Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 211 (2008) 87–98

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Maintaining Coherence Between Models With Distributed Rules: From Theory to Eclipse

Paolo Bottoni[a] Francesco Parisi-Presicce[a,c] Simone Pulcini[a]
Gabriele Taentzer[b]

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

[b] *Technische Universität Berlin - Germany*

[c] *George Mason University - USA*

Abstract

Integrated Development Environments supporting software and model evolution have to deal with the problem of maintaining coherence between code and model despite changes which may occur on both sides. Rather than going through model reingeneering or code regeneration, it would be better to build a full correspondence between the starting models and keep it updated in an incremental way after each evolutionary step. In a series of previous papers, it was shown how distributed graph rewriting could support such updates. Here, we show how to construct a distributed graph from individual models, through the use of synchronized rules. In particular, we discuss the case of Java code and UML models, and propose an Eclipse implementation of the approach.

Keywords: Distributed graphs, model morphism, software evolution.

1 Introduction

Integrated Development Environments (IDEs) are increasingly devoted to enable their users to move through the different processes of design and implementation, providing tools to keep some form of coherence between the design models and the produced code. In particular, several tools support refactoring, usually providing the possibility of combining simple refactorings into complex ones, managing aspects such as assessment of preconditions and modifications of model components, typically class diagrams.

In previous papers we have made the case for keeping into account other views of the design model, such as sequence and state diagrams, and have proposed the use

⋆ Work supported in part by the European Community's Human Potential Programme under contract HPRN-CT-2002-00275, SegraVis and by the European Network of Excellence Interop

doi:10.1016/j.entcs.2008.04.032

of distributed graph rewriting [3,4] for an integrated management of modifications in the code and in the global UML model underlying a software artifact. The approach is based on identifying mappings between software elements, represented by an Abstract Syntax Tree (AST) derivable from the code, and model elements, expressed in UML terms. Both AST and UML models are seen as instances of their respective metamodels, interpreted as graph types. In this context, the construction of the correspondence between them amounts to that of their (typed) interface graph. In such a graph, each node corresponds to some abstract concept common to the two models. At the instance level, morphisms between nodes in the interface graph and the corresponding nodes are constructed.

In this paper, we show how to construct the interface graph and the associated morphisms, based on the assumption that the two models (AST and UML) already exist and are coherent in the sense that elements with the same (qualified) name refer to the same concept. The approach can be easily extended to the case of two incoherent models, so that reasons for failure can be identified. On the other hand, by assuming one of the two models as correct, repair actions can performed on the other one.

In particular, we express the sequences of actions performing the morphism construction as transformation units [11,2], which are specializations of a general transformation pattern and illustrate how such specializations can be generated. We also present guidelines for implementing the rules using the Eclipse API system [6]. The discussion is illustrated by presenting transformation units for the construction of mappings between some particular types.

The rest of the paper develops as follows. After a brief recall of Theory in Section 2, we present the general pattern of transformation some of its specific instantiations in Section 3. Section 4 presents the Eclipse implementation and conclusions are given in Section 5.

2 Theory and models overview

For correspondence construction, we rely on the DPO approach [5], and in particular, to the theory of *distributed graphs and graph transformation* [13], allowing the concurrent construction of the interface graph, of the morphisms between it and individual graphs, and of morphisms between corresponding nodes in the different graphs, so that diagrams such as the one of Figure 1 commute. Figure 1 also illustrates the convention adopted in the rest of the paper: corresponding nodes are identified by the same name, primed in the code graph and doubly primed in the UML graph. This allows us to deal with the existence of morphisms and of a node with corresponding name in the interface graph implicitly.

In the examples of the paper, we show pairs of local rules working in a synchronized manner. Rules are defined on the metamodels specifying the type graphs for the two models.

A *rule* $p : L \xleftarrow{l} I \xrightarrow{r} R$ is given by two morphisms l and r. Given an object G and a rule $p : L \xleftarrow{l} I \xrightarrow{r} R$, a *match* of p to G is a morphism $m : L \to G$. A *direct*

$$\begin{tikzcd}
 & p \arrow[dl, "\mu'"'] \arrow[dr, "\mu''"] & \\
p' \arrow[rr, bend left, "c2u"] & & p'' \arrow[ll, bend left, "u2c"]
\end{tikzcd}$$

Fig. 1. The general form of morphisms.

derivation d from G to H by p and match m, $d : G \Rightarrow_{p,m} H$, is given by a double pushout (see Figure 2). Rules may have application conditions, both positive and negative (NACs), as well as attribute evaluation actions associated. In Figure 2, the NAC is an object N and an injective total morphism n; a rule is applicable only if match m cannot be extended to m' such that $n \circ m' = m$. Several objects N_i, and the associated morphisms n_i, can be associated with one L, indicating that no extension of m should exist for any i. The *derived rule* from a direct derivation $d : G \Rightarrow_{p,m} H$ is $p_d : G \overset{g}{\leftarrow} D \overset{h}{\rightarrow} H$.

$$\begin{tikzcd}
N \arrow[dr, "m'"'] & L \arrow[l, "n"'] \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & I \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] \arrow[dr, phantom, "(2)"] & R \arrow[d] \\
 & G & D \arrow[l, "g"] \arrow[r, "h"'] & H
\end{tikzcd}$$

Fig. 2. Double Pushout rule with a negative application condition.

A transformation unit controls rule application through control conditions specified by expressions over a set $Names$ of rule names. The class $\mathcal{C}$ of *control expressions* is recursively defined by

(i) $Names \subseteq \mathcal{C}$,

(ii) **forall** n **end** $\in \mathcal{C}$, if $n \in Names$,

(iii) $C_1; C_2 \in \mathcal{C}$, if $C_1, C_2 \in \mathcal{C}$,

(iv) **asLongAsPossible** C **end** $\in \mathcal{C}$, if $C \in \mathcal{C}$,

(v) **if** B **then** C **end** $\in \mathcal{C}$, if $C \in \mathcal{C}$,

where B is a logical expression constructed using the logical operators OR and AND on atoms of the form `applicable(`r`)`, with r a named rule and `applicable` a predicate which evaluates to true only if r is applicable in the current graph. If an expression consists of a name $r \in Names$ only, the rule with name r is applied to the current host graph. The operator in (ii) applies the rule with name n at all different matches in parallel to the same host graph. The operator ; is left associative and applies first the expression C_1 and then the expression C_2. The operator in (iv) sequentially applies expression C as long as its application is possible. The operator in (v) prescribes the execution of the expression C conditioned on the success of B (typically this will contain names of rules to be applied first). Transformation units have a transactional interpretation, i.e. they either succeed or fail completely.

In this paper we exploit the metamodels resulting from the definition of the abstract syntax of the Java language, as per the JavaML DTD [1], and the UML Metamodel [9].

3 Correspondence Construction

In this section, we illustrate the approach to the construction of the correspondence, by showing a general template for the used transformation units and illustrating it by an example. The complete construction is described in [12]. While the identification of corresponding elements is based on type and name identities, the main problem lies in the identification of the context, i.e. the *namespace*, in which to check identities. A general search template has therefore been specifically devised to address this problem.

In general, we consider the Java AST as the basis for the construction process, so as to exploit the facilities for tree visit provided by Eclipse. For the sake of simplicity, a slightly abstract form of Java and UML model elements are used in the rules. Where necessary, adaptations of the rules to the real metamodels are discussed.

Templates for Correspondence Construction

In several situations, establishing a correspondence between elements requires recognizing the correspondence of the embedding contexts. In particular, we rely on the notion of parenthood as provided by the tree model. As the number of sibling elements is arbitrary, we adopt transformation units to force an exhaustive search of such elements.

In particular, we observe that a common structure exists for transformation units to build correspondences between elements in a well defined pattern. A correspondence can be established between elements so that the element p' in the AST is the root of some subtree, and children of p' correspond to elements which are linked according to some suitable association with p''.

We can therefore define a template for transformation units to be properly instantiated with a suitable set of rules to resolve the correspondence for a specific pattern. The transformation unit is constructed from 4 basic steps.

Step 1 : Identify the corresponding parent elements to ensure the presence of a context for the rest of the transformation unit.

Step 2 : The construction of the correspondence for children of a mapped element requires a mapping for each corresponding pair of children. Hence, the rule establishing the correspondence has to be applied in the context of the parent and to each different pair of corresponding children.

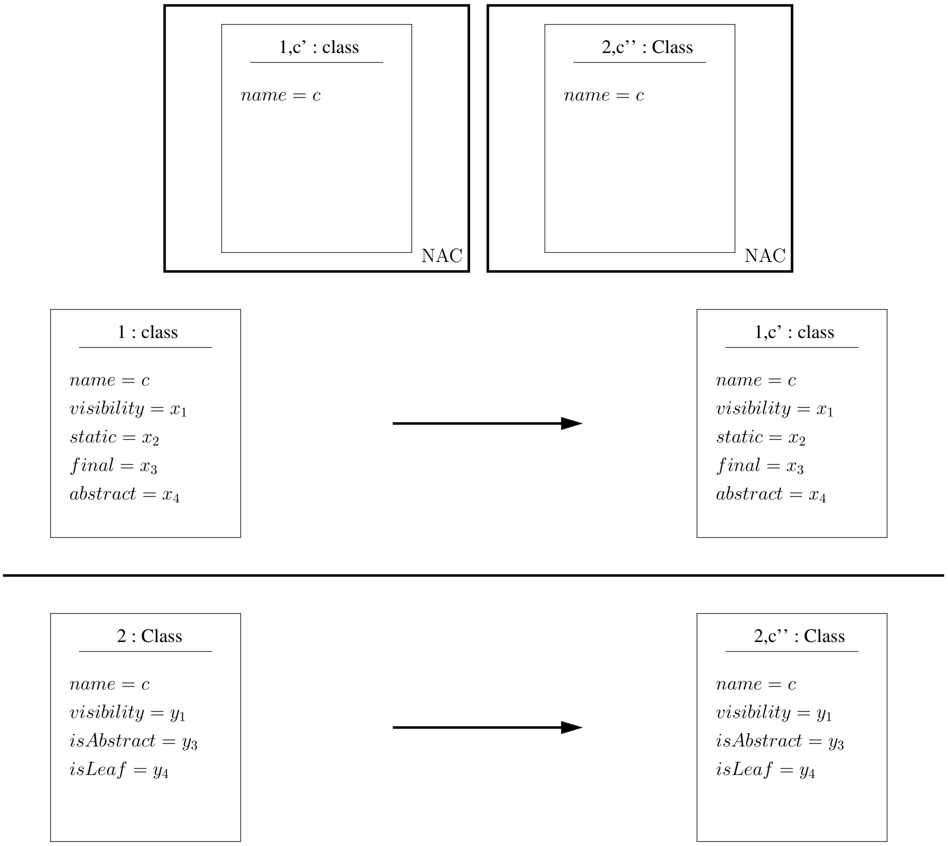
The template for transformation units is expressed as

```
CorrespondenceConstruction()
forall mapParent();
forall mapChild() end
```

This template can be compared to amalgamated graph transformation as presented in [13].

Sample Correspondence Construction

Now we illustrate the specialization of the template presented above to study the case in which the context is a Java class declaration; as stated in JavaML DTD, together with zero or more field declarations in its scope. The construction of the mapping between a Java *class* and a UML *Class* is realized by the rule `mapClass()` in Figure 3, an example of instantiation of `mapParent()`, while the construction of mappings between Java *fields* and UML *Attributes* requires the instantiations of `mapChild()` in the form of `mapField2Attribute()`, as shown in Figures 4.



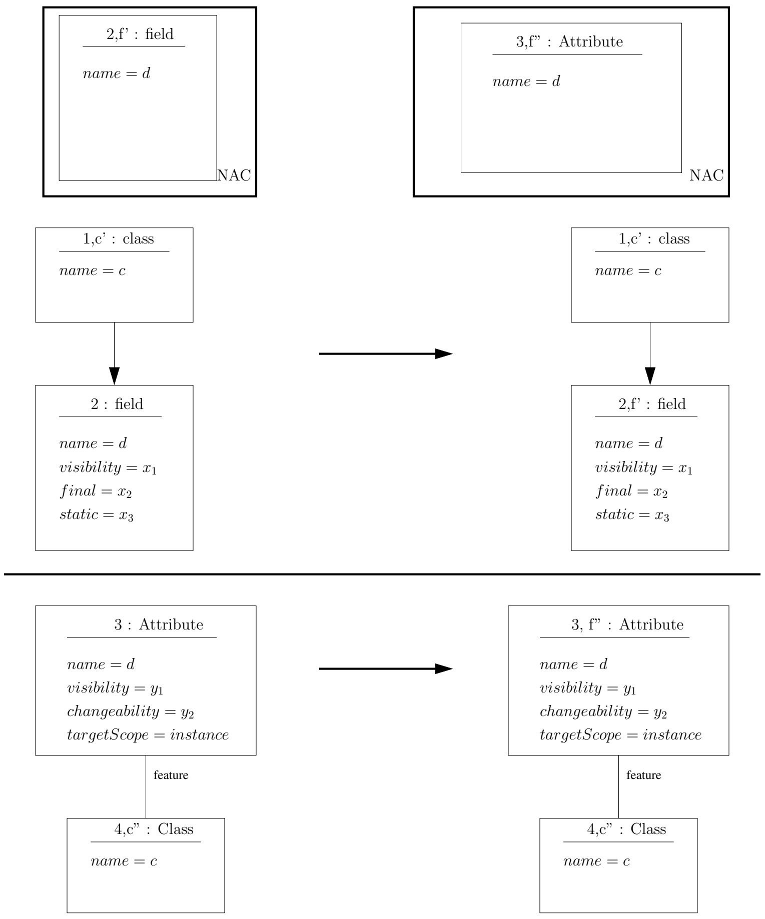
$x_1 \in \{public, protected, private\} \cup \{nil\}$
$x_2 \in \{true, false\} \cup \{nil\}$
$x_3 \in \{true, false\} \cup \{nil\}$
$x_4 \in \{true, false\} \cup \{nil\}$

$y_1 = x_1 \notin \{nil\}?x_1 : package$
$y_3 = x_3 \notin \{nil, false\}?true : false$
$y_4 = x_4 \notin \{nil, false\}?true : false$

Fig. 3. Rule `mapClass()`.

The rules in Figure 3 show several application conditions on the class properties:

- If visibility in the AST is undefined, then the UML side assumes default, *package*, visibility. Otherwise, *visibility* is the same for both elements;
- The UML counterparts of the `abstract` and `final` JavaML attributes are `isAbstract` and `isLeaf` respectively;
- No counterpart for the JavaML *static* attribute is available from the UML meta-



$x_1 \in \{public, protected, private\} \cup \{nil\}$
$x_2 \in \{true, false\} \cup \{nil\}$
$x_2 \notin \{false, nil\}?y_2 = frozen : y_2 = changeable$
$x_1 \in \{nil\}?y_1 = package : y_1 = x_1$

Fig. 4. Rule `mapField2Attribute()`.

model for outer classes.

In the rules of Figure 4 specific issues of concern are [1] :

- `targetScope` is specified with the `instance` value according to the metamodel semantic. By doing so, `Attribute` is not used to store meta-information but behaves as a normal model attribute;
- `changeability` represents the UML 1.5 way to specify a Java `final` attribute modifier.

The transformation unit which establishes the correspondence between classes,

[1] The field mapping rules shown in this article are a simplified version; some attributes are omitted and a more complex pattern on the UML side is not shown in order to keep the presentation of the Eclipse implementation simpler.

fields, and attributes results from the specialization of the template given above and is expressed as follows:

```
Field2Attribute()
forall mapClass());
forall mapField2Attribute() end
```

4 Correspondence Construction between Java and UML in Eclipse

This section discusses the implementation of template instances in an Eclipse plugin, `com.spulci.C2MCM` (**Code to Model Consistency Maintainer**). C2MCM is based on the Eclipse AST framework, residing in the `org.eclipse.jdt.core.dom` package tree, and on the UML2 Eclipse tool project in package `org.eclipse.uml2` [9]. C2MCM manipulates structures generated by these APIs to search for semantic equivalent nodes inside them. C2MCM also creates a representation of the interface graph within an XML file. A brief introduction to the Eclipse platform, *AST* framework and the *UML2* plugin is given as needed.

4.1 The Eclipse Platform

Eclipse is a platform centric IDE which offers tools to develop and maintain software taking into account various project aspects. The whole Eclipse architecture is extensible and open. Indeed, tools belonging to the platform are structured as plug-ins. Each plug-in can define one or more **extension-points**, places where another plug-in can attach itself to provide new capabilities and offer an interface to the existing ones.

4.2 Java Abstract Syntax and UML2 in Eclipse

We rely here on the definitions of the Java Abstract Syntax and of UML2 as provided by the Eclipse core, in which the instances of these metamodels are stored as separate files without reference between them. The basic assumption is that matching names refer to corresponding elements.

Classes from `org.eclipse.jdt.core.dom` and `org.eclipse.uml2` are imported to manage the Java *AST* and *UML2* models. The AST of some Java file is taken as input, allowing the search for semantically equivalent nodes in the UML2 model during the AST visit.

Correspondence construction in C2MCM is started by a call to the method *startEngine(ICompilationUnit icu)*, where the actual value for *icu* is an instance implementing the `ICompilationUnit` interface, specified by the user through the plug-in GUI. This is the root of an AST built from a `.java` file. Besides loading the AST, this method evaluates the URI of the UML2 model on which to construct the mapping and passes it to the *loadModel(URI uri)* method which actually loads it.

The realization of the approach takes advantage of the implementation of the `Visitor` pattern supported by Eclipse which can be advantageously used to implement the template as developed in the previous section.

Actually, visiting the tree according to the node types allows the interleaving of rules from different transformation units. However, this does not alter the final result with respect to the normal execution of these transitions. Indeed, each transformation unit resulting from the instantiation of the template produces, as its net effect, the construction of a node in the interface graph and of the mappings to UML2 and AST models, without eliminating any existing node or edge. As a result, no derived rule for each such instantiation may disrupt the positive context for the application of another (i.e. to consume something in the left-hand side of a rule). Hence, building a correspondence between some elements cannot prevent the construction of other correspondences between elements in their context. We can thus conclude that any interleaving of rules from different transformation units produces the same result, provided that any partial order between rules in the same transformation unit is respected.

loadModel() returns a **Package** model class instance with the same name as the **Package** Java class. To avoid namespace conflicts, we adopt the convention of always using the fully qualified name **org.eclipse.uml2.Package**. The model is loaded through a call to an EMF method, as the UML2 plug-in is an extension of the *Eclipse Modelling Framework*.

4.3 Code Skeleton

The first step to the Eclipse implementation of a transformation unit is to identify the nodes that should be visited in the AST. The visit is started on the nodes for which a transformation unit is defined. This results in the mappings prescribed by instantiations of `mapChild`, and possibly in those prescribed in the instantiations of `mapParent`, which are optionally applied. According to the AST Eclipse API, it is necessary to override the appropriate *visit()* method for each node type that has to be visited by the framework[2] The steps below analyze the template core notions and show the skeleton followed to build the Eclipse implementation:

Context Identification and Applicability: The identification of the context (schematised in the template as parent) for the node under examination is done by navigating the tree starting from the current node and looking for the pattern described in the `mapParent()` rule, also checking the applicability conditions. In most cases, this is simply done by navigating upwards until a node of a specific type is found. As node visits proceed from the root downwards, a mapping for the found parent may have been constructed in the visit of some other type with the same context (e.g. fields and methods in a class).

Node Mapping: The visiting policy adopted in the Java AST Framework provides

[2] The abstract syntax node type is passed as parameter to *visit()* Hence, a *visit(A x)* method codes a visit for a node **x** of Java type **A**. To grant children visit for the current node, the value *true* must be returned by each implementation.

an implementation of the `forall mapChild() end` construct, invoking a visit each time it finds a node of a certain type. This assures that a node of a certain type is visited at most once for each visit. Actually, it proceeds in a sequence in which the leftmost child of a node is always the first to be visited, and the subsequent siblings are visited in the order of declaration.

Name checking: As the mapping relies on name identification, the method `getFullyQualifiedName()` is used on AST nodes. On the UML side, the obtained name is used to construct an argument for *findNamedElements()*, which returns a *Collection* of nodes (typically at most two elements, if a variable and a method in the same class have the same name). The node of the correct kind is then extracted from the collection.

Application Conditions: An application condition in a rule is directly coded as a Boolean clause which performs checks on the attribute values specified in the rule.

Mapping construction: If the check is passed, the mapping is represented by adding an XML node to three different documents, one representing the Interface Graph, one for the *Java to UML* correspondences, the last for the reverse UML to Java mappings.

4.4 AstDecorator class: AST visit to find equivalent nodes

The bulk of the work is realized within the `AstDecorator` class in Listing 1, by which AST nodes are visited to find semantic equivalences. The constructor initializes a reference to the UML2 model passed as argument and stores the UML2 model name, to be used to construct fully qualified UML2 names. For each AST node type a version of the *visit()* method is defined. The actual node parameter is passed at runtime by the framework while the returned boolean value is set to true to allow visits to children nodes. In particular, for each node of the AST, a reference to the corresponding element in the UML model is set, and vice versa. Moreover, a node of the interface graph is constructed with references to the nodes put in correspondence. This also provides the correct context for the visit to the children.

In particular, we show the code for a `TypeDeclaration` node in the Java Language Specification[3] in Figure 3, and for a `VariableDeclarationFragment`, a JLS grammar element containing JavaML Field node items, together with their parent `FieldDeclaration`(see Figure 4).

A **TypeDeclaration** can be specialized as either an Interface or a Class Declaration; we consider here only the latter. The corresponding `Class` element in the UML2 model is found using the Eclipse *findNamedElement()* method. Inside the **if** clause body, the concrete coding of the mapping is performed. (See Listing 2.)

Field declarations require some additional work; a field identifier can be found inside a **VariableDeclarationFragment**, child of a **FieldDeclaration**. As our matching technique is based on name searching, it is better to define a visit on the

[3] we follow here JLS3, i.e. the version described in the third edition of [10]

former instead of the later. As explained before, a check is needed to find the context of that node. This time the context will be a class declaration and is searched by the method in Listing 3.

The visit implementation is shown in listing 4. Its structure is quite similar to the **TypeDeclaration** visit, exploiting the Java context to find an UML Class that contains a semantic equivalent field.

4.5 XML Document for the Interface Graph

Correspondences built by **C2MCM** are maintained both as new elements of the XML files for AST and UML2 and in a specific XML Document representing the Interface Graph. Nodes in this document have the following structure:

- The name of the node is the name of the rule which built it.
- The attribute *JAVANAME* contains the fully qualified name of the Java Ast element mapped by the rule.
- The attribute *UMLNAME* contains the fully qualified name of the corresponding element in the loaded UML2 model

As an example, the following code snippet constructs the node for the `mapClass()` rule mapping, using the DOM4J open source API [7]:

```
    Element igChild = igRoot.addElement("Class2Class"); //Node name
igChild.addAttribute("JAVANAME", packageName+"."+
className.getFullyQualifiedName()); //Java name
igChild.addAttribute("UMLNAME",md.getQualifiedName()+"::"+
className.getIdentifier());//UML name
```

5 Conclusion

In conclusion, we have shown how synchronized rules defined on the meta levels of Java abstract syntax and UML2 can be used to establish correspondences between instance models. This can be used for several purposes, including navigation from code to model and viceversa, and is particularly suited to allow consistency management between refactored code and model, without having to recur to reverse engineering or recompilation.

References

[1] Badros, G., *Javaml: A markup language for java source code*, 9th Int. World Wide Web Conference (2000).
URL http://www.badros.com/greg/JavaML/

[2] Bottoni, P., M. Koch, F. Parisi Presicce and G. Taentzer, *Automatic consistency checking and visualization of OCL constraints*, in: *UML 2000 - The Unified Modeling Language* (2000), pp. 294–308.

[3] Bottoni, P., F. Parisi Presicce and G.Taentzer, *Specifying Integrated Refactoring with Distributed Graph Transformation*, in: *Applications of Graph Transformations with Industrial Relevance*, LNCS **3062** (2004), pp. 220–235.

[4] Bottoni, P., F. Parisi Presicce and G. Taentzer, *Specifying Coherent Refactoring of Software Artefacts with Distributed Graph Transformations*, in: P. v. Bommel, editor, *Transformation of Knowledge, Information, and Data: Theory and Applications* (2004), pp. 95–125.
URL http://tfs.cs.tu-berlin.de/%7Egabi/gBPT04.pdf

[5] Corradini, A., U. Montanari, F. Rossi, H. Ehrig, R. Heckel and M. Löwe, *Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach*, , **I**, World Scientific, 1997 pp. 163–246.

[6] D'Anjou, J., S. Fairbrother, D. Kehn, J. Kellerman and P. McCarthy, "The Java Developer's Guide to Eclipse 2nd Edition," Addison Wesley, 2004.

[7] Dom4J Group, *Dom4J API Project*, http://www.dom4j.org/.

[8] Eclipse Organisation, *Eclipse 3.1.x Official Documentation*, http://help.eclipse.org/help31/index.jsp.

[9] Eclipse Organisation, *UML2 project*, http://www.eclipse.org/uml2/.

[10] Gosling, J., B. Joy, G. Steele and G. Bracha, "Java^{TM}Language Specification, Third Edition," The Java^{TM}series, Addison Wesley, 2005, 3rd edition.

[11] Kreowski, H.-J. and S. Kuske, *Graph transformation units with interleaving semantics*, Formal Aspects of Computing **11** (1999), pp. 690–723.

[12] Pulcini, S., "Evoluzione concorrente di Modelli Basata su Grafi Distribuiti," Master's thesis, University "La Sapienza" of Rome, Italy (2005).

[13] Taentzer, G., "Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems," Ph.D. thesis, TU Berlin (1996), shaker Verlag.

A Listings

```
public class AstDecorator extends ASTVisitor {
  public Model md; public String modelName;; //UML2 Model and Model name
  public AstDecorator(Model md){super(); this.md = md; this.modelName = md.getName();}
  ...
  public boolean visit(TypeDeclaration node){ // see Listing 2
   return true;
  }
  public boolean visit(VariableDeclarationFragment node){ // see Listing 4
   return true;
  }
}
```

Listing 1: AstDecorator class

```
public boolean visit(TypeDeclaration node){
  //check for class declaration
  if (!node.isInterface()){ SimpleName className = node.getName(); //get node simple name
   Collection c = UML2Util.findNamedElements((Resource) md.eResource(),
       modelName+"::"+packageName+"::"+className.getFullyQualifiedName());
   Iterator it = c.iterator(); //iterate on found model elements if any
   while(it.hasNext()){
     if (it instanceof org.eclipse.uml2.Class){
       org.eclipse.uml2.Class cl = (org.eclipse.uml2.Class)it.next(); //cast to Class
       VisibilityKind visibilityKind = cl.getVisibility(); //get UML visibility
       int modifiers = node.getModifiers(); //get AST node modifiers bit mask
       Modifier.ModifierKeyword keyword = Modifier.ModifierKeyword.fromFlagValue(modifiers);
   //boolean clause for application conditions
       boolean test = (cl.isAbstract() & Modifier.isAbstract(modifiers)) |
              (cl.isLeaf() & Modifier.isFinal(modifiers)) |
              (keyword.toString().contains(visibilityKind.getName()));
       if (test){ // code for mapping construction}
     }
   }
  }
}
```

Listing 2: visit(TypeDeclaration node) body

```
private TypeDeclaration getClassDeclaration(ASTNode node){
  ASTNode tempNode = node.getParent();
  while(!(tempNode instanceof TypeDeclaration)){ tempNode = tempNode.getParent(); }
  return (TypeDeclaration) tempNode;
}
```

Listing 3: getClassDeclaration(ASTNode node) body

```
public boolean visit(VariableDeclarationFragment node){
 TypeDeclaration parent = getClassDeclaration(node); //get the ClassDeclaration context
 if (!parent.isInterface()) { //check for parent node to be a class
   String parentName = parent.getName().getFullyQualifiedName();
 //search for the class inside the UML model
   org.eclipse.uml2.Classifier classifier;
   Collection c = UML2Util.findNamedElements(md.eResource(),
     md.getName()+"::"+packageName+"::"+parentName);
   Iterator it = c.iterator();
   while(it.hasNext()){
     classifier = (org.eclipse.uml2.Classifier)it.next();
 //find UML Attribute with same Java Field name
     Property attribute = classifier.getAttribute(node.getName().getFullyQualifiedName());
 //find fragment modifiers
     FieldDeclaration parentNode = (FieldDeclaration) node.getParent();
     int modifiers = parentNode.getModifiers();
     Modifier.ModifierKeyword keyword = Modifier.ModifierKeyword.fromFlagValue(modifiers);
     if (attribute != null){
       VisibilityKind attributeVisibility = attribute.getVisibility();
 //boolean clause for the application conditions
     boolean test = (keyword.toString().contains(attributeVisibility.getName()) |
       (attribute.isReadOnly() & Modifier.isFinal(modifiers)) |
       (attribute.isLeaf() & Modifier.isStatic(modifiers)));
     if(test){ // code for mapping construction }
     }
   }
 }
 return true;
}
```

Listing 4: visit(VariableDeclarationFragment node) body