Electronic Notes in Theoretical Computer Science 74 No. 4 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume74.html 12 pages

Coordinated Distributed Diagram Transformation for Software Evolution ⋆

Paolo Bottoni [a] Francesco Parisi-Presicce [a] Gabriele Taentzer [b]

[a] *Università di Roma "La Sapienza - Italy*

[b] *Technische Universität Berlin - Germany*

Abstract

We present an approach to maintaining consistency between code and specification consistent during refactoring, where the specification consists of several diagrams from different UML diagram families. Distributed graph transformations are applied in a coordinated way over the different diagrams. Refactoring itself is modelled as a graph transformation over a flowgraph representing the code, while the UML diagrams are taken to represent different views of the code.

1 Introduction

Refactoring has been a common practice of programmers ever since the possibility of writing subprograms arose. Now, it has become a central practice of what is called *extreme programming* [2] and has undergone a systematisation work, as witnessed for instance in Fowler's book [6]. Although refactoring can apply to any programming paradigm, it becomes particularly significant in object-oriented languages, and largely benefits from the synergy with research on design patterns, as this makes new possibilities for refactoring apparent.

Refactoring is usually meant to preserve a program behaviour while improving its reusability and flexibility, but does so by relying on a purely functional view of behaviour, as expressed by a mapping between inputs and outputs. Actually, refactoring can have several consequences if one considers the computing process, as expressed for instance by the sequence of method calls or the state changes of an object or an activity. On the other hand, as refactoring is usually performed at the source code level, it becomes difficult to maintain consistency between the code and its specification,, as expressed for instance through UML diagrams, which refers to the original version of the code. Two strategies can then be adopted: either recover the specification

⋆ Partially supported by the EC under Research and Training Network SeGraVis.

after each change (or any chosen set of changes), using for instance tools such as Fujaba [12]; or define the effects of a refactoring on the different parts of the specification. This last option is easily realised on structural specifications, as transformations on such diagrams are notationally equivalent to the lexical transformations on the source code. It is not so easy to define these effects on behavioural specifications.

This work discusses an approach to the problem of maintaining consistency between source code and both structural and behavioural diagrams, grounded in the formal framework of graph transformations. In particular, UML specification diagrams are represented as graphs, and an abstract representation of the source code is also expressed through suitable attributed graph structures. The UML diagrams as well as the code are seen as different specification views of a software system, so that consistency management between the views and the code is modelled as a graph transformation distributed on several graphs at once. Complex refactorings, as well as the checking of complex preconditions are broken into collections of distributed transformations whose application is managed by control expressions in suitable transformation units.

Paper outline. After reviewing some approaches to refactoring and to the more general problem of specifying software evolution through graph rewriting in Section2, we provide the background notions on distributed graph transformation in Section 3. In Section 4, after reformulating the problem of maintaining consistency between different forms of specification and with code as the specification of suitable distributed graph transformations, we illustrate our approach by means of two refactorings. Finally, conclusions are given in Section 5.

2 Related Work

Formal methods have been applied towards a clear definition of the conditions under which refactoring takes place, from Opdyke's seminal thesis [14], where preconditions for behaviour preservation are analysed, to Robert's thesis, where the effect of refactoring is formalised in terms of postconditions [15], so that composite refactorings can be discussed, in which the preconditions for a refactoring are guaranteed by the postconditions for a previous one.

Recent work in the UML community has expressed the effects of refactoring on UML diagrams, mainly class or state diagrams [16]. Similarly, Mens' work has studied how such transformations can be expressed in terms of graph transformations, by mapping diagrams onto type graphs [10]. Such type graphs are equivalent to abstract syntax class diagrams as found in the metamodel for UML. The work by Mens *et al.*, documented in [11], exploits several techniques that we also use in this paper, such as control expressions, negative conditions, and parameterised rules. However, all these studies focus on the effect of refactorings on single types of diagrams, and do not investigate, for example, the coordination of a change in a class diagram, with that in a sequence or state diagram.

In this paper we try to attack this problem, by showing that some types of refactorings require modifications in several diagrams at once, and proposing a constrained way of rewriting different graphs, as the model for a full formalisation of the refactoring process. To this end, we propose an approach derived from distributed graph transformation, which is grounded in the double pushout approach to graph transformation. Indeed, this is classically based on a hierarchical view of distributed systems, where high-level "network" graphs define the overall architecture of a distributed system, while low-level "specification" ones refer to the specific implementation of local systems [17]. Such an approach has also been applied to the specification of ViewPoints, a framework to describe complex systems where different views and plans have to be coordinated [7]. In the ViewPoint approach, inconsistencies between different views can be tolerated [8], while in the approach proposed here different graphs have to be modified in a coordinated way so that overall consistency is always maintained.

3 The Formal Background

We work with distributed graphs where all nodes and edges can be typed and attributed. Distributed rule application follows the double-pushout approach to graph transformation as described in [17], exploiting rules with negative application conditions. For further control on distributed transformations, transformation units are used. In [4], they are defined based on a general approach. Here, we use transformation units on distributed graph transformation. This combination provides us with a global control on structured graph manipulations.

3.1 Distributed Graph Transformation

Distributed graph transformation is a form of structured graph transformation on two abstraction levels: the network and the object level. The network level contains the description of the system's architecture by a network graph, and its dynamic reconfiguration at runtime by network rules. At the object level, graph transformation is used to manipulate local object structures. To describe a synchronized activity on distributed object structures a combination of graph transformations on both levels is needed. A *distributed graph* consists of a network graph where each network node is refined by a local object graph. Network edges are refined by graph morphisms on local object graphs which describes how the object graphs are interconnected. Analogously, a *distributed graph morphism* m is defined by a network morphism n – which is a normal graph morphism – together with a set S of local object morphisms being graph morphisms on local object graphs. Each node mapping in network morphism n is refined by a graph morphism of S on the corresponding local graphs. Each mapping of network edges guarantees a compatibility between the corresponding local object morphisms.

Following the double-pushout approach, a *distributed graph rule* $p : L \stackrel{l}{\leftarrow} I \stackrel{r}{\rightarrow} R$, is defined by the two distributed graph morphisms l and r. When applied to a *distributed host graph* G,it transforms G into a *target graph* G' if and only if an *injective match* $m : L \rightarrow G$ is found which is a distributed graph morphism again. A comatch $m' : R \rightarrow G'$ specifies the embedding of R in the target graph. The elements in the *interface graph* I must be preserved in the transformation. Due to space reasons, in the presentation of the rules below, the graph I is implicit as the intersection of L and R A rule may also contain a set of *negative application conditions (NAC)* to express that something *must not* exist for a rule to be applicable. The negative condition is defined by a set of distributed graph morphisms $\{N_i \stackrel{n_i}{\rightarrow} L\}$ with $i \in I$ and can of course refer to values of attributes [17]. For the rule to be applicable, no graph present in the set of NACs must be matched in the host graph .

In the figures below we will use schemes of graph productions rather than actual rules. In these schemes *set nodes* are also employed, which can be mapped to any number of nodes in the host graph, including zero. The matching of a set node is in any case exhaustive of all the nodes in the host graph satisfying the condition indicated by the rule.

3.2 Transformation Units

Transformation units are used to further control rule application, with the control condition specified by expressions over rules [4]. The concept of transformation units is defined independently from any given approach to graph transformation. Actually, each transformation unit refers to a certain graph transformation approach $\mathcal{A}$ consisting of a class of graphs $\mathcal{G}$, a class of rules $\mathcal{R}$, a rule application operator $\Longrightarrow$ yielding a binary relation on graphs for every rule of $\mathcal{R}$, a class $\mathcal{E}$ of graph class expressions, and a class $\mathcal{C}$ of control conditions. A *transformation unit* consists of an initial and terminal graph class expression, defining which graphs serve as valid input and output graphs. Moreover, a set of rules and a set of references to transformation units used are contained, together with a control condition over $\mathcal{C}$ describing the way the rules an other transformation units have to be applied. Typically, $\mathcal{C}$ contains expressions on sequential application of rules and units as well conditions or application loops, e.g. applying a rule as long as possible. An explicit presentation of typical control expressions is given in [9].

In the following, we apply transformation units on distributed graph transformation. I.e. we use the following underlying graph transformation approach: $\mathcal{G}$ is the class of distributed graphs, $\mathcal{R}$ the class of distributed rules, and $\Longrightarrow$ the DPO way of rule application, as defined in [17]. In [9], class $\mathcal{C}$ is described. Class $\mathcal{E}$ is not needed in the following. It can trivially be left empty meaning that no special initial and terminal graph classes can be specified.

In this paper, we will use two additional mechanisms to specify control on rule application. The first is the possibility to label edges with *path expressions* summarising possible concatenations of edges, or alternative paths between elements. The check of the existence of such paths could be realised by using suitable transformation units defining how to check the existence of a sequence of links, so that path expressions are here intended as a notational shortcut. The second mechanism is the use of parameterised rules, so that actual rules are obtained by instantiating the parameters to the context at hand.

4 Refactoring

In this section, we analyse some examples of refactoring which involve transformations in more than one UML diagram. Following [15] refactorings are expressed by pre- and post-conditions. The typical interaction with a refactoring tool can be modelled by the following list of events.

(i) The user selects a segment of code.

(ii) The user selects one from a list of available (possible composite) refactorings.

(iii) The tool checks the preconditions for refactoring.

(iv) If the preconditions are satisfied, refactoring takes place, with effects as described in the postconditions. Otherwise a message is issued to the user.

The choice to perform a specific refactoring is usually left to the software designer. However, complex refactorings are resolved as sequences of individual refactoring steps. In what follows we consider the effect of complex refactorings as specified by individual graph transformation rules, possibly distributed over different diagram graphs, and we do not model the processes which lead to the choice of any particular refactoring. Actually, the effect of refactoring on different diagrams can be expressed through schemes of graph rewriting rules, which have to be instantiated with the proper names for, say, classes and methods, as indicated in the code transformation.

Preconditions are usually checked on the textual code, but involve the analysis of properties which are properly structural, such as the visibility of variables in specific portions of code, or the existence of calls to some methods.

4.1 Graph Representation of Diagrams and Code

In the line of [10], we consider type graphs as defining the abstract syntax for concrete visual languages, such as those defined in UML. In particular, we refer to UML class, sequence, and state diagrams, with the type graphs defined in coherence with the metamodel definition of such languages in the official UML documentation [13].

We also assume the possibility of representing the source code in the form

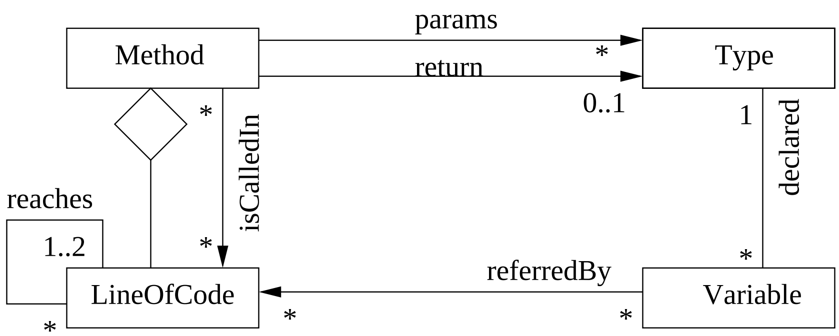

Fig. 1. The type graph for code representation

of a flowgraph for a method, as is typical in compiler construction [1]. This is a directed graph where nodes are lines of code and an edge exists between two nodes if a program execution can make the line represented by the first node to be followed by the line represented by the second node. Moreover, each *line* node is attached to a set of nodes describing the variables referred to (for definition or usage) in the line and the methods, if any, called in the line. Finally, a set of *parameter* nodes describing the types of the arguments to a method and a *return* node, describing the type of the result, are also considered to be present. Figure 1 describes the resulting type graph. Such a type graph is simpler than the one in [11] for the representation of relations among software entities, which also considers inheritance among classes, and the presence of subexpressions in method bodies. Here, we deal with inheritance in the UML class diagram and, since we are not interested in representing the whole body of a method we only keep trace of references to variables and methods and not of complete expressions. On the other hand, we maintain a representation of code lines and of reachability relations among them, which allows us to have a notion of *block* that will be used in Section 4.2.

A similar representation of diagrams as graphs can be used for sequence diagrams, following the ideas in [5], and based on the UML metamodel. To simplify matters, we describe transformations directly on the concrete syntax of sequence diagrams.

Distributed graphs are well suited to describe interrelationships between diagrams and code fragments. Following the approach of [7], a network graph (see Figure 2) describes the specification of the whole software system at some stage of the evolution process. A network node is either associated with one local object graph representing a UML diagram or the code flowgraph (we call such nodes *diagram* nodes), or it is an *interface* node. For each pair of diagram nodes, a common interface node exists. The refinement of the interface nodes at the local level shows the common graph parts of two diagrams in the current state. Network edges connect diagram nodes and interface nodes and are refined at the local level by defining how common interface parts are embedded in diagrams. Hence, an interface graph is related to its parent graphs by two graph embeddings (being injective graph morphisms).

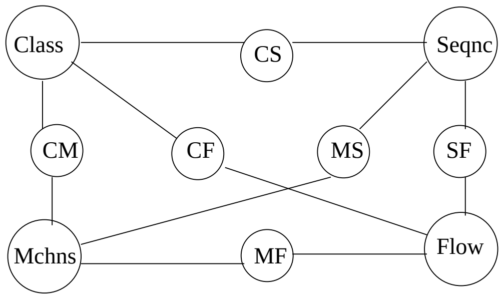

Fig. 2. The type graph for network graphs describing software system specifications

4.2 Code Extraction

As a first example, consider the `extract_code_as_method` refactoring by which a segment of code is isolated, given a name, and replaced in the original code by a call to the newly formed method.

The preconditions for such a refactoring are that the code to be extracted can be seen as a *block*, i.e. it must have only one entry point and one point of exit, although it does not have to be a maximal block, i.e. it can be immersed in a fragment of code which has itself the properties of being a block. Moreover, the name to be given to the method must not exist in the class hierarchy to which the affected class belongs. The post-conditions for this refactoring assert that a new method is created containing the extracted code, that such a method receives as parameters all the variables which are not visible to the class and which are used in the code (they had to be passed or be local to the original method), and that the code is replaced in the original method by a proper call to the new method.

Figure 3 describes, in the form of a rule scheme, the effect of such refactoring on the graph representing the code. In this figure, a set node is used to indicate the lines of code in the original version of the method which are affected by the transformation. The rest of the method is left untouched. The programmer must provide a specific instantiation of this scheme by listing the lines to be moved. The other set nodes in Figure 3 indicate the variables and methods referred to by the moved lines. Labels are here and in what follows used as variables to identify sets and nodes. The two negative application conditions express the requirement that the moved lines must constitute a block by stating that there must not be no o line in the code that can reach or be reached from the moved lines following two distinct paths.

In Figure 3 the negative application condition is complemented by a textual condition. An operational semantics for OCL conditions exploiting transformation units has been given in [3] and can be integrated in the current approach. In general, complex preconditions can be expressed as rules with identical left- and right- sides. A transformation unit can then define a control mechanism such that the actual transformation occurs only if all positive precondition is satisfied and no negative precondition is satisfied.

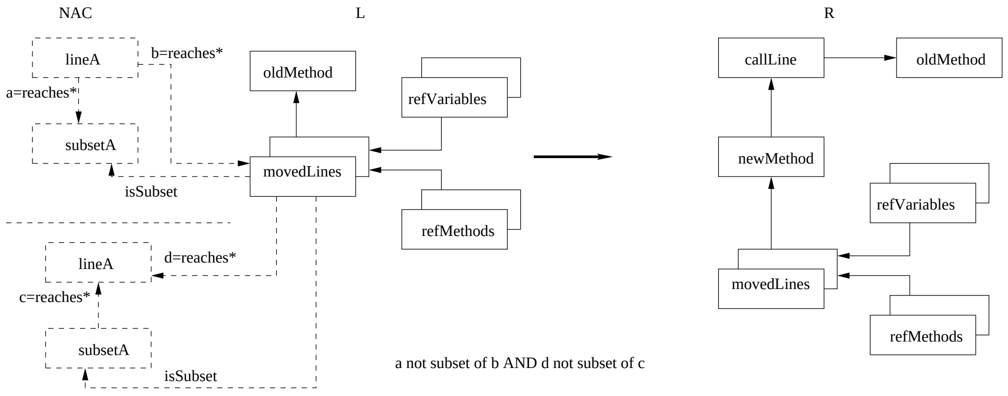

Fig. 3. The transformation scheme for code modification in the `extract_code_as_method` refactoring

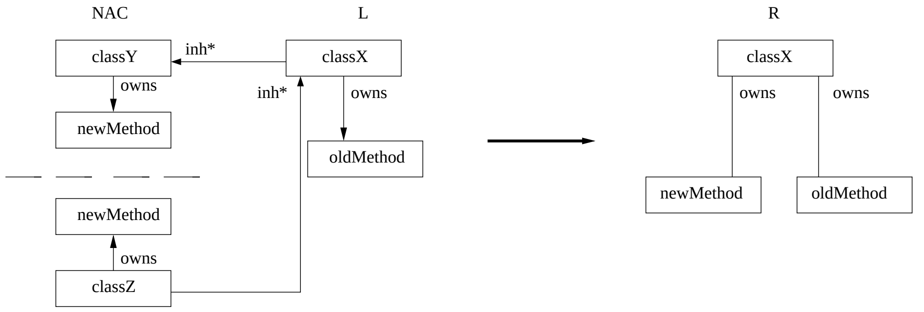

Fig. 4. The transformation scheme for class diagram modification in the `extract_code_as_method` refactoring.

Figure 4 describes the effect of this refactoring on class diagrams. At the structural level only the existence of a new method (with a fresh name) in the class can be shown. The effects on the referred variables and the existence of a call for this method, observable in the textual description, are not reflected in the structural diagram. The two negative application conditions state that a method with the same signature as the new one must not appear in any class higher or lower in the hierarchy than the modified class. These conditions make use of path expressions to indicate the transitive closure of the *inheritance* relation.

The newly created call is observed at the behavioural level, as shown in Figure 5. Referring now to concrete sequence diagrams, and not to their abstract representation as graphs, we use two oblique lines as a notational shortcut to indicate that anything can precede the activation of the old method. This can be expressed in the abstract syntax graph by a path expression labelled with *first_e next_e** from the node representing the receiving instance to the node representing the action.

Finally, in case the execution of the original code is represented by some state or process in an activity or a state diagram, such a state or process should be replaced by states and state transitions which distinguish between states

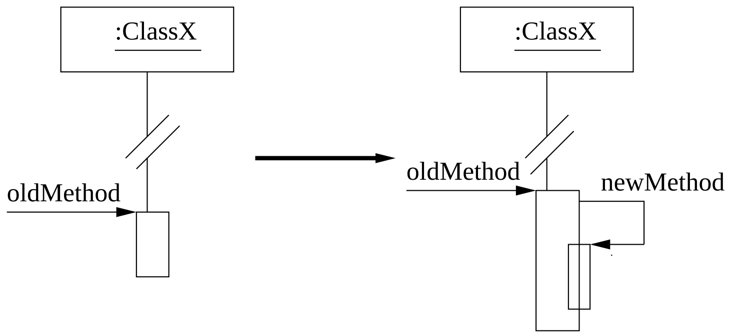

Fig. 5. The rule scheme for sequence diagram modification in the extract code as method refactoring.

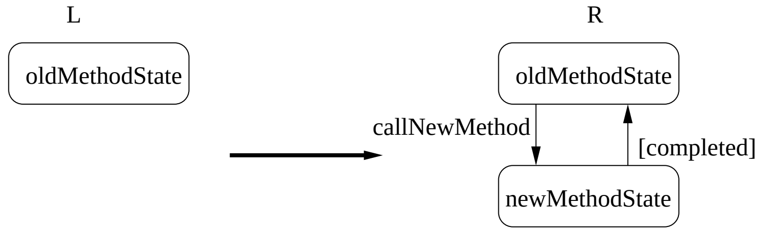

Fig. 6. The rule scheme for state diagram transformation in the extract_code_as_method refactoring

prior to, coinciding with, or subsequent to the execution of the extracted code. Figure 6 describes such a transformation. Incidentally, this suggests that more stringent preconditions could be applied, to the effect of checking if a state depicts a situation in which parts of code are executed which are not subsets or supersets of the extracted code. If this is the case, there would be at least two states such that the system would transit from one to the other during the execution of the extracted code. In this case, the insertion of an intermediate state corresponding to such execution would be more significant.

All the transformations above have to be applied to maintain consistency of diagrams and code. The network level transformations will simply rewrite nodes into themselves. With each such rewriting a transformation of the associated local graph occurs. The transformation units state that nodes have to be rewritten as long as possible. This amounts to rewriting diagram nodes at the network level as long as it is possible to transform the associated local graph if affected by the refactoring, and applying transformations on one local graphs as long as possible as well. For example, in the modification of sequence diagrams, an activation of the old method can occur several times on the same lifeline. Hence, the instantiation of the relevant transformation scheme must be applied as long as possible in any sequence diagram. If necessary, rules can be modified to tag elements to which a transformation has already been applied and negative application conditions can be used to avoid applying a rule twice to the same element. The transformation unit can then be completed by removing the tags.

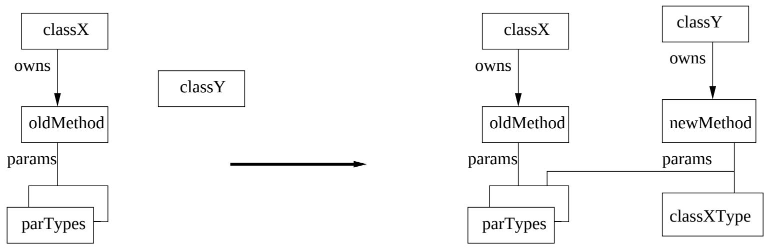

Fig. 7. The rule scheme for class diagram transformations for the move_method refactoring.

It is to be noted that preconditions for the refactoring were distributed among different graphs, in this case the one representing the class diagram and the one representing the code. As the whole process is regulated by control expressions in a transformation unit, one can condition the application of the refactoring to the satisfaction of preconditions in all the relevant graphs.

4.3 Method movement

The code of a method can be moved from its defining class to a different class in which it takes a new name. The original method is replaced with a forwarding method that simply calls the new method in the destination class. Since the original method could refer to members of its original class, the signature for the method is enriched with a reference to the original class, as indicated by the node labelled with `ClassXType` in Figure 7. The case where the method has to be moved to the superclass would be expressed by a rule scheme similar to the one in Figure 7, but which would require the existence of an inheritance relation between nodes labelled `ClassX` and `ClassY`.

Besides class diagrams, sequence diagrams have to be modified as well, according to the transformation scheme depicted in Figure 8. Indeed, while the call to the forwarding method can be said not to modify the behaviour of the class, it has to be reflected in this diagram to prevent subsequent refinements of this diagram from violating the correct sequence of calls.

In Figures 7 and 8 we have exemplified the case where the method is an instance method. It is also possible to consider the case for static methods with the obvious modifications.

Again, a transformation unit will be used to prescribe that all the network nodes of type *ClassDiagram* and *SequenceDiagram* whose local nodes contain references to the moved method will be affected by the transformation.

5 Conclusions

We have presented an approach to maintaining consistency between code and model diagrams in the presence of refactorings. Each refactoring is described through a set of coordinated graph transformation schemes which have to be

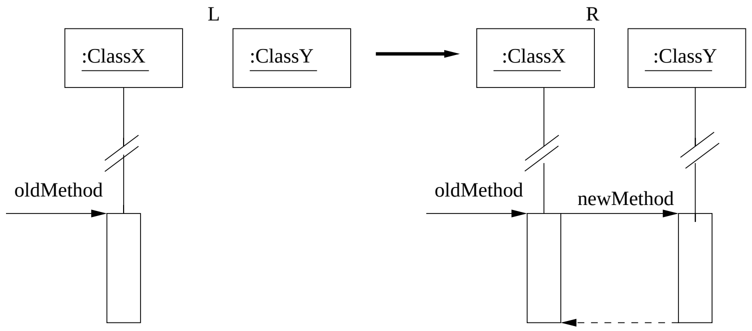

Fig. 8. The rule scheme for sequence diagram transformations for the move_method refactoring.

instantiated according to the specific code modification and applied to the diagrams affected by the change.

While this can be seen as a way to avoid reverse engineering to reconstruct the models from the modified code, the model can also be seen as a way to maintain consistency in diagrams through re-engineering steps, before proceeding to the actual code modification.

A more thorough study of existing refactorings, and experimentation on actual code, is needed to produce a library of distributed transformations which can be used in practical cases, based on existing tools for management of distributed graph transformations and transformation units.

Acknowledgements

Thanks are due to the anonymous referees of this paper for several comments and suggestions, and to Tom Mens for providing comments and making available to us a preprint of reference [11].

References

[1] Appel, A. W., “Modern Compiler Implementation in Java,” Cambridge University Press, 1998.

[2] Beck, K. and M. Fowler, “Planning Extreme Programming,” Addison Wesley, 2001.

[3] Bottoni, P., M.Koch, F. Parisi-Presicce and G.Taentzer, *Automatic consistency checking and visualization of ocl constraints*, in: A. Evans and S. Kent, editors, *UML 2000 - The Unified Modeling Language*, number 1939 in LNCS (2000), pp. 294–308.

[4] Drewes, F., P. Knirsch, H. J. Kreowski and S. Kuske, *Graph transformation modules and their composition*, in: M. Nagl, A. Schürr and M. Münch, editors, *Proc. AGTIVE 1999*, 2000, pp. 15–30.

[5] Engels, G., M. Andries and J. Rekers, *How to represent a visual specification*, in: K. Marriott and B. Meyer, editors, *Visual Language Theory*, Springer, 1998 pp. 245–260.

[6] Fowler, M., K. Beck, J. Brant, W. Opdyke and D. Roberts, "Refactoring: Improving the Design of Existing Code," Addison Wesley, 1999.

[7] Goedicke, M., B. Enders, T. Meyer and G. Taentzer, *Towards integration of multiple perspectives by distributed graph transformation*, in: M. Nagl, A. Schürr and M. Münch, editors, *Proc. AGTIVE 1999*, 2000, pp. 369–377.

[8] Goedicke, M., T. Meyer and G. Taentzer, *Viewpoint-oriented software development by distributed graph transformation: Towards a basis for living with inconsistencies*, in: *Proc. 4th IEEE International Symposium on Requirements Engineering*, 1999, pp. 92–99.

[9] Koch, M. and F. Parisi-Presicce, *Describing policies with graph constraints and rules*, in: *Proc. ICGT02*, to appear, 2002.

[10] Mens, T., *Conditional graph rewriting as a domain-independent formalism for software evolution*, in: M. Nagl, A. Schuerr and M. Muench, editors, *Applications of Graph Transformation with Industrial Relevance*, 1999, pp. 127–143.

[11] Mens, T., S. Demeyer and D. Janssens, *Formalising behaviour preserving program transformations*, in: *Proc. ICGT 2002* (2002), to appear.

[12] Niere, J., J. Wadsack and A. Zündorf, *Recovering UML Diagrams from Java Code using Patterns*, in: J. Jahnke and C. Ryan, editors, *Proc. of the* 2^{nd} *Workshop on Soft Computing Applied to Software Engineering, Enschede, The Netherlands* (2001).

[13] OMG, *UML specification version 1.4*, http://www.omg.org/technology/documents/ formal/uml.htm (2001).

[14] Opdyke, W. F., "Refactoring Object-Oriented Frameworks," Ph.D. thesis, University of Illinois at Urbana-Champaign (1992).

[15] Roberts, D. B., "Practical Analysis for Refactoring," Ph.D. thesis, University of Illinois (1999).

[16] Sunyé, G., D. Pollet, Y. L. Traon and J.-M. Jézéquel, *Refactoring UML models*, in: M. Gogolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference*, LNCS **2185** (2001), pp. 134–148.

[17] Taentzer, G., I. Fischer, M. Koch and V. Volle, *Visual Design of Distributed Systems by Graph Transformation*, in: H. Ehrig, H.-J. Kreowski, U. Montanari and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency, Parallelism, and Distribution* (1999), pp. 269–340.