ELECTRONIC COMMUNICATIONS OF THE EASST

EMF Model Refactoring based on Graph Transformation Concepts

Enrico Biermann *, Karsten Ehrig **, Christian Köhler *, Günter Kuhns *, Gabriele Taentzer *, Eduard Weiss *
*Department of Computer Science, Technical University of Berlin,
Germany,{enrico,bunjip,jaspo,gabi,eduardw}@cs.tu-berlin.de
**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. Within model driven software development based on EMF, refactoring of EMF models become a key activity. In this paper, we present an approach to define EMF model refactoring methods as transformation rules being applied in place on EMF models. Performing an EMF model refactoring, EMF transformation rules are applied and can be translated to corresponding graph transformation rules, as in the graph transformation environment AGG. If the resulting EMF model is consistent, the corresponding result graph is equivalent and can be used for validating EMF model refactoring. Results on conflicts and dependencies of refactorings for example, can help the developer to decide which refactoring is most suitable for a given model and why.*

Keywords: Model refactoring, Eclipse Modeling Framework, graph transformation

1 Introduction

In the world of model-driven software development, the Eclipse Modeling Framework (EMF) [EMF06] is becoming a key reference. It is a framework for describing class models and generating Java code; it supports the creation, modification, storage, and loading of model instances. 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. That means defining a transformation approach for EMF, it will become also applicable to the other technologies.

Refactoring within the model-driven software development process means to refactor the corresponding

models. Basing the model-driven approach on EMF models, refactoring of EMF models becomes a key activity which should be supported. Since EMF unifies three different technologies, i.e. Java, XML and UML, the EMF refactoring can also be used to restructure Java programs, XML schemas and UML models. Considering especially UML, a number of refactoring methods are already available, see e.g. [SPTJ01], [Por03], [MB05], which all cover at least refactoring of class models. Since EMF models resemble very much UML class models, UML refactoring methods can also be considered for EMF model refactorings.

Different approaches have been considered for model refactorings which can be categorized as model transformations in general, optimizing models of a given modeling language. Most of the refactoring approaches presented (e.g. [SPTJ01], [MB05]) use OCL constraints to describe the pre- and post-conditions of refactorings in a declarative way. Another kind of approaches use transformation rules (e.g. [Por03]). In [MT04], declarative approaches based on pre-/post-conditions are compared with graph transformation approaches.

In this paper, we consider two selected refactorings of instances of the Ecore model being the EMF meta model and as such also an EMF model. Besides the Ecore model, we could also choose any other EMF model, such as the UML2 model, for refactoring. Our transformation approach is based on graph transformation and adapted to EMF models.

In our running example, we consider an EMF model which stores the abstract syntax of simple place/transition Petri nets and refactor it in order to get a more object-oriented model. (See the original Petri net model in Fig. 1.) This model is restructured by pulling up the common attribute ”name” of classes ”Place”, ”Transition” and ”PetriNet” to a new superclass ”NamedElement”. (See the refactored model in Fig. 2.) In the following, we consider “create superclass” and “pull up attribute” as sample EMF model refactorings.

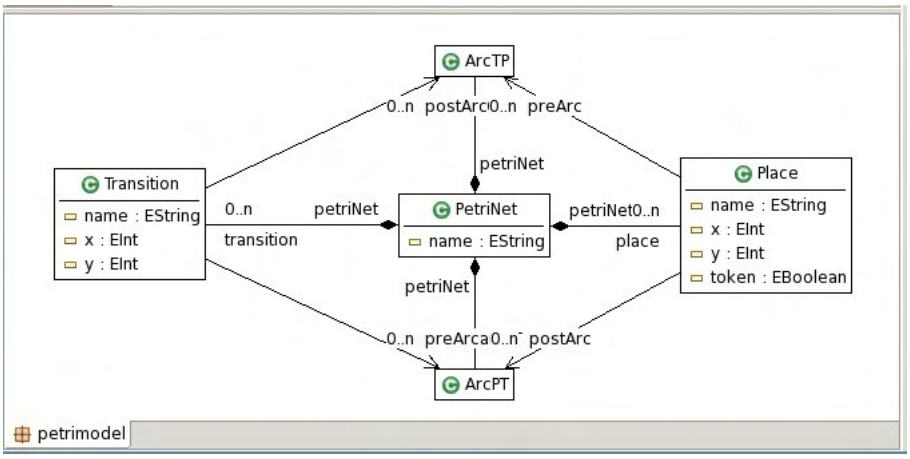

Figure 1: Petri net model before refactoring

EMF model refactoring can be considered as endogenous model transformation [MVG06] performing some kind of model optimization. When applying a refactoring method to an EMF model, this model shall be modified, i.e. it shall be transformed in-place. Considering the current transformation approaches

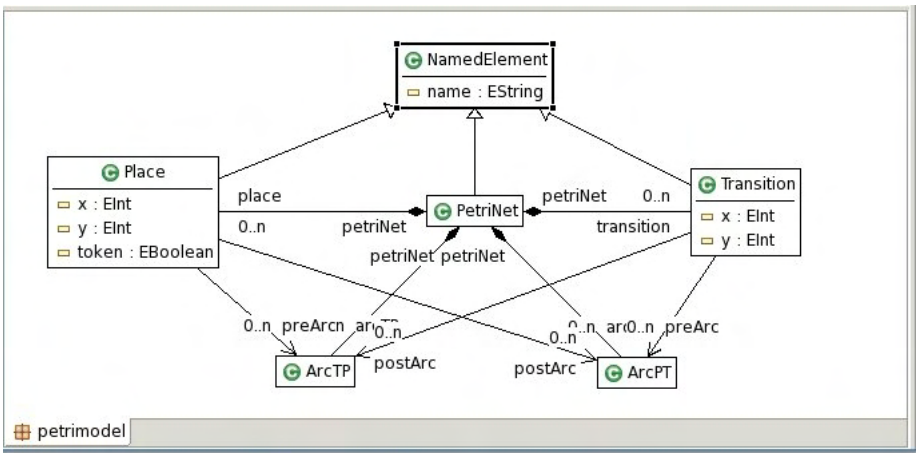

Figure 2: Petri net model after refactoring

for EMF such as Tefkat [LS05], ATL [JK06], MTF [MTF05], Merlin [Mer06], and the transformation engines developed within the Eclipse Project General Model Transformer (GMT), a transformation engine for in-place transformation and with validation facilities for model transformations is not yet available.

Therefore, we recently developed an endogenous EMF model transformation engine [BEK+] which can perform in-place model transformations, based on graph transformation concepts. The transformation description can be compiled to Java code using those EMF classes already generated. Furthermore, it is possible to translate the rules to AGG [AGG06], a tool environment for algebraic graph transformation where the transformation might be further analyzed.

The analysis of transformations and transformation rules is helpful in deciding what to refactor and when. Termination of refactoring operations as well as conflicts and dependencies between different refactorings are important issues to be analysed to inform the user about the refactorings which can be performed. The dependency analysis has already been considered in [MTR04].

2 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [EMF06] 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 [EMO06]. 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. 3, the main part of Ecore (without attributes) 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. Fur-

thermore, there are some abstract classes to better structure the Ecore model, such as ENamedElement, ETypedElement, etc.

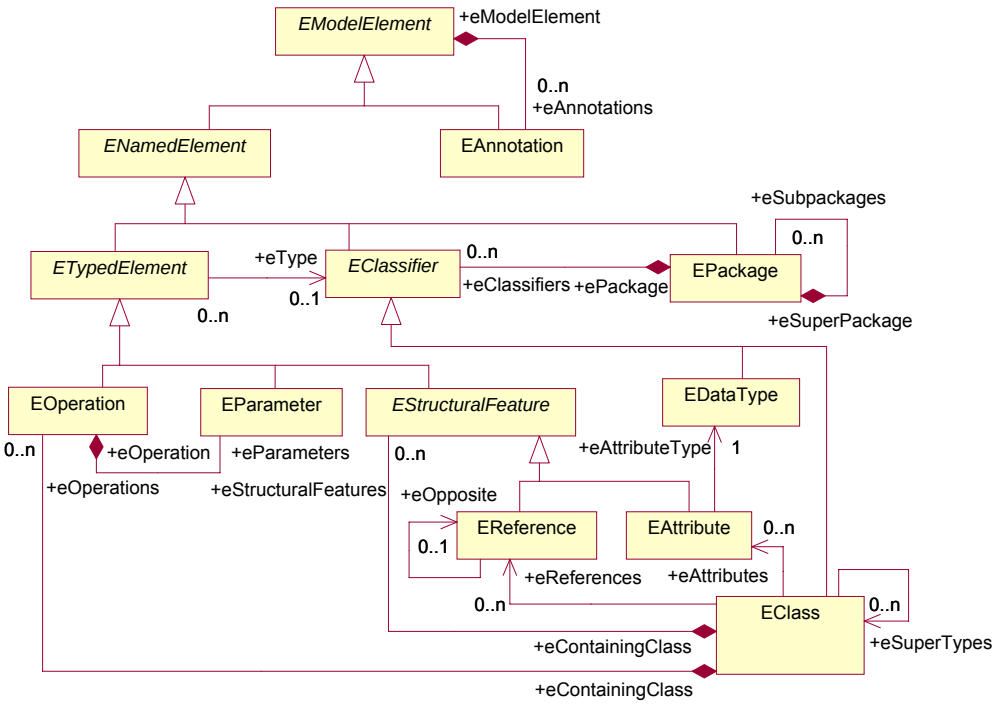

Figure 3: Kernel of Ecore model

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

3 Specification of EMF Model Refactorings

Before presenting the specification of concrete EMF model refactoring methods, we present the transformation approach used to manipulate EMF models. The transformation concepts are closely related to the algebraic graph transformation concepts [EEPT06]. The main reason for this design decision is the basic opportunity to validate EMF model transformations which is possible only if some form of consistency can be reached by the EMF transformation. We will discuss consistency issues in the next section.

3.1 EMF Model Transformation Approach

Basically, an 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 an EMF core model which itself is again typed over Ecore. Refactoring can take place on two levels: (1) Refactoring rules are typed over the Ecore model and are applied to EMF models. The running example of this paper containing refactorings of a Petri net model (being an EMF core model) is of this kind. (2) Refactoring rules are typed over some EMF core model (e.g. the Petri net model in Fig. 1) and refactor EMF instance models (e.g. Petri net instance models).

A *Transformation System* consists of a set of transformation rules. Furthermore, it has a link to the core model its instances are typed over. 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 objects 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 objects 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 of the LHS not mapped to the RHS define the part which shall be deleted, and all symbols and links of the RHS to which nothing is mapped, define the part to be created.

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 which is the target of a mapping from the LHS. This feature is useful to prohibit structures connected 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 an 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 may exist, and (2) several rules may 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 (in any order) before going over to the next layer. The transformation stops after having execute the last layer.

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 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.

3.2 Selected Refactoring Methods for EMF Models

Based on the presented transformation approach for EMF models we show two selected refactoring methods for EMF models. All transformation rules are typed over the Ecore model, in more detail over the Ecore section shown in Fig. 3. In the following, we define the simple refactorings "create superclass" and "connect superclass" where a new superclass is created for a given class and can be become superclass of further classes. Moreover, the complex refactoring "pull up attribute" is shown. If each subclass contains an attribute with the same name and type, it can be pulled up to their common superclass.

Refactoring rule "CreateSuperclass(EString c, EString s, boolean a)" in Fig. 4 has parameters "c" and "s" to determine the name of the child class and of the new super class. Moreover, we have to decide if the new super class shall be abstract. The LHS describes the pattern to be found for refactoring consisting of a class which will be the child and the package it belongs to. The RHS shows the new pattern after refactoring where a new class with name "s" has been created which is the super class of the given class. The super class shall be contained in the same package as the class its subclass. Fig. 4 shows the left and the right-hand sides of the rule. Objects which are preserved occur in both parts. If two objects correspond to each other, they are colored and numbered equally.

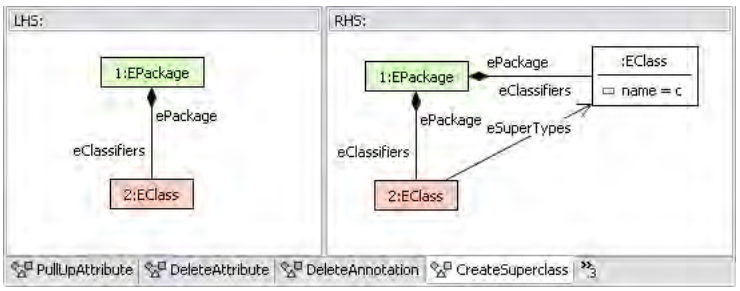

Figure 4: Rule "CreateSuperclass"

After creating a super class, it might become super class of more than one class which can be restructured by rule "ConnectSuperclass(EString s, EString c)" in Fig. 5. A class is allowed to become subclass of a given class, if that class does not have attributes and references yet. These additional conditions are expressed by two NACs "No Attribute" and "No Reference" which check that the class does not have an EAttribute and does not have an EReference to some class.

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. 6 checks for the class named "c" if there is a subclass not containing an attribute named "a". This

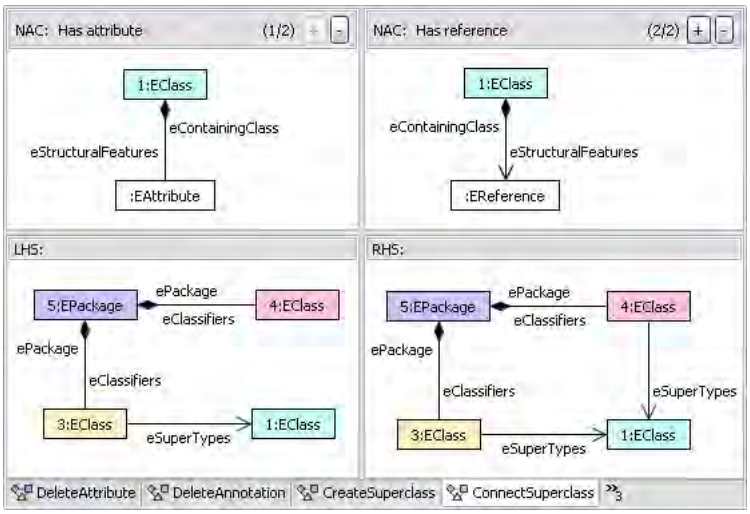

Figure 5: Rule ”ConnectSuperclass”

rule can be applied at most once, since there are NACs which check if there is already a subclass with this annotation. Thereafter, we try to apply rule “PullUpAttribute(EString c, EString a)” in Fig. 7. If there is no subclass of the class named “c” which has an annotation with source ” no attribute” and if the class named “c” has not already an attribute named “a”, it looks for a subclass which has an attribute named “a”. After the refactoring, an existing attribute with name “a” is pulled up. 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.

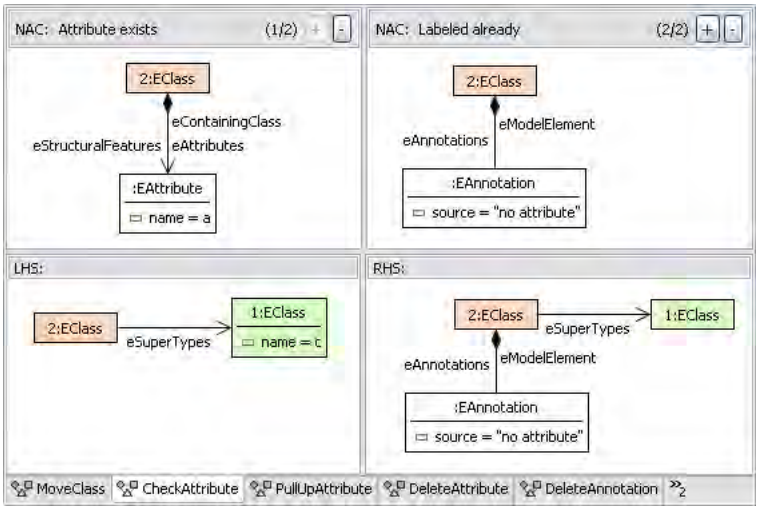

Figure 6: Rule ”CheckAttribute”

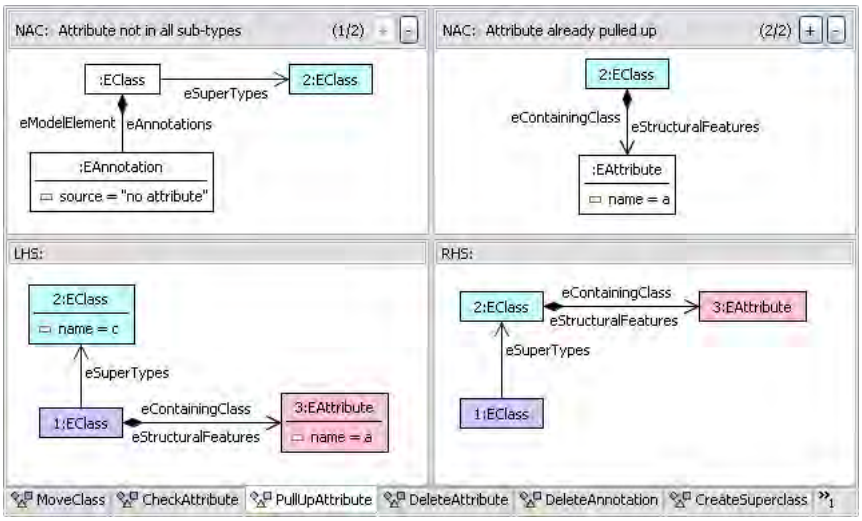

Figure 7: Rule ”PullUpAttribute”

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. 8 applying it as long as possible. Finally, if the refactoring was not successful, all new annotations have to be deleted again which is performed by rule “DeleteAnnotation()” in Fig. 9. 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 Table 1.)

Layer	Rule
1	CreateSuperclass, ConnectSuperclass, CheckAttribute
2	PullUpAttribute
3	DeleteAttribute
4	DeleteAnnotation

Table 1: Control flow for refactoring “pull up attribute”

Compared to the first refactorings, the implementation of refactoring “pull up attribute” looks rather complicated. Four rules instead of one are needed, since our approach does not support complex preconditions which allow to check for-all-conditions. For example, refactoring “pull up attribute” is allowed only, if all direct subclasses contain that attribute to be pulled up. After this refactoring, the resulting model has to be updated in the sense that the subclasses do not have to contain that attribute anymore. Thus, rules should have a for-all-operator for deletion and/or creation of graph parts. We restricted our transformation approach such that for-all-conditions and -operators are supported, because we provide algebraic graph transformation as formal basis for validation purposes, provided by AGG. Since there are also formal concepts for graph transformation with for-all-conditions and -operators around, it is up to

future work to extend the analysis techniques for such extended graph transformation and to implement them.

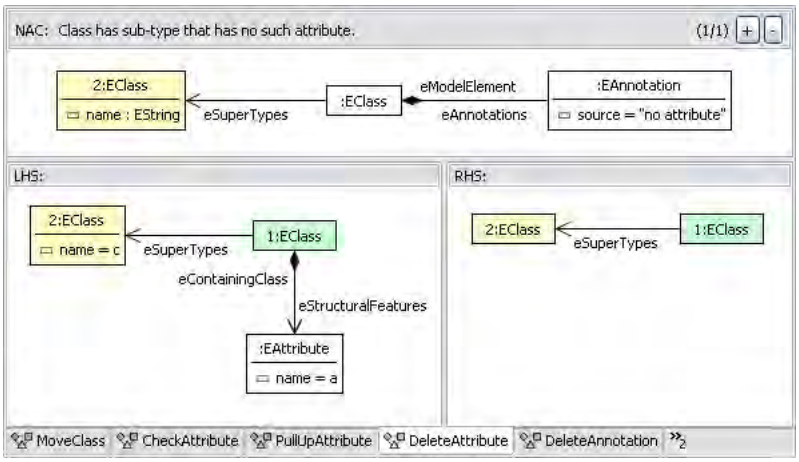

Figure 8: Rule ”DeleteAttribute”

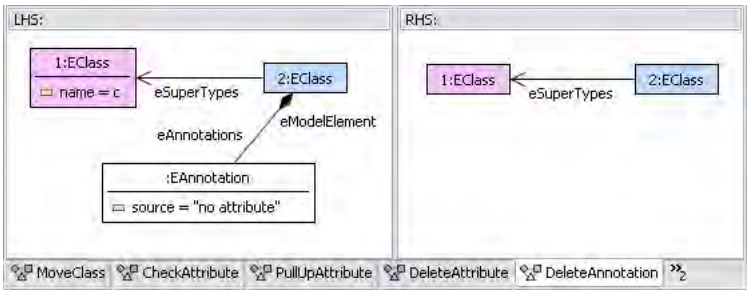

Figure 9: Rule ”DeleteAnnotation”

Applying first rule “CreateSuperclass(“Transition”, “NamedElement”, “true”) to class “Transition” of the Petri net model in Fig. 1. Second, rule applications “ConnectSuperclass(“Place”, “NamedElement”)” and “ConnectSuperclass(“Petrinet”, “NamedElement”)” are performed. Thereafter, attribute “name” is pulled up by first applying “PullUpAttribute” once, e.g. to class “Place”. Thereafter, rule “DeleteAttribute” is applied twice (to classes “Petrinet” and “Transition” in this case). Rule “CheckAttribute” does find a match, since all subclasses of “NamedElement” have attribute “name”. Similarly, rule “DeleteAnnotation” is not applicable.

4 Consistency of EMF Model Refactorings

Consistency of model refactorings can be understood in different ways: (1) In the software development process different kinds of artefacts such as models, programs, documentations,etc. occur. If the model

is changed, the corresponding code and documentation has to be changed accordingly (see [MT04] for further information). Following the model-driven software development paradigm thoroughly, code would be changed accordingly as soon as the code generator uses the refactored model. Thus, consistency between model and code could be easily achieved in this case. (2) More basically, consistency of model refactorings should also mean a kind of syntactic correctness in the sense that the refactored model is still an element of the given modeling language and fulfills certain validation properties. In the following, we concentrate on this second kind of consistency, while the first kind is out of scope of this paper.

Similarly to MOF, modeling languages are defined with EMF by a class model defining the model elements and their relations. Since all refactoring rules are typed over the EMF core model, i.e. the meta model, the refactored model is still correct wrt. its meta model. If the meta model contains additional consistency constraints, they have to be checked after each refactoring to make sure that the resulting model is still consistent.

4.1 Consistency with Graph Transformation

To open up the possibility for formal validation of EMF model refactorings, another kind of consistency is needed. In the following, we consider EMF model refactorings as consistent if they can be compared with graph transformations. In this case, the formal analysis techniques for graph graph transformation become available also for EMF refactoring. Results on conflicts and dependencies of refactorings for example, can help the developer to decide which refactoring is most suitable for a given model and why.

Although EMF models show a graph-like structure and can be transformed similarly to graphs [EEPT06], 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 classes. Since an EMF model may have non-containment references in addition, the following question arises: What if a class which is transitively contained in a root class, has non-containment references to other classes not transitively contained in some root class? In this case we consider the EMF model to be inconsistent.

A transformation can make an EMF model inconsistent, if its rule deletes one or more objects. For example an inconsistent situation occurs, if one of these objects transitively contains an object referred to by a non-containment link. To restore the consistency, all objects to be deleted have to be determined. Thereafter, all non-containment references to these indicated objects have to be removed, too. To ensure consistent transformations only, rules which delete objects or containment links or redirect them, have to be equipped with additional NACs.

If a containment link is deleted, the corresponding contained object has to be deleted, too. This object is not allowed to contain further objects. This constraint has to be required with a separate NAC. If a containment link is set or redirected and a maximum multiplicity is set on the container's end, an application condition is needed which checks that this multiplicity has not yet been reached, i.e. there do not exist objects which would be without container after rule application.

Similarly to the handling of deleted structures, consistency recovery is also applied to newly created objects. If a rule creates objects which are not transitively contained in one of the root objects, the consistency recovery will remove these objects at the end of a rule application. It is easily possible to forbid the application of those rules entirely, since inconsistencies on creation of objects can be determined

statically.

Our visual editor for EMF transformation rules is able to check the consistency with graph transformation discussed above. Every time an inconsistency is found, the inconsistent part is highlighted and the inconsistency is further described to inform the user.

4.2 Consistency of Selected EMF Model Refactorings

All selected refactoring rules are typed over the Ecore model, thus also the refactored EMF models will be typed over Ecore.

Considering the consistency of the selected EMF model refactorings, we define all objects of type "EPackage" as root objects. Now we check the consistency of those refactoring rules defined in the previous section: Rules "CreateSuperclass", "ConnectToSuperclass", "CheckAttribute", and "PullUpAttribute" are consistent by definition. Rule "DeleteAttribute" could cause an inconsistent model in principle, but does not do so, since "EAttribute" objects cannot have children and all possible links occur in the LHS. In contrast, rule "DeleteAnnotation" defines the deletion of an object which may have children due to the Ecore model, but does not have any applying only the refactoring rules. To ensure a consistent rule application only, rule "DeleteAnnotation" could be extended by a NAC checking that there is no EObject connected to the given EAnnotation. In this way we get a set of consistent refactoring rules only which can be translated to a set of corresponding graph transformation rules for further validation. In this case, the EMF refactoring rules can be translated to AGG, a tool environment for algebraic graph transformation where they might be further analyzed. Since all refactoring rules in the running example preserve the consistency of EMF models, analysis techniques such as critical pair analysis, termination checks, etc. are available also for EMF model refactorings. For example in [MTR04], critical pair analysis was used to detect conflicts and dependencies between refactorings of class models.

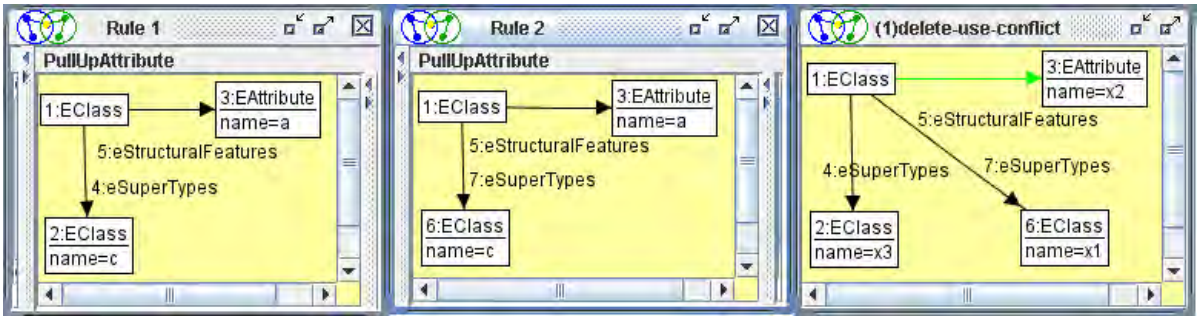

Figure 10: Sample critical pair in AGG

As a sample critical pair a "delete-use-conflict" is shown in Fig. 10 produced by the critical pair analysis of AGG between rule "PullUpAttribute" two times. The overlapping graph at the right-hand side of Fig. 10 depicts a situation where edge "5:eStructuralFeatures" is critical, since it is deleted during the first application and cannot be used again during the second application of "PullUpAttribute". Considering our example, this situation is not critical, since one and the same "EAttribute" is intended to be pulled up only once.

5 Conclusion

In this paper we presented an approach to specify the refactoring of EMF models by endogenous, in-place EMF model transformation. We use a recently developed EMF model transformation engine [BEK+] which is based on algebraic graph transformation concepts. Due to this formal basis, a formal analysis of conflicts and dependencies of EMF model refactorings can be performed. A necessary presumption for this kind of formal validation is the consistency of EMF refactorings with graph transformations. We could show that the selected EMF model refactoring rules fulfill this kind of consistency.

It is up to future work to investigate further EMF model refactorings, to answer the following questions: How far are EMF refactorings similar to UML refactorings? Which of the EMF refactorings are consistent with graph transformation (as defined above) and thus, can be formally analysed concerning conflicts and dependencies?

Future work also includes the development of a comprehensive environment for EMF model refactoring. A visual editor and a transformation engine with code generator presented in [BEK+] are already available at http://tfs.cs.tu-berlin.de/emftrans. Further tools such as a visual debugger and validation tools would be helpful.

References

[AGG06] *AGG-System* http://tfs.cs.tu-berlin.de/agg/, 2006.

[BEK+] E. Biermann, K. Ehrig, G. Kuhns, C. Köhler, G. Taentzer, and E. Weiss. Graphical Definition of Rule-Based Transformation in the Eclipse Modeling Framework. In *Springer LNCS. 9th Int. Conf. on Model Driven Engineering Languages and Systems.* To appear.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer, 2006.

[EMF06] *Eclipse Modeling Framework (EMF)* http://www.eclipse.org/emf, 2006.

[EMO06] *Essential MOF (EMOF) as part of the OMG MOF 2.0 specification* http://www.omg.org/docs/formal/06-01-01.pdf, 2006.

[JK06] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, NfC, MDD, WUsCAM, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers, LNCS 3844, edited by Jean-Michel Bruel. Springer Berlin / Heidelberg, pages 128–138*, 2006.

[LS05] M. Lawley and J. Steel. Practical Declarative Model Transformation With Tefkat. In J. Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*. Springer, LNCS 3844, 2005.

[MB05] Slavisa Markovic and Thomas Baar. Refactoring ocl annotated uml class diagrams. In *MoDELS*, pages 280–294, 2005. URL: http://lgl.epfl.ch/pub/Papers/baar-2005-models.pdf.

[Mer06] *Merlin Generator* http://sourceforge.net/projects/merlingenerator/, 2006.

[MT04] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004.

[MTF05] *IBM Model Transformation Framework* http://www.alphaworks.ibm.com/tech/mtf, 2005.

[MTR04] 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.

[MVG06] 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.

[Por03] Ivan Porres. Model Refactorings as Rule-Based Update Transformations. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 159–174. Springer, 2003. URL: http://citeseer.ist.psu.edu/porres03model.html.

[SPTJ01] Gerson Sunye, Damien Pollet, Yves Le Traon, and Jean-Marc Jezequel. Refactoring UML models. In *The Unified Modeling Language*, pages 134–148, 2001.