



Proceedings of the
Third International ERCIM Symposium on
Software Evolution
(Software Evolution 2007)

Exploring a Method to Detect Behaviour-Preserving Evolution Using
Graph Transformation

Javier Pérez, Yania Crespo

10 pages

Exploring a Method to Detect Behaviour-Preserving Evolution Using Graph Transformation

Javier Pérez¹, Yania Crespo²

¹ jperez@infor.uva.es, www.infor.uva.es/~jperez

² yania@infor.uva.es, www.infor.uva.es/~yania

^{1,2} Departamento de Informática, ETSI Informática
Universidad de Valladolid, Valladolid, Spain

Abstract: One of the problems of documenting software evolution appears with the extensive use of refactorings. This paper explores a method, based on graph transformation, to detect whether the evolution between two versions of a software system can be expressed by means of a sequence of refactoring operations. For this purpose we extend a graph representation format to use it for simple Java programs, and we show a sample implementation of the method using the AGG graph transformation tool. In case a refactoring sequence exists, our technique can help reveal the functional equivalence between the two versions of the system, at least, as far as refactorings can assure behaviour preservation.

Keywords: finding refactoring sequences, documenting software evolution, behaviour preservation, functional equivalence, graph transformation

1 Introduction

Refactorings [FBB⁺99, Opd92] are structural transformations that can be applied to a software system to perform design changes without modifying its behaviour. Efforts to include refactorings as a regular technique in software development have led refactoring support to be commonly integrated into development environments (*e.g.* Eclipse Development Platform, IntelliJ® Idea, NetBeans, etc.). Finding refactorings, now that they are extensively used, is one of the problems of documenting and understanding software evolution [DDN00].

We explore a method, based on graph transformation [EEKR99, Roz97], to search for a refactoring sequence which can describe the evolution of a software system. A toy Java system will be used, as an example, to explain our approach. The AGG graph transformation tool [ERT99] is used for a sample implementation of the method.

The paper is organised as follows. Firstly, in [Section 2](#) we show the example used to illustrate our proposed method. [Section 3](#) introduces an extension of a graph format to allow representing simple Java programs as graphs, and Java refactorings as graph transformation rules. In [Section 4](#) we describe our method to find refactorings using graph parsing. [Section 5](#) describes a sample implementation using the AGG graph transformation tool and, in [Section 6](#) we show the test-run of this implementation with the previously formulated example. The analysis and comparison to related work of our approach is performed in [Section 7](#). Finally, in [Section 8](#) we present our future work and conclude. inheritance

2 A short example

We will use a toy Java system, slightly inspired in the LAN simulation example [JDM03, MT04], as a running example to illustrate our approach. Our original system (see Figure 1(a)) was designed to model printers for different document types and a hub to connect them, providing a single access point. A hierarchy of default and specific printers allows to add more printers when needed. When the administrator notices that no other type than pdf documents are sent, the original system is rashly modified to simplify the printer hierarchy (see Figure 1(b)). When a new system administrator arrives, he finds the two versions of the system, and no documentation about the changes performed between them. The problem here is not only about documenting the changes performed, but mainly to test whether the new system is functionally equivalent to the old one. For space reasons we have simplified the example removing some method's bodies.

```
//-----PrinterHub.java
public class PrinterHub {
    public void printWithPDF(PDFPrinter p){
        p.printPDF();
    }
}
//-----PDFPrinter.java
public class PDFPrinter extends Printer{
    public void printPDF(){
        // body of printPDF method
    }
}
//-----Printer.java
public class Printer {
    public String content;
    public void setContent(String c){
        this.content = c;
    }
    public void printDefault(){
        // body of printDefault method
    }
}
```

(a) The original printing system

```
//-----PrinterHub.java
public class printerHub {
    public void print(Printer p){
        p.print();
    }
}
//-----Printer.java
public class Printer {
    public String content;
    public void setContent(String c){
        this.content = c;
    }
    public void print(){
        // body of print method
    }
}
```

(b) The modified printing system

Figure 1: The printing system example

We know that the new system was obtained after applying the following sequence of refactorings over the old one: **1)** *removeMethod* applied to *Printer.printDefault()*; **2)** *pullUpMethod* applied to *PDFPrinter.printPDF()*, placing it at *Printer.printPDF()*; **3)** *renameMethod* executed over *Printer.printPDF()*, which is renamed as *Printer.print*; **4)** *renameMethod* executed over *PrinterHub.printWithPDF()*, which gets renamed as *PrinterHub.print*; **5)** *useSuperType* applied over *Printer.print(PDFPrinter p)*, resulting the new method signature *Printer.print(Printer p)*; **6)** *removeClass* performed over the class *PDFPrinter*.

This sequence uses only refactorings supported by a refactoring tool. This restriction implies that it is composed by widely tested refactorings, being also widely recognised that they are behaviour-preserving. This guarantees that the sequence is behaviour-preserving and both versions of the system are functionally equivalent. These transformations belong to the set supported by the Eclipse Development Platform¹. The *removeMethod* and *removeClass*, whose definitions can be found, for example, in [Opd92], are not provided by Eclipse. These refactorings can be performed manually, checking the preconditions and deleting the desired source code.

¹ Eclipse Development Platform homepage: <http://www.eclipse.org>

3 Representing refactorings with graph transformation

To tackle the problem we use a formal representation for refactorings and Object-Oriented software. Refactorings involve modification of the system structure, and refactoring definitions must include preconditions and postconditions in order to guarantee that these changes are behaviour-preserving. Therefore, we have selected graph transformation [EEKR99, Roz97] as the underlying formalism because its naturally focused on description and manipulation of structural information and allows to model refactoring conditions too. Formalisation of refactorings with graph transformation is described and validated in [EJ05, MVDJ05]. In that work, programs and refactorings are represented with graphs, in a language-independent way, using a kind of abstract syntax trees with an expressiveness level which is adequate for the problem.

3.1 Java programs as graphs

We use the part of the formalism from [MVDJ05] referenced as “program graphs” and related to software representation to represent Java programs. As it is suggested in that work, it is necessary to extend the graph format when representing programs containing specific elements and constructions of a particular language. Bearing this fact in mind, we have developed an extension to represent simple Java programs which we have named “Java program graphs”.

We describe the edges and nodes which build up this representation format within Tables 2(a) and 2(b), including the elements of the original format and our modifications. Observing the contribution of our extension, it can be noticed, for example, that some new node types have been added to represent packages, interfaces, literals and operators, altogether with their relationships edges. We also included a *type* attribute to *E* nodes, in order to represent control structures and language constructions like a *return* or an instantiation (*new*) expression. Added *seq* attributes to *E* and *P* nodes allow to represent the order of the sentences within a method body and the order of the parameters within the signature of a method.

Using our extension, we can represent the old version of the printing system (see Figure 1(a)) as displayed in Figure 3. The graph shows that a prefix *s:* has been added to each node and edge type label. This prefix is needed by the implementation that will be described later.

3.2 Refactorings as graph transformation rules

Graph transformation rules are similar to the derivation rules of string grammars. To apply a rule, a graph transformation system looks for a subgraph, within the graph currently being transformed, which matches the left-hand side of the rule. Then, the left-hand side matched subgraph is transformed into the subgraph of the right-hand side. Rule constraints, such as negative application conditions, can be added. Negative application conditions (NAC) describe subgraphs which forbid the application of a rule if a match for the NAC exists. The basis of formalising refactorings as graph transformation rules is presented in [EJ05, MVDJ05].

Figure 4(c) shows how we specify a simplified version of the *removeMethod* refactoring with the AGG graph transformation tool. The leftmost part of Figure 4(c) is the left-hand side of the rule. It represents a method signature (node type *M*) dynamically linked (edge type *l*) to a method definition (node type *MD*) which is a member (edge type *m*) of a class (node type *C*). The

Concept	Node	Attributes	Description
package	PKG		Java package
		name: string	name
interface	I	(same as package)	Java interface
class	C		Java class
		name: string	name
		visibility: enum	visibility: [non-public, public, protected, private]
		hierarchy: enum	order within a inheritance hierarchy: [abstract, middle, final]
		static: boolean	is static?: [true, false]
method	M	(same as class)	signature of a method
	MD		implementation of a method
variable	V	(same as class)	attribute or local variable
	VD		attribute or local variable declaration
parameter	P		formal parameter
		seq: integer	position of the parameter within the parameter list
expression	E		expression, subexpression or sentence
		type: enum	type of expression: [new, return, if, else, for, loop, ...]
		seq: integer	order within a sequence of subexpressions sharing the root node, i.e. sentence order
operator	O		operator
		type: string	string that represents the operator
literal	L		literal or constant
		value: string	string representation of the literal value

(a) nodes and attributes

Label	Edge	Description
i:	C \rightarrow C	inheritance relationship between classes (reference on an "extends" subexpression)
	I \rightarrow I	inheritance relationship between interfaces (reference on an "extends" subexpression)
	C \rightarrow I	class which implements an interface (reference on an "implements" subexpression)
o:	C \rightarrow I	an interface is observable from a class ("import" subexpression)
	C \rightarrow C	a class is observable from a class ("import" subexpression)
	I \rightarrow I	an interface is observable from an interface ("import" subexpression)
	I \rightarrow C	a class is observable from an interface ("import" subexpression)
t:	M \rightarrow C	type of a method
	V \rightarrow C	type of a variable or attribute
	P \rightarrow C	type of a formal parameter
	L \rightarrow C	type of a literal/constant
m:	MD \rightarrow C	body of method MD appears in class C
	VD \rightarrow C	declaration of an attribute within a class
	VD \rightarrow MD	declaration of a local variable within a method's body
	PKG \rightarrow PKG	a package belongs to another package
	C \rightarrow PKG	a class belongs to a package
	I \rightarrow PKG	an interface belongs to a package
	M \rightarrow I	method's signature belongs to an interface
	VD \rightarrow I	attribute declaration belongs to an interface
l:	M \rightarrow MD	binding of method's signature to a method's body
	V \rightarrow VD	binding of an attribute or a local variable to its declaration
p:	M \rightarrow P	formal parameter declaration within a method's signature
e:	MD \rightarrow E	expression within a method's body
	E \rightarrow E	subexpression or hierarchical relationship between expressions
d:	E \rightarrow VD	local variable declaration in subexpression
c:	E \rightarrow M	method call in subexpression
	E \rightarrow O	operator used in subexpression
a:	E \rightarrow V	access to a local variable or attribute
	E \rightarrow P	access to a parameter
	E \rightarrow L	access to a literal
u:	E \rightarrow V	variable update expression
	E \rightarrow P	parameter update expression

(b) edges

Figure 2: Description of nodes and edges of Java program graphs

right-hand side describes the result of the rule application. All the elements that represent the method found by the left-hand side matching subgraph, have been removed from that subgraph. Figure 4(b) describes a NAC: if any expression (node type *E*) exists, which represent a call (edge type *c*) to the method in the left-hand side, the rule can not be applied. The numbers appearing in the graph elements stand for identity mappings between elements from the different parts of the rule (left-hand side, right-hand side and NAC). Attribute values, like *name=mName*, specify variables, parameters of the rule, which are substituted with values from the graph to find a match. When the left-hand side of the rule is matched to a subgraph and a match for the NAC can not be found, the subgraph is transformed into the right-hand side of the rule. Application of the rule of Figure 4(c) removes a method from the class where it is declared.

4 A method to detect refactoring sequences

Using the representation format described in the previous Section, we can apply graph parsing algorithms to search for a refactoring sequence between two different versions of a software system. We address the problem of finding this transformation sequence as a state space search problem. With this approach we identify: **the original/old system** as a start state, **refactoring operations** as state changing operations (edges), **the refactored/new system** as the goal state, **the problem of whether a refactoring sequence exists** as a reachability problem, and a **refac-**

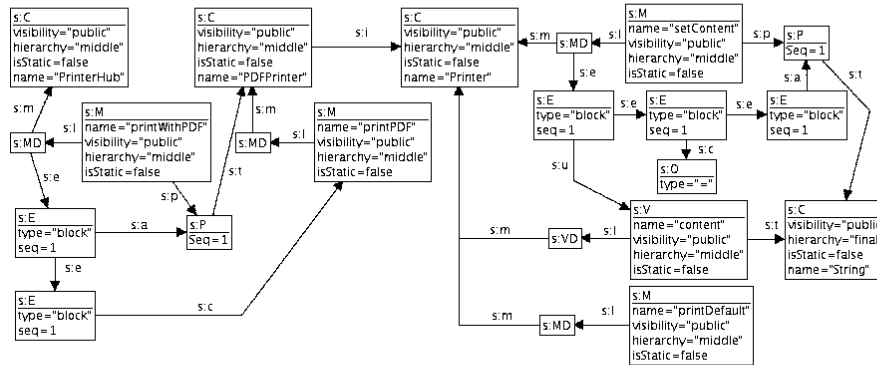


Figure 3: The original system of the example

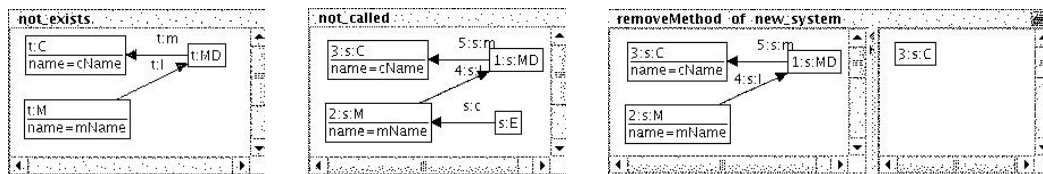


Figure 4: Rule to search *removeMethod*

toring sequence as the path from the start state to the goal state. We will refer to the old version of the system as “*source*”, to the new version of the system as “*target*” and we will use “*current*” to refer to the state being explored at a certain point. The “*current*” state will represent a system that is being transformed into the new version during the searching process.

We propose a basic search algorithm to search for refactoring sequences (see Figure 5). In order to allow some kind of guided search and the future addition of heuristics, we use preconditions and postconditions of refactorings. So our approach needs refactoring definitions which include preconditions and postconditions, as they are used in [KK04].

5 Implementation in the AGG graph transformation tool

We have prepared a prototype with version 1.6.0 of AGG [ERT99]. AGG is a rule based visual language, and tool, supporting an algebraic approach to graph transformation². We have chosen AGG mainly because it allows rapid prototyping of graph transformation systems. Additionally, it supports graph parsing, what can be used to perform depth first search with backtracking, allowing us to explore our algorithm straightly, with little implementation effort.

² AGG home page, graph grammar group, Technische Universität Berlin: <http://tfs.cs.tu-berlin.de/agg>

```

Graph source, current, target
List rules_to_apply, ruleset, sequence
Rule rule; Pair(Graph, List) node; Stack stack

initialise source, target, ruleset

copy source to current
rules_to_apply = find_rules(current, target, ruleset)
rule = rules_to_apply.remove_first
sequence = empty

while current is not isomorphic to target and
  rules_to_apply is not empty
do
  if rule is not empty then // candidate rule
    sequence.add_last(rule)
    current = apply_rule(current, rule)
    rules_to_apply = find_rules(current, target, ruleset)
    rule = rules_to_apply.remove_first
    stack.push(node(current, rules_to_apply)) // state
  else
    sequence.remove_last
    node = stack.pop // recover a previous state
    if node is not empty then
      current = node.graph; rules_to_apply = node.rules
      rule = rules_to_apply.remove_last
    end //if
  end //if
end //while
if current is isomorphic to target then
  return sequence //success
else
  return empty //fail
end //if

List find_rules(Graph source, Graph target, List ruleset)
do
  List rules_to_apply = empty;
  Rule rule;

  foreach rule in ruleset do
    if rule.precondition holds in source and
       rule.postcondition holds in target
    then
      rules_to_apply.add_last(rule)
    end //if
  end //foreach
  sort(rules_to_apply, random)
  return rules_to_apply
end //find_rules

```

Figure 5: Basic refactoring searching algorithm

AGG graph parsing, which is mainly oriented to visual languages parsing [RS97], allows checking whether a particular graph belongs to the graph language generated by a graph grammar or not. A graph grammar includes an initial graph and a set of graph transformation rules. The set of graphs generated by a graph grammar defines a graph language. To perform graph parsing, AGG needs a “parsing grammar”, a grammar whose transformation rules are reversed so they can transform a given graph to reduce it to the initial graph of the grammar. The tool needs at least: a set of graph transformation rules, the graph to be transformed, called “host graph” and the initial graph, called a “stop graph”. The parsing process is based on a simple backtracking algorithm. The tool randomly chooses graph transformation rules from the parsing grammar and iteratively applies them to the host graph. The process stops when there are no more applicable rules or when the host and stop graphs become isomorphic.

To implement our method we use the source system representation as the host graph, and the target system representation as the stop graph. The host graph also acts as the “current” graph, the graph being transformed during the searching process. To guide the search, we have added another subgraph to the host and to the stop graph: a guidance graph which is another copy of the target system graph. This graph allows to prioritise the selection of refactorings whose “effects” or postconditions can be found within the target system. Each iteration, to select a refactoring for the sequence, its preconditions are searched within the “current” graph and its postconditions within the guidance graph, which does not change during the parsing process. To distinguish both subgraphs, we have added the prefix *s:* to labels in the source and target system graph, and *t:* to labels in the guidance graph.

This can be seen in the rule of Figure 4. First, we implement a refactoring operation by specifying the transformation source and context in the left-hand side of the rule and the result on the right-hand side. These elements have their type labels prefixed with *s:* (see Figure 4(c)). Additional refactoring preconditions, which do not form part of the refactoring context, must be expressed as NACs formulated over the current graph (see Figure 4(b)). Refactoring postcondi-

tions are specified as conditions over the guidance graph. Positive postconditions are included within the left-hand side of the rule, while negative postconditions must be expressed as NACs. Figure 4(a) express that the removed method does not exist in the target system graph.

6 Parsing the example

We launch the parser using the graph representation of the old system (see Figure Figure 3) as the host graph, the graph representation of the modified system as the stop graph, and the search rules for the refactorings mentioned in Section 2 as parsing rules. In order to explore our approach, we have only implemented the rules needed by the example. Given the small complexity of experiment, the state space resulted is finite and thus the parser does not have problems finding a valid sequence. Termination of the algorithm depends on the size of the state space, which we believe can be restricted with the use of refactoring postconditions in the searching process.

To test our approach we just needed a raw output from the parser. We obtained the parser's debugging information to know which rules have been applied, when backtracking occurs, the intermediate graphs, etc. From that debugging information, we have extracted that the parser has found the refactoring sequence: $1 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 6$. This sequence differs from the one we proposed in Section 2 but it is an equivalent one for our purposes, and thus is a valid result. The main goal of our approach was not to find the exact refactoring sequence which was originally applied. Our goal is to find whether the system evolution was behaviour-preserving and to try to formulate this evolution in terms of a suitable refactoring sequence.

This result is enough to detect the functional equivalence and to document the general intention of the changes applied. Even if the developer does not know to have applied a behaviour-preserving sequence of changes, if this has been the case, it can be revealed. More experiments, with more refactoring rules, bigger systems, and different graph transformation tools are planned, to test the scalability and obtain the performance details of the approach.

7 Discussion

7.1 Related work

As far as we know, there are not many works dealing with finding refactorings, and the efforts have been mainly addressed to mining refactorings that occur mixed with other changes which are not behaviour-preserving. Our approach focuses on computing if a refactoring sequence exists between two versions of a software systems or not. What we offer, through this refactoring sequence calculation, is a kind of test for functional equivalence between systems as long as the refactorings we support assure behaviour preservation.

Demeyer *et al.* [DDN00] use metrics to reveal refactorings with the purpose of helping a programmer to understand the evolution of a system. Their technique applies to a scenario where human examination of the source code is performed after the automatic analysis. The main problems of the approach are that it does not behave well with renamings and that it loses effectiveness when many changes have been applied to the same piece of code.

With our approach, multiple refactorings executed over the same element can be found easily.

To apply a refactoring searching rule we initially impose the condition of finding the refactoring postcondition in the refactored system graph. Refactoring postconditions included in the rule definition can be more or less restrictive. Less restrictive postconditions allow the rule to be applied whether the refactoring immediate result can be observed on the refactored system graph or not. The more restrictive postconditions we include in the refactoring searching rules definitions, the smaller the state space to search is. We can fine-tune the refactoring searching rules to balance state space size against searching capability.

Handling renamings is not a problem either in our approach. The previous argumentation applies in this case too. Renamings are expressed in the same way as any other refactoring, and we can increase the capability of finding renamings making their postconditions less restrictive.

Görg and Weißgerber [GW05] present an approach to find refactorings from consecutive transactions in CVS repositories. A transaction, in this context, is each new version of a system submitted to the repository. This work is based in pairing and comparing elements (classes and methods) between consecutive transactions, using element attributes such as name, parameters, return type, visibility, etc. The method presents good empirical results, but can not detect refactorings when more than one change has been applied to a program element. This work also shows an interesting technique of change visualisation that helps in software comprehension.

As already said, our approach allows to fine-tune the searching rules to maximise the searching capability. We can adjust a rule so it can be selected even if the effects of a refactoring are overridden by another one making the first disappear from the refactored system graph.

Dig *et al.* [DCMJ06] have developed an effective approach which has been proved to offer good empirical results. Their technique of shingles computing is oriented to find refactorings that affect software APIs and behaves well at that level of detail. The limitation suffered by this method is that it can not deal with changes performed within method bodies.

The advantage of our approach relies on its graph transformation basis. Our graph transformation approach is based on a structural representation of the source code which can be as detailed as needed in order to support refactorings at any detail level.

7.2 Known problems and limitations

Our approach presents a technique that aids to check the functional equivalence between two versions of a software system. A parsing analysis having an affirmative result will reveal behaviour preservation between two versions of a system and will generate a refactoring sequence. The main limitation of our approach arrives when analysing two versions which are not functionally equivalent, because the state space we search is probably not finite. Our searching algorithm is only partially correct. An affirmative result will undoubtedly generate a valid refactoring sequence, but it is not complete because termination can not be guaranteed. Our parsing grammar does not belong to a graph grammar category with termination condition, such as layered graph grammars [EEL⁺05]. So, it is not guaranteed that our searching method terminates. We believe that this can be overcome if we formulate the refactoring searching rules in a way they could restrict the search space to a finite state space.

In spite of the parsing support provided by AGG, this tool lacks some key features in order to be able to fully specify any refactoring searching rule. The rules that we have presented are sufficient to show the validity of our method, but it is quite obvious that they did not fully conform

to “real” refactoring operations implemented in a development tool. In rule *removeMethod* (see [Figure 4](#)), e.g. a method is specified just by its name, and its context is defined just by its container class (which is also identified just by its name). This problem appears because the AGG graph language allows to specify only one context per rule. We need to be able to specify a set of contexts within a single rule, and more expressiveness is needed for that. This can be achieved using, for example, path expressions as they are supported on the PROGRES graph transformation tool [[Mün99](#)].

It has also been difficult to represent some kind of rules in AGG, those with an iterative nature, which take an undetermined number of steps to be executed. The transformation control that AGG has been supporting was based in rule layering and this is not sufficient to formulate that kind of rules. More complex execution control has been recently implemented in the last versions of AGG, that allows to gather a set of rules and to designate a main rule to fire them.

8 Results and future work

We have presented a technique based on graph transformation to find whether a refactoring sequence exists between two versions of a software system or not. Despite of showing some limitations, the sample implementation that has been reviewed can be seen as a proof of concept, and it offers very promising results.

The main limitation of our approach is that our searching method is only partially correct, and it could not terminate in certain cases. In order to solve this problem an exhaustive analysis of the state space must be done. We must determine if we can formulate the refactoring searching rules to restrict the search space to a finite state space.

Once we have proved the validity of our approach, our immediate objective is to implement refactoring searching rules to support more refactoring operations and to measure the scalability of our technique over industrial-size systems. This will include improving rule descriptions to take benefit of features in the newest versions of the AGG tool.

From the raw information the parser outputs we are able to identify the refactoring sequence it finds, but this is only valid for the purpose of testing our approach. There is a clear need to develop a front-end tool to show up the refactoring sequence in a more convenient way. We also believe that building tool is a fundamental step to demonstrate an approach. Thus this is planned to be provided as a plugin for the Eclipse Development Platform, which has a strong refactoring support. Up to date, we have already developed an initial Eclipse plugin [[MA06](#)] to automatically obtain a Java program graph representation and to show the parser results.

Bibliography

- [DCMJ06] D. Dig, C. Comertoglu, D. Marinov, R. Johnson. Automatic Detection of Refactorings in Evolving Components. In *ECOOP 2006 - Object-Oriented Programming; 20th European Conference, Nantes, France, July 2006, Proceedings*. 2006.
- [DDN00] S. Demeyer, S. Ducasse, O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA*. Pp. 166–177. 2000.

- [EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (eds.). *Handbook of Graph Grammars and Computing by Graph Transformations, Volume II: Applications, Languages and Tools*. Volume 2. World Scientific, 1999.
- [EEL⁺05] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, S. Varró-Gyapay. Termination Criteria for Model Transformation. In Cerioli (ed.), *FASE*. Lecture Notes in Computer Science 3442, pp. 49–63. Springer, 2005.
- [EJ05] N. V. Eetvelde, D. Janssens. Refactorings as Graph Transformations. Technical report, Universiteit Antwerpen, 2005.
- [ERT99] C. Ermel, M. Rudolf, G. Taentzer. *The AGG approach: language and environment*. Volume 2 in [EEKR99], chapter 14, pp. 551–603, 1999.
- [FBB⁺99] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, 1999.
- [GW05] C. Görg, P. Weißgerber. Detecting and Visualizing Refactorings from Software Archives. In *IWPC*. Pp. 205–214. 2005.
- [JDM03] D. Janssens, S. Demeyer, T. Mens. Case Study: Simulation of a LAN. *Electr. Notes Theor. Comput. Sci.* 72(4), 2003.
- [KK04] G. Kniesel, H. Koch. Static Composition of Refactorings. *Science of Computer Programming* 52(1-3):9–51, 2004. Special issue on Program Transformation, edited by Ralf Lämmel, ISSN: 0167-6423, <http://dx.doi.org/10.1016/j.scico.2004.03.002>.
- [MA06] B. Martín Arranz. Conversor de Java a grafos AGG para Eclipse. Master's thesis, Escuela Técnica Superior de Ingeniería Informática, Universidad de Valladolid, September 2006.
- [MT04] T. Mens, T. Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* 30(2):126–139, 2004.
- [Mün99] M. Münch. PROgrammed Graph REwriting System PROGRES. In Nagl et al. (eds.), *AGTIVE*. Lecture Notes in Computer Science 1779, pp. 441–448. Springer, 1999.
- [MVDJ05] T. Mens, N. Van Eetvelde, S. Demeyer, D. Janssens. Formalizing refactorings with graph transformations. *Journal on Software Maintenance and Evolution: Research and Practice* 17(4):247–276, July/August 2005.
- [Opd92] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992. also Technical Report UIUCDCS-R-92-1759.
- [Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations, Volume I: Foundations*. Volume 1. World Scientific, 1997.
- [RS97] J. Rekers, A. Schürr. Defining and Parsing Visual Languages with Layered Graph Grammars. *J. Vis. Lang. Comput.* 8(1):27–55, 1997.