Specifying Integrated Refactoring with Distributed Graph Transformations*

Paolo Bottoni[1], Francesco Parisi-Presicce[1,2], Gabriele Taentzer[3]
[1]University of Rome "La Sapienza",
[2]George Mason University, [3]Technical University of Berlin

Abstract. With refactoring, the internal structure of a software system changes to support subsequent reuse and maintenance, while preserving the system behavior. To maintain consistency between the code (represented as a flow graph) and the model (given by several UML diagrams of different kinds), we propose a framework based on distributed graphs. Each refactoring is specified as a set of distributed graph transformations, structured and organized into transformation units. This formalism could be used as the basis for important extensions to current refactoring tools.

1 Introduction

Refactoring is the process of changing a software system so as to preserve its observable behavior while improving its readability, reusability, and flexibility. Refactoring has its origins in some circles of the Smalltalk community, though its principles can be traced back to the idea of writing subprograms to avoid repetitious code fragments. It is now a central practice of *extreme programming* [BF01], and can be carried out in a systematic way, as witnessed for instance in [FBB+99]. Although refactoring techniques can be applied in the context of any programming paradigm, refactoring is particularly effective with object-oriented languages and is usefully combined with the notion of design pattern.

The input/output view of the behavior of the system is not intended to change with refactoring. However, the changes can have several consequences for the computing process, as expressed for instance by the sequence of method calls, or by state changes of an object or an activity. Since refactoring is usually performed at the source code level, it becomes difficult to maintain consistency between the code and its model, expressed for example with UML diagrams, usually fitting to the original version of the code.

Several tools have been developed to assist refactoring: some are packaged as stand-alone executables, while others have integrated refactorings into a development environment. Many tools refer directly and exclusively to specific languages: **C# Refactory** (http://www.xtreme-simplicity.net/) supports 12 refactorings on pieces of C# code, including extraction of methods, superclasses and interfaces, and renaming of type, member, parameter or local value; **CoreGuide6.0** (http://www.omnicore.com) is based on Java and provides refactorings such as extract method and move/rename class/package. **Xrefactory**

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

(http://www.xref-tech.com) manages refactoring in C and Java, including pushing down and pulling up of fields and methods, and insertion/deletion/shift/exchange of parameters, in addition to the usual method extraction and various renamings. None of these tools mentions diagrams or the effect that these refactorings have on other views of the system, including documentation.

Two strategies can be adopted to preserve consistency among the different views: either recover the specification after a chosen set of changes of the code, as in Fujaba [NWZ01], or define the effects of a refactoring on the different parts of the model. The latter option is quite easily realized on structural models, where transformations on such diagrams are notationally equivalent to the lexical transformation on the source code, less so on behavioral specifications.

The class diagram, referred to as 'the model', is considered in **objectiF** (http://www.microtool.de/objectiF) which, in addition to supporting a wide variety of languages, allows transformations both in the code and in the class model, with the changes propagated automatically to both views. No mention is made, however, of diagrams describing the behavioral aspects of the system. **Eclipse** (http://www.eclipse.org) integrates system-wide changes of code with several refactor actions (such as rename, move, push down, pull up, extract) into the Java Development Tools (JDT) that automatically manages refactoring. Class diagrams are implicitly refactored, too. Finally, **JRefactory** (http://jrefactory.sourceforge.net) supports 15 refactorings including pushing up/down methods/fields and extract method/interface. The only diagrams mentioned are class diagrams, which are reverse engineered from the .java files.

Class diagram editors do not extend changes to all related diagrams, limiting their "automation" to source code. Hence, direct intervention is needed to restore consistency among the different UML diagrams representing the same subsystem.

We discuss an approach to maintaining consistency between source code and both structural and behavioral diagrams, using the formal framework of graph transformations. In particular, the abstract syntax of UML diagrams is represented through graphs, and an abstract representation of the source code is expressed through suitable attributed graph structures. The UML diagrams and the code are seen as different views on a software system, so that the preservation of consistency between the views is accomplished by specifying refactoring as a graph transformation distributed on several graphs at once. Complex refactorings, as well as the checking of complex preconditions, are decomposed into collections of distributed transformations, whose application is managed by control expressions in appropriate transformation units. It is then possible to describe 'transactional' refactorings [TB01] as combinations of primitive transformations that, individually, may not be behavior-preserving, but that, in appropriate compositions under appropriate control, may globally preserve behavior.

Paper outline. In Section 2 we review some approaches to refactoring and to software evolution using graph rewriting and in Section 3 we present a motivating example. Background notions on distributed graph transformation are given in Section 4. In Section 5, we first reformulate the problem of maintaining consistency among different forms of specification and code as the specification of

suitable distributed graph transformations, and then we illustrate our approach by means of two refactorings. Conclusions are given in Section 6.

2 Related Work

Since Opdyke's seminal thesis [Opd92], where preconditions for behavior preservation are analysed, formal methods have been applied towards a clear definition of the conditions under which refactoring takes place. More recently, the approach has been complemented in Robert's thesis, where the effect of refactoring is formalized in terms of postconditions [Rob99], so that it is possible to treat and verify composite refactorings, in which the preconditions for a primitive refactoring are guaranteed by the postconditions for a previous one. In general, it is very difficult to provide proofs of the fact that refactorings preserve the behavior of a program, due mostly to the lack of a formal semantics of the target language. The approach in [Opd92] is based on proofs that the enabling conditions for each refactoring preserve certain invariants, without proving that the preservation of the invariants implies the preservation of the behavior.

Fanta and Rajlich proposed a collection of algorithms which implement a set of refactorings on C++ code, exploiting the abstract semantic tree for the code provided by the GEN++ tool [FR98]. These algorithms directly access the tree, but are not formalized in terms of tree modification.

Recent work in the UML community has dealt with the effects of refactoring on UML diagrams, mainly class or state diagrams [SPTJ01]. As these works consider model refactoring prior to code generation, they do not take into account the possible associated modifications of code.

Mens [Men99] studies how to express code transformations in terms of graph transformations, by mapping diagrams onto type graphs. Such type graphs are comparable to the abstract syntax description of UML models specified in the UML metamodel. The work by Mens *et al.* in [MDJ02] exploits several techniques also used in this paper, such as control expressions, negative conditions, and parameterized rules. As these studies focus on the effect of refactorings on the source code, they do not investigate the coordination of a change in different model views with that in the code.

Since some refactorings require modifications in several diagrams, we propose a constrained way of rewriting different graphs, as a model for a full formalization of the refactoring process, derived from distributed graph transformations. This is 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 [TFKV99]. Such an approach has also been applied to the specification of ViewPoints, a framework to describe complex systems where different views and plans must be coordinated [GEMT00]. In the ViewPoint approach, inconsistencies between different views can be tolerated [GMT99], while in the approach proposed here different graphs have to be modified in a coordinated way so that the overall consistency of the specification is always maintained.

3 An example

We introduce an example which represents in a synthetic way a situation often encountered when refactoring, i.e. classes have a similar structure, or follow common patterns of collaboration. In particular, we illustrate the problem of exposing a common client-server structure in classes otherwise unrelated.

Consider a set of client classes, indicated as `Client⟨X⟩`, and a set of server classes, indicated as `Server⟨X⟩`, providing services to the clients (the names are arbitrary and have been chosen only to facilitate reading).

```
class Client⟨X⟩ {
    protected Server⟨Y⟩ serv⟨Y⟩;
    protected Server⟨Y⟩ getServer⟨Y⟩() { // dynamic server lookup }
    protected void exploitService⟨X⟩() {
        serv⟨Y⟩ = getServer⟨Y⟩();
        Object result = serv⟨Y⟩.service⟨Y⟩(this);
        // code using result
    }
}

class Server⟨Y⟩ {
    public Object service⟨Y⟩(Object requester) { // service implementation }
}
```

Several types of primitive refactorings can be combined to extract the client-server structure[1]. First of all, we *rename* all the variables `serv⟨Y⟩` as `serv` using the `rename_inst_variable` refactoring. Similarly we *rename* all the methods of type `getServer⟨Y⟩`, `exploitService⟨X⟩`, and `service⟨Y⟩` to their common prefixes using the `rename_method` refactoring. `extract_code_as_method` is then applied to all client classes to *extract* different versions of the method `exploit`, containing the specific code which uses the `result` provided by the call to `service`. We then *add* an abstract class, through the `add_class` refactoring, to which we *pull up* the instance variables `serv` and the now identical methods `exploitService`. The client classes are subclasses of this abstract class. We proceed in a similar manner to *create* an interface `Server` declaring a method `service`, of which the server classes are implementations (we can use interfaces here as we do not have either structure or code to *pull up*).

The result of this sequence of refactoring is described by a new collection of classes, where the parts in bold indicate the modifications.

```
abstract class AbstractClient {
    Server serv;
    abstract Server getServer();
    void exploitService() {
        serv = getServer();
```

[1] We use the names of refactorings in the *Refactoring Browser* [Rob99].

```
        Object result = serv.service(this);
        exploit(result);
    }
    abstract void exploit(Object arg);
}

class Client⟨X⟩ extends AbstractClient {
    Server getServer() { //code from getServer⟨Y⟩ }
    void exploit(Object arg) { // previous code using result }
}

interface Server {
    Object service(Object requester);
}

class Server⟨Y⟩ implements Server {
    public Object service(Object requester) { return service⟨Y⟩(requester); }
    Object service⟨Y⟩(Object requester) { // service implementation }
}
```

Each individual refactoring implies changes in class diagrams, to modify the class structures and inheritance hierarchies. However, some refactorings also affect other forms of specification. For example, `extract_code_as_method` affects sequence diagrams, as it introduces a self call during the execution of `exploitService()`. State machine diagrams may be affected as well, if states such as `WaitingForService`, `ResultObtained` and `ResultExploited` are used to describe the process realized by `exploitService`. To reflect process unfolding, the latter state can be decomposed into `ExploitStarted` and `ExploitCompleted`.

4 The Formal Background

Distributed rule application follows the double-pushout approach to graph transformation as described in [TFKV99], using rules with negative application conditions. For further control on distributed transformations, transformation units [KKS97] on distributed graph transformation are used.

4.1 Distributed Graph Transformation

We work with distributed graphs containing typed and attributed nodes and edges. Edge and node types for a given family of graphs $\mathcal{F}$ are defined in a *type graph* $T(\mathcal{F})$ and the typing of a graph $G \in \mathcal{F}$ consists of a set of injective mappings from edges and nodes in G to edges and nodes in $T(\mathcal{F})$. Distributed graph transformations are graph transformations structured at two abstraction levels: the *network* and the *object* level. The network level describes the system's architecture by a network graph, and allows its dynamic reconfiguration through network rules. At the object level, graph transformations are 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 describe how the object graphs are interconnected. 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,which are graph morphisms on local object graphs. Each node mapping in 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. The morphisms must also be consistent with the attribute values.

Following the double-pushout approach, a *distributed graph rule* $p : L \stackrel{l}{\leftarrow} I \stackrel{r}{\rightarrow} R$ is given by two distributed graph morphisms l and r. It transforms a *distributed host graph* G, into a *target graph* G' if an *injective match* $m : L \rightarrow G$ exists, which is a distributed graph morphism. 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. In our case, I can be understood to simply be the intersection of L and R. A rule may also contain *negative application conditions (NAC)* to express that something *must not* exist for the rule to be applicable. These are a finite set of distributed graph morphisms $NAC = \{L \stackrel{n_i}{\rightarrow} N_i\}$ and can refer to values of attributes [TFKV99]. Several morphisms $L \stackrel{n_i}{\rightarrow} N_i$ can become necessary in an NAC to express the conjunction of basic conditions. For the rule to be applicable, no graph present in a NAC must be matched in the host graph in a way compatible with $m : L \rightarrow G$. We also allow the use of *set nodes*, which can be mapped to any number of nodes in the host graph, including zero. The matching of a set node is exhaustive of all the nodes in the host graph satisfying the condition indicated by the rule.

The application of distributed graph transformations is synchronized via *subrules*. A subrule of an object level rule identifies that portion of the rule which modifies nodes or edges shared among different local graphs, so that the modifications have to be synchronized over all the involved rules. Hence, synchronous application is achieved by rule amalgamation over the subrules. For details, see [TFKV99]. In this paper we use dotted lines to denote NACs and grey shading to indicate subrules. Non-connected NACs denote different negative application conditions (see Figure 3). The rules in the following figures are only the local components of distributed transformations. For the case discussed in the paper, the network components of the rules are usually identical rules. We will discuss later the few cases in which non-identical rules are needed at the network level.

4.2 Transformation Units

Transformation units are used to control rule application, with the control condition specified by expressions over rules [KKS97]. The concept of a transformation unit is defined independently from any given approach to graph transformation.A graph transformation approach $\mathcal{A}$ consists 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. Given an approach $\mathcal{A}$, a *transformation unit* consists of an initial and a terminal graph class expression in $\mathcal{E}$ (defining which graphs serve as valid input and output graphs), a set of rules in $\mathcal{R}$ and a set of references to other transformation units, whose rules can be used in the current one, together with a control condition over $\mathcal{C}$ describing how rules can 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. We adopt here the syntax for control expressions in [KP02].

Here, we apply transformation units to distributed graph transformation, so that $\mathcal{A}$ is thus defined: $\mathcal{G}$ is the class of distributed graphs, $\mathcal{R}$ the class of distributed rules, and $\Longrightarrow$ the DPO way of rule application, as defined in [TFKV99]. The control expressions in $\mathcal{C}$ are of the type mentioned above and described in [KP02], while the class $\mathcal{E}$ is not needed here. It can trivially be left empty to indicate that no special initial and terminal graph classes need to be specified.

We relate rule expressions to graph rules by naming rules and passing parameters to them, to be matched with specific attributes of some node. By this mechanism, we can restrict the application of rules to those elements which carry an actual reference to the code to be refactored. To this end, the rules presented in the transformation units are meant as rule schemes to be instantiated to actual rules, assigning the parameters as values of the indicated attributes.

5 Refactoring by Graph Transformation

In this section, we analyse some examples of refactoring involving transformations in more than one view, i.e. in the code and at least one UML diagram. Following [Rob99], refactorings are expressed by pre- and post-conditions. A typical interaction with a refactoring tool can be modelled by the following list of events: **(1)** The user selects a segment of code. **(2)** The user selects one from a list of available (possible composite) refactorings. **(3)** The tool checks the preconditions for refactoring. **(4)** 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, and we ignore here the relative process. We consider the effect of refactorings by defining graph transformation rules, possibly distributed over different view graphs, and managed in transformation units. Complex refactoring are modeled as sequences of individual refactoring steps. The composition of transformation units can give rise to complex refactorings. The effect of refactoring on different views is 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, even if checked on the textual code, involve the analysis of structural properties, such as the visibility of variables in specific portions of

code, or the existence of calls to some methods, which may be distributed over diagrams of different types.

5.1 Graph Representation of Diagrams and Code

In the line of [Men99], we consider type graphs for 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 type graphs given by the metamodel definitions in [OMG01].

We also assume the possibility of representing the source code in the form of a flow graph for a method, as is typical in compiler construction [App98]. This is a directed graph where nodes are lines of code and a *prev/next* relation exists between two nodes if they are consecutive lines, or the line represented by the source node can branch execution to the variable represented by the target node. Moreover, each *Line* node is attached to a set of nodes describing the variables mentioned in the line (for definition or usage), and the methods, if any, called in the line. Some of these variables can be *local* to the *Method* environment, while the other variables used in the method, but not local to it (and not defined in the class) are necessarily passed to the method. Local variables are *defined in* specific lines of code. Finally, a set of *Parameter* nodes describing the types of arguments to a method, and *Classifier* nodes, describing types of variables, parameters and methods are also present. Figure 1 describes the resulting type graph. Such a type graph is simpler than the one in [MDJ02] 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 5.2.

Distributed graphs are suited to describe relationships between diagrams and code fragments. Following the approach of [GEMT00], a network graph NG describes the type graph for the specification of the whole software system at some stage of the evolution process. In NG, a network node is either associated with one local object graph – representing either a UML diagram or the code flow graph (we call such nodes *content* nodes) – or it is an *interface* node. Here, we consider only the *Class*, *Sequence*, and *StateMachines* families of diagrams discussed in the text, and the *Code Flowgraph*. For each pair of diagram nodes, a common interface node exists. Interface nodes are refined at the local level by 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). For example, an interface between Class diagrams and Flow graphs will present *Method*, *Variable*, and *Class* nodes, an interface between State Machine diagrams and Sequence diagrams may have nodes for

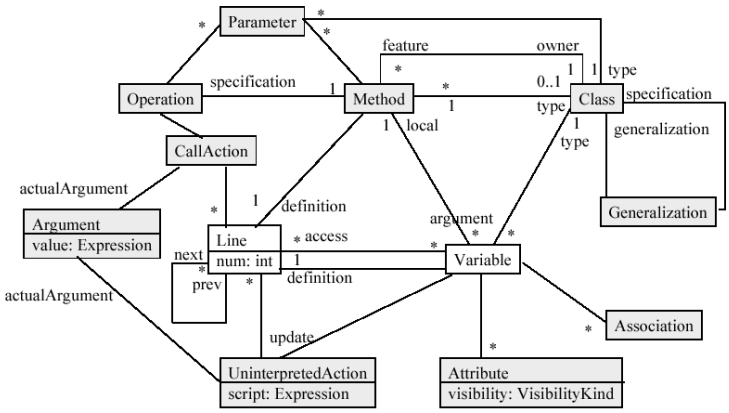

Fig. 1. The type graph for code representation.

states and transitions dealing with the call of a method[2] or other events, and an interface between Code and Sequence Diagrams will contain nodes for call actions. Several network nodes of the same type can be used in the specification of a software system. For instance, different sequence diagrams are used to depict different scenarios, or a class can be replicated in different class diagrams to show its relationships with different sets of other classes.

The following table indicates the need for modifications in different diagrams, where the first column presents the name of some refactorings implemented in the *Refactoring Browser* [Rob99], involving modifications of class structures; the second column lists the types of diagrams affected by the refactoring; the third column indicates if a new diagram of the indicated type has to be added (+) or removed (-), involving a modification at the network level. The = sign indicates that no modification at this level need occur. Basically, only state machine diagrams can be added or removed. From a practical point of view, the creation of new state diagrams is usually not needed for small classes.

Refactoring	**Affected Diagrams**	**Creation/Deletion**
Add Class	Class	State+
Remove Class	Class, Sequence	State-
Rename Class	Class, Sequence	=
Remove Method	Class, Sequence, State	=
Add Parameter to Method	Class	=
Extract Code As Method	Class, Sequence, State	=
Push Up/Down Method	Class, Sequence, State	=

[2] We could as well refer to Activity Diagrams in a similar way.

5.2 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 a newly formed method.

A precondition for such a refactoring is that the code to be extracted constitute a *block*, i.e. it has 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 the block property itself. 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: 1) a new method is created whose body is the extracted code; 2) 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 to be local to the original method); 3) the code in the original method is replaced by a call to the new method.

Figure 2 contains the local rule for the code graph. It is part of a distributed rule, named *ecamAll* after the initials of the refactoring. In Figure 2, a set node indicates the lines of code, in the original version of the method, which lie between the *first* and *last* lines of the block to be moved. Another set node indicates the variables to be passed as parameters to the new methods. These are variables referenced in the block, but neither defined within it, nor in the class. The rest of the method is left untouched. However, one has to check that no branches exist from lines in the block to other parts of the method, nor must branches exist from other parts of the method to the middle of the block. This is checked in the negative application conditions. The values of *first* and *last*, as well as the names *morig* and *mnew* of the original method and of the new one, are provided by the rule expression activating the application of the transformation. Nodes of type *Parameter*, *Class* and *Method* are shaded, as they and their interrelations are shared with other diagrams, thus identifying the subrules.

Figure 3 describes the local rule of *ecamAll* acting on class diagrams. At the structural level, only the existence of a new method 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 here. 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 inheritance hierarchy of the modified class. These conditions make use of additional *gen* edges, produced by the rules of Figure 4 which construct the transitive closure of the *Generalization* relation in the inheritance hierarchy in class diagrams. The set of rules to compute the closure is completed by a rule *insert_down_gen* which creates *gen* edges betwen a class and its descendants, like *insert_gen* creates edges between a class and its ancestors. An inverse sequence of operations eliminating the *gen* edges must be performed at the end of the refactoring process.

The newly created call is also observed at the behavioral level, as shown in Figure 5. It inserts a `CallAction` to the new `Method`, named *mnew*, and the corresponding activation of the `Operation` spawning from an activation of the old `Method`, *morig*, in the `Class` *cname*. As the subrule identifies these elements

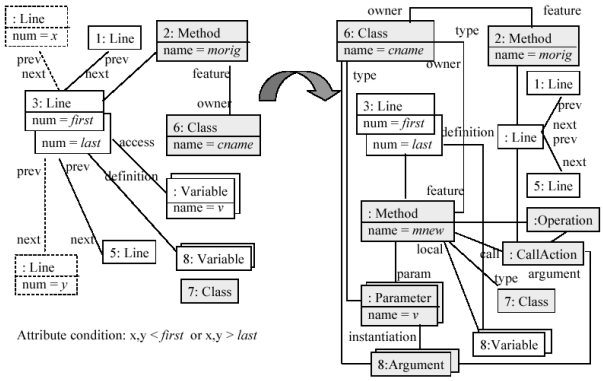

Fig. 2. The component of *ecamAll(cname, morig, mnew, first, last)* for the code flow graph.

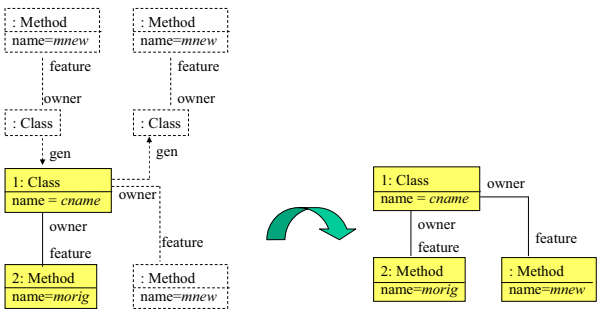

Fig. 3. The component of *ecamAll(cname, morig, mnew, first, last)* for class diagrams.

for amalgamation, this transformation occurs in a synchronized way on all the sequence diagrams presenting an activation of *morig*. However, as several activations of the same method can occur in one sequence diagram, this local rule has to be applied as long as possible on all sequence diagrams. The transformation of the sequence diagrams should be completed by transferring all the `CallActions`, originating from *morig* and performed in lines that have been extracted to *mnew*, to the activation of *mnew*. A rule *completeEcamSequence*, not presented here for space reasons, is responsible for that.

Finally, the component of *ecamAll* which operates on state machine diagrams inserts a new state and a couple of transitions for all the states to which a `Transition` labelled with a `CallEvent` for *morig* exists. The new state is reached with a `Transition` labelled with a `CallEvent` for *mnew*. Return to the previous state occurs by a "completed" `Event`, indicating the completion of the operation.

All the transformations above have to be applied in a synchronized way to maintain consistency of diagrams and code. The network level transformations

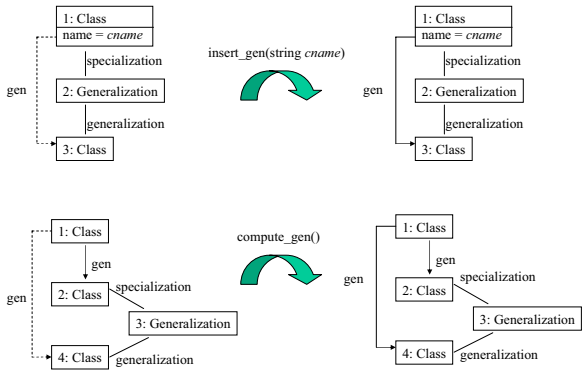

Fig. 4. The rule scheme for computing the transitive closure of the *Generalization* relation.

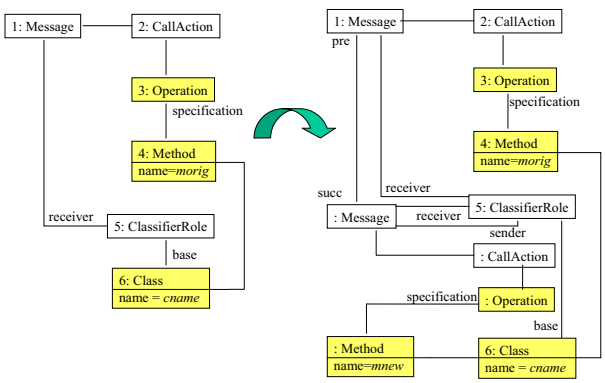

Fig. 5. The component of *ecamAll(cname, morig, mnew, first, last)* for sequence diagrams.

simply rewrites nodes into themselves. With each such rewriting, a transformation of the associated local graph occurs. An overall transformation unit describes this refactoring. This can be expressed as:

ExtrCodeAsMthd(String cname, String morig, String mnew, int first, int last) =
  asLongAsPossible insert_gen() end;
  asLongAsPossible insert_down_gen() end;
  asLongAsPossible compute_gen() end;
  if applicable(ecamAll(cname, morig, mnew, first, last)) then
    asLongAsPossible ecamAll(cname, morig, mnew, first, last) end;
    asLongAsPossible completeEcamSequence(cname, morig, mnew) end;
  else null end;
  asLongAsPossible remove_gen() end;

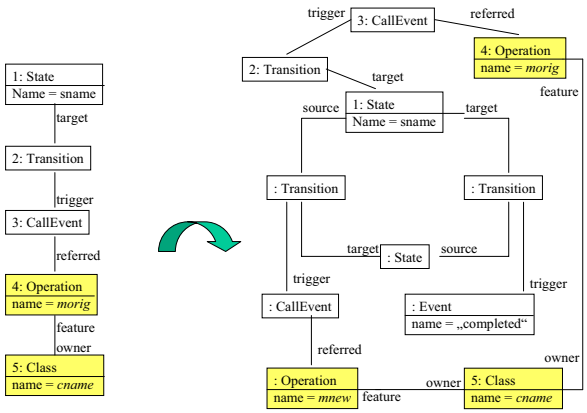

Fig. 6. The component of *ecamAll(cname, morig, mnew, first, last)* for state machine diagrams.

As transformations occur both at the network and local level (but at the network level they are identical), applying them as long as possible allows the transformation of all local graphs affected by refactoring, while at each time the associated network node is rewritten into itself. Note that the rules presented above should be complemented so that the elements to which a transformation has already been applied are tagged and the presence of the tag must be checked to prevent the process from applying a rule again to the same element. The transformation unit is then completed by removing the tags.

5.3 Method movement

As shown in Figure 7, the code of a `Method` *morig* can be moved from its defining *source* `Class` to a different *target* `Class` in which it takes a new name *mnew*. A Method with the name *mnew* must not already appear (higher or lower) in the hierarchy of the target Class, as indicated by the NACs. 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 `Parameter` node whose attribute *name* takes the value "orig" in Figure 7, illustrating the component of the distributed rule *mmAll(morig, source, mnew, target)* for class diagrams. (Again, the name *mmAll* derives from the initials of the refactoring.) 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 the nodes labelled `ClassX` and `ClassY` and where one of the NACs would be omitted. In the code flow graph, the code of *morig* has to be replaced with a forwarding method that simply calls *mnew* in the target class. We do not show this transformation here.

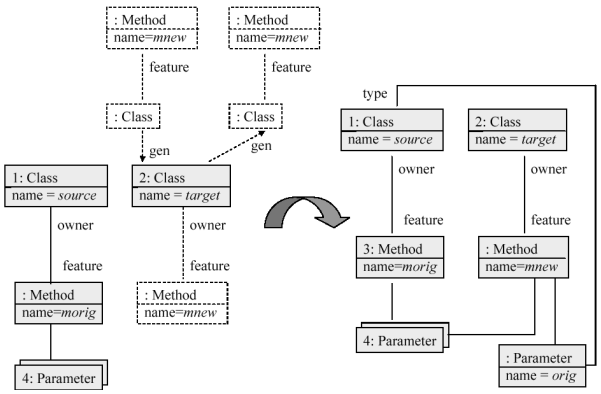

Fig. 7. The component of *mmAll(morig, source, mnew, target)* for class diagrams.

Besides class diagrams, sequence diagrams have to be modified as well, according to the transformation scheme depicted in Figure 8. Hence, a `CallAction` for *mnew* towards the *target* `Class` must be inserted. While the call to the forwarding method does not modify the behavior of the class, it has to be reflected in this diagram to prevent subsequent refinements of the diagram from violating the correct sequence of calls. Again, the transformation of diagrams should be completed by a transfer of all calls originating from the *morig* method to the activation of the *mnew* method.

The overall transformation unit is controlled by a rule expression of the form:

```
MoveMethod(String morig, String source, String mnew, String target) =
    asLongAsPossible insert_gen() end;
    asLongAsPossible insert_down_gen() end;
    asLongAsPossible do compute_gen() end;
    asLongAsPossible mmAll(morig, source, mnew, target) end;
    asLongAsPossible completeEcamSequence(morig, source, mnew, target) end;
    asLongAsPossible remove_gen()
```

which causes the flow graph for the old method, together with all the network nodes of type *ClassDiagram* and *SequenceDiagram* whose local nodes contain references to the moved method, to be affected by the transformation.

In Figures 7 and 8 we consider the case where methods are instance methods and the code is moved between unrelated classes. The cases for static methods, or for calls to methods in classes in the same hierarchy (event is a self call), require some obvious modifications.

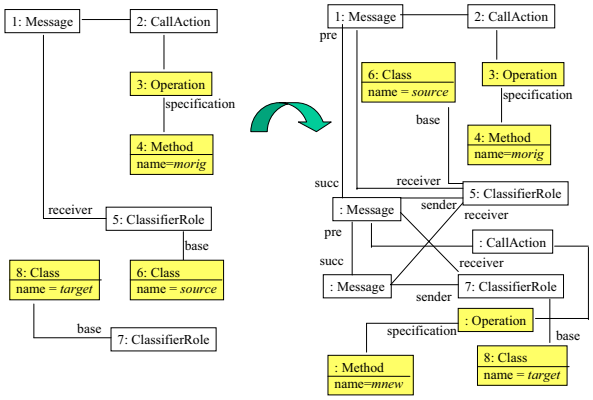

Fig. 8. The rule scheme for sequence diagram modification in `move_method`.

6 Conclusions

We have presented an approach to maintaining consistency between code and model diagrams in the presence of refactorings. Each refactoring is described by a transformation unit, with parameters depending on the specific code modification, applied to the diagrams affected by the change. This framework avoids the need for reverse engineering, which reconstructs the models from the modified code, by propagating only the incremental changes to the original model. The model proposed can also be seen as a way to maintain consistency in the different diagrams through re-engineering steps, before proceeding to the actual code modification. Since there is no 'preferred' object in a distributed graph, the model can be used to propagate changes made on any diagram to the remaining ones, and not necessarily from the code to the documentation, as in refactoring. 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. The formalism can then be exploited to analyze possible conflicts or dependencies between different types of refactorings. We are now investigating the use of abstract syntax as a replacement for representation of concrete code, which would also favor the integration of the approach in existing refactoring tools based on abstract syntax representations.

References

[App98] A. W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[BF01] K. Beck and M. Fowler. *Planning Extreme Programming.* Addison Wesley, 2001.

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[FR98] R. Fanta and V. Rajlich. Reengineering object-oriented code. In *Proceedings of ICSM 1998*, pages 238–246. IEEE Computer Society Press, 1998.

[GEMT00] M. Goedicke, 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*, pages 369–377, 2000.

[GMT99] M. Goedicke, T. Meyer, and G. Taentzer. Viewpoint-oriented software development by distributed graph transformation: Towards a basis for living with inconsistencies. In *Proc. 4th IEEE Int. Symp. on Requirements Engineering*, pages 92–99, 1999.

[KKS97] H.-J. Kreowski, S. Kuske, and A. Schürr. Nested graph transformation units. *Int. J. on Software Engineering and Knowledge Engineering*, 7(4):479–502, 1997.

[KP02] M. Koch and F. "Parisi Presicce". Describing policies with graph constraints and rules. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Transformation. Proc. ICGT02*, pages 223–238, 2002.

[MDJ02] T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Transformation. Proc. ICGT02*, pages 286–301, 2002.

[Men99] T. Mens. 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*, pages 127–143, 1999.

[NWZ01] J. Niere, J.P. Wadsack, and A. Zündorf. Recovering UML Diagrams from Java Code using Patterns. In J.H. Jahnke and C. Ryan, editors, *Proc. of the 2^{nd} Workshop on Soft Computing Applied to Software Engineering.* Centre for Telematics and Information Technology, University of Twende, The Netherlands, February 2001.

[OMG01] OMG. UML specification version 1.4. *http://www.omg.org/technology/documents/formal/uml.htm*, 2001.

[Opd92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Rob99] D. B. Roberts. *Practical Analysis for Refactoring.* PhD thesis, University of Illinois, 1999.

[SPTJ01] G. Sunyé, D. Pollet, Y. Le Traon, and J.-M. Jézéquel. Refactoring UML models. In M. Gogolla and C. Kobryn, editors, *Proc. UML 2001*, pages 134–148. Springer, 2001.

[TB01] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, 8:89–120, 2001.

[TFKV99] G. Taentzer, 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*, pages 269–340. World Scientific, 1999.