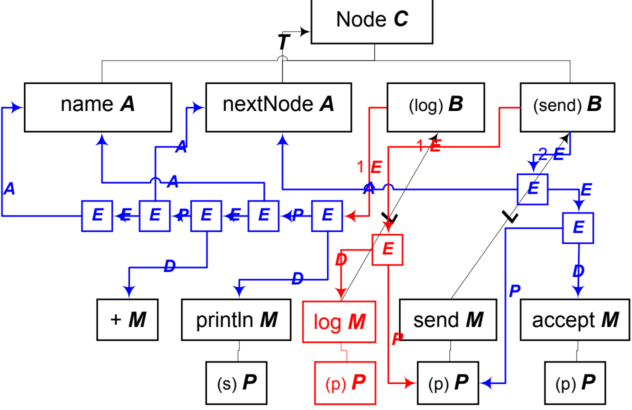
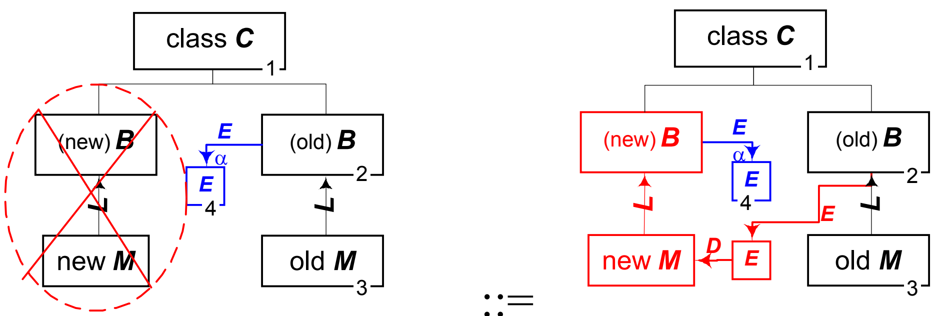


Figure 11: After extracting logging behaviour from the *send* method

Graph rewriting

The above refactoring can be expressed by the occurrence **ExtractMethod**(*Node,send,log,{1}*) of the parameterised graph production **ExtractMethod**(class,old,new,StatList) that is formally defined as follows:



$\forall$ edge $\alpha \in$ StatList

Given the body of method *old* in *class*, all statements specified in *StatList* are redirected to the body of a parameterless method *new*. Obviously, we could also define a more general version of this refactoring where local variables of the old method are passed to the new method as parameters, but this only makes the example more complex without contributing anything more to this paper.

Behaviour preservation invariants

- The *access preservation*, *update preservation* and *type preservation* invariants are automatically satisfied since the graph production **ExtractMethod** has no influence on these graph invariants. It does not modify, introduce or remove any attribute accesses, attribute updates or type information.
- The *statement preservation* invariant is satisfied since the edge path of length two on the left-hand side from ***M***-node *3* to ***E***-node *4* via ***B***-node *2* is replaced by a path (***L-E-D-L-E***) of five edges from ***M***-node *3* to ***E***-node *4*, and the last edge in this path is still an ***E***-edge.

4.4 Replace data value with object

Motivation

On page 175 of [8], the refactoring **ReplaceDataValueWithObject** is motivated as follows:

You have a data item that needs additional data or behavior.

Turn the data item into an object.

Example with Java source code

In the LAN example, this refactoring has been used to introduce *Document* objects. This step is necessary in order to prepare the framework for different kinds of documents that can be printed on different kinds of printers. Before the refactoring, a *Packet* contains an attribute *contents* of type *String*. This variable is only accessed through the accessor methods *getContents* and *setContents*. After the refactoring, the attribute *contents* of type *String* is moved into a new class *Document*. *Packet* can still refer to it indirectly by means of a new attribute *doc* of type *Document*. Consequently, all accesses and updates of the attribute must take place through an extra indirection. Fortunately, this indirection can be encapsulated in the accessor methods *getContents* and *setContents*.

```
// *** BEFORE the refactoring ReplaceDataValueWithObject ***
public class Packet {
  ...
  private String contents;
  public String getContents() {
    return this.contents;
  }
  public void setContents(String s) {
    this.contents = s;
  }
}
```

```
// *** AFTER the refactoring ReplaceDataValueWithObject ***
public class Packet {
  ...
  private Document doc;
  public String getContents() {
    return this.doc.contents;
  }
  public void setContents(String s) {
    this.doc.contents = s;
  }
}

public class Document {
  ...
  public String contents;
}
```

Example in graph format

The graph representation before the refactoring is depicted in Figure 12. The graph representation after the refactoring is shown in Figure 13.

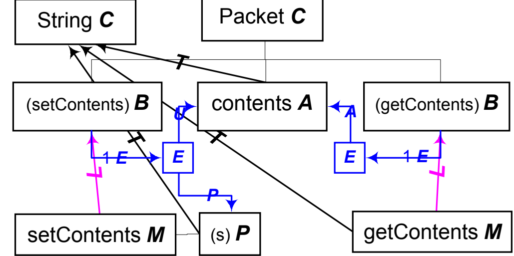


Figure 12: Before turning the attribute *contents* into a *Document* object

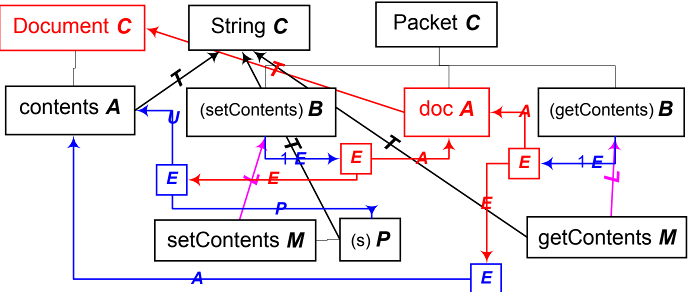
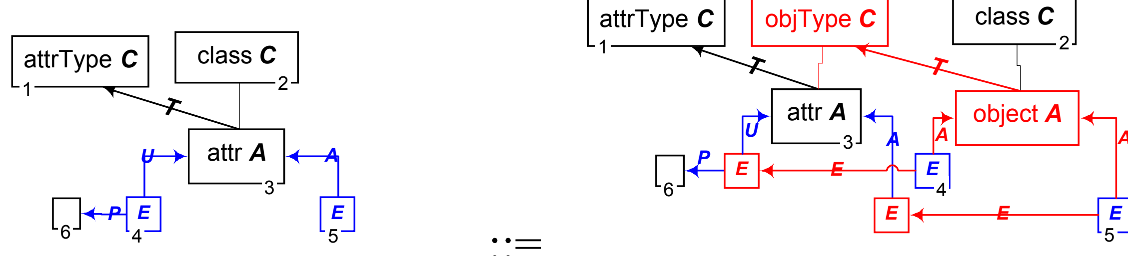


Figure 13: After turning the attribute *contents* into a *Document* object

Graph rewriting

The above refactoring can be expressed formally by an occurrence **ReplaceDataValueWithObject**(*Packet,contents,String,doc,Document*) of the parameterised graph production **ReplaceDataValueWithObject**(class,attr,attrType,object,objType) that is shown below:



Behaviour preservation invariants

- The *access preservation* invariant is satisfied since the direct ***A***-edge from ***E***-node *5* to ***A***-node *3* is replaced by a path (***E-A***) of two edges from ***E***-node *5* to ***A***-node *3*, and the last edge in this path is still an ***A***-edge.
- The *update preservation* invariant is satisfied since the direct ***U***-edge from ***E***-node *4* to ***A***-node *3* is replaced by a path (***E-U***) of two edges from ***E***-node *4* to ***A***-node *3*, and the last edge in this path is still an ***U***-edge.
- The *statement preservation* invariant is automatically satisfied since the graph rewriting **ReplaceDataValueWithObject** has no influence on this graph invariant.
- The *type preservation* invariant is also satisfied. Before and after the graph rewriting, there is an edge path from ***E***-node *5* to ***C***-node *1*, and another edge path from ***E***-node *4* to ***C***-node *1*. Note that this is only a very weak form of type preservation. In order to know more precisely whether all type information is preserved, more extensive type inferencing is needed.

5. Open Questions

This paper is only a first step towards a complete formal foundation for object-oriented refactoring, and there is still a lot of work to be done. Below we have compiled the most important things that remain to be done in the form of a list of open questions with partial answers.

- *Which other variants of behaviour preservation are important?*

  For each such variant, motivate why, and give a concrete situation or example.
- *Which graph rewriting formalism is most appropriate for the purpose of refactoring (or software evolution in general)?*

  This paper used conditional graph rewriting (based on the category-theoretical single-pushout approach) [5] [11] [12]. In the research literature, many other graph rewriting formalisms can be found, and these formalisms may be more appropriate for our needs. For example, Luqi and Goguen [17] propose hierarchical evolutionary hypergraphs as an alternative formal model for software evolution.
- *How can the graph rewriting formalism be made as language independent as possible?*

  In this paper, we only focussed on Java refactorings. However, the proposed formalism seems very well-suited to specify object-oriented refactoring and its properties (such as behaviour-preservation) in a language-independent way? If we apply the formalism for different languages, how should the language-independent and language-specific features be separated? (**In Java**: static typing, interfaces, packages, abstract keyword, constructors, visibility information such as public, private, protected, final. **In Smalltalk**: dynamic typing, metaclasses, namespaces.)

  Note that [31] presents an informal discussion on language-independence issues of object-oriented refactorings.
- *How can we express dynamic (or run-time) information about the software?*

  In the current graph representation we have excluded the following information since it requires a more extensive data-flow and control-flow analysis:
  - Order of messages, e.g., `p.setOriginator(this); this.send(p)`
  - Conditional messages, e.g., `if (p.addressee = this) this.print(p) else super.accept(p)`
- *Can the graph rewriting formalism help with retroactive analysis of the refactorings?*

  In this paper, we only considered **proactive** support for refactoring. If all refactorings can be applied automatically, it is guaranteed that the refactorings do not lead to inconsistent programs, and that the behaviour is preserved by means of graph invariants.

An alternative would be to provide **retroactive** support, by allowing the user to manually apply the refactoring, with the option the check afterwards whether the applied refactoring is correct in the sense that it preserves the program behaviour. The question is whether and how the graph rewriting formalism can provide support for this. Another question is whether the graph rewriting formalism can be used to analyse an arbitrary software evolution step to identify which (sequence of) refactoring(s) has been applied.

- *How do refactorings affect quality factors*?

  Typically, refactorings aim to improve encapsulation, reduce complexity, and remove code redundancy. Nevertheless, some refactoring increase the complexity of the class structure, yet proponents claim that this extra complexity improves understandability. A formal classification of the different refactorings in terms of the quality factors they aim to improve is therefore essential.

- *How can a graph rewriting formalism help to cope with the interaction and composition of refactorings*?

  Graph rewriting seems to be an intuitive formalism to combine primitive refactorings into more complex refactorings. Relevant material in the work on graph rewriting includes work on modularity and hierarchical graphs [7] [24]. Graph rewriting can also prove helpful to cope with unexpected interactions between refactorings, as such causing potential conflicts when these refactorings are applied in parallel. See [18] [19] for the use of graph rewriting in the context of detecting evolution conflicts. Especially the formal theorems about local confluence, parallel and sequential independence are relevant here. For example, local confluence guarantees that, under certain conditions, a sequence of refactorings can be reordered without influencing the end-result. This can be used to reorder the refactorings into a canonical form, or to remove redundant refactorings.

- *(How) should we provide support for non-behaviour-preserving refactorings?*

  In practice, many complex refactorings can be broken down in more primitive transformations. Often, these transformations are only behaviour preserving if they are applied in combination with the other primitive transformations. Hence, a formalism for refactoring should also be able to deal with transformations that are only partially behaviour preserving.

- *How do refactorings affect design models*?

  Refactorings are typically applied at the code-level, however these changes will also affect representations at design level. Given the behaviour-preserving nature of refactorings, it should be possible for certain representations of the software to identify which parts of the design remain unaffected and which parts must be refactored. Possible formal approaches that can help with this are *triple graph grammars* and *critical pair analysis* [6] [15] [28].

6. Conclusion

This paper presented a feasibility study of the use of graph rewriting as a formal foundation for refactoring object-oriented software. While the initial results look very promising, there is still a considerable amount of work needed, from the formal as well as the practical side.

From a formal side, …

From the practical side, automated tool support for refactoring is crucial, since refactoring is a time-consuming and error-prone process. An ideal refactoring tool would enable its user to

- identify places in the program where refactoring is needed (see, e.g., [Simon&al 2001] [Kataoka&al 2001])
- visualize the program, or program part, considered a candidate for refactoring
- suggest possible refactorings for the program part considered, taking into account user-specified quality factors
- detect whether a refactoring is applicable to a given piece of code. If not, suggest concrete changes that can make the refactoring applicable
- automatically apply a given refactoring to the source code
- visualize the effect or impact of a refactoring on the program
- check retroactively whether the behaviour preservation invariants of a refactoring are satisfied if the refactoring was performed through a manual process
- compose sequences of refactorings, and store frequently used composite refactorings for later use
- analyse two sequences of refactorings that have been applied in parallel with the aim to (1) detect potential inconsistencies between these refactorings; (2) detect commonalities between these refactorings

- optimise sequences of refactorings by (1) reordering them into a canonical form; (2) removing redundancy in the sequence

8. References

[1] K. Beck, *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

[2] P. L. Bergstein. "Maintenance of Object-Oriented Systems During Structural Evolution," *TAPOS Journal* 3(3): 185-212, 1997

[3] E. Casais, "An Incremental Class Reorganization Approach," *Proc. ECOOP'92*, O. Lehrmann Madsen (Ed.), LNCS 615, Springer-Verlag, 1992.

[4] A. Corradini, H. Ehrig, M. Löwe, U. Montanari and J. Padberg, "The Category of Typed Graph Grammars and their Adjunction with Categories of Derivations," *Proc. 5th Int'l Workshop on Graph Grammars and their Application to Computer Science*, LNCS, Springer-Verlag, 1996.

[5] H. Ehrig and A. Habel, "Graph Grammars with Application Conditions," *The Book of L*, G. Rozenberg and A. Salomaa, eds., Springer-Verlag, 1986, pp. 87-100.

[6] H. Ehrig and A. Tsioalikis. "Consistency analysis of UML class and sequence diagrams using attributed graph grammars". In H. Ehrig and G. Taentzer, editors, *ETAPS 2000 workshop on graph transformation systems*, March 2000.

[7] G. Engels and A. Schürr, "Encapsulated Hierarchical Graphs, Graph Types and Meta Types," Joint Compugraph/Semagraph Workshop on Graph Rewriting and Computation. *Electronic Notes in Theoretical Computer Science*, Vol. 2, Elsevier, 1995.

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

[9] W. G. Griswold, *Program Restructuring as an Aid to Software Maintenance*. PhD Thesis, University of Washington, August 1991.

[10] W. G. Griswold and D. Notkin, "Automated assistance for program restructuring," *ACM Trans. Software Engineering and Methodology*, 2(3): 228-269, July 1993.

[11] A. Habel, R. Heckel and G. Taentzer, "Graph Grammars with Negative Application Conditions," *Fundamenta Informaticae*, Special Issue on Graph Transformations, Vol. 26, No 3,4 (June), 1996, pp. 287-313.

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

[13] R. Heckel and A. Wagner, "Ensuring Consistency of Conditional Graph Grammars: A Constructive Approach," *Lecture Notes in Theoretical Computer Science*, Vol. 1, Elsevier Science, 1995.

[14] Y. Kataoka, M. D. Ernst, W. G. Griswold and D. Notkin, "Automated Support for Program Refactoring Using Invariants," *Proc. Int. Conf. Software Maintenance*, pp. 736-743, IEEE Computer Society Press, 2001.

[15] M. Lefering. *Development of incremental integration tools using formal specifications*. Technical Report, RWTH Aachen, 1994.

[16] M. Löwe, "Algebraic Approach to Single-Pushout Graph Transformation," *Theoretical Computer Science*, Vol. 109, 1993, pp. 181-224.

[17] Luqi, J. A. Goguen, "Formal Methods: Promises and Problems," *IEEE Software*, pp. 73-85, January 1997.

[18] T. Mens, *A Formal Foundation for Object-Oriented Software Evolution*, PhD Thesis, Vrije Universiteit Brussel, Belgium, Dept. Computer Science, September 1999.

[19] T. Mens, "Conditional Graph Rewriting as a Domain-Independent Formalism for Software Evolution," *Proc. Int'l Agtive '99 Conf.*, LNCS 1779, pp. 127-143, Springer-Verlag, 2000.

[20] T. Mens and T. Tourwé, "A Declarative Evolution Framework for Object-Oriented Design Patterns," *Proc. Int. Conf. Software Maintenance*, IEEE Computer Society Press, 2001.

[21] Ivan Moore, "Automatic Inheritance Hierarchy Restructuring and Method Refactoring," *Proc. OOPSLA '96*, ACM Press, 1996.

[22] W.F. Opdyke, *Refactoring Object-Oriented Frameworks*, PhD Thesis, Univ. of Illinois at Urbana-Champaign, 1992. Tech. Report UIUC-DCS-R-92-1759.

[23] W.F. Opdyke and R.E. Johnson, "Creating abstract superclasses by refactoring," *Proc. ACM Computer Science Conference*, pp. 66-73, ACM Press, 1993.

[24] A. Poulovassilis and M. Levene, "A Nested-Graph Model for the Representation and Manipulation of Complex Objects," *ACM Transactions on Information Systems*, 12(1): 35-68, ACM Press, 1994.

[25] D. Roberts, J. Brant and R.E. Johnson, "A Refactoring Tool for Smalltalk," J. Theory and Practice of Object Systems, Vol. 3, No. 4, 1997, pp. 253-263.

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

[27] A. Schürr, "PROGRES: A VHL-Language Based on Graph Grammars," *Proc. 4th Int'l Workshop on Graph Grammars and their Application to Computer Science*, LNCS Vol. 532, Springer-Verlag, 1991, pp. 641-659.

[28] A. Schürr. *Specification of graph translators with triple graph grammars*. Technical Report AIB 94-12, RWTH Aachen, 1994.

[29] F. Simon, F. Steinbruckner and C. Lewerentz, "Metrics Based Refactoring," *Proc. 5th European Conference on Software Maintenance and Reengineering*, pp. 30- , IEEE Computer Society Press, 2001.

[30] G. Sunyé, D. Pollet, Y. LeTraon and J.-M. Jézéquel, "Refactoring UML models," *Proc. UML 2001*, LNCS 2185, pp. 134-148. Springer Verlag, 2001.

[31] S. Tichelaar, *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD Thesis, University of Bern, 2001.

[32] L. Tokuda and D. Batory, "Evolving Object-Oriented Designs with Refactorings," *Proceedings ASE'99*, IEEE Computer Society Press, 1999

[33] M. M. Werner, *Facilitating Schema Evolution with Automatic Program Transformation*. PhD Thesis, Northeastern University, July 1999.