Formalizing Software Refactoring in the Distributed Environment by aedNLC Graph Grammar

Leszek Kotulski, Adrian Nowak

Institute of Computer Science, Jagiellonian University
Nawojki 11, 30-072 Kraków, Poland
kotulski@agh.edu.pl, nowaka@ii.uj.edu.pl

Abstract. Being a commonly used technique to enrich the software structure, refactoring – as well as any software changes performed every day – still lacks a good formal definition. Especially in the distributed environment there is a great need for a better mechanism allowing to avoid conflicts and properly merge the changes introduced by different developers. In this paper we continue our project of a core of distributed environment based on graph repository, which helped us to defeat and significantly decrease problems of refactoring conflicts. We focus on technical aspects of the environment and present precise description of the refactorings with the help of aedNLC graph grammar and graph transformation mechanisms. We also discuss some other properties of the graph repository including its abilities to store dynamic software description. Presented approach is based on UML notation, however it could be easily extended for any object-oriented language. The graph repository concept alone could lead to a model of a modern integrated software development environment.

1 Introduction

Modifying and maintaining existing software has become an important part of the job of software developers. Any changes made to the software (code or model) should contribute to this software evolution and maturity. Some operations might change the behavior of the software while others just modify the structure. These changes which improve object-oriented software while preserving its behavior are well known as refactorings [1, 2]. When applied properly, refactorings help in many ways to improve not only the software itself [1] but also the whole process of software development and maintenance.

Nowadays there exists a number of tools to support such operations for many different programming languages, e.g. Refactoring Browser [3] for Smalltalk or Eclipse [4] for Java. A great deal of research in this area was conducted, but not much focused on formalizing the refactoring and its properties. Furthermore, the distributed environment, used naturally in case of any application developed by a team, was not taken into account. We try to deal with both these issues in the paper – formalize refactoring in the distributed environment – since it is essential to take into account all the factors which may have any influence on the refactoring operation.

As pointed in [5, 6, 7] many problems appear when two developers decide to make refactorings, in a parallel way, on the same software. As a very simple example, even an Encapsulate Variable and a Move Variable refactorings applied to the same variable by different developers cause a structural conflict, due to lost of the variable identification in the system [7].

As a formal framework, we use graph-based representation, utilizing Mens's notation [8]. However we extend the approach by introduction of the graph repository concept [7] and online graph transformations controlled by aedNLC graph grammar [9]. We compose refactorings from simple grammar productions, and provide atomicity by a special execution environment. This formalism allows us to describe and synchronize refactoring operations and also – under some conditions – exclude many conflicts.

In the next section of the paper we present a concept of graph representation of the software structures, where refactorings are represented as graph transformations. Section 3 overviews appearance of refactoring conflicts in a collaborative environment and some other common refactoring problems specific for team software development. Section 4 introduces formal definitions. Section 5 describes details of representing refactorings as aedNLC grammar productions. Section 6 shows how the refactoring conflicts can be automatically excluded using this approach. Some other properties of the repository are also discussed. Finally, section 7 concludes our work and proposes some future research.

2 Software as a graph

An idea of representing software as a graph is very reasonable and quite natural, hence commonly used in research [8, 10] and tools [4]. Compared with tree based representations it does not only allow to represent static relations between program elements but also dynamic relations such as method call, variable access and late polymorphism binding.

2.1 Metamodel

A graph representation of all allowed connections between potential software components as well as all necessary attributes is known as a metamodel. Formally it is also called a type graph [11, 8]. An example of a simplified metamodel for object-oriented programming language (or UML class diagram) was presented in [7] and is now extended to distinguish method definitions – following Mens [8] – see figure 1.

Graph nodes are labeled by: "Class" for nodes representing classes (or types), "MethodDef" for method definitions, "Method" for method signatures, "VariableDef" for variable definitions, "Variable" for variable signatures and "Parameter" for method parameters. The separation between the definition and the method or variable itself is crucial as we have to provide a possibility to introduce many definitions of a single component within a hierarchy (due to late binding and polymorphism). We use the UML notation, relying on the composition (depicted as filled rhombi) as the most suitable for representing strong inclusion between main and part components – further

called a <<member>> relation. Other relations are represented by attributed references. Multiplicities of the relations are written along edges as number, range, or an * (asterisk) in case of being not strictly defined. We do not interfere with methods bodies (as [8] by introducing an expression component) – this is another hierarchy level (after package and class levels) in hierarchical graph which we normally use as a full metamodel – not essential to present in this paper. The OCL[12] in turns allows us to express constraints to exclude illegal components and relations in the instance graph, that is for example exclude two methods or two variables with the same signature within a class.

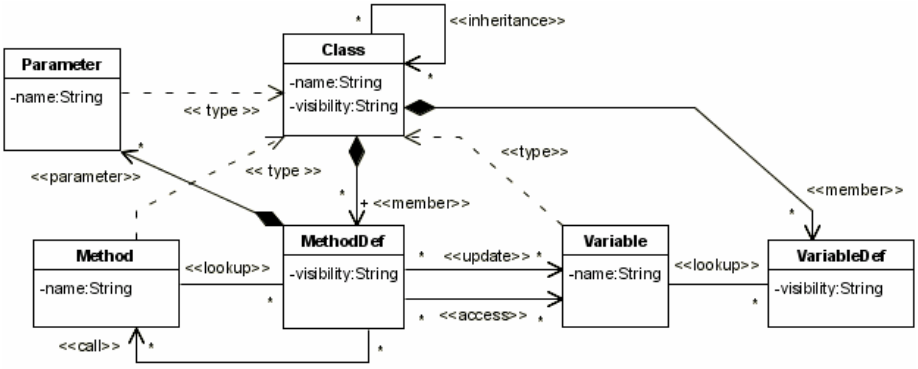

Fig. 1. Simple metamodel.

For better understanding of the metamodel let us take a look at its sample instance (Figure 2). Now we operate on particular components instances with given names and attribute values.

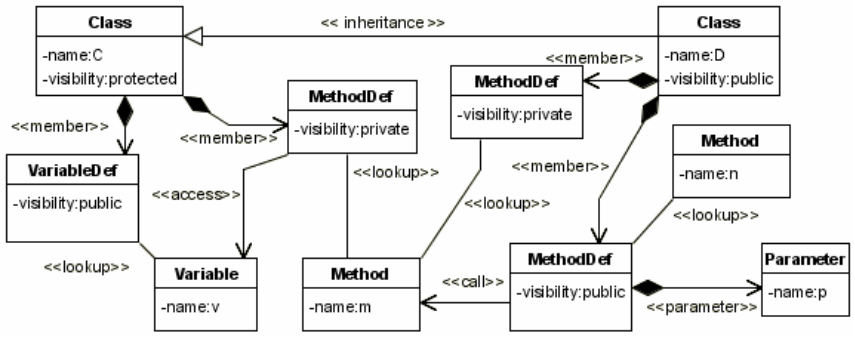

Fig. 2. Example of the metamodel instance.

The relations “m is a member method of class C” or “v is a member variable of the C” or “m calls n” can be expressed by a suitable node interconnection, presented appropriately by compositions and associations. A real example of Local Area Network with program code and method definitions was presented in [8].

2.2 Refactorings

We assume that any changes made to the software will have one-to-one mapping inside the software instance graph. The result of such changes might be very simple, like a single attribute change, but can be also quite complicated like change to thousands of expressions appearing throughout the project.

An example of a Move Variable refactoring representation is depicted in figure 3. There are two graphs describing the software fragment – before and right after the transformation (as a result of applying a single grammar production or a whole set of productions). Usually refactorings can be processed only when particular preconditions are met [2, 10] (here the Move Variable can be applied when the variable called "v" is not already defined within the class named "D").

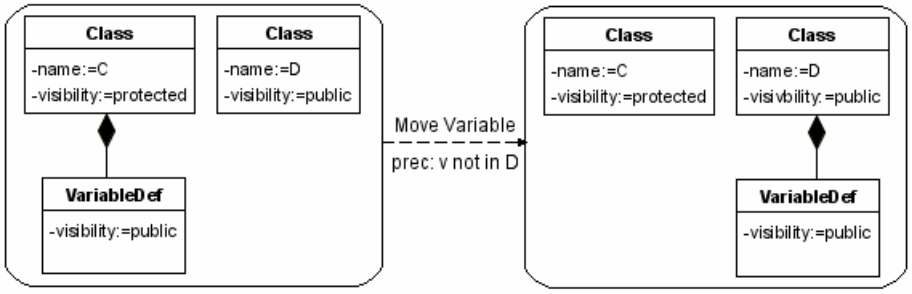

Fig. 3. Move Variable refactoring and its precondition - *v is not in D*.

3 Distributed environment

Software is usually developed by teams, which members are in different locations. These developers usually work in their own local environments on their own copies of the software, without any knowledge about modifications made by others – and this may cause the problems considered below.

3.1 Refactoring conflicts

The main problem of applying changes in collaborative environment are conflicts, usually detectable quite late, in the phase of software merging. These could appear as syntactic, structural or semantic conflicts [6] – the first two types will be further considered in the paper.

Let us get back to the example mentioned in the introduction. Suppose two developers decide to make changes involving the same variable from the same class. One of them performs the Encapsulate Variable refactoring (changing visibility and adding setter and getter methods) while another performs the Move Variable refactoring (from the class named "C" to another named "D"). When, probably after a sequence of other modifications, developers finally decide to share new versions with the rest

of the team, the second developer should encounter a problem - conflicts will appear in all places where the variable was accessed or updated. The issue is that when using any of currently existing commercial tools (e.g. the Eclipse [4] with external version control system like CVS) the source codes of all classes are sent as text. This way we are able to detect only very basic conflicts, and often there is no possibility to avoid them. If we just try to automate the merging, we may lose the result of one refactoring or get two similar variables [7]. In consequence, quite frequently a developer has to realize what was really modified and decide which operation should be accepted. In many cases it might be even necessary to redo some operations. Moreover, sometimes such decisions must be discussed with the rest of the developer team.

3.2 Graph repository

Number of conflict possibilities arise together with the number of changes being applied. If we are able to imagine a sequence of refactoring operations on the same piece of code or a group of such sequences, then it is easy to imagine that conflicts number might increase dramatically[1].

Further, when concerning large refactorings performed step by step even through few days (and so interacting with many other changes, no matter how carefully planed), it is easy to notice that a kind of long term control mechanism is necessary.

The mentioned mechanism should also support ordinary developers work, that is provide possibilities such as undo the changes (including refactorings) or history and version management.

Refactoring tools already utilize a full code description but currently all analysis is done in complete separation from the rest of the distributed environment. We suggest to utilize a graph repository, introduced by us in [7]. The internal graph describing current state of the maintained software will be modified by a graph transformation system [13] according to software changes. The term “graph repository” is used on purpose, to put the emphasis on concurrent access to the graph. We assume that the graph should give us a possibility of unique components identification. It is an instance of the metamodel and additionally may hold some technical attributes for better description and analysis of refactoring preconditions as well as performance issues.

4 Formal definitions

The solution presented in the paper is supported by an aedNLC graph grammar [9, 13, 14, 15], so the basic properties of this grammar should be outlined.

[1] Important issue here, not only applicable when considering distributed environment, is providing a possibility of operation grouping in order to get better performance e.g. by gathering preconditions or finding context once.

4.1 EDG Graph

The graph generated by grammar consists of nodes and directed edges; both nodes and edges are labeled (its general properties are established e.g. a can node represent class or method) and attributed (the individual components properties are defined e.g. class name).

An attributed directed node- and edge-labeled graph, EDG graph, over Σ and Γ is a quintuple $H = (V, D, \Sigma, \Gamma, \delta)$, where:

V – is a finite, non-empty set of nodes, to which unique indices are ascribed, defining the order within the set

Σ – is a set of attributed node labels

Γ – is a set of attributed edge labels

D – is a set of edges in the form of (v,μ,w) where $w, v \in V$ and $\mu \in \Gamma$

$\delta: V \rightarrow \Sigma$ – is a function, which labels the nodes.

For the metamodel presented in the section 2.1 we should have:

Σ = {“Class”, “Method”, “MethodDef”, “Variable”, “VariableDef”, “Parameter”}

Γ = {<inheritance>, <member>, <type>, <parameter>, <call>, <access>, <update>, <lookup>}

4.2 Graph transformation

Any graph grammar production P is represented by a left-hand (L) and a right-hand side (R) graphs and an embedding transformation E, thus P=(L, R, E). Modification of the graph H, describing a current state of the system, is made by applying graph grammar production. First, a subgraph of the H, that is homomorphic (by a homomorfism h) to L is localized (so the subgraph is equal to h(L)), next h(L) is removed from H and the right-hand side graph R is placed instead; the embedding transformation E specifies a way in which the nodes of the graphs R and H-h(L) should be associated by edges. The left-hand and right-hand sides of productions could be easy presented graphically, but the embedding transformation is rather described using a special notation.

The equation $E(\gamma, in, v) = \{(Q, (X, \pi), \mu, in)\}$ is interpreted as follows: every edge labeled by “γ” and coming into (thus “in” or “out” will be used to show a direction) the node h(v) within the graph H should be replaced by an edge connecting a node (w) labeled by “Q” from the right–hand side graph R with a node labeled by “X” from the rest of the graph (H-h(L)) on condition that the formula π is fulfilled (for the nodes belonging to this edge). Newly introduced edge will be labeled by “μ” and will come into the node w. In order to simplify the notation, we assume that the dangling edges (not described by E) will be connected to a node (inside right-hand side graph R) with the nodes designated as follows:

- if the removed node ($u \in L$) appears in the right-hand side of production (i.e. exist node with the same index as u) the edge will be connected to this node
- otherwise the edge will be connected to the least node inside R with the same label as u.

The above rule we will call COPY_REST embedding transformation rule.

The homomorphism used have to be unambiguously defined, so when the left-hand-side graph of the considered production consists with a single node v_L then we assume the homomorphism is defined as a unique homomorphism from the node v_L to the node for which the production is applied that is v. Note that, in such a case the embedding transformation is equivalent to the one introduced in [9, 16].

Application of productions should be done in context of the repository graph H by a special Derivation Control Environment (DCE). The proposition of DCE usage is based on previous solutions that were utilized to control the software allocation process in a distributed system [14] and to describe a behavior of the mobile agent systems [13].

The DCE services developers and system requests; when a request appears either a waiting control thread is activated or a new thread of control is created for starting point.

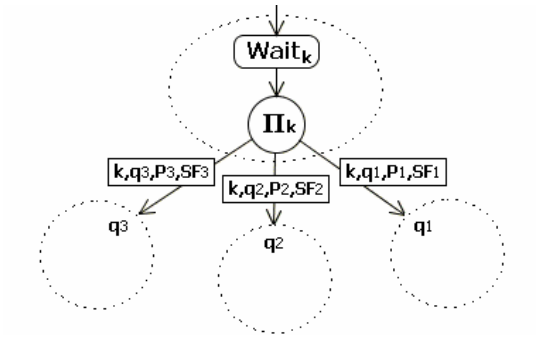

Fig. 4. Derivation control environment.

The DCE can be interpreted as a diagram (see Figure 4) connecting control points (the dotted circles) inside which a synchronizing functions $Wait_k$ (if exist) and a selector Π_k are sequentially evaluated. The synchronizing function $Wait_k$ is evaluated basing on the current graph value and the queue of requests (that have to be sent to the DCE). If this evaluation fails then the control point activity will be delayed until the environment changes (i.e. a new request appears), otherwise the selector Π_k is evaluated (also basing on the same elements) and designates the proper transition. The transition not only moves the activity to the next control point but also both a semantic function and the graph grammar production (pointed out as edge attributes – SF_i and P_i) are performed. The semantics function (associated with the transition):

- adds new request to the order queue (requesting some actions from refactoring system),
- removes the request, which is serviced from the queue,
- evaluates parameters of the right-hand side graph of the production.

When the production P is applied to the current graph H a new graph H' is created in a way defined by the transformation rules of the graph grammar associated with this derivation control diagram.

Introduction of the concurrent threads of control simplifies the DCE description, however to assure proper data modifications we have to introduce a general synchronization rule: each thread of control has exclusive access to the data representing

graph H and to the requests queue in the period beginning from $Wait_k$ evaluation to the moment when a new graph H' is created.

5 Refactorings as aedNLC grammar productions

As described in the section 2.2 any refactoring corresponds to the graph repository transformation. To introduce such transformation we need to be able to apply an adequate grammar production. However, due to restrictions on the graph, graph grammar and performance issues, we will usually need several productions to define a single refactoring. For this we will further utilize the derivation control mechanism described in the previous section. In order to easily describe refactorings we will use parameterized productions – to locate nodes of the left-hand side L of production by graph indices and to avoid ambiguous definition of the homomorphism h. For simplicity, due to one-one mapping, we will also incorporate all information from "VariableDef" and "MethodDef" nodes into "Variable" and "Method" accordingly. Let us introduce productions for considered refactorings.

5.1 Move Variable

The MoveVariable refactoring should take effect not only in origin (C) and destination (D) classes but also in all places where the variable was updated or accessed (e.g. in Java by adding new imports or class prefixes). Fortunately this information is associated with VariableDef node, so embedding transformation consist only of the COPY_REST rule.

An adequate part of the derivation control diagram is presented in Figure 5. The condition Π_1 corresponds to the MoveVariable refactoring precondition that is "*v is not in D*". The Move variable production P_{mv} is the same as the transformation shown previously in Figure 3.

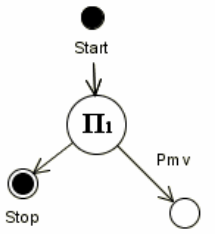

Fig. 5. A part of the appropriate DCE for MoveVariable(v, C, D).

5.2 Encapsulate Variable

The EncapsulateVariable should make the considered variable v a private, add get() and set() methods as well as introduce calls to these methods in all places where the variable was accessed or updated. We can provide three separated productions – two

for introducing the methods (Figure 6a, 6b) and one for changing the variable attribute (Figure 6c) – the sequence is controlled by dedicated DCE fragment (Figure 6d). The embedding transformations are, accordingly:

$$E_1(<access>, in, 2)) = \{(Method, (Method, true), <call>, out)\}$$

$$E_2(<update>, in, 2)) = \{(Method, (Method, true), <call>, out)\}$$

$$E_3 \Leftrightarrow COPY\_REST$$

The condition Π_1 is checking whether the methods set() and get() already exist in class "D", Π_2 is always true and Π_3 checks if the variable is public (in case it is not no production is applied).

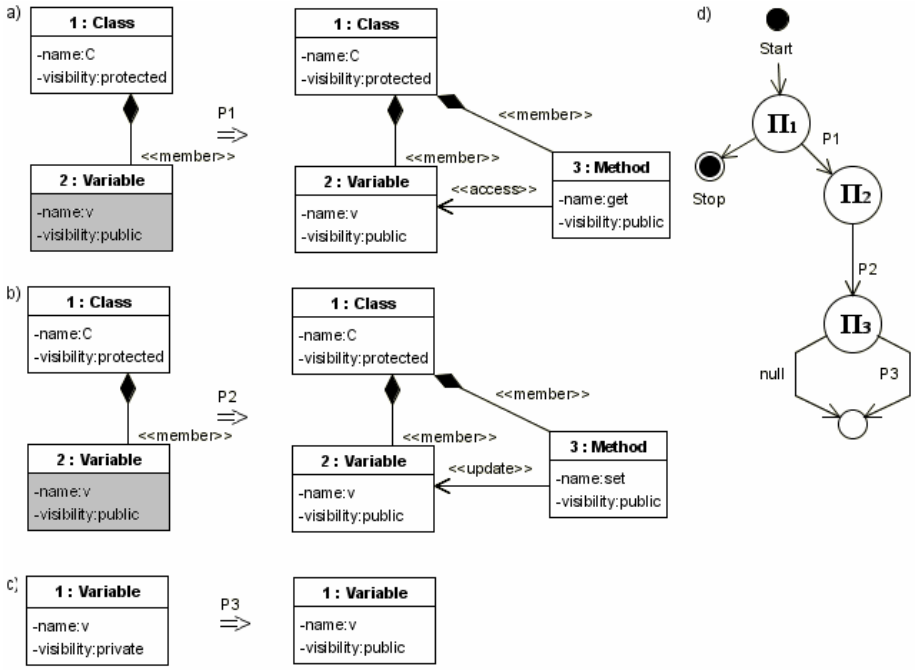

Fig. 6. EncapsulateVariable(v, C). a) production P1 - introduces get() method, b) production P2 - introduces set() method, c) production P3 - changes the variable visibility to private, d) DCE used to control application of these productions.

6 Graph repository idea revisited

Our approach supporting global refactoring consists of Graph Management System[7] (GMS) and a few Local Refactoring Environments (LRE). The GMS maintains the graph repository. Each of LRE performs the sequence of the following tasks:

- asking the GMS for searching a part of the graph associated with the refactoring operation (and possibly, synchronize it with the others) – find_context request
- performing the refactoring operation basing on the code,
- informing the GMS that the refactoring operation has to be performed – Move_variable, Encapsulate_variable requests
- updating the code maintained by LRE on GMS demands.

The first task performed by the GMS is a simple semantic action of searching for information in the graph. The second GMS task is associated with applying a graph grammar production (modifying the graph repository) and with execution of the semantic action (which requests all LREs to update the code maintained by nodes modified by this production).

Let us trace the above schema on the example (Figure 7a). Both developers are choosing some components to modify – the repository is looking for the right context. For unique identification of the components and better performance the GMS maintains unique indices for every node in the graph. An attempt to apply the Encapsulate Variable to "v" results on identifying nodes with indices 1 and 2,similarly an application of the Move Variable to move "v" from "C" to "D" results in identifying nodes with indices 1, 2 and 3.

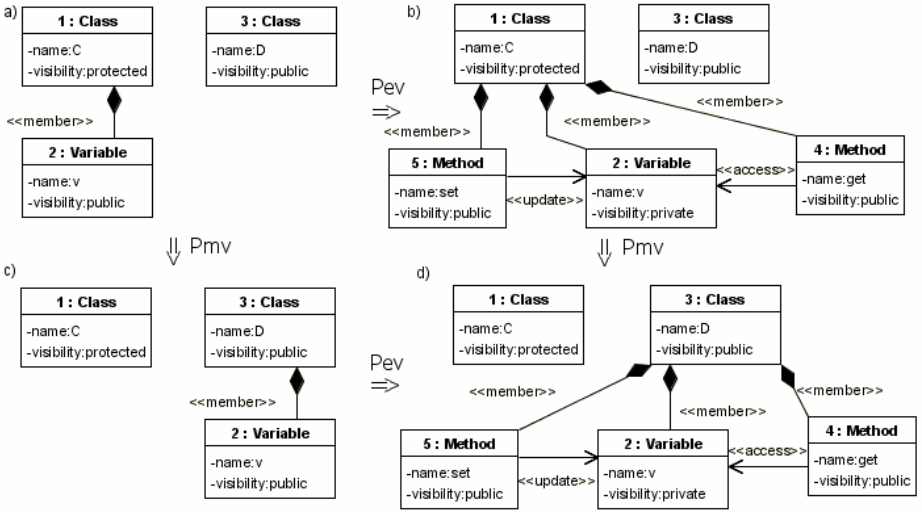

Fig. 7. Synchronization process using indexed EDG graphs.

After the context of these operations has been established we can read and share this part of graph in a parallel way. The Move variable and Encapsulate variable requests on the nodes can be serviced by DCE. However sometimes the request can not be served or can be only served partially with respect to preserving some of preconditions associated with the request. Let us note the important role of the semantic actions, that are tracing the graph modification (associated with the embedded transformation) and creating the refactoring orders for all LREs maintaining the source code associated with this graph modification.

Finally, in cooperation, both LREs and the GMS update the graphs to new instances. It is easy to notice that the order in which the developers make synchronization does not matter (Figure 7d). The key assumption for this method is that both productions have to preserve nodes indices.

It is easy to find out that the following conflicts pointed by [5] can be excluded in the same way:

- Rename Variable and Move Variable applied to the same variable,
- Rename Variable and Variable Encapsulate applied to the same variable,
- Rename Variable and Pull Up Variable applied to the same variable,
- Rename Method and Pull Up Method applied to the same method,
- Rename Method applied twice (separately by two developers) to the method within the same class,
- Rename Variable applied twice to a variable within the same class.

While dealing with other conflicts, when adding new components or deleting existing ones, the proposed method is not useful. To avoid such conflicts we have to exclude concurrent execution of the conflicted refactoring operations. This is a simple task from synchronization point of view (some operations can be delayed until global predicates, based on attributes, are fulfilled). This solution is still difficult to approve by developer teams. One of the developers should wait, however now the time of waiting is considerably decreased (only one refactoring operation should be completed instead of a full sequence). Moreover, introduction of the graph repository causes that developers are informed about conflict just in the time when it appears, while earlier they were informed after finishing of sharing a new software version.

7 Conclusions

In the paper we propose a solution dealing with refactoring conflicts based on [5] classification. Introduced graph repository concept, properly transformed (under control of graph grammars) is completely enough to defeat the kind of conflicts where the key problem was losing method or variable identification while merging the changes. The introduced environment allows us to solve these conflicts automatically.

In order to prove the theoretical value of the method the centralized service of the graph repository is sufficient, but in the practical solution it seems that the repository should be distributed (together with system source code). Fortunately, for the aedNLC graph grammar semi-parallel derivation mechanism over the distributed graph has been introduced [13]. Moreover, the parser of aedNLC graph grammar is based on

ETPL(k) graph grammar with O(n2) computational complexity [16] and the effectiveness is the most important issue in the system working online. Graph parsing will be useful when describing and allocating the nested distributed system [6], we can utilize it to exchange whole subgraphs in case of complex refactorings.

The graph repository could be further utilized by holding additional attributes of the software, also including dynamic parameters suitable to calculate metrics and using these to perform automate refactorings. Derivation control diagram is able to manage refactoring compositions to introduce patterns and should be extended to manage plans of large refactorings (under interactive control of a developer).

References

1. Fowler, M.: Refactoring: Improving the Design of Existing Programs. Addison-Wesley (1999)
2. Opdyke, W.F.: Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign (1992)
3. Roberts, D., Brant, J., Johnson, R.: A Refactoring Tool for Smalltalk, Theory and Practice of Object systems (1997) 253-263
4. Eclipse Foundation, http://www.eclipse.org/eclipse/, The Eclipse Project (2005)
5. Mens, T., Taentzer, G., Runge, O.: Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. Electronic Notes in Theoretical Computer Science, Vol. 127(3) (2005) 113-128
6. Mens, T.: A state-of-the-art survey on software merging, IEEE Transactions on Software Engineering 28(5) (2002) 449-462
7. Kotulski, L., Nowak, A.: Graph Repository As a Core of Environment for Distributed Software Restructuring and Refactoring, 24th IASTED International Conference on Applied Informatics, Insbruck (2006)
8. Mens, T., Eetvelde, N., Janssens, D., Demeyer, S.: Formalising Refactoring with Graph Transformations, Journal of Software Maintenance and Evolution (2004) 1001-1025
9. Flasiński, M., Kotulski, L.: On the Use of Graph Grammars for the Control of a Distributed Software Allocation, The Computer Journal, 35(1) (1992) 167-175
10. Roberts, D.: Practical Analysis for Refactoring, Ph.D. thesis, University of Illinois at Urbana-Champaign (1999)
11. Engels, G., Schurr, A.: Encapsulated Hierarchical Graphs, Graph Types and Meta Types, Electronic Notes in Theoretical Computer Science (1995) 2
12. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML, Addison-Weslay (1998)
13. Kotulski, L.: Parallel Allocation of the Distributed Software Using Node Label Controlled Graph Grammars, Krakow, Poland, Jagiellonian University, Inst. of Comp. Science (2003)
14. Kotulski, L.: Model systemu wspomagania generacji oprogramowania współbieżnego w środowisku rozproszonym za pomocą gramatyk grafowych (in Polish), Krakow, Poland, Jagiellonian University Press (2000)
15. Kotulski, L.: Graph representation of the nested software structure, Proc. 5th International Conference on Computational Science, Atlanta, GA (2005) 1008-1011
16. Flasiński M.: Power Properties of NLC Graph Grammars with a Polynomial Membership Problem, Theoretical Computer Science, 201(2) (1998) 189-231