Specifying Coherent Refactoring of Software Artefacts with Distributed Graph Transformations[i]

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

Refactoring is an important source of software transformation, which changes the internal structure of a software system, while preserving its behavior. Even though the input/output view of a system's behavior does not change, refactoring 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. Such modifications must be reflected in the system model, generally expressed through UML diagrams. We propose a formal approach, based on distributed graph transformation, to the coordinated evolution of code and model, as effect of refactorings. The approach can be integrated into existing refactoring tools. Due to its formal background, it makes it possible to reason about the behavior preservation of each specified refactoring.

1. Introduction

Software and processes specifications are valuable company assets. The diffusion of UML in large programming organizations is already creating a vast amount of documentation in the form of collections of UML diagrams, being inspected by developers and other designers, and in some cases used to communicate with the shareholders involved in a project. Repositories of documentation facilitate software reuse and design pattern identification. Ideally, refinements or adaptations of existing programs should maintain a trace to the original material, and software transformations should be reflected back to the documentation.

An important source of software transformation is *refactoring*. In refactoring, the internal structure of a software system changes, while preserving its behavior, as expressed by the

[i] Partially supported by the EC under Research and Training Network SeGraVis.

input/output relation. However, refactoring 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. Several types of refactoring are now known and widely used (Fowler, 1999). For these types, it is demonstrable that they preserve program behavior, and it is usually known in which way they modify its static specification, in the form of class diagrams. Refactoring can also occur in design, involving modifications of interaction, state machine, or activity diagrams.

However, it is not always the case that all transformations induced by refactoring are actually mapped back to the relevant documentation. Since refactoring is usually performed at the source code level, it becomes difficult to maintain consistency between the code and its specification - expressed for example with UML diagrams - which usually refers to the original version of the code. In particular, one has to identify which diagrams are interesting when modifying a piece of software in the first place. In such situations, it is easy to lose consistency between the code and the specification. Two strategies can be adopted to preserve consistency: either recovering of the specification after a chosen set of changes, or coherently defining the effects of each refactoring on the different artefacts of a software project. While changes in structural specifications are notationally equivalent to lexical transformations on the source code, transformations of the behavioral specifications may be significantly more intricated.

We discuss an approach to the problem of maintaining consistency between source code and diagrams, both structural and behavioral, using the formal framework of graph transformation. In particular, Abstract Syntax Trees describe the source code, while UML diagrams are represented as graphs, conforming to the abstract syntax presented in the UML metamodel. The UML diagrams and the code are hence seen as different views on a software system, so that consistency between the views and the code is preserved by modeling coherent refactorings as graph transformations distributed on several graphs. Complex refactorings, as well as checking of complex preconditions, are decomposed into collections of distributed transformations whose application is managed by control expressions in appropriate transformation units.

Paper organization. In the rest of this introduction, we set the background for our work, by introducing the refactorings used in the motivating example of Section 2, reviewing some approaches to refactoring and to software evolution using graph rewriting, and illustrating motivations for the coherent refactoring of code and models. Background notions on graph transformation are given in Section 3. In Section 4, we reformulate the problem of maintaining

consistency among different forms of specification and code as the definition of suitable distributed graph transformations, and illustrate our approach by two important refactorings. Section 5 discusses the principles under which one can establish correspondences between abstract representations of the code and of the model. Section 6 discusses forms of behavior preservation and sketches the way to how formal results for graph transformation help in reasoning about it. Conclusions are given in Section 7.

1.1 Selected refactorings

The number of refactorings which are seen useful in practice, proposed in literature, or implemented in systems, is continually increasing. While a complete coverage of refactorings is beyond the scope of this paper, we illustrate here the basic refactorings used in the example of Section 2. A rich set of refactorings, both primitive and complex is given in (Fowler, 1999). In general, all refactorings require that no name clashes are generated as a consequence of it. For instance, if a new method is introduced or has its name changed, a check is needed to ensure that no method with the same signature is already present in the inheritance hierarchy. Hence, we only mention additional checks other than checks for name clashes for specific refactorings.

`RenameVariable` and **`RenameMethod`** change the name of a variable or method to highlight structural or behavioral analogies in a set of classes: all references to these features must be renamed. **`RenameMethod`** is one of the constituents of the **`ChangeMethodSignature`** refactoring, with sub-refactorings such as **`ChangeReturnType`** and **`ChangeParameterType`**, or addition and removal of parameters. The **`EncapsulateVariable`** refactoring hides information by making a variable private and providing public *getter* and *setter* methods for accessing and updating it. All direct references to the variable are replaced by dynamic calls to these methods. **`InsertClass`** expands the inheritance hierarchy by introducing a new class `B` between a class `A` and its original superclass `C`. `B` becomes the superclass for `A` and has `C` as its superclass. **`PullUpMethod`** allows replicated methods to be moved from subclasses into a common superclass. To apply this refactoring, the body of the pulled up method must not refer to any variable only defined in subclasses. **`ExtractMethod`** derives from the composition of several primitive refactorings, but it is so widespread that it can be considered as a single one. It removes a block of code from a method and uses it to create a new method, substituting the code in the original method by a call to the new one. Beyond avoidance of name clashes, preconditions for it require that all variables which are accessed by the extracted code and have a local scope be passed as parameters, and that

the removed code forms a block, i.e. it has a single entry point and a single exit point.

1.2 Related Work

Several tools have been developed to assist refactoring; some are packaged as stand-alone executables, while others integrate refactorings into a development environment. Many tools refer directly and exclusively to a specific language, for example **C# Refactory** (http://www.xtreme-simplicity.net/) for C#, or **CoreGuide6.0** (http://www.omnicore.com) for Java. **Xrefactory** (http://www.xref-tech.com) assists in modifying code in C and Java. All of these provide a variety of refactorings, typically renamings and method extraction. None of these tools mentions diagrams and the effects on other views of the system, including documentation.

The class diagram, referred to as 'the model', is instead considered in **objectiF** (http: //www.microtool.de/objectiF) which, in addition to supporting a variety of languages, allows transformations of both the code and the class model, with changes propagated automatically to both views. Other kinds of diagrams, especially those describing behavioral aspects of the system, are not refactored. **Eclipse** (http://www.eclipse.org) integrates system-wide changes of code with several refactor actions (such as rename, move, push down, pull up, extract). 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.

Reverse engineering is also present in **Fujaba** (Niere *et al*., 2001), where the user can reconstruct the model after a chosen set of changes of the code. A more efficient option would be to define the effects of a refactoring on the different parts of the model. This is more easily realized on structural models, where transformations on such diagrams are notationally equivalent to the lexical transformation on the source code, than on behavioral specifications. Modern refactoring tools, however, work on abstract representations of the code, rather than on the code itself, typically in the form of an *Abstract Syntax Tree* (AST), following Roberts' (1999) line.

Refactorings are defined also on model diagrams. Sunyé *et al*. (2001) illustrate refactoring of statecharts, typically to extract a set of states to be part of a composite state. Transformations of concrete diagrams are specified by pre and post conditions, written as OCL constraints. Metz *et al*. (2002) consider the UML metamodel to propose extensions to use case models, which would allow significant refactorings of such models and avoid improper current uses. These papers, however, do not consider the integration with possible source code related to these models.

Current class diagram editors do not extend changes to all other related diagrams, limiting their "automation" to the source code, with the result that direct intervention is needed to restore consistency among possibly various UML diagrams representing the same subsystem. We adopt UML metamodel instances and draw a correspondence between these and abstract syntax trees representing code. Hence, a common graph-based formalism can be used as basis for an integrated management of refactoring both, the code and the model in an integrated way.

Graph rewriting has been introduced as a basis for managing transformations induced by refactoring in work by Mens, alone (2000, 2001) and with others (2002). In these papers, a non-standard graph representation of code is used, so that this approach is not able to leverage from the availability of AST representations in existing tools. Moreover, integrated refactoring of model and code by graph transformation has not been considered up to now.

1.3 Outline of the approach

Our approach aims at precisely specifying integrated refactoring of model and code, maintaining the consistency between them achieved during the development of a software project.

Indeed, software development follows typical patterns. Authors can start some modelling activity, update and refine their models, start writing code, and modify it. If a CASE tool is available, code can be generated from models, e.g. skeletons of classes or collaboration skeletons for some pattern. Moreover, developers can decide to compile the code, reverse-engineer, and update some parts of the model from the generated abstract syntax tree. As a result of this first phase of *development*, parts of the code and of the model have had an independent evolution, so that they can be consistent or not, while other parts have evolved in a coordinated way. The model and the code can then be *checked* to identify parts consistent with one another, either by circumstance or by construction. To this end, we assume that at any time a pair of graphs exists providing abstract representations of the code and of the model. The code representation is in the form of an AST, while the model is given through UML diagrams, and constitutes an instance of the UML meta-model. The checking phase produces an *interface graph IG* and a pair of graph morphisms from *IG* to the AST and UML graphs respectively. The morphisms establish the correspondence between code and models. The relations between these of graphs (compare Figure 1) must be managed by the CASE tool.

The subsequent phases of *refactoring* can then be triggered on the code or the model. For any refactoring supported by the CASE tool in which the user acts on the code or on the model, the

corresponding modifications on the other graph must be enforced. After refactoring, the cycle can start again with new developments, and so on. Such a process is depicted in Figure 2.

Figure 1. The graphs to be managed by a CASE tool.

While refactoring tools work on both abstract and concrete representations of code, they are usually restricted to the manipulation of structural aspects of the model, namely class diagrams. Although this is intuitively justifiable by the stated assumption that refactoring does not affect the behavior of systems, the combination of refactoring with other forms of code evolution can lead to inconsistencies between the model and the code. This could be avoided by a careful consideration of what a refactoring involves, as shown in the following two subsections.

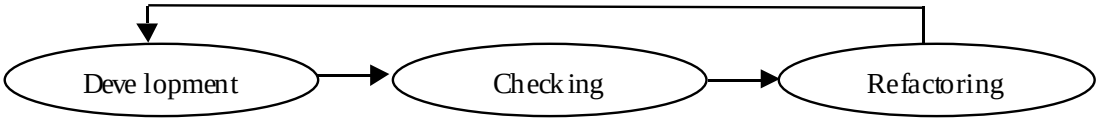


Figure 2. An activity diagram depicting the sequence of activities at the basis of our approach.

1.3.1 Modification of collaborations

Consider refactoring **`ExtractMethod`** in which a block of code *blk* is removed from a method *morig* in a class *C*, a new method *mnew* is created, block *blk* is inserted as the body of *mnew* and a call to *mnew* replaces the original code in *morig*. If the execution of *morig* is represented in a collaboration diagram (as in Figure 3a)), but the refactoring tool cannot manipulate such a diagram, then the activation of *mnew* cannot be explicitly represented. Now suppose that *mnew* is subsequently moved to another class *D* which is coupled to *C*, and finally modified so that some new activities are executed during its execution, involving a call to a method *meth* in a third class *E*. The designer can now assume that the last addition of the call to *meth* is significant enough to show it in the collaboration. But as we have lost the consistency between the model and the code, a simple-minded addition of the call to *meth* as a descendant of the call to *morig* would result in an inconsistent model, as shown in Figure 3b). This can be avoided if all the steps in this process are reflected in the collaboration: the extraction of *blk* to *mnew* would be reflected by a self-call stemming from the activation for *morig*, the movement of *mnew* to class *D* would transform the self-activation to a call to this new class and the consequent activation of *mnew* in it, so that the

call of *meth* would now be in the correct position, as in Figure 3c).

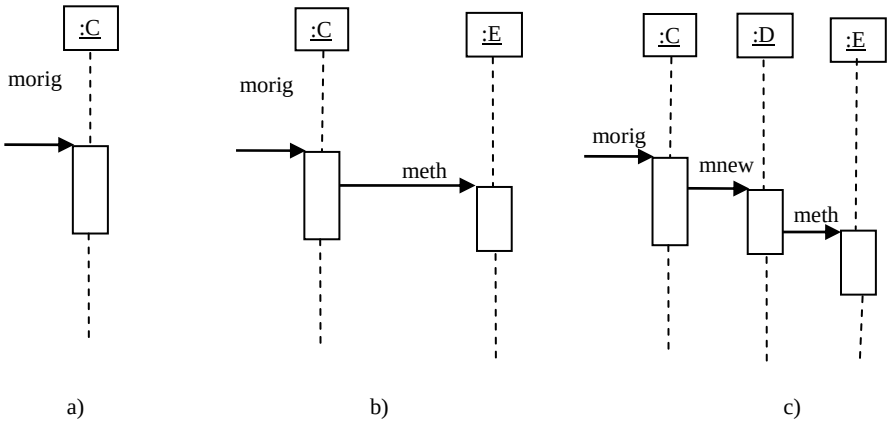


Figure 3. a) original sequence diagram; b) inconsistent situation from not reflecting the extraction of *mnew*; c) desired consistent situation.

1.3.2 Modification of activity graphs

Activity graphs are special types of state machines used for describing complex processes, involving several classifiers, where the state evolution of the involved elements is modelled. Suppose that an action is performed to the effect of setting a field variable to some value, say `x = 15`. Hence, a state *s* appears in the model indicating that an assignment has to occur at that time. If the **`EncapsulateVariable`** refactoring is subsequently applied to the variable *x*, the code `x = 15` is replaced by `setX(15)`. The state in the activity diagram now becomes a `CallState` *s'*. (Compare similar modifications in activity diagrams in Figures 5 c) and 6 c).)

2. An Example of Refactoring

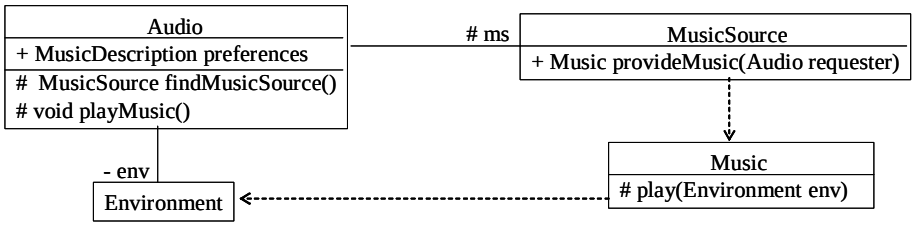
We illustrate the refactorings of Section 1.2 with a typical case in software development. Let us consider the design of an intelligent `Audio` player, able to dynamically identify a `MusicSource`, for example on the basis of some `preferences`, and to obtain from this source a piece of `Music`. It then sets up an environment, in which it require the received piece to play itself.

A first version produces the following, strongly coupled, set of classes. Moreover, the player must expose its preferences in a public variable for the music source to select some piece of music. This prevents reuse of the `Audio` class in a non-controlled environment. Figure 5 shows components of the UML model: class (5a), sequence (5b) and two activity diagrams (5c).

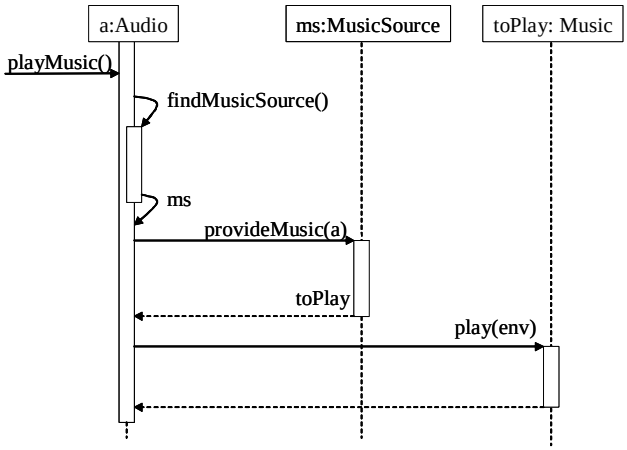
```
class Audio {
  protected MusicSource ms;
  private Environment;
  public MusicDescription preferences;
  protected findMusicSource() { // lookup for a music source }
  protected void playMusic() {
       ms = findMusicSource();
       Music toPlay = ms.provideMusic(this);
       // code to set the playing environment env
       toPlay.play(env);
  }
}
class Music {
   void play(Environment env) { // code to play in the environment env }
}
class MusicSource {
   public Music provideMusic(Audio requester) {
   MusicDescription desc = requester.preferences;
   // code to retrieve music according to desc and sending it back as result
   }
}
class Environment { // fields and methods to define a playing environment }
```

With a view to the possibility of reuse, the programmer decides to protect the preferences, by applying the `EncapsulateVariable` refactoring, discussed in Section 1.2. After this first step, the affected code looks as follows, where the parts in bold mark the changed elements. The new situation is reflected in the model diagrams of Figure 6.

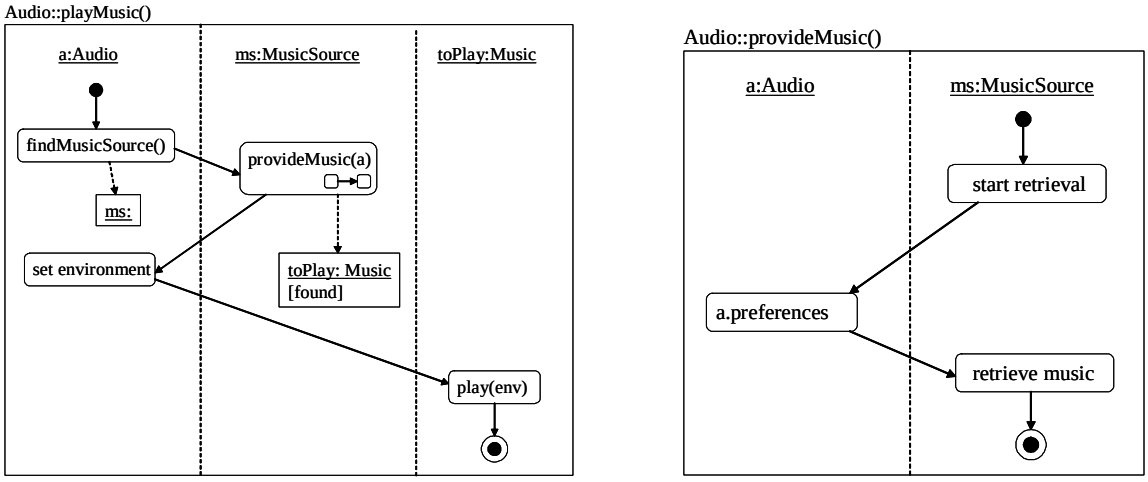
```
class Audio {
      protected MusicSource ms;
      private Environment;
      private MusicDescription preferences;
      protected findMusicSource() { // same implementation as before }
      protected void playMusic() { // same implementation as before }
      public MusicDescription getPreferences() { return preferences;   }
      public void setPreferences(MusicDescription desc) { preferences = desc; }
}
class MusicSource {
      public Music provideMusic(Audio requester) {
            MusicDescription desc = requester.getPreferences();
            // same code using desc as before
      }
}
```



(a)



(b)

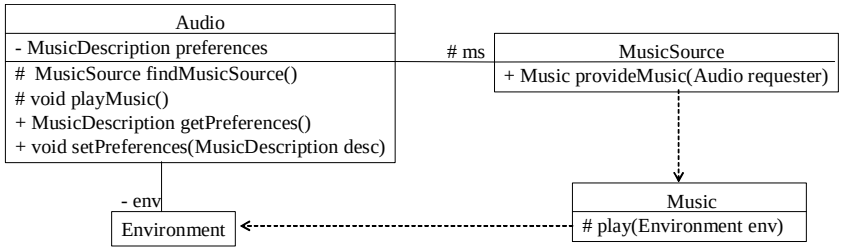


(c)

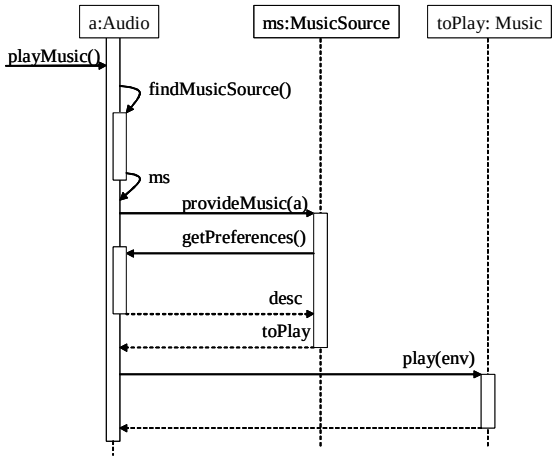
Figure 5. Components of the UML model for the first version of code. a) Class diagram. b) Sequence diagram. c) Activity diagrams.

The code above presents several possibility for refactorings, allowing the introduction of an abstract notion of player, able to retrieve a content source, interrogate it in order to obtain some content and setting an environment for it to be played. The concrete players will differ from each other for the type of source they have to retrieve and the way in which they define the

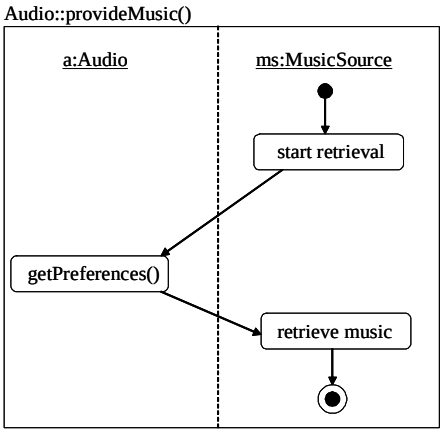
environment. On the other hand, content sources must have a generic ability to accept a player and sending the appropriate content to it, while the different forms of content will have specific realizations of the `play` method. To this end, a first step is to *extract* the code for playing in an environment from the `playMusic` method into a `setEnvironment` method. Method `playMusic` is then *renamed* to `playContent`. Analogously, `findMusicSource` is *renamed* to `findSource` and the variable `musicSource` to `source`, while in class `Music`, `provideMusic` is *renamed* to `provideContent`. Refactorings are then performed to introduce new classes and interfaces in an existing hierarchy, by *creat*ing and *insert*ing the abstract class `AbstractPlayer` and the interfaces `Content` and `ContentSource`. We can now *pull up* methods and variables from `Audio` to `AbstractPlayer`. Finally, all return and parameter types referring to the concrete classes are now *changed* to the newly inserted types. The resulting code is reported below. Again, parts in bold show the modified parts with respect to the previous version. The reader can reconstruct the UML diagrams according to these modifications.



(a)



(b)



(c)

Figure 6. The UML diagrams after variable encapsulation. a) class diagram, b) sequence diagram, c) activity diagram.

```
abstract class AbstractPlayer {
      protected ContentSource source;
      private Description preferences;
      private Environment env;
      protected abstract ContentSource findSource();
      protected abstract void setEnvironment();
      protected void playContent() {
            source = findSource();
            Content toPlay = source.provideContent(this);
            setEnvironment();
            toPlay.play(env);
      }
      Description getPreferences() { return preferences; }
      void setPreferences(Description desc) { preferences = desc; }
}
class Audio extends AbstractPlayer {
      ContentSource findSource() { // code from findMusicSource }
      void setEnvironment() { // previous code used to set env }
}
interface ContentSource { Content provideContent(AbstractPlayer requester); }
class MusicSource implements ContentSource {
      Content provideContent(AbstractPlayer requester) {
            Description desc = requester.getPreferences();
            // previous code from provideMusic exploiting desc;
      }
}
interface Content { void play(Environment env); }
class Music implements Content { // same implementation as before }
class Environment { // same implementation as before }
```

3. The Formal Background

The algebraic approach to graph transformation (Corradini *et.al.*, 1997) is the formal basis for our work on refactoring, as graphs are natural means to represent code and model structures. Their modification is performed by applying graph rules. To handle model and code-related graphs in a separate but consistent way, we apply concepts of distributed graph transformation. Finally, the concept of transformation units is used to obtain a global control on structured graph manipulations, useful to specify complex refactorings.

3.1 Graph Transformation

Graphs are often used as abstract representation of code and diagrams, e.g. of UML diagrams. Formally, a graph consists of a set of vertices V and a set of edges E such that each edge e in E

has a source and a target vertex *s(e)* and *t(e)* in *V*, *resp*. Each vertex and edge may be attributed by some data value or object, formally expressed by elements of an algebra on some algebraic signature Σ. The theory of graph transformation is largely independent of the notion of graph, so that one can be chosen which best reflects domain-specific structures. Here, we consider typed attributed graphs. Graph manipulation is performed by the so-called double-pushout approach to graph transformation, *DPO* (Corradini et.al., 1997), based on category theory. Using typed graphs, structural aspects occur on two levels: the type level (modelled by a type graph *T*) and the instance level (modelled by an instance graph). An instance graph is correctly typed if it can be mapped in a structure-preserving manner to *T*, formally expressed by a graph homomorphism.

A graph rule $r: L \rightarrow R$ consists of a pair of *T*-typed instance graphs *L, R* such that the union $L \cup R$ is defined. That means graph objects which occur in both, *L* and *R*, have the same type and attributes and, if edges, the same source and target vertices. The left-hand side *L* represents the pre-conditions of a modification, while the right-hand side *R* shows the effect of the modification. Vertex identity is expressed via names, while edge identity is deduced from the identity of the connected vertices. Additionally, graph rules comprise attribute computations where left-hand sides may contain constants or variables of set *X*, while right-hand sides capture the proper computations, denoted as elements of term algebra $T_{\Sigma(X)}$.

A rule may also contain a set of *negative application conditions (NAC)*, expressing graph parts that *must not* exist for the rule to be applicable. NACs are finite sets of graphs $NAC=\{N_i \mid L \subseteq N_i, i \geq 0\}$, expressing the conjunction of basic conditions, and can refer to values of attributes (Fischer et.al, 1999). For a rule to be applicable, none of the prohibited graph parts $N_i - L$ present in a NAC may occur in the host graph *G* so that this occurrence is compatible with a rule match *m*. A match is an injective graph homo-morphism $m: L \cup R \rightarrow G \cup H$, such that $m(L) \subseteq G$ and $m(R) \subseteq H$, i.e. the left-hand side of the rule is embedded into *G* and the right-hand side into *H*. In this paper we use dotted lines to denote NACs. Non-connected NACs denote different negative application conditions (see Figure 14 for an example). A graph transformation from a graph *G* to a graph *H*, $p(m): G \Rightarrow H$, is given by a rule *r* and a match *m* with $m(L - R) = G - H$ and $m(R - L) = H - G$, i.e. precisely that part of *G* is deleted which is matched by graph objects of *L* not belonging to *R* and symmetrically, that part of *H* is added which is matched by new graph objects in *R*. Operationally, the application of a graph rule is performed as follows: First, find an occurrence of *L* in graph *G*. Second, remove all the vertices and edges from *G* matched by *L - R*. Make sure that the remaining

structure $D= G-m(L-R)$ is still a proper graph, i.e. no edge is left which dangles because its source or target vertex has been deleted. In this case, the *dangling condition* is violated and the application of the rule at match *m* is not possible. Third, glue *D* with *R-L* to obtain graph *H*. A typed graph transformation system *GTS=(T,I,R)* consists of a type graph *T* and a finite set *R* of graph rules with all left and right-hand sides typed over *T. GTS* defines formally the set of all possible graphs by $Graphs(GTS)=\{G|I \Rightarrow^*_R G\}$ where $G \Rightarrow^*_R H \equiv G \Rightarrow_{r1(m1)} H_1 \ldots \Rightarrow_{rn(mn)} H_n = H$ with *r1, ..., rn* in *R* and $n >= 0$. It follows from the theory that each graph *G* is correctly typed.

3.2 Distributed Graph Transformation

Distributed graph transformation (Fischer et. al., 1999) is graph transformation structured at two abstraction levels: the *network* and the *object* level. The network level contains the description of a system's architecture by a *network graph*, and its dynamic reconfiguration by *network rules*. At the object level, graph transformations manipulate local object structures. To describe a synchronized manipulation on distributed graphs, a combination of graph transformations on both levels is needed. A *distributed graph* consists of a network graph where each network vertex is refined by a local graph. Network edges are refined by graph homomorphisms on local graphs which describe how the local graphs are interconnected. Each local graph may be typed differently, only restricted by the fact that an interface type graph is fully mapped to all connected local type graphs. In the following, we use distributed graphs where the network graphs consist of three vertices: for the model, for the code and for their interface. Furthermore, two network edges are needed, starting from the interface vertex and going to the model and code vertices, respectively. The corresponding refinement graphs are called *model graph, code graph* and *interface graph*. The interface graph holds exactly that subgraph which describes the correspondences between the other two local graphs.

A *distributed graph rule r* is defined by a network rule *n* – which is a normal graph rule – together with a set *S* of local rules – graph rules on local graphs – for all those network vertices which are preserved. Each preserved network edge guarantees a compatibility between the corresponding local rules. The rules must also be consistent with common attribute values. In the following, the network rules will always be identical, meaning that the network is not changing.

Two local rule applications on the model and the code graph will be synchronized by applying a common subrule on their interface graph. The rules in the following figures are local rules where we will use grey levels to indicate subrule parts. We introduce two operators to assemble a

distributed rule from local ones: *asOftenAsPossible* means to apply a local rule as often as possible at different matches in parallel, while || just denotes the distributed application of rules.

3.3 Transformation Units

Transformation units (Kreowski *et al.*, 1997) are used to control rule application, with the control condition specified by expressions over rules. The notion is defined independently of any given approach to graph transformation. It just assumes a certain graph transformation approach *A* consisting of a class of graphs *G*, a class of rules *R*, a rule application operator yielding a binary relation on graphs for every rule of *R*, a class *E* of graph class expressions, and a class *C* of control conditions. A *transformation unit* consists of: an initial and a terminal graph class expression (defining valid input and output graphs); a set of rules; a set of references to other transformation units, whose rules can be used in the current one; and a control condition over *C* describing how rules can be applied. Typically, *C* contains expressions on *sequential application* of rules and units, as well as conditions and loops, e.g. applying a rule *as long as possible*. We adopt here the syntax for control expressions presented in (Koch, Parisi Presicce, 2002).

In this paper, we apply transformation units to distributed graph transformation, so that *G* is the class of distributed graphs, *R* the class of distributed rules, and the DPO way of rule application, as defined in (Fischer et.al., 1999), is the rule application operator. We adopt the class *C* described in (Koch, Parisi-Presicce, 2002), while the class *E* is not needed here and can trivially be left empty to indicate that no special initial and terminal graph classes need be specified.

We relate rule expressions to graph rules by giving names to rules and passing parameters to them, to be matched to specific attributes of some vertex. 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.

4. Refactoring by Graph Transformation

We present the general setting of refactoring by graph transformation and analyse a sample refactoring which involves transformation of the code and more than one UML diagram. Furthermore, we show the use of transformation units over distributed graph transformations to enforce synchronization and atomicity of the transformations in different diagrams.

4.1 Graph Representation of Diagrams and Code

The abstract representations of code and UML models are given in the form of graphs, obeying the constraints imposed by a type graph. For the code, we refer to the JavaML definition of an abstract syntax for Java (Badros, 2000), and we consider the type graph provided by its DTD. Indeed, any JavaML document is structured as a tree, i.e. a special kind of graph where an XML element is represented by a typed vertex and its attributes by vertex attributes. The graph edges show the sub-element relation and are untyped and not attributed. We call this graph the *code graph*. For UML (OMG, 2002), the abstract syntax of the UML metamodel provides the type graph to build an abstract representation of the diagram, that we call the *model graph*.

As an example, Figure 7 shows the code graph for class `Audio`. Due to reasons of space, we omit the representation of the fields `ms` and `env` and of the method `findMusicSource`. Similarly, Figure 8 presents the model graph for the class diagram of Figure 5a (without dependencies). Here and in the following figures, only the important fields of model elements are shown. Details of model elements occurring in more than one figure are shown only in one. One can notice that the vertices that would be directly connected to a `class` vertex in the code graph, appear in the model graph as *feature* elements for which the class is an *owner*. Figures 9 and 10 present the components of the model graph for the sequence and activity diagrams of Figure 5.

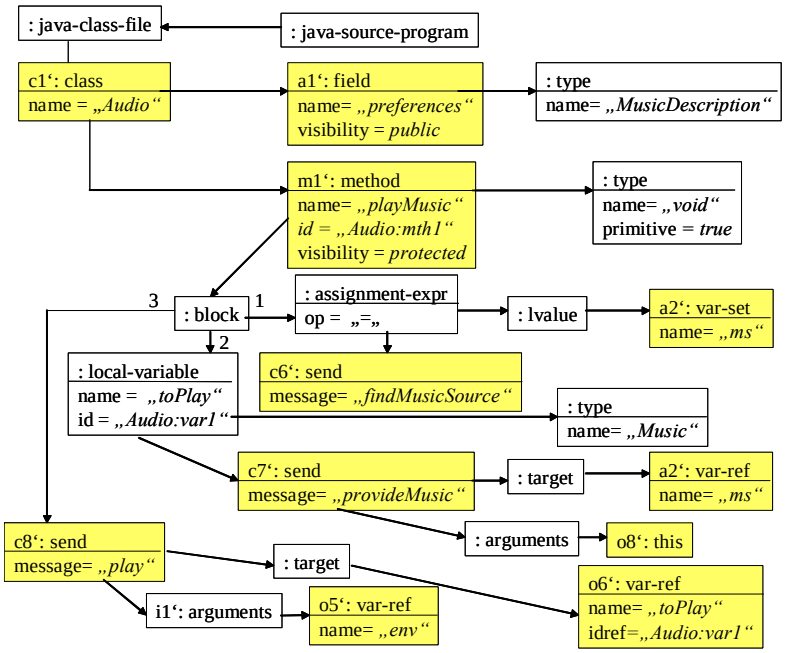


Figure 7. A part of the code graph for the first version of the code of class `Audio`.

The model graphs, though presented here separately, are construed as different views on one

large graph representing the whole model. Indeed, the behavioral diagrams are associated with the model elements which own or contribute to the model's behavior. As an example, object `m1:Method` for `playMusic` appears in Figure 8 as a behavioral feature of class `Audio`, and in Figure 10 as the element whose behavior is defined by the component of the model graph for activity diagrams. Conversely, object `o2:operation` appears as the specification for the same method in Figure 8, and as the operation for a `CallOperationAction` object in Figure 9. In the next section, we will show how transformation units for distributed transformations can be used to modify both graphs, code and model, in a coordinated way. This coordination is based on the existence of correspondences between elements in the two graphs, indicated by similar names (e.g. "c1' " in Figure 7 and "c1" in Figure 8) and further discussed in Section 5.

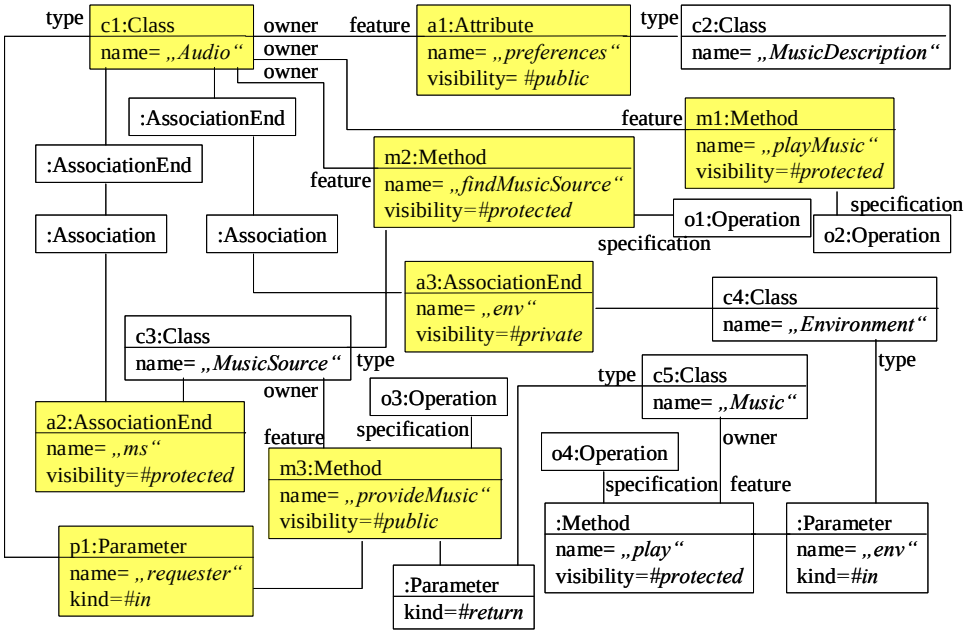


Figure 8. The abstract graph for the class diagram of Figure 5a

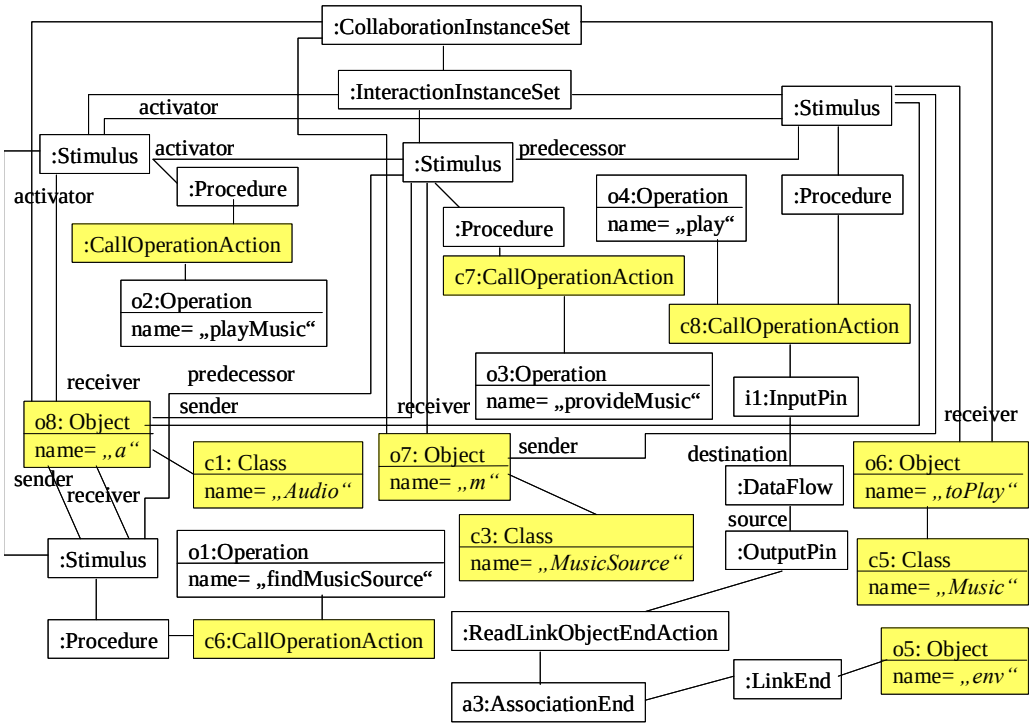


Figure 9. The abstract graph for the sequence diagram of Figure 5b

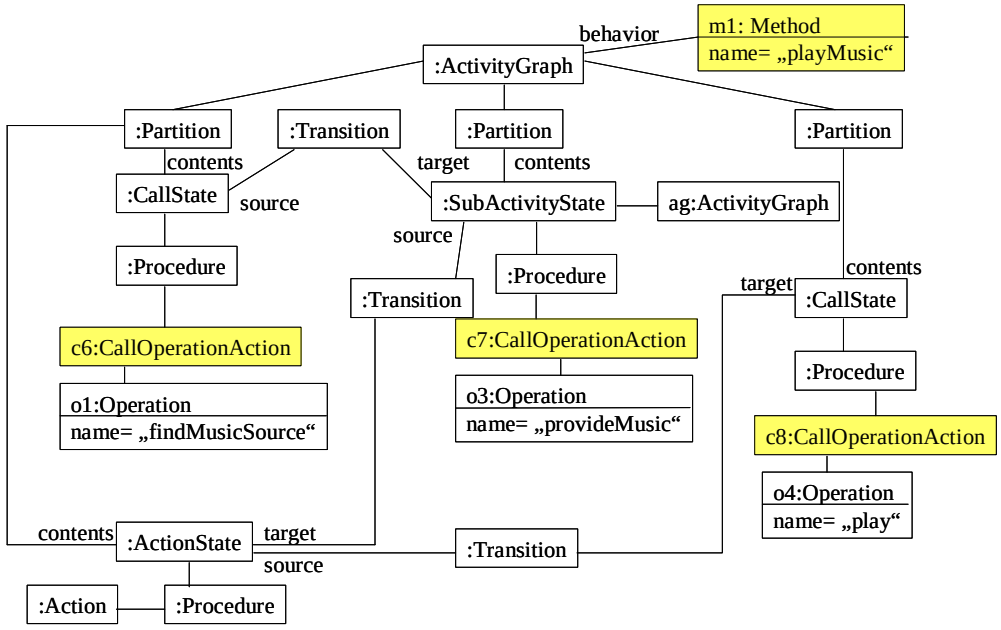


Figure 10. The abstract graph for the activity diagram of Figure 5c for executing `playMusic`.

4.2 Encapsulation of Variables

The preconditions for **`EncapsulateVariable`** require that no method exist in the hierarchy with

the same signature as the setter and getter methods to be created. So, we use NACs on rules transforming the code and model graphs. Neither graph expresses the hierarchy directly, so that the transitive closure of inheritance must be evaluated before checking the precondition. Rule `insert_up_gen_code` in Figure 11 starts the identification of a class ancestors by inserting an edge, labelled `gen`, between a class and its superclass in the code graph. The input to the rule is the name of the class for which the hierarchy is built. Direct subclasses are found by an analogous rule `insert_down_gen_code`. The hierarchy construction is completed by the rule of Figure 12. In all these cases, the dotted `gen` line in the LHS indicates a negative application condition: a `gen` edge must not already exist between the classes to be related.

`insert_up_gen_code(inString cname):`

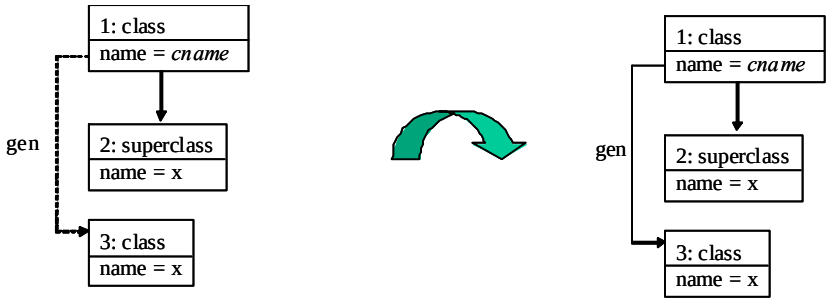


Figure 11. The rule to start the computation of the ancestors of the class named `cname`.

`compute_gen_code():`

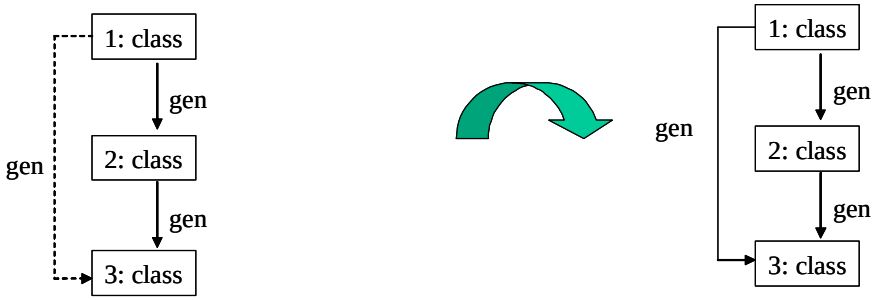


Figure 12. The rule to complete the construction of the hierarchy.

Code graph transformation is now specified by rule `encapsulate_variable_code` in Figure 13, where `cname` identifies the class to be modified, and `varname` the name of the variable to be encapsulated. This rule is complemented by NACs, two of which are shown in Figure 14. These two check the absence of methods with the same signature in the class, while the others check the absence in the whole hierarchy, i.e. in a class associated with `cname` through a `gen` edge.

All uses of the variable are substituted by calls of the corresponding methods. Hence, Figure 15 shows the rule replacing direct access to the variable with a call of the getter, while Figure 16

shows the rule taking care of value updates.

`encapsulate_variable_code(in String cname, in String varname):`

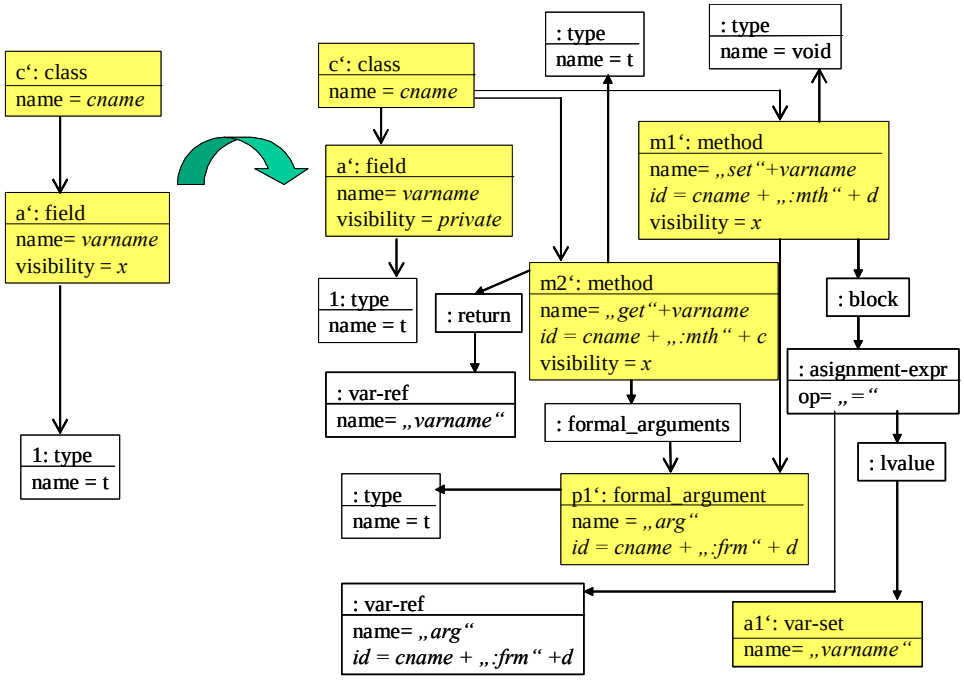


Figure 13. LHS and RHS of the rule for variable encapsulation in the code graph.

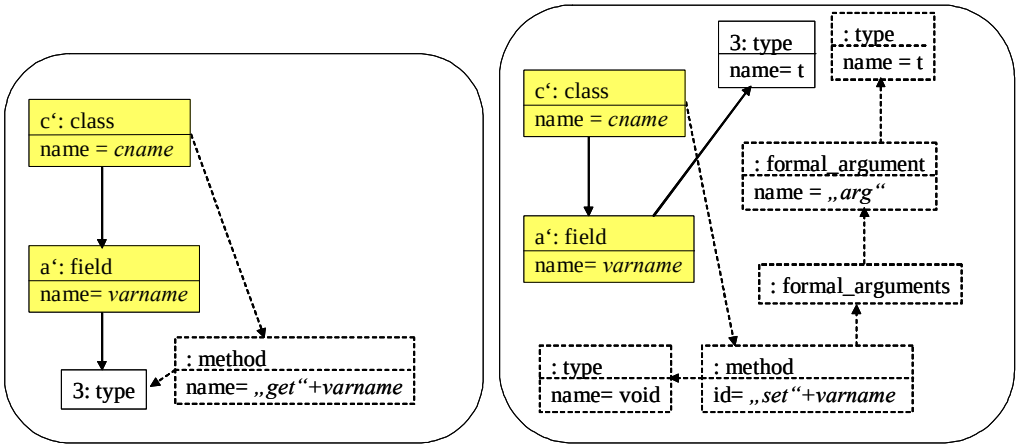


Figure 14. Two NACs for the rule in Figure 13, to check that no method exists with the same signature as the inserted setter and the getter methods.

Consistency between model and code is maintained by applying a set of rules which operate locally on the components of the model graph for the diagrams considered above. Figure 17 shows the `encapsulate_variable_model` rule acting on the class diagram. Negative application conditions analogous to those for the code graphs are also used, guaranteeing a check of the overall consistency of the representations. Consequently, we need to compute the transitive closure of the inheritance relation also for model graphs, through rules `insert_up_gen_model`,

`insert_down_gen_model`, and `compute_gen_model`. Rules `encapsulate_variable_model` and `encapsulate_variable_code` are applied in parallel along their common subrule shown in grey.

`field-access(in String cname, in String varname):`

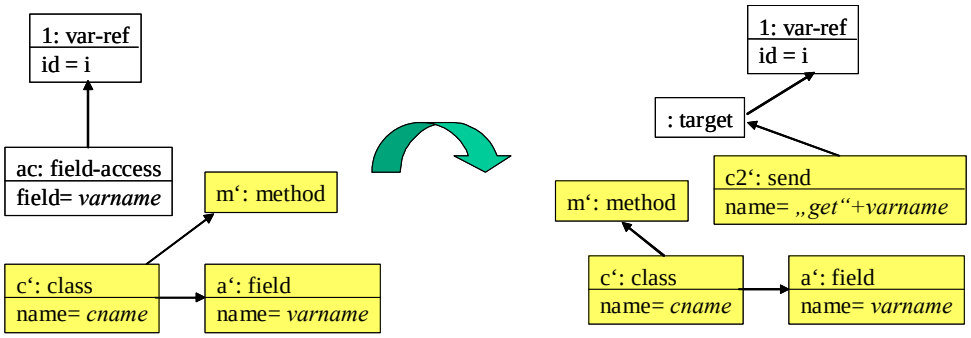


Figure 15. The rule to replace accesses to `varname` in `cname` with calls to the getter.

`field-set(in String cname, in String varname):`

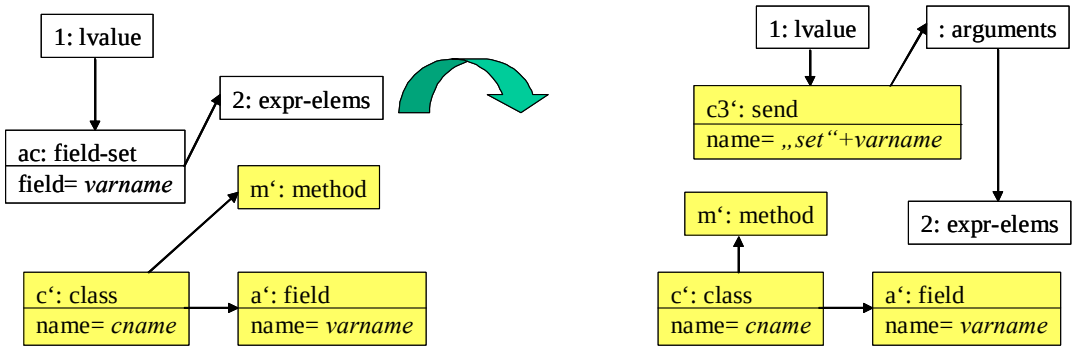


Figure 16. The rule to replace updates of `varname` in `cname` with calls to the setter.

The effect on activity diagrams is shown by the rule `getEncVarInActivity` in Figure 18, replacing variable access with a call of a getter. For simplicity, we omit all input and output pins. We do not present the whole distributed rule, but show only the local rule acting on the model graph. If the variable replacement in the model graph corresponds to some variable replacement in the code graph, all possible rule applications of `getEncVarInActivity` have to be applied synchronously with code rule `field-access` along their common subrule, which is shown in grey. An analogous rule exists for replacing variable updates with calls of the setter method.

Finally, we consider the required modifications for sequence diagrams, for the case of variable encapsulation. Since sequence diagrams do not show read and write actions on attributes, the encapsulation does not directly cause a refactoring. In order to maintain a consistent model, the user has to specify if and where the refactoring should be represented for this part of the model. In particular, whenever a method `m` is called, in which the encapsulated variable is used, it is

necessary to introduce a `Stimulus s'` to call the relevant setter or getter method. From the ordering of subtrees in the code graph of `m`, one can identify the stimulus `s` for which `s'` is the successor (or predecessor) in the new activation sequenc, and pass it as a parameter to the rule. For space reasons, we omit the representation of the relative rule `getEncVarInInteraction`.

The rules in Figures 15, 16, and 18 must be applied at all possible instances of their left hand side in the distributed graphs. There may be several such instances, and we want to apply a transformation in a transactional way, i.e. the overall application is possible only if corresponding parts can be coherently transformed. We use therefore transition units to specify some form of control on the application. In particular, we use the control construct *asOftenAsPossible* which indicates that a local rule must be applied in parallel on all (non conflicting) instances of the antecedent. In particular, contextual elements can be shared by different instances, but no overlapping is possible on elements which are removed or transformed by the rule. Moreover, we use the construct || which indicates the distributed application of two or more rules.

To sum up, `EncapsulateVariable` is expressed by a transformation unit as follows:

```
EncapsulateVariable(in String cname, in String varname):=
   asLongAsPossible insert_up_gen_code() || insert_up_gen_model() end;
   asLongAsPossible insert_down_gen_code() || insert_down_gen_model() end;
   asLongAsPossible compute_gen_code() || compute_gen_model() end;
   encapsulate_variable_code(cname, varname) ||
      encapsulate_variable_model(cname,varname);
   asLongAsPossible field_access(cname,varname) ||
      (asOftenAsPossible getEncVarInActivity(cname,varname) end) end;
   asLongAsPossible field_set(cname,varname) ||
      (asOftenAsPossible setEncVarInActivity(cname,varname) end) end;
```

The user can also decide to request a modification of interaction diagrams. In this case, he or she has to interactively provide a value for the `stimulus` after or before which to place the new call, and the transformation unit is completed by the following construct.

```
asOftenAsPossible getEncVarInInteraction(cname,varname,stimulus) end
asOftenAsPossible setEncVarInInteraction(cname,varname,stimulus) end
```

By applying the transformation unit, both code and model graphs are transformed to reflect the existence and usage of the new methods. As an example, the code graph for the class `Audio` now appears as shown in Figure 19 (the body of `playMusic` is not shown as it remained unchanged). With respect to Figure 7, the visibility of field `preferences` has changed from `public` to private, and two new methods have been inserted, as specified in rule `encapsulate_variable_code`. The graph in Figure 19 is a subgraph of the code graph, obtained by applying the transformation unit

`EncapsulateVariable`, i.e. it results from applying the local rule `encapsulate_variable_code` once with arguments `cname = “Audio”` and `varname = “preferences”`.

`encapsulate_variable_model(in String cname, in String varname):`

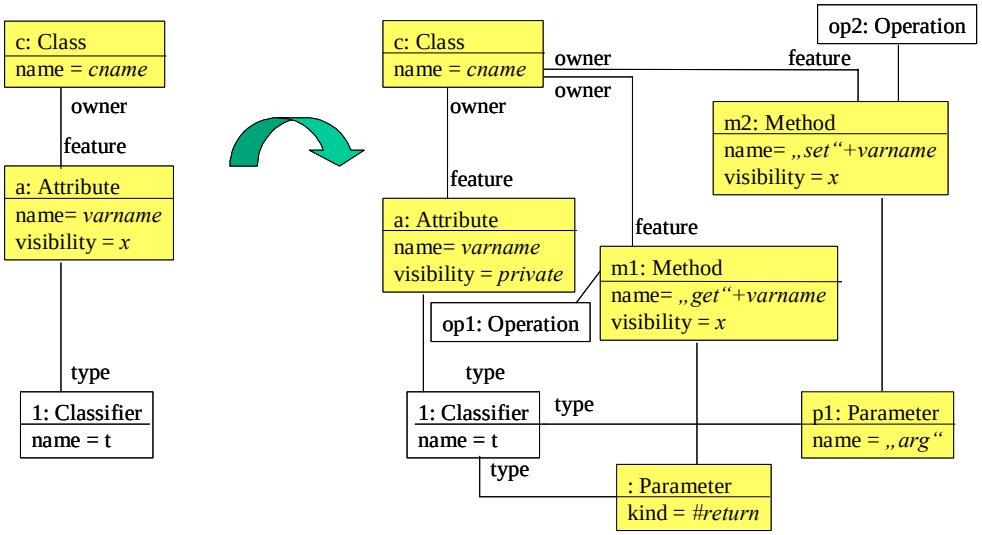


Figure 17. LHS and RHS of the rule for variable encapsulation on the class diagram component of the model graph.

`getEncVarInActivity(in String cname, in String varname):`

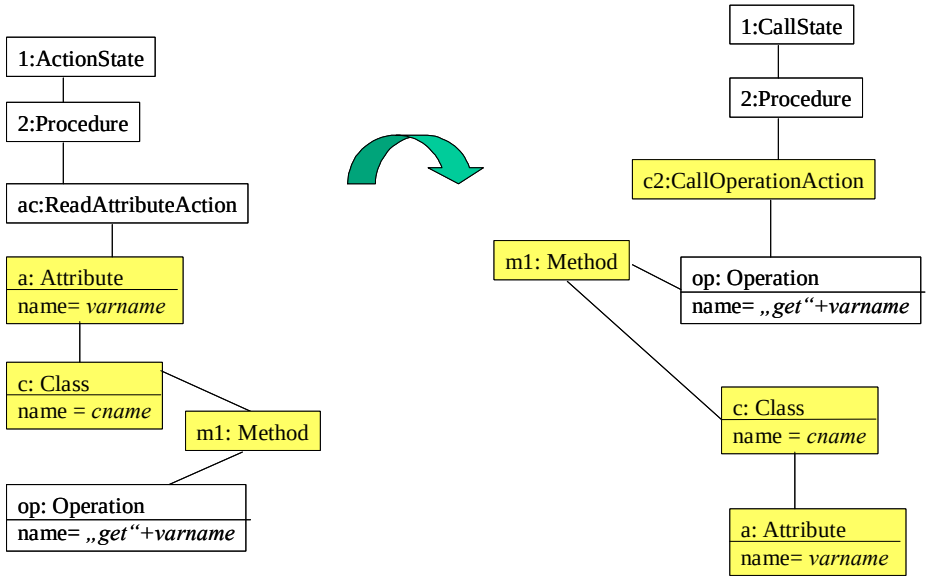


Figure 18. The rule for modifying variable access in activity diagrams.

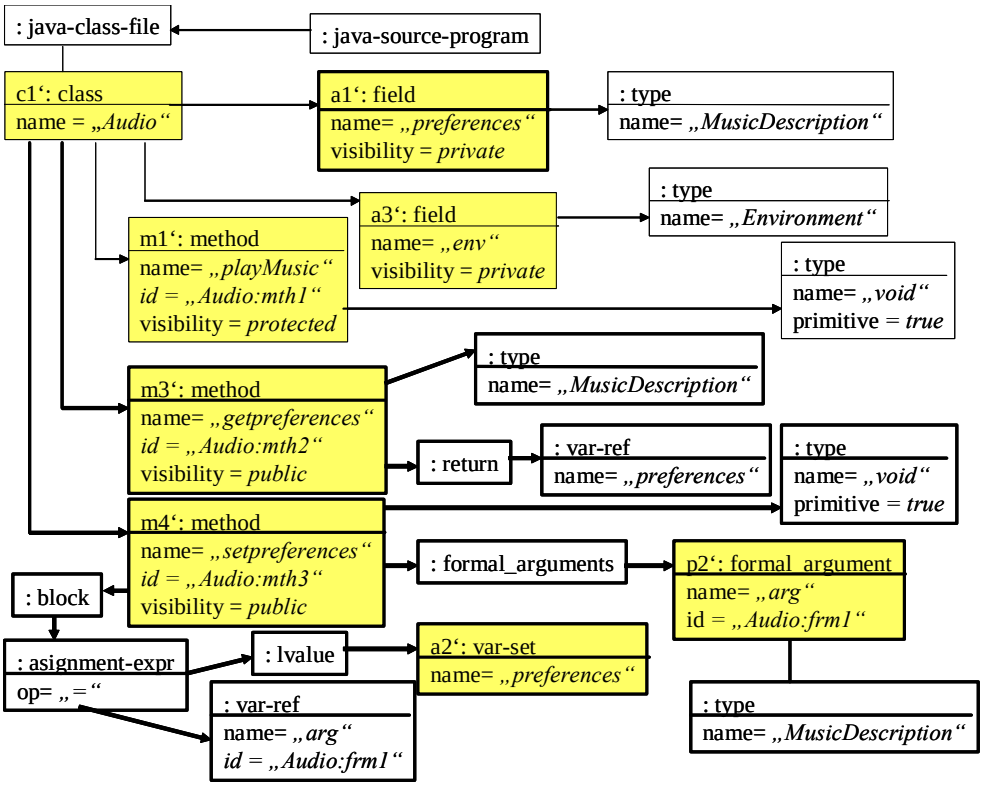


Figure 19. The code graph for class `Audio` after the `EncapsulateVariable` refactoring.

4.3 Extract Method

In the words of Martin Fowler, «If you can do Extract Method, it probably means you can go on more refactorings. It’s the sign that says "I’m serious about this"». Hence, we present our approach to managing this refactoring, without figures due to lack of space. A more detailed version of this specification, but with a different code representation, is in (Bottoni *et al*., 2003).

The precondition that the name for the new method does not exist in the class hierarchy is checked as for variable encapsulation. In general, we can assume that the code and model graphs are complemented by all the needed `gen` edges. The precondition that the code to be extracted constitute a *block* is easily checkable on the code graph. Indeed, this code can be a whole subtree rooted in a `block`, `if`, `switch`, `loop`, `do-loop` vertex, or be a collection of contiguous subtrees of a same `method` vertex, composed of `stmt-exprs` not comprising any construct `try`, `throw`, `return`, `continue`, `break`, `synchronized`, and such that no `label` appears in them.

We then need to identify all the variables to be passed to the new method. We can inspect the code graph to identify all the `var-set` and `var-ref` elements where the name of the variable is not the name of a `formal-argument` of the original `method` or a `name` for a `local-variable`

declaration present in the subtree to be moved. Additionally, if the subtree presents some `local-variable` vertex, we have to check that there are no `var-set` or `var-ref` elements for that variable, in the subtrees remaining with the original `method`. The creation of the call for the new method is achieved by substituting the removed subtrees with a `send` element which has the name of the new method as value of the attribute `message`, target `this`, and the list of `formal-arguments` as derived before. In the model, we modify the class diagram by simply showing the presence of the new method in the class, as the effects on the referred variables and the existence of a call for this method are not reflected at the structural level.

For the activity diagrams, we need to identify the `Action` associated with a given `Operation`. Such an `Action` can be further detailed through a collection of `Actions` associated with it. So, we need to identify all those vertices which correspond to roots of the moved subtrees, detach them from the description of the `Operation`, and exploit them to create the description of the `Operation` associated with the new method.

For interaction diagrams, the situation is different from that for variable encapsulation. The system can identify the existing instances of `Stimulus` which occur before and/or after the extracted code, and insert the `Stimulus` to a `CallOperationAction`, for an `Operation` with the name of the new `Method`, on the same receiver as the sender. Moreover, each instance of `CallOperationAction`, originating from the original `Operation` instances, and related to a vertex in the extracted subtrees, must now be related to an instance of `Stimulus` whose activator is the `Stimulus` for the new `Operation`. The existing `predecessor` and `successor` associations for the first and last such instances of `Stimulus` must be transferred to the new `Operation`. All these transformations must be applied as often as possible, so as to affect all the descriptions of the behavior of the refactored method. Indeed, calls to such methods can occur in different scenarios, meaning that the sequence diagrams for all such scenarios must be modified.

5. Building Correspondences between Code and Model Graphs

In order to manage distributed transformations involving the Abstract Syntax tree (AST) (viewed as a graph) and the graph representing the UML model, we need to establish an interface graph *IG* and two morphisms μ_{AST} and μ_{UML} from it to the two graphs. This requires the construction of a correspondence between types of vertices in the two graphs. To this end, we adopt for AST the concrete representation given by JavaML, an XML-based specification, while the graph for the

UML model is constructed in accordance with the UML metamodel.

In this section, we sketch the principles directing the construction of the correspondences, as the complete construction is beyond the scope of (and the space allowed for) this paper. In particular, we consider the structural and behavioral aspects of the specification separately.

From the structural point of view, we can proceed top-down and see that JavaML `class` vertices correspond to UML `Class` vertices, and a JavaML `field` to a UML `Attribute`. However, care must be taken in managing aspects of the structural definition involving relations to other classes. For example, the subclass relation is represented in UML by the presence of a pattern involving two `Class` vertices, a `Generalization` vertex, and two associations relating thee latter vertex to the other two, one with the role of `specialization`, the other being a `generalization`. In JavaML, a `superclass` vertex, with a `name` attribute, constitutes a leaf of the tree rooted in the `class` vertex. In such a case *IG* would contain only a `CLASS` vertex mapping to a `class` vertex in AST through μ_{AST}, and to a `Class` vertex in UML through μ_{UML}. The definition of the morphisms requires checking that the superclass relation is consistently represented in the two graphs. A similar situation occurs for the `implements` construct.

As concerns behavioral aspects, the `method` vertex in JavaML contains all the information present in the code to characterize the method, in particular its signature and its body. However, in the UML metamodel, this information is distributed across an `Operation` vertex, maintaining information about the signature, and a `Method` vertex which simply contains the code of the method body. As regards the signature, similarly to before, we relate `method` and `Operation` vertices, and we check the agreement of the type information, without associating the `type` subvertices for `method` to the `Classifier` vertices describing those types in UML. This is due to the fact that a type vertex is present in JavaML every time it is necessary, but need to be present only once, and associated with other vertices an arbitrary number of times, in a UML diagram

As we are interested in modelling not only the static declaration of a method, but also its behavior, as modelled in collaboration, sequence, state, or activity diagrams, we recur to *action semantics* as defined in (OMG, 2003). Here, a `Method` is associated with a `Procedure`, which has a `Composition` relation with an `Action` vertex. We put such an `Action` in correspondence with the `stmt-elems` vertex, usually a `block`, which is the root of the subtree for the description of the `method` vertex. In general, we want to put into relation semantically equivalent elements, so we will consider the different types of `Action` that can be associated with `stmt-elems`. A major

difference exists, though. The JavaML file presents the `stmt-elems` of a `block` in an order which corresponds to the sequence of statements in the original code. The UML model on the other hand, does not require that an order is specified for independent actions. Control flow actions indeed exist, such as `ConditionalAction` or `LoopAction`, and also idioms such as `Iteration` can be expressed. However, actions not related through some chain of `DataFlow` objects, need not be realized in any given order. If desired, though, the modeller can prescribe the existence of `ControlFlow` objects, defining `predecessor` and `successor` `Action`s.

The process of building such correspondences, i.e. of introducing elements in the interface graph and establish the morphisms from this to the code and model graphs can be modelled by rewriting rules. Figure 20 shows two local rules whose distributed application on the code and model graph, respectively, produces the following effect: if there are, both in the code and model graph, elements representing a class `s` which is a superclass for a class `c` whose representations in the two graphs have already been put in correspondence, as witnessed by the identifiers `c1` and `c1′` for the two instances, then the two representations of class `s` are put in correspondence, as witnessed by the production of the identifiers `c1` and `c1′`.

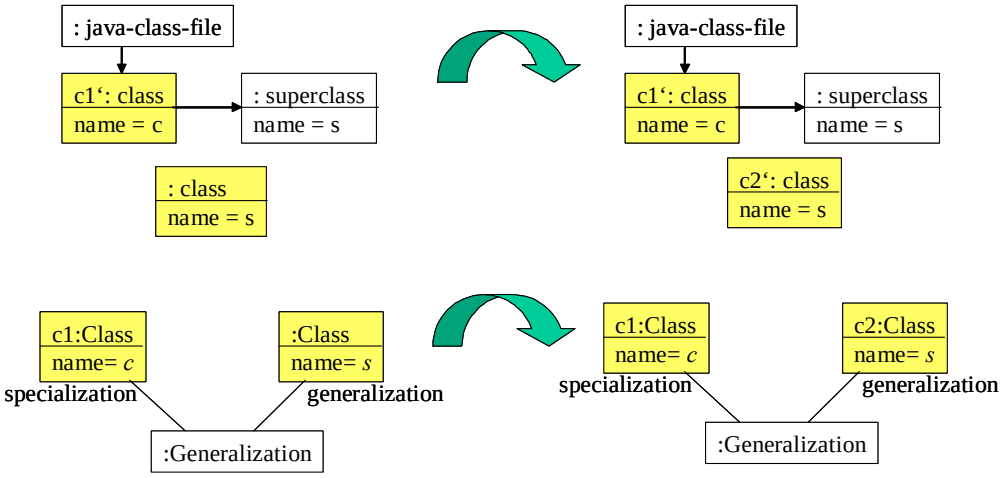


Figure 20: Two rules to establish correspondences concerning class inheritance.

6. Behavior Preservation in Refactoring

The refactorings in Section 4 are historical ones, whose behavior preservation properties are widely discussed. In general, it is important to have a methodology to formally verify properties of (new) refactorings. The approach based on graph transformations provides a formal basis to perform these checks and to generate conditions ensuring the desired preservation properties. We

restrict our discussion to checks that can be carried out statically, not based on flow analysis.

Several kinds of behaviour preservation are relevant to the refactorings studied here. Some of them are briefly described next (Mens *et al*. 2002):

- *Type preservation* is the behaviour exhibited by a refactoring that does not change the type of any entity not deleted.
- *Access preservation* requires that the refactoring maintains the access of a method to at least the same variables accessed before the refactoring. The access may now be through one or more intermediate method calls.
- *Update preservation* occurs when each method produces, after the refactoring, the same variable changes produced before the refactoring.
- *Call preservation* is the behaviour exhibited by a refactoring when each method causes the execution, after the refactoring, of at least the same methods called before the refactoring.

The refactorings presented here are based on typed graph transformations. Since these transformations always guarantee that all the resulting graphs are correctly typed over the same type graph, type preservation (according to the type graph) is always exhibited by our refactorings. No additional verification is needed.

This kind of behaviour preservation is not sufficient to ensure that the resulting graph is an acceptable code or model graph. Well-formedness constraints are needed to rule out undesired configurations of the produced graph (instance of the type graph). For example, we have seen the requirement that no names, whether for variable or method, are in conflict in any class.

Now, in order to verify that these constraints are satisfied after the application of refactoring rules, we can enlist the help of established results on consistent graph transformations in (Heckel *et al*., 1995). The approach there consists of considering a subgraph forbidden from the result of the application of a rule and deriving from it (by applying the rule backwards) a number of NACs for the rule, preventing the application of the rule if it causes the construction of the forbidden subgraph. The construction in (Heckel *et al*., 1995), applied to the above constraint and to the rule for variable encapsulation in Figure 13, yields NACs of the type shown in Figure 14.

Not all constraints can be expressed by such simple ‘forbidden’ graphs. More general constraints can be defined by using propositional logic (Matz, 2002) to compose ‘atomic’ constraints, formed by simple forbidden graphs, and injective graph morphisms describing the conditional existence of graph (sub)structures (Koch, Parisi Presicce, 2002),.

For example, to express the fact that no method can have the same name and the same formal argument as another method in the same class, we can write the proposition `NOT (constraint_parameter AND NOT two_parameter_constraint)`, where the two graphs are presented in Figure 21. This proposition is satisfied only if it is not the case that the two methods named `mnew` have each exactly one parameter of the same type.

Another type of constraints, refactoring-specific, addresses the problem of unwanted side effects of a refactoring. These constraints can be expressed with pre- and/or post conditions to the refactoring: with the latter, if the post-condition is not met, the transformation must be 'undone' and the previous model restored, while with the former (more efficient) method, application conditions are checked to prevent the transformation by a refactoring if it produces unwanted effects. For example, a new method `m` defined in class `C` should not override an existing method `m` with the same signature in a subclass of `C`, or be overridden by an existing method with the same signature defined in a superclass of `C`. This constraint is needed, for example, in both sample refactorings presented in Section 4. The graph `constraint_parameter_gen` in Figure 22, must be used, together with the version for inheritance of `two_parameter_constraint` above, to express the constraint for the case of a method with exactly an argument. When the name of the method (`mnew`) is either `set+varname` or `get+varname`, this technique can be used to produce the appropriate NACs for the rule in Figure 17. If we know the maximum number of arguments for a method in the code, we can construct similar forbidden configurations for each number of arguments. Otherwise, we have to define graph schemes, typically using set vertices (Bottoni *et al.*, 2000), to be matched to any number of arguments.

`constraint_parameter` `two_parameter_constraint`

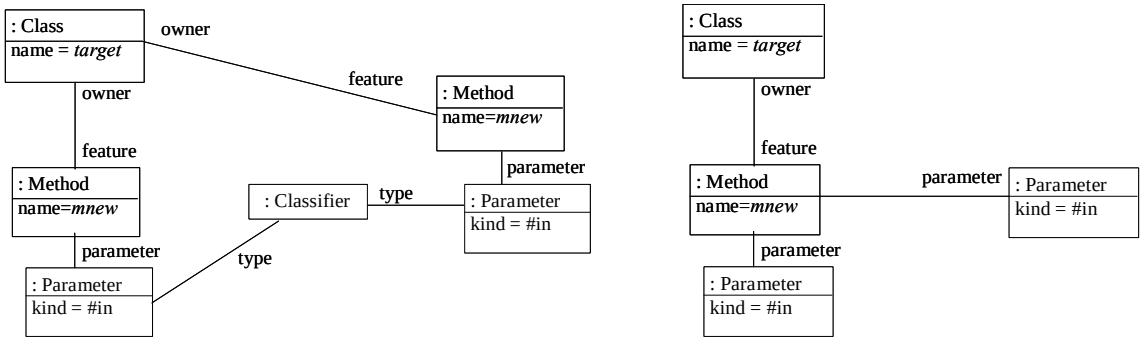


Figure 21. The two graphs to express the constraint for methods with only one parameter.

The above techniques can add NACs individual rules to obtain the desired effect. But what about transformation units? As an example, `EncapsulateVariable` requires the application of

different rules, some of them `asLongAsPossible`. It is possible to prove that this refactoring has a "functional" behaviour, i.e., that it always produces a resulting graph (termination of the sequence of transformations), which does not depend on the order in which some of the elementary (even non behaviour-preserving) transformations are applied. The latter property is called *confluence* and can be argued as follows, without giving a formal proof, by relying on results for attributed graph transformations, easily adaptable to the distributed ones. The possible choices inside the transformation unit depend on the possibility of applying rules in parallel. Rules `insert_up_gen_code` and `insert_up_gen_model` are sequentially independent since they only add elements to the graph; therefore both sequential orders of application produce the same effect as their simultaneous application. The construct `asLongAsPossible` does eventually terminate because each application reduces the number of pairs of vertices not connected by an edge labelled `gen`. A similar argument applies for the `insert_down` and `compute` rules, while for the rules `encapsulate_variable`, the application is required to be simultaneous, overlapping on the shaded entities. In general, by adapting to distributed graph transformations existing results on attributed graph transformations, critical pair analysis could be applied after taking the control structure in the transformation units into account, i.e. ignoring pairs of rules in sequential segments and considering only pairs of rules which may be applied in parallel. Termination can be proved in general by finding a well-founded ordering associated to the application of the rules indicating that the rules obey some layering conditions (Bottoni *et al.*, 2000).

`constraint_parameter_gen`

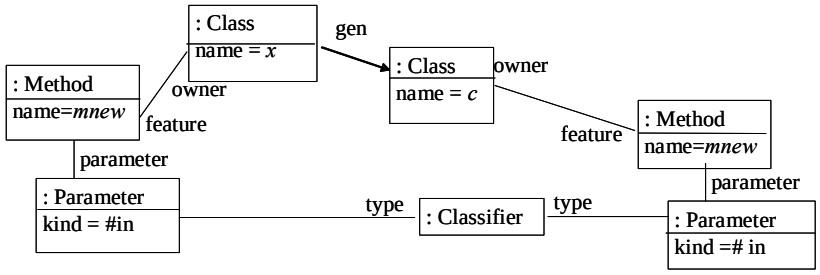


Figure 22. Forbidden graph for avoiding overriding between existing methods and a new method.

Historically, two factors have contributed to the limited use of formal methods to verify the preservation of behaviour in refactoring. One factor is due to the need of defining a formal semantics for the target language, and a formal semantics is not always available for real complex languages such as C++. The other reason is the need to automate refactorings, which requires a significant engineering effort to build appropriate tools to manipulate ASTs and UML diagrams.

Our approach alleviates in part the second problem, since tools for graph transformations, such as AGG (Ermel *et al.* 1999), are available to deal with attributed typed graph transformations with pre- and post- conditions (Matz 2002).

7. Conclusions

We have presented an approach, based on graph transformation, to maintain consistency between code and model diagrams in the presence of refactorings. The approach allows the coordinated transformation of two graphs representing the abstract syntax, as derived from the code by a parser, and the UML model of the software system. A correspondence is established between these two graphs, starting from the correspondence between types of vertices in the abstract syntax trees, as defined by the JavaML markup language, and types of elements and associations in the UML diagrams, as defined by the UML meta-model.

Although the approach has been concretely demonstrated using the Java language and the JavaML coding of the abstract syntax, it can be applied to any type of abstract syntax for object-oriented languages, provided that a non-ambiguous correspondence between the abstract syntax and the components of the UML model can be established.

As a consequence, an integrated tool can be devised, able to perform refactoring on code and model diagrams, so as to maintain the original correspondences between these components. This would require the integration of the current ability of modern refactoring tools to manipulate ASTs, with a more general interpreter for transformation units. Indeed, it is not needed that the tool exploits graph transformations in order to manipulate the tree. As all refactorings are individually described by a transformation unit, and a tool has a finite number of them available, it is sufficient that the tree transformation is wrapped so as to communicate the parameters which characterize it to the other parts of a distributed transformation. If the transformation occurs on a part of the code for which the corresponding parts of the model have been identified, the relevant modifications would automatically be performed.

The opposite process could also be envisaged in which a refactoring of the model would reflect into a modification of the corresponding code. This can be easily performed on structural diagrams, for which we have seen that there is a close correspondence between elements of JavaML and of the UML meta-model. Future work will have to identify refactorings in the behavioral diagrams for which it is possible to identify the needed transformations in the code.

8. References

Badros, G. (2000). JavaML: A Markup Language for Java Source Code. 9th Int. World Wide Web Conference. JavaML-Homepage: http://www.cs.washington.edu/homes/gjb/JavaML.

Bottoni P., & Schuerr A., & Taentzer, G., (2000), Efficient Parsing of Visual Languages based on Critical Pair Analysis (and Contextual Layered Graph Transformation)", *IEEE Symposium Visual Languages 2000*, pp.59-61, 2000.

Bottoni, P., & Parisi Presicce, F., & Taentzer, G., (2003). Specifying Integrated Refactoring with Distributed Graph Transformations. In Proc. AGTIVE'03, to appear.

Corradini, A., & Montanari, U., & Rossi, F., & Ehrig, H.,& Heckel, R. & Löwe, M. (1997). Algebraic approaches to graph transformation part {I}: Basic concepts and double pushout approach. In Rozenberg, G., editor, Handbook of Graph Grammars and Computing by Graph transformation, Volume 1: Foundations, World Scientific 163--246.

Fischer, I., & Koch, M., & Taentzer, G. & Volle, V. (1999). Visual Design of Distributed Systems by Graph Transformation. In Ehrig, H., Kreowski, H.-J., Montanari, U. & Rozenberg, G. (eds.). Handbook of Graph Grammars and Graph Transformation, Volume 3: Concurrency, Parallelism, and Distribution, 269 – 340.

Fowler, M. (1999). Refactoring: Improving the Design of Existing Programs. Addison-Wesley.

Heckel, R., & Wagner A. (1995). Ensuring Consistency in Conditional Graph Grammars: A Constructive Approach, in Proceedings SEGRAGRA'95 Graph Rewriting and Computation, Electronic Notes in Theoretical Computer Science vol.2, 1995 - http://www.elsevier.nl/locate/entcs/volume2.html

Koch, M., & Parisi Presicce, F. (2002). Describing policies with graph constraints and rules, in: Corradini, A., Ehrig, H., Kreowski, H.-J. & Rozenberg, G. (eds). Graph Transformation, LNCS 2505, Springer, 223—238.

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

Matz, M. (2002). Design and Implementation of a Consistency Checking Algorithm for Attributed Graph Transformation. (in German) Diploma Thesis, Technical University of Berlin.

Mens, T. (2000). Conditional Graph Rewriting as a Domain-Independent Formalism for Software Evolution. In Proc. AGTIVE'99, vol. 1779 of Lecture Notes in Computer Science, 127-143 Springer-Verlag.

Mens, T. (2001) Transformational Software Evolution by Assertions. Proceedings of theWorkshop on Formal Foundations of Software Evolution. 67-74

Mens, T., & Demeyer, S., & Janssens, D. (2002). Formalising Behaviour Preserving Program Transformations. In Proc. IGCT2002, vol. 2505 of Lecture Notes in Computer Science. 286-301. Springer-Verlag.

Metz, P., & O´Brien J., & Weber W, Use Case Model Refactoring: Changes to UML's Use Case Relationships, Internal Research Report, Department of Computer Science, Darmstadt University of Applied Sciences, July, 2002

Niere, J., & Wadsack, J.P., & Zündorf A. (2001).Recovering UML Diagrams from Java Code using Patterns, Proc. of the 2nd Workshop on Soft Computing Applied to Software Engineering. Fujaba-Hompage: http://www.uni-paderborn.de/fachbereich/AG/schaefer/ag_dt/PG/Fujaba

OMG (2003). UML Spcification 1.5. http://www.omg.org/uml

Opdyke, W.F. (1992). Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois,. Available at: ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdykethesis.ps.Z

Roberts, D. (1999) Practical Analysis for Refactoring, Ph.D. Thesis, University of Illinois at Urbana-Champaign,.

Sunyé, G., & Pollet, D., & LeTraon, Y., & Jézéquel, J.-M. (2001) Refactoring UML models. In: Proc. UML 2001 vol. 2185 of Lecture Notes in Computer Science, 134-138 Springer-Verlag.