Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework

Enrico Biermann[1], Karsten Ehrig[2], Christian Köhler[1], Günter Kuhns[1], Gabriele Taentzer[1], and Eduard Weiss[1]

[1] Department of Computer Science, Technical University of Berlin, Germany, {enrico,jaspo,bunjip,gabi,eduardw}@cs.tu-berlin.de
[2] Department of Computer Science, University of Leicester, UK, karsten@mcs.le.ac.uk

Abstract. The Eclipse Modeling Framework (EMF) provides a modeling and code generation framework for Eclipse applications based on structured data models. Although EMF provides basic operations for modifying EMF based models, a framework for graphical definition of rule-based modification of EMF models is still missing. In this paper we present a framework for in-place EMF model transformation based on graph transformation. Transformations are visually defined by rules on object patterns typed over an EMF core model. Defined transformation systems can be compiled to Java code building up on generated EMF classes. As running example different refactoring methods for Ecore models are considered.

1 Introduction

In the world of model-driven software development the Eclipse Modeling Framework (EMF) [7] is becoming a key reference. It is a framework for describing class models and generating Java code which supports to create, modify, store, and load instances of the model. Moreover, it provides generators to support the editing of EMF models.

EMF unifies three important technologies: Java, XML, and UML. Regardless of which one is used to define a model, an EMF model can be considered as the common representation that subsumes the others. I.e. defining a transformation approach for EMF, it will become also applicable to the other technologies.

In model-driven development, the transformation of models belongs to the essential activities. Different kinds of model transformations [24] are distinguished: endogenous transformations, such as refactoring or optimization in general, modify models within the same language. Exogenous transformation translate models between different languages. A prominent example for exogenous transformations are mappings from Platform Independent Models (PIMs) to Platform Specific Models (PSMs) in the Model-Driven Architecture (MDA) approach [12]. Although different in the intention, exogenous and endogenous transformations can simulate each other in a certain sense. An exogenous transformation with the same source and target language can be considered as endogenous one. Corresponding transformation engines usually work with two models, the source and the target model, in the exogenous case. This is not adequate for endogenous transformations where mostly in-place model updates are needed. Vice versa, endogenous transformations can emulate exogenous ones by constructing the product of all source and target languages and using it as underlying language.

Furthermore, we can distinguish model-to-model transformation to be used on a higher abstraction level, while model-to-text transformation to be defined by approaches like JET [10], refer to e.g. code generation. In the following, we focus on model-to-model transformations.

It has been shown that source-driven transformation languages such as XSLT being used to transform XML documents, are well suitable for the transformation of documents, but less suited for model transformations [18, 27].

In contrast to common model-to-model transformation approaches for EMF, we present an approach for in-place model-to-model transformations. As running example, we will consider model refactorings in EMF. We will introduce a visual notation for transformation rules which differs largely from that of QVT. Relations are a key concept in QVT which does not fit well to endogenous transformations, since relationships between model elements are not of primary interest. In contrast, the transformation approach presented focuses on structure modification and is inspired by graph transformation [19]. Transformation rules contain left and right-hand sides being object structures; moreover, negative object patterns may be defined, restricting the rule application. Since the transformation concepts are close to graph transformation concepts, it is possible to translate the rules to AGG [2], a tool environment for algebraic graph transformation where they might be further analyzed. For efficient execution of model transformations, the rules can be translated to Java code using generated EMF classes.

2 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [7] provides a modeling and code generation framework for Eclipse applications based on structured data models. The modeling approach is similar to that of MOF, actually EMF supports Essential MOF (EMOF) as part of the OMG MOF 2.0 specification [8]. The type information of sets of instance models is defined in a so-called core model corresponding to metamodel in EMOF. The core or metamodel for core models is the Ecore model. It contains the model elements which are available for EMF core models in principle. In Fig. 1, the main part of Ecore is shown. The kernel model contains elements EClass, EDataType, EAttribute and EReference. These model elements are needed to define classes by EClass, their attributes by EAttribute and interrelations by EReference. EClasses can be grouped to EPackages which might be again structured into subpackages. In addition, each model element can be annotated by EAnnotation. Furthermore, there are some abstract classes

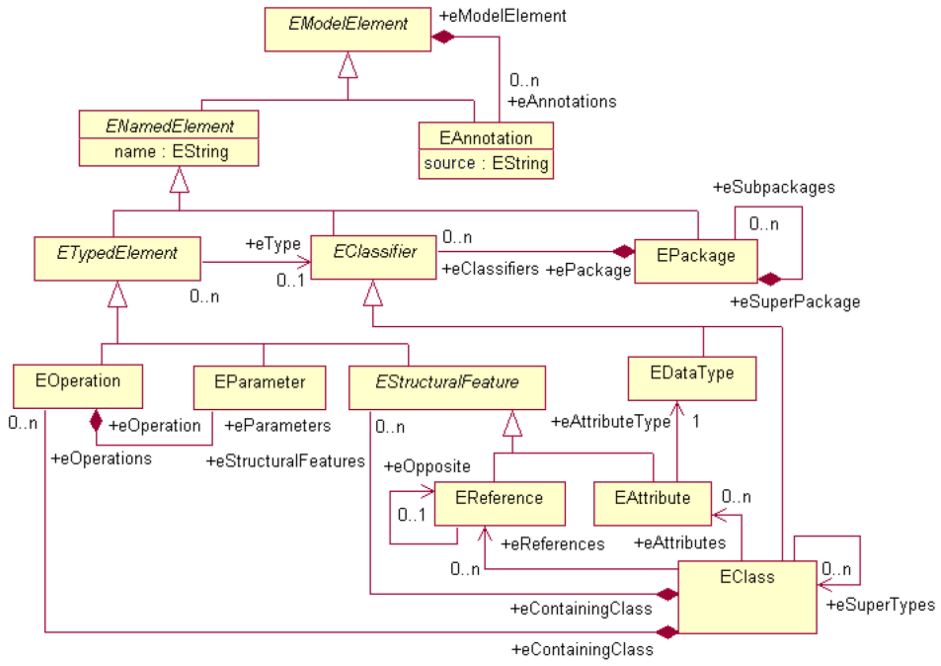

Fig. 1. Kernel of Ecore model

to better structure the Ecore model, such as ENamedElement, ETypedElement, etc.

It is important to note that the EMF metamodel (Ecore) again is a core model. That means that the metaclasses EClass, EDatatype, EReference etc. actually cannot just be interpreted as, but in fact *are* classes of an EMF core model. This is of great importance for our approach, since it enables us to use native EMF notions (elements of the metamodel) for the definition of transformation rules and interprete these notions in terms of formal graphs and graph transformations.

From an EMF model, a set of Java classes for the model and a basic, tree based editor can be generated. The generated classes provide basic support for *creating/deleting* model elements and persistency operations like *loading and saving*. Relations between EMF model classes are handled by special EMF lists, extending the Java list classes. Moreover, EMF models can be used as underlying models in new application plugins. But in many cases, the EMF model by its own is not powerful enough to express the complete model behavior. Therefore the generated code can be extended by the developer in order to add new functionalities that are not expressed in the EMF model.

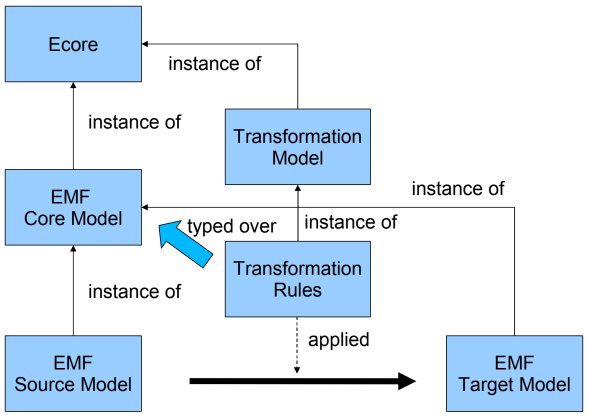

Fig. 2. Transformation Overview

3 Visual Definition of Endogenous Transformations

Basically, an in-place EMF transformation is a rule-based modification of an EMF source model resulting in an EMF target model. Both, the EMF source and target models are typed over the same EMF core model which itself is again typed over Ecore. The transformation rules are typed over the *Transformation Model* shown in Fig. 3 which itself is an instance of Ecore again (see Fig. 2). Since the transformation model is an EMF model, a tree-based editor can be generated automatically. For more convenient editing of the rules we developed an additional visual editor being an Eclipse plug-in based on EMF and GEF [5]. Figs. 4 - 8 show screenshots of this editor.

A *Transformation* consists of a *RuleSet* containing the set of *Rules* for the transformation. Furthermore, it has a link to the core model its instances are typed over. If needed, a start structure can be defined as well to have a fixed starting point for the transformation available. A transformation together with a start structure forms an EMF grammar.

Rules are expressed mainly by two object structures LHS and RHS, the left and right-hand sides of the rule. Furthermore, a rule has mappings between obects and links of the LHS and the RHS indicated by numbers preceding the class names. The left-hand side LHS represents the pre-conditions of the rule, while the right-hand side RHS describes the post-conditions. Those symbols and links of the LHS which are mapped to the RHS, describe a structure part which has to occur in the EMF source model, but which is not changed during the transformation. All objects and links of the LHS not mapped to the RHS define the part which shall be deleted, and all objects and links of the RHS to which nothing is mapped, define the part to be created. Attributes in the LHS have to occur in the EMF source model in addition while they can be reassigned with different values in the RHS of the rule.

The applicability of a rule can be further restricted by additional application conditions. As already mentioned above, the LHS of a rule formulates some kind of positive condition. In certain cases also *negative application conditions* (NACs) which are pre-conditions prohibiting certain object structures, are needed. If several NACs are formulated for one rule, each of them has to be fulfilled. A NAC is again an object structure. Moreover, mappings between the LHS and a NAC can be defined. This feature is useful to prohibit structures in relation to the LHS.

The rule's LHS or a NAC may contain constants or variables as attribute values, but no Java expressions, in contrast to a RHS. A NAC may use the variables already used in the LHS or new variables declared as input parameters. The scope of a variable is its rule, i.e. each variable is globally known in its rule. The Java expressions occurring in the RHS, may contain any variable used within the LHS or declared as input parameter. Multiple usage of the same variable is allowed and can be used to require equality of values.

A rule-based transformation system may show two kinds of non-determinism: (1) for each rule several matches can exist, and (2) several rules can be applicable. There are techniques to restrict both kinds of choices. The choice of matches can be restricted by using input parameters. Moreover, some kind of control flow on rules can be defined by applying them in a certain order. For this purpose, rules are equipped with layers. All rules of one layer are applied as long as possible before going over to the next layer. Later on, we will show how to use Java for controlling rule applications.

Running Example: Refactoring of EMF Models: To illustrate the presented transformation approach for EMF models we show two refactoring methods for EMF models. All transformation rules are typed over the Ecore model, in more detail over the Ecore section shown in Fig. 1. In the following, we define the simple refactoring "move class" where a class is moved from one package to another. Moreover, the complex refactoring "pull up attribute" is shown. If each subclass contains an attribute with the same name, it can be pulled up to their common superclass.

Refactoring rule "MoveClass(EString n, EString p)" in Fig. 4 has two input parameters "n" and "p" to determine the names of the class to be moved and the package it shall be moved to. The LHS describes the pattern to be found for refactoring consisting of the class with name "n", the package it is currently in, and the package with name "p" it shall be moved to. The RHS shows the new pattern after refactoring where the class is contained in the package named "p". In addition, the rule has a NAC which checks if the package named "p" already contains a class named "n".

Refactoring "PullUpAttribute" is more complex, i.e. it cannot be defined by just one rule, but four rules are needed to check the complex pre-condition, to do the kernel refactoring, and to make the model consistent afterwards. For checking the pre-condition, rule "CheckAttribute(EString c, EString a)" in Fig. 5 checks for the class named "c" if there is a subclass not containing an attribute named "a". This rule can be applied at most once, since there are NACs which check if

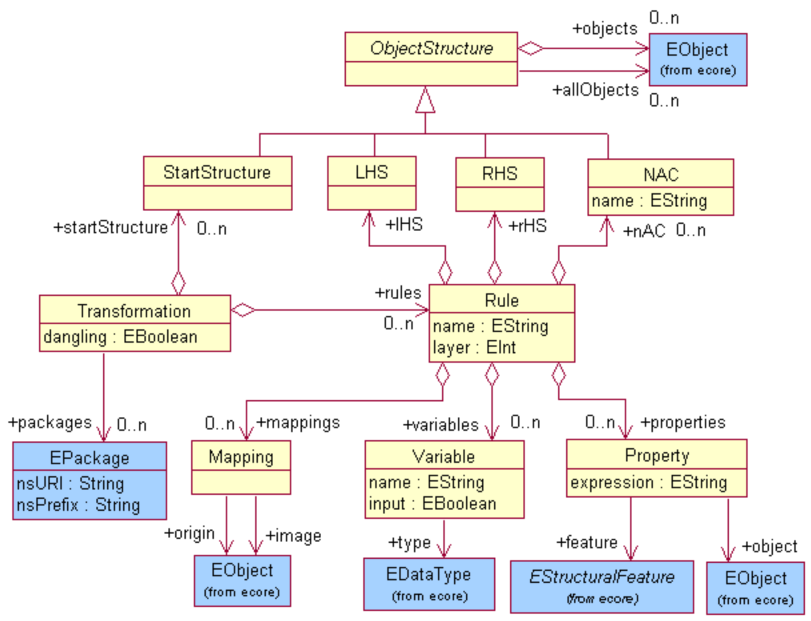

Fig. 3. Transformation Model

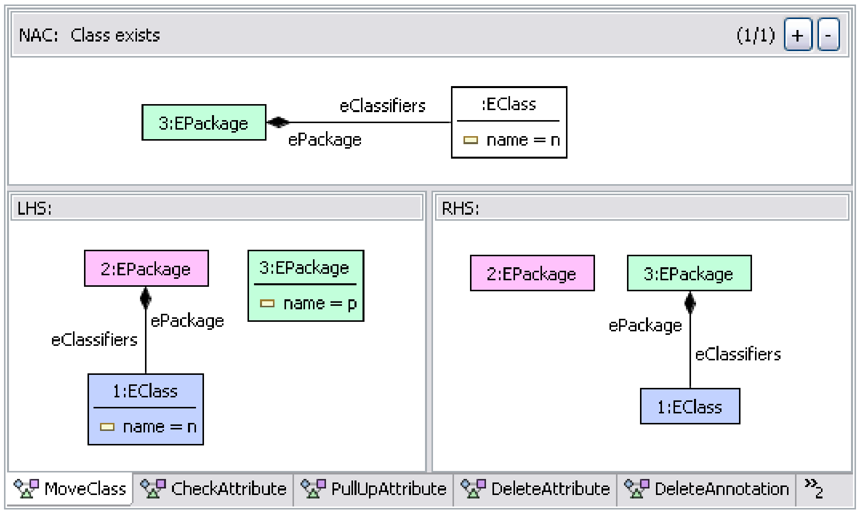

Fig. 4. Rule ”MoveClass”

there is already a subclass with this annotation. Thereafter, we try to apply rule ”PullUpAttribute(EString c, EString a)” in Fig. 6. If there is no subclass of the class named ”c” which has an annotation with source ”no attribute” and if the

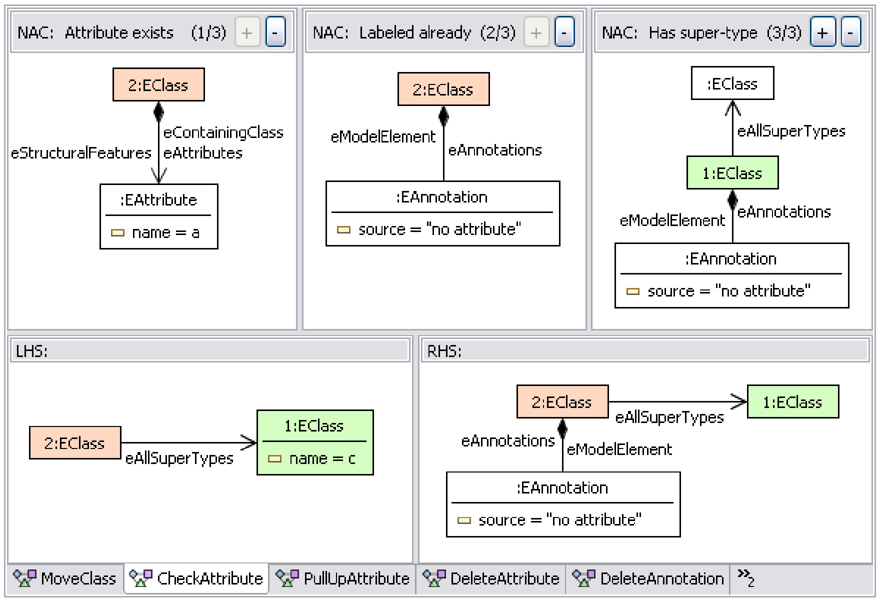

Fig. 5. Rule "CheckAttribute"

class named "c" has not already an attribute named "a", it looks for a subclass which has an attribute named "a". After the refactoring, the attribute with name "a" is pulled up from one subclass. This rule is applicable at most once. Thereafter, NAC "Attribute already pulled up" will not be satisfied anymore. NAC "Attribute not in all sub-types" checks a necessary pre-condition.

If "PullUpAttribute" was successful, i.e. there is no subclass with a corresponding annotation, all attributes named "a" being still contained in subclasses have to be deleted. This is done by rule "DeleteAttribute(EString c, EString a)" in Fig. 7 applying it as long as possible. Finally, if the refactoring was not successful, all new annotations of the class named "c" have to be deleted again which is performed by rule "DeleteAnnotation(EString c)" in Fig. 8. The application control for these rules just described can be realised by putting each of the rules to consecutive layers in the order of description. (See attribute "layer" of model element "Rule" in the transformation model in Fig. 3.)

4 Execution of EMF Transformations

To apply the defined transformation rules on a given EMF model, we either select and apply the rules step-by-step, or take the whole rule set and let it apply as long as possible. A transformation step with a selected rule is defined by first finding a match of the LHS in the current instance model. A pattern is matched to a model if its structure can be found in the model such that the types and attribute values are compatible. In general, a pattern can match to

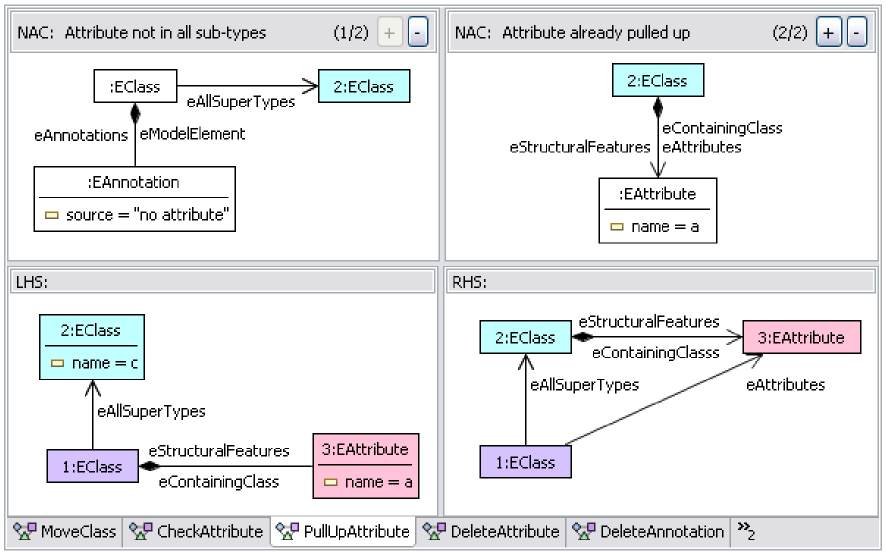

Fig. 6. Rule "PullUpAttribute"

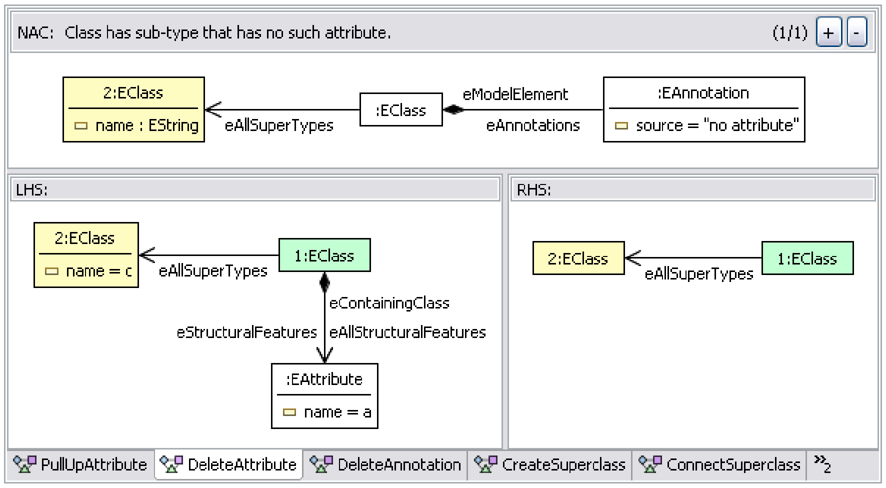

Fig. 7. Rule "DeleteAttribute"

different parts of a model. In this case, one of the possible matches has to be selected, either randomly or by the user.

Performing a transformation step which applies a rule at a selected match, the resulting object structure is constructed in two passes: (1) all objects and links present in the LHS but not in the RHS are deleted; (2) all object and links in the RHS but not in the LHS are created. A transformation, more precisely a transformation sequence, consists of zero or more transformation steps.

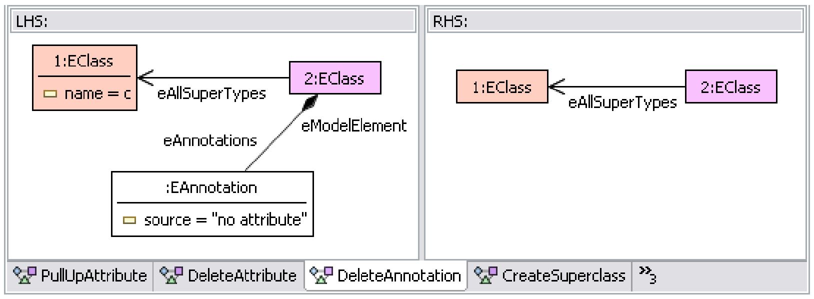

Fig. 8. Rule "DeleteAnnotation"

Consistency recovery: Although EMF models show a graph-like structure and can be transformed similarly to graphs [19], there is a main difference in between. In contrast to graphs EMF models have a distinguished tree structure which is defined by the containment relation between their classes. An EMF model should be defined such that all its classes are transitively contained in the root class. Since an EMF model may have non-containment references in addition, the following question arises: What if a class which is transitively contained in the root class, has non-containment references to other classes not transitively contained in the root class? In this case we consider the EMF model to be inconsistent, since e.g. it cannot be made persistent anymore. A transformation can make an EMF model inconsistent, if its rule deletes one or more objects or containment links. For example an inconsistent situation occurs, if one of these objects transitively contains an object included by a non-containment reference. To restore the consistency, all objects to be deleted or to be disconnected from their containing objects, have to be determined. Thereafter, all non-containment references to these indicated objects have to be removed, too. Similar to the handling of deleted structures, consistency recovery is also applied to newly created objects. If a rule creates objects which are not contained in the tree structure, the consistency recovery will remove these objects at the end of a rule application. It is possible to forbid the application of those rules entirely, since inconsistencies on creation of objects can be determined statically.

4.1 Interpreter Approach

For executing the defined transformation by the EMF Interpreter, a new Interpreter instance has to be created first. (See the following code snippet.)

```
Interpreter interpreter = new Interpreter(eObject);
interpreter.loadTransformation(filename);
interpreter.transform();
interpreter.applyRule(rulename, parameter, mapping);
```

An *eObject* can be any class in the model instance which should be transformed. After creating the interpreter, the transformation file with name "filename" is loaded. It has to be ensured that the loaded transformation contains the same classes that are used by the instance model to be transformed. After loading a transformation, rules can be applied. For example, invoking *transform()* results in the application of all rules as long as possible. For applying a specific rule, method *applyRule* is called. The first parameter of *applyRule* is simply the name of the rule to be applied. Afterwards the value of each input parameter needs to be specified. A sample use of class Parameter is given in the following example. The third parameter of method *applyRule* contains a vector of *EObjects* which defines a partial match between rule objects and instance objects. If a rule shall be matched automatically, this parameter is set to null.

Here is the sample code snippet for the application of rule *MoveClass* to an EMF model for a library. Assuming you want to move class *Book* from package *Bookshelf* to package *Library.*

```
Interpreter interpreter = new Interpreter(eClass1);
interpreter.loadTransformation("refactoring.tfm");

Parameter parameter = new Parameter();
parameter.addParameter("n","Book", "String");
parameter.addParameter("p", "Library", "String");

Vector mapping = new Vector(2);
mapping.add(eClass1);
mapping.add(ePackage1);

interpreter.applyRule("MoveClass", parameter, mapping);
```

Interpreting EMF Transformations by Graph Transformations: Since EMF models show a graph-like structure and can be transformed similarly to graphs, we have chosen an interpreter approach where an EMF model is translated to a corresponding graph. Furthermore, the EMF transformation rules are translated to graph rules. After having performed the corresponding graph transformation, the result graph is translated back to an EMF model. For the execution of graph transformations, we take AGG [2], a transformation engine for typed, attributed graphs.

As first step, the EMF core model of the transformation is translated to a so-called type graph. Classes are translated to node types and references to edge types. Please note that bidirectional references are mapped to two opposite edge types. Class attributes become node type attributes on the graph side. EMF instance models are translated to graphs. Since each consistent instance model has root objects which contain all other objects, we can navigate from given EObjects being the roots for all linked objects and translate them to graph nodes. All references are mapped to edges. Each EMF rule is translated to a graph rule in a straightforward way.

After having performed the corresponding graph transformation, the resulting graph has to be translated back to an EMF model. As described above, it might happen that the resulting EMF model is not consistent, i.e. non-containment references which make the model inconsistent, have to be removed.

Having a translation of EMF transformation to graph transformation (and back again) at hand, the available analysis techniques may be useful to validate EMF transformations. This is not always possible, but only if the EMF models remain consistent during the transformation which is the case if objects with subtrees are not deleted or uncoupled by removing the reference to their container.

All refactoring rules in the running example preserve the consistency of EMF models. Thus, analysis techniques such as critical pair analysis, termination checks, etc. are available also for these EMF model refactorings. For example in [23], critical pair analysis was used to detect conflicts and dependencies between software refactorings. For example, one conflict between two different applications of rule "PullUpAttribute" reported by AGG occurs, if a class has several subclassses where attributes named "a" occur. In this case only one of these attributes is pulled up. Since all these attributes in the subclasses are equal and are deleted afterwards, the refactoring result is independent of the concrete attribute pulled up. Thus, this conflict can be resolved [22].

4.2 Compiler Approach

Besides interpreting an EMF transformation as graph transformation, transformation rules can also be compiled to Java methods to be used together with previously generated EMF code. For the translation of transformation rules to code we use JET, the code generator in EMF [7].

For each transformation rule, two classes are generated to do the rule matching and the transformation. E.g. for refactoring rule "MoveClass", Java classes "MoveClassRule.java" and "MoveClassWrapper.java" are generated. The first class contains methods for execution, undo and redo functionality. The second class is needed for the matching process. Rule matching is formulated as a constraint solving problem where the LHS objects are variables, the objects of the EMF instance model form the domain, and typing, linking und attribute values form the set of constraints. Formulating pattern matching in this way, its efficiency is directly dependent on the constraint solving algorithm as well as on the ordering of variables and domain elements. This form of pattern matching is influenced by graph pattern matching as done in AGG [25].

To apply one rule you create an instance of the generated rule class. This class and the dependent wrapper class contain all information about the intended changes of instances by the rule and how to find a match for the LHS. To have at least one reference to the instance, on which the rule shall be applied, it must be set by method "setInstanceSymbol(eObject)". Its parameter can be an arbitrary EObject of the instance model. Input parameters can be given by setters, which have name "set" followed by the variable name in the rule. Matches for the LHS are either found automatically or are given by setters, which have the form set+Type+Counter (for objects of type "Type" further distinguished by "Counter"). By method "execute()" the given partial match is completed and the rule applied. Here is a short code example for the application of the rule "MoveClass". Let's assume you want to move class *Book* from package *Bookshelf* to package *Library*.

```
MoveClassRule moveClassRule = new MoveClassRule();
moveClassRuse.setInstanceSymbol(eClass1);

moveClassRule.setParN("Book"); // set Name
moveClassRule.setParP("Library"); // set Package

moveClassRule.setEClass0(eClass1);
moveClassRule.setEPackage0(ePackage1);

moveClassRule.execute();
```

There is also a way to apply a rule with the same parameters as the Interpreter. To do so you call method "applyRule()" in class "Transformation-Interface". This class also needs a reference to the instance which is given in the constructor. Additionally it allows to start a transformation by calling the method "transform()".

```
Transformation transformation = new Transformation(eClass1);
transformation.applyRule("MoveClass", parameter, mapping);

transformation.transform();
```

While transform applies the rule arbitrarily in this example, the rule application can also be controlled by Java constructs.

5 Related Work

In this paper we presented a model transformation approach based on graph transformation concepts and the Eclipse technology. There are already several model transformation tool environments around being based on graph transformation and/or Eclipse. Most of these tool environments are designed for exogenous model transformation, i.e. model transformations between different languages, and do not allow in-place model updates. This fact is one of the differences to our approach which is especially designed for endogenous model transformation, i.e. model transformation within the same language. In the following, we look a little closer to several approaches and distinguish between EMF-related and graph transformation related approaches.

5.1 EMF-Related Approaches

A rather simple approach to EMF model transformation is given by the Merlin Eclipse plug-in [11] which can perform model-to-model and model-to-code transformations. Focussing on the first type of transformations type mappings and simple mapping rules consisting of conditions - actions pairs can be performed. Type mappings and rules are defined in a textual form.

Sub-projects in Eclipse GMT [4] like Tefkat [20], ATL [3], MTF [1] and MOMENT [16] support a much more elaborated transformation approach which is mainly declarative and close to the concepts of QVT, but might also allow imperative feature, as in the case of ATL. Similarly, our approach is mainly rule-based, but allows native method calls in attribute computations (as ATL does). In contrast to ours, model transformations are formulated in textual forms in all studied approaches.

Each of the QVT-related approaches considered provides a transformation engine based on EMF which might be integrated in other applications as well as a tool environment (IDE) which consists of at least an editor and a debugger provided as Eclipse plug-ins. While also offering a transformation engine and a (visual) editor, our approach lacks from an integrated debugger. For this purpose, a model transformation has to be translated to AGG where the stepwise execution of transformations is supported.

The MOMENT project contains an EMF transformation engine which is based on algebraic specifications as implemented in Maude [?]. Similarly to ours, this approach has a clear formal background. But in contrast to MOMENT our EMF transformations are based graph transformation concepts which can be used for verify properties of model transformations such as termination, confluence, and constraint checking, and can be executed by the AGG graph transformation engine.

5.2 Graph Transformation Related Approaches

There are a number of graph transformation-based approaches to model transformation, as e.g. supported by VIATRA2 [15], VMTS [21], AToM3 [17], GReAT [9], MOFLON [13], Gmorph [26] and MOTMOT [14]. While all dealing with graphs and their manipulation, these approaches differ heavily concerning the kind of graphs used and the transformation concepts supported. All indicated approaches support exogenous model transformations.

Besides standard graph transformation concepts, such as rules with left and right-hand sides and integrated attribute computations, a number of advanced transformation concepts are supported. Additional forms for structuring rule sets are supported by all of the related approaches. We decided to keep our transformation model rather simple by supporting the standard transformation concepts with negative application conditions for rule in addition. As advantage, graph transformations of this form can be verified based on the theory of algebraic graph transformation. Additional structuring of transformation rules has to be expressed by additional Java code and is not yet taken into account for verification.

Most of these graph transformation-related approaches do not offer EMF import/export facilities. While VIATRA2 is able to perform EMF transformations in an interpretative mode, it is not able to generate Java code for endogenous EMF transformations. MOFLON combines MOF with graph transformation and supports the generation of JMI compliant Java code, but does not offer verification facilities.

6 Conclusion and Future Work

In this paper we presented an approach for the graphical definition of in-place model transformations. As running example, we considered model refactorings in EMF. Our visual notation for transformation rules pretty differs from that of QVT. Relations are a key concept in QVT which does not fit well to endogenous transformations, since relationships between model elements are not of primary interest. In contrast, the transformation approach presented focuses on structure modification and is inspired by graph transformation. Transformation rules contain left and right-hand sides being object structures; moreover, negative object patterns may be defined, restricting the rule application. Since the transformation concepts are closely related to graph transformation concepts, it is possible to translate the rules to AGG, a tool environment for algebraic graph transformation where they might be further analyzed. For efficient execution of model transformations, the rules can be translated to Java code to be integrated into generated EMF classes. The presented tool can be downloaded at *http://tfs.cs.tu-berlin.de/emftrans*.

Further application of endogenous EMF model transformation may include the execution of editing operations in EMF-based editors such as generated by the Eclipse Graphical Modeling Framework (GMF) [6]. Orienting the transformation model at the concepts of algebraic graph transformation techniques, we started with a rather simple transformation model. Further concepts may be formulated on top of the approach presented such that the well-developed analysis techniques for algebraic graph transformations can still be used.

References

1. *IBM Model Transformation Framework http://www.alphaworks.ibm.com/tech/mtf*, 2005.
2. *AGG-System http://tfs.cs.tu-berlin.de/agg/*, 2006.
3. *ATL: The Atlas Transformation Language Home Page http://www.sciences.univ-nantes.fr/lina/atl*, 2006.
4. *Eclipse Generative Modeling Tools (GMT) http://www.eclipse.org/gmt*, 2006.
5. *Eclipse Graphical Editing Framework (GEF) http://www.eclipse.org/gef*, 2006.
6. *Eclipse Graphical Modeling Framework (GMF) http://www.eclipse.org/gmf*, 2006.
7. *Eclipse Modeling Framework (EMF) http://www.eclipse.org/emf*, 2006.
8. *Essential MOF (EMOF) as part of the OMG MOF 2.0 specification http://www.omg.org/docs/formal/06-01-01.pdf*, 2006.
9. *GReAT: Graph Rewriting And Transformation http://www.isis.vanderbilt.edu/Projects/mobies/downloads.asp*, 2006.
10. *Java Emitter Templates (JET) as part of the Eclipse Modeling Framework (EMF) http://www.eclipse.org/emf*, 2006.
11. *Merlin Generator http://sourceforge.net/projects/merlingenerator/*, 2006.

12. *Model Driven Architecture (MDA). http://www.omg.org/mda*, 2006.
13. *MOFLON http://gforge.echtzeitsysteme.org/projects/moflon/*, 2006.
14. *MoTMoT: Model driven, Template based, Model Transformer http://www.fots.ua.ac.be/motmot/index.php*, 2006.
15. *VIATRA2 (VIsual Automated model TRAnsformations) framework http://dev.eclipse.org/viewcvs/indextech.cgi/ checkout /gmt-home/subprojects/VIATRA2/index.html*, 2006.
16. A. Boronat, J. Carsi, and I. Ramos. Algebraic Specification of a Model Transformation Engine. In *Springer LNCS 3922. Fundamental Approaches to Software Engineering (FASE'06). ETAPS'06. Vienna (Austria).*, 2006.
17. J. de Lara and H. Vangheluwe. ATOM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling. In R. Kutsche and H. Weber, editors, *Proc. Fundamental Approaches to Software Engineering (FASE'02), Grenoble, April 2002*, pages 174 – 188. Springer LNCS 2306, 2002.
18. K. Duddy, A. Gerber, M.J. Lawley, K. Raymond, and J. Steel. Declarative Transformation for Object-Oriented Models. In *In Transformation of Knowledge, Information, and Data: Theory and Applications, edited by P. van Bommel. Idea Group Publishing*, 2005.
19. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer, 2006.
20. M. Lawley and J. Steel. Practical Declarative Model Transformation With Tefkat. In *In Proc. Model Transformation in Practice Workshop, Models Conference*, 2005.
21. T. Levendovszky, L. Lengyel, G. Mezei, and H. Charaf. Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS. In *2nd International Workshop on Graph Based Tools (GraBaTs), workshop at ICGT 2004, Rome, Italy*, 2004.
22. T. Mens, G. Taentzer, and O. Runge. Detecting Structural Refactoring Conflicts unsing Critical Pair Analysis. In *In R. Heckel and T. Mens, editors, Proc. Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra'04), Satellite Event of ICGT'04), Rome, Italy*, 2004.
23. T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and System Modeling*, 2006. to appear.
24. T. Mens and P. Van Gorp. A Taxonomy of Model Transformation. In *Proc. International Workshop on Graph and Model Transformation (GraMoT'05)*, number 152 in Electronic Notes in Theoretical Computer Science, Tallinn, Estonia, 2006. Elsevier Science.
25. Michael Rudolf. Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In *6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), LNCS 1764*, pages 238–251. Springer Verlag, 2000.
26. S. Sendall. Combining Generative and Graph Transformation Techniques for Model Transformation: An Effective Alliance? In *18th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.
27. G. Taentzer and G. Toffetti Carughi. A Graph-Based Approach to Transform XML Documents. In L. Baresi and R. Heckel, editors, *Proc. Fundamental Approaches to Software Engineering (FASE)*, volume 3922 of *LNCS*. Springer, 2006.