A Visual Model Transformation Environment for the Eclipse Modeling Framework

A Diploma Thesis submitted at the Institute for
Software Technology and Theoretical Computer Science,
Technical University of Berlin

Christian Köhler
October 2006

Die selbstständige und eigenhändige Anfertigung dieser Diplomarbeit versichere ich an Eides statt.

October 18, 2006 Christian Köhler

Abstract

Model driven engineering as a methodology for designing and implementing systems and processes is established for years now. The industry on one hand and Open Source communities on the other hand provide a large variety of frameworks for model driven development processes. The Eclipse Modeling Framework (EMF) implements a modeling approach by providing code generation facilities for structural data models. Although there already exist proposals for EMF model transformations, a graph-based approach, where model transformations can be defined in a visual, rule-based manner has not been considered yet.

In this thesis, a formal interpretation of modeling concepts is presented and a model transformation approach for EMF is introduced, based on formal graph transformation. An implementation is provided in the form of a graphical editor for the Eclipse platform, that allows the visual definition of in-place transformations for EMF-compliant models. EMF model instantiations are interpreted as attributed typed graphs with special *containment edges*. The order properties of these containment edges are formalized and conditions are stated, that must be satisfied to apply transformation rules to this kind of graphs. Further, a concept of consistency is introduced, that ensures well defined instantiations of EMF models. Applications of the framework are given through two examples, one for endogenous and one for exogenous model transformations.

Zusammenfassung

Modell-getriebene Software-Entwicklung als Methodik um Systeme und Prozesse zu entwerfen und implementieren, hat sich schon seit Jahren durchgesetzt. Die Industrie auf der einen Seite und Open Source Communities auf der anderen, haben eine Vielzahl von Modellierungswerkzeugen und -umgebungen hervorgebracht. Das Eclipse Modeling Framework (EMF) realisiert einen solchen Modellierungsansatz, indem es Codegenerierung für strukturelle Datenmodelle ermöglicht. Zwar existieren bereits Modell-Transformationsansätze für EMF, ein auf formaler Graph-Transformation basierender Ansatz, bei dem Transformationen regelbasiert und visuell definiert werden können, wurde aber bisher noch nicht betrachtet.

In dieser Arbeit wird eine formale Betrachtung von Modellierungskonzepten gegeben und ein Modell-Transformationsansatz für EMF vorgestellt, basierend auf Graph-Transformationssystemen. Eine Implementierung liegt in Form eines graphischen Editors für die Eclipse Plattform vor, der eine visuelle Definition von sogenannten *In-place* Transformationen für EMF Modellen ermöglicht. Instanziierungen von EMF Modellen werden als attributierte, getypte Graphen mit speziellen *Containment*-Kanten interpretiert. Es werden die Ordnungseigenschaften dieser Kanten formal definiert und Bedingungen angegeben, unter denen eine Transformation von solchen Graphen möglich ist. Weiter wird ein Konsistenz-Begriff eingeführt, der wohldefinierte EMF Modell-Instanziierungen definiert. Als Anwendung werden zwei Beispiele diskutiert, eines für endogene und ein weiteres für exogene Modell-Transformationen.

Acknowledgments

I would like to thank Dr. Gabriele Taentzer, Dr. Karsten Ehrig and Prof. Dr. Hartmut Ehrig for providing me with a challenging project and supporting me in a professional yet very personal way for the whole time.

Regarding the project's implementation, I would like to thank Eduard Weiss, Günter Kuhns and Enrico Biermann for their professional help.

For their physical and mental support I am very grateful to Philipp Jacob, Kristin Müller, Nina and Tobias Hacker, Igor Tunjic and Holger Lewin.

I would also like to thank my parents and my grandmother for their support during the past 5 years.

For her constant support I am especially grateful to my fiancée, Carola Krause.

Contents

Chapter 1

Introduction

1.1 Motivation

Model driven development is a methodology for describing and handling complex systems and processes. Examples for complex systems are not only large distributed environments, but also rather small systems like the real-time computation units in a car. Complexity is a general phenomenon. It is neither a domain specific problem nor limited to technical fields only.

Most approaches that try to solve this problem, describe systems and processes by defining *models* for them and use these models for further implementation and analysis. Modeling approaches in this way are able to handle the great complexity of large systems. However, the complexity can not really be removed, but only transferred to another level, because even if these models are able to simplify and state the structure or behavior of a system in a better way, analyzing and handling dependencies between multiple, often evolving models is a new upcoming complexity. Popular certified development processes like the Rational Unified Process[1] or the V-Modell[2] for instance define a number of phases, each including usually multiple models defined in multiple languages with a number of tools. In reality these processes involve the work of sometimes hundreds of developers, who often communicate only by exchanging models or specifications.

Therefore it is of great importance to be able to handle models and their relations, which is a goal of this thesis. By 'relations between models' the constructive concept of model transformation is meant, e.g. to automatically generate a so called *platform*

[1] `http://www.ibm.com/software/awdtools/rup/`
[2] `http://www.v-modell.iabg.de/`

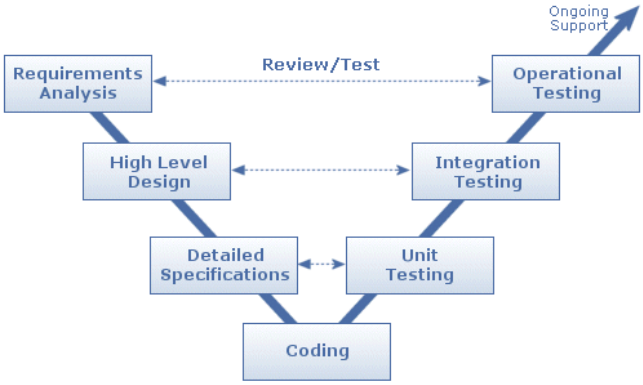


Figure 1.1: Work flow for a development process based on the V-Modell

specific model from a *platform independent model*, which is a major problem that occurs in almost every model driven development process. Another example for model transformations is the case of *refactoring* where it is the goal to modify a system's structure while keeping its behavior.

In this thesis, a combination of formal methods and popular modeling techniques is presented by implementing a graph transformation based approach. Since there is a large variety of modeling languages implementing different modeling paradigms, which can not simply be mapped to one holistic approach, we choose a concrete modeling technology, called EMF, and define transformations for this framework with the aim of being formal. The implementation of the presented approach is on one hand very intuitive, because its a visual language, and provides on the other hand reliable assertions about the transformation's semantics through formal analysis.

1.2 Models and Model Transformation

The term *model transformation* is used a lot in the context of model driven software development. The goal of model driven engineering in general is to give a complete specification for a system by defining a model describing all structural and behavioral features of the system. Such a model is often given in the form of diagrams representing views on multiple aspects of the system (structural, behavioral etc.). The notion of model is central to this approach and so is the one of model transformation. Since these terms are so important and basically come from the world of model driven development, it is interesting to see how these notions can be interpreted in a

formal way. As a motivation for how a connection between the world of model driven development and formal methods can be established, existing definitions of the term model / model transformations are compared in the following.

1.2.1 Formal View on Models

Models are usually thought of as some kind of semantic giving functions, e.g. a grammar of a programming language can be seen as an abstract definition for the class of valid programs for this language. On the other hand a specific program can be seen again as a formal definition for the class of all possible applications of this program (e.g. for different input values), which is called it's *operational semantics*.

Formal definitions of the term *model* are often made in the form of a mapping from an abstract schema (usually referred to as *syntax*) into a semantical domain. In first order logic the syntax is given by a set of formulas or a theory T, while it's model class $Mod(T)$ is defined as the class of structures fulfilling these formulas - again a mapping into a semantical space.

In category theory, the notion of model is usually defined as a so called *model functor*, which is basically a mapping from an abstract schema category into a semantical category (see [BW99, p. 42-43] for a precise definition). The abstract schema can be compared to what was referred to as syntax before. This schema is a definition of the common structure of all models of this schema. It is basically a *language definition*. For instance, the category generated by the graph S shown in (1.1) can be seen as such an abstract schema.

$$E \underset{target}{\overset{source}{\rightrightarrows}} V \tag{1.1}$$

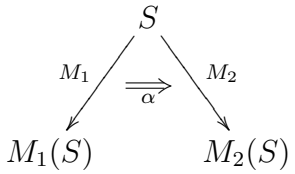
A model functor of the form $M : S \to \mathbf{Set}$ maps the objects E and V to certain sets and the arrows *source* and *target* to functions from $M(E)$ to $M(V)$. Since we interprete S as a language definition, we need to somehow state the *static semantics* of this definition. This is the *model class*, as mentioned earlier. So the static semantics of this language definition is the class of all functors into some semantical category.

Since $M(V)$, $M(E)$ are sets and $M(source)$, $M(target) : M(E) \to M(V)$ are functions, the structure described by this schema are graphs. The set $M(V)$ can be interpreted as nodes and $M(V)$ as edges of a graph. So in essence, the graph S is a schema whose models in **Set** are graphs again (!).

This principle, to use a language to define itself is usually referred to as *bootstrapping* of a structure or a language. Bootstrapping can be found at various points in computer science. One that will also appear in the presented approach later and that

turned out to be very important is the definition of a model based language using a *metamodel*, e.g. UML 2 is defined through a metamodel that itself is a UML 2 compliant model[3].

So in categorical terms, a model can be defined as a functor from an abstract schema into some semantical space. Category theory is one of the most generic theoretical frameworks and therefore has a lot of very abstract terminologies. One of them is the so called *natural transformation* between funtors, which is some kind of mapping from one functor to another. This terminology allows to define *model transformations* formally as natural transformations between model functors. Such mapping from one model functor to another can be visualized as shown in (1.2). The precise definitions would go beyond the focus of this thesis. For details see [BW99, p. 43] or [EB05].



(1.2)

1.2.2 Model Driven Engineering View

In the terms of model driven engineering, the notion of *metamodel* is central. A metamodel is just a usual model, but it is used in a special way. It defines a modeling language. Such a metamodel can be interpreted formally as an abstract schema in the way described before. A model is essentially the process of metamodel instantiation, as shown in Figure 1.2. While there is an endless number of models, the metamodel is fixed. All models have something in common and this common structure is defined by the metamodel. The relation between metamodel and model is called *instantiation* in one direction and *typed by* in the other one. While this relation is usually thought of as vertical, a mapping from one model to another is horizontal (on the same level). In the terms of model driven engineering, such a mapping is called a *model transformation*. Mathematically, it would rather be called a *homomorphism* in the simplest case (a mapping that respects certain constraints). Recalling the categorical interpretation, model transformations were defined as natural transformation between model functors, which is essentially the same as a homomorphism for the language defined by the abstract schema / metamodel (see [BW99]).

[3]The UML 2 metamodel actually is not defined using UML 2 but a language called MOF. Nevertheless, MOF is a part of UML 2 (*package merge*)

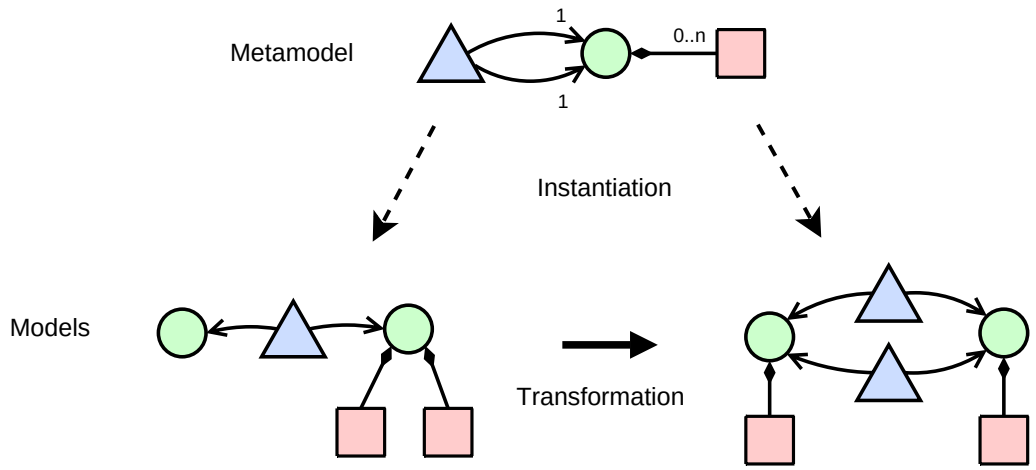


Figure 1.2: Instantiation as vertical relation between models and their metamodels. Transformation as horizontal relation between models typed by the same metamodel.

These categorical definitions are just a motivation for a formal point of view on notions of the world of model driven engineering. The terminology of model driven engineering is often quiet blurry and it is even harder to define a precise mathematical view on these notions, but as partially shown, it is possible to define these notions in a formal, mathematical way. Now why is this important? Because it is the only way to define terms like *correctness* or *consistency* for them. There is simply no common notion of correctness of a model transformation for instance. This is one of the biggest issues, when applying model driven engineering to sensitive fields like real-time environments, e.g. the very complex software of jumbo jets or other areas where safety and therefore correctness of models and the software that is generated by them is essential.

1.2.3 Structural Data Models

Structural data models define domain specific languages. EMF is all about these domain specific languages and therefore is defined by a metamodel for structural data models. These data models are basically class diagrams. They define a language through a number of classes and associations between them, that can include certain constraints, like multiplicities.

Such class diagrams can be formally interpreted as *attributed typed graphs* with additional semantical features, like inheritance and multiplicity constraints. Formal

transformations between such graphs can be either defined using graph homomorphisms directly between two data models, or rule-based on the level of model instantiations (*in-place* transformations). These rule-based transformations turned out to be very generic and are therefore the central concept in this thesis, forming the basis for model transformations for EMF.

1.3 Overview

In chapter 2, formal definitions for graphs and graph transformation system are made. They involve attribution and typing of graphs and the so called *gluing condition*, that is required to apply transformation rules wrt. a given match into a graph.

A short introduction to the concepts of the Eclipse Modeling Framework (EMF) is given in chapter 3. It is pointed out what features of EMF are required to define languages and how EMF models can be used in actual implementations. Further, a short comparison to another metamodeling language, called MOF, is given.

In chapter 4, the formal concepts of graphs introduced before are extended with formal versions of specifical features of EMF and other modeling languages. These are inheritance, multiplicities and containment associations. Graph transformations are extended to transformations of EMF model instantiations. A so called *containment condition* is stated that is required to have a proper defined transformation result. Further, it is shown what other constraints must be satisfied for instantiations of EMF models. This is done by introducing a concept of concistency of transformations.

Afterwards, the provided implementation is discussed. This mainly includes a graphical editor, that is based on an underlying transformation model.

Applications of the presented approach are given in chapter 6. EMF-Refactoring as an endogenous transformation on one hand and a mapping from simple class diagrams to relational database schema as an example for exogenous transformations on the other hand are discussed.

A conlusion and proposals for possible future work can be found in the last chapter. The appendix includes proofs for two theorems that are concerned with the concept of containment edges. An introductionary user guide for the model transformation framework can also be found in the appendix.

Chapter 2

Graph Transformation Systems

The basis of the presented model transformation approach are formal graph transformation systems. For that reason, a short introduction to typed attributed graphs, graph homomorphisms and transformations is given in the following. For details that go beyond the definitions made here, see [EEPT05] (graph transformation) and [BW99], [EB05] (category theory).

2.1 Typed Attributed Graphs

The first goal is to state the structure of EMF models and later of EMF model transformations in a formal manner. That is the starting point for describing important properties like termination, confluence and consistency of model transformations. As mentioned before, EMF models are interpreted as *graphs* in our approach.

Definition 2.1.1 (Directed graph)**.** A directed graph $G = (V, E, source, target)$ consists of a finite set of nodes V (vertexes), a finite set of edges E and two functions $source, target : E \to V$ that assign to each edge a source and a target node.

All kinds of graphs used here are defined in this way and therefore always have directed edges. This very basic definition can be extended in a couple of ways, e.g. the nodes and/or edges can be labeled, can have certain constraints or basically can have some kind of other structure added. In particular, adding algebraic information to the nodes/edges is usefull, which leads to what is called *attributed graphs*. Before adding any structure, it is first necessary to define a concept of mappings between these graphs.

Definition 2.1.2 (Graph homomorphism)**.** A graph homomorphism $h = (h_V, h_E) : G_1 \to G_2$ is a pair of functions $h_V : V_1 \to V_2$ and $h_E : E_1 \to E_2$ with the property that $h_E \circ source_2 = source_1 \circ h_V$ and $h_E \circ target_2 = target_1 \circ h_V$, which is the same as that the following diagrams commute.

$$\begin{array}{ccc} E_1 & \xrightarrow{source_1} & V_1 \\ {\scriptstyle h_E}\downarrow & = & \downarrow{\scriptstyle h_V} \\ E_2 & \xrightarrow[source_2]{} & V_2 \end{array} \qquad \begin{array}{ccc} E_1 & \xrightarrow{target_1} & V_1 \\ {\scriptstyle h_E}\downarrow & = & \downarrow{\scriptstyle h_V} \\ E_2 & \xrightarrow[target_2]{} & V_2 \end{array} \tag{2.1}$$

A graph homomorphism h is injective if both h_V and h_E are injective. Having these basic notions we want to extend them by adding type information. Nodes and edges of a graph can be typed by nodes / edges of a given *typing graph*. Homomorphisms for this kind of graphs should preserve the typing information.

Definition 2.1.3 (Typed graph)**.** Given a distinguished *typing graph* T, a graph G is typed by T, if there is a graph homomorphism $type : G \to T$. This *typing graph homomorphism* assigns to each node and edge in G a node type / edge type.

Definition 2.1.4 (Typed graph homorphism)**.** For a typed graph homomorphism $h : G_1 \to G_2$, both graphs G_1 and G_2 must have the same typing graph T and the following diagram has to commute.

$$\begin{array}{ccccc} & & T & & \\ & {\scriptstyle type_1}\nearrow & = & \nwarrow{\scriptstyle type_2} & \\ G_1 & & \xrightarrow[h]{} & & G_2 \end{array} \tag{2.2}$$

Attribution of graphs is not given formally here. For each node and edge in a typing graph, a set of attributes can be declared. For an application in a context of a modeling language like EMF, attributes of edges are not considered here. In a node of a typing graph, the attributes are identified through a unique name and have a specified type. Here we only consider primitive-valued attributes, i.e. strings, integers etc. Each instantiation of a typing graph (a graph that is typed by it) includes a set of nodes and edges that have a specific node/edge type. The nodes of the typed graph can further include values for the attributes as defined in the typing graph. Formal definitions of attributed graphs are based on an integration of Σ-algebras into graphs (see [EEPT05]).

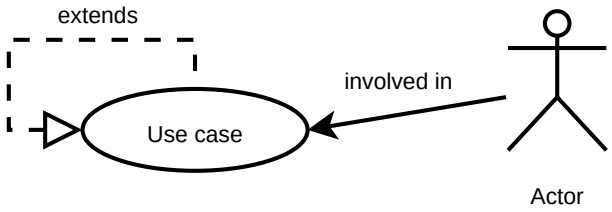


Figure 2.1: Typing graph for Use case diagrams in concrete syntax. There are two node types (*Use case*, *Actor*) and two edges types (*extends*, *involved in*).

2.2 Rule based Transformations of Graphs

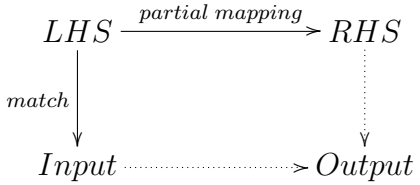
Graph transformations are defined using transformation rules. These rules consist of a left-hand side, a right-hand side, a mapping between those two and possible negative application conditions. A left-hand side (LHS) of a rule is a graph that stands for the structural preconditions that must be fulfilled to apply the rule. Accordingly a right-hand side (RHS) is a graph that describes the result (or postconditions) of a rule. Negative application conditions (NACs) are also graphs, that describe structural conditions that must not be fulfilled to apply the rule.

Nodes in the LHS of a rule can be mapped to nodes in the RHS and also to nodes in the NACs. Those nodes in the LHS, which are mapped to the RHS, will be preserved during transformation. Nodes in the LHS, which have no mapping to the RHS are being removed. Accordingly, nodes in the RHS without a mapping source will be created during rule application.

To apply a rule it is necessary to define a match from the LHS into the graph that should be transformed. Such a match can be given explicitly or can be found using match finding algorithms. In general, finding a match of two graphs is a problem that cannot be solved efficiently. However, typing information decreases the complexity of finding a match a lot and usually the transformation rules are also rather small (consist of a few nodes and edges). Often there is also at least a partial match given (see the example of refactoring in chapter 6). Even though the theoretical problem is very complex, computing matches for typed graph transformations is feasible in all realistic applications.

In (2.3) it is shown that a rule consists of a left-hand side, a right-hand side and a partial mapping between these two graphs (plus possible negative application

conditions). Such a rule together with an input graph and a match from the left-hand side into the input graph are the necessary information to compute a rule application, i.e. a direct transformation step. While the solid lines in (2.3) must be given, the dotted lines represent the actual computation that is performed to derive the transformation result.



(2.3)

As already mentioned, the match describes the structural pattern that must be found in the input graph. Adding the algebraic features of typed attributed graphs, it is also possible to define pre- and postconditions for attribute values. For this, primitive-typed variables are used which can be evaluated from a match or given explicitly as an input value.

In the following, the concept of graph transformation rules and graph transformations (also called *productions*) is introduced. We make these definitions on simple graphs only and do not consider attributes and types. Negative application conditions are also handled in a informal way. For further details see [EEPT05].

Definition 2.2.1 (Graph transformation rule)**.** A graph transformation rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ consists of three (typed) graphs L, K, R together with two injective (typed) graph homomorphisms $l : K \rightarrow L$ and $r : K \rightarrow R$.

The graph K can be seen as a gluing graph which connects the left-hand side with the right-hand side. As described before matches between a LHS and a RHS are in general partial maps. To avoid the difficulties with partial maps the graph K in combination with injective graph homomorphisms is used. There are basically two approaches to define transformations in this context, both are concepts coming from category theory. We use here the latter one of the following two:

- Single-Pushouts (SPOs): rules are partial maps between LHS and RHS, like shown in 2.3.
- Double-Pushouts (DPOs): rules are a span of injective homomorphisms, like in definition 2.2.1.

Definition 2.2.2 (Direct graph transformation)**.** Given a graph transformation rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$, a direct graph transformation $M \overset{p}{\Rightarrow} N$ consists of two pushouts in the category of graphs:

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & (PO) & {\scriptstyle c}\downarrow & (PO) & {\scriptstyle n}\downarrow \\ M & \xleftarrow{g} & C & \xrightarrow{h} & N \end{array} \tag{2.4}$$

As a generalized version, a graph transformation $M \overset{*}{\Rightarrow} N$ is a sequence of direct graph transformations $M = M_0 \overset{p_1}{\Rightarrow} M_1 \overset{p_2}{\Rightarrow} ... \overset{p_n}{\Rightarrow} M_n = N$.

The morphism m is the match and can be either (partially) given or being computed by solving a constraint problem. In general, it is not always possible to construct the context graph C in a proper way. Therefore it is necessary to ensure some properties of the match m in combination with the left mapping l of the transformation rule, so that the transformation in fact can be computed with a valid and unique result. The following definitions exactly reflect these constraints.

Definition 2.2.3 (Gluing condition)**.** Given a graph transformation rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ and a match $m : L \rightarrow M$ the gluing condition is defined as follows:

- Gluing points: $GP = l_V(V_K) \cup l_E(E_K) = l(K)$
- Dangling points: $DP = \{v \in V_L | \exists e \in E_M \; m(E_L) : m_V(v) = source(e) \vee m_V(v) = target(e)\}$
- Identification points: $IP = \{v \in V_L | \exists v' \neq v : m_V(v') = m_V(v)\} \cup \{e \in E_L | \exists e' \neq e : m_E(e') = m_E(e)\}$
- Gluing condition: $DP \cup IP \subseteq GP$

The sets GP, DP and IP represent certain nodes and edges in the left-hand side of the rule (graph L). Gluing points are those nodes or edges which are being preserved during rule application. Dangling points are those nodes, which have an image in M that is a source or a target of an edge in M, while this edge has not an origin in L. These dangling edges would produce something like shown in Figure 2.2, that is not a valid graph anymore. On the other hand, if the match is not injective it is not always possible to decide whether a node should be deleted or preserved by

the transformation rule (see Figure 2.3). The set of identification points IP includes all nodes and edges that are non-injectivly matched to points in M. The gluing condition states, that dangling points and identification points also must be gluing points. So these points must be preserved by the rule. Otherwise, the result of the transformation is not a valid graph. For a more detailed description see [EEPT05].

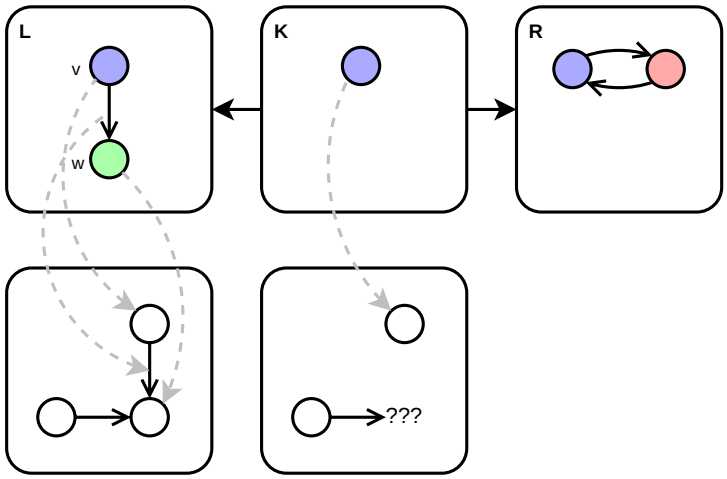


Figure 2.2: Gluing Condition: node w is a dangling point, but not a gluing point.

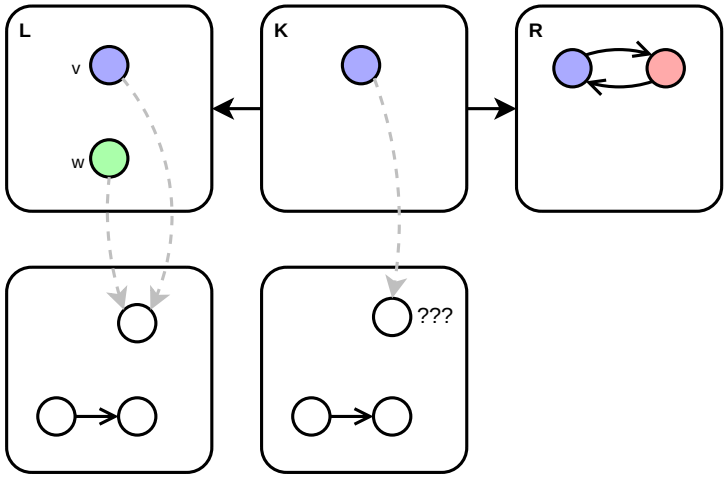


Figure 2.3: Gluing Condition: node w is a identification point, but not a gluing point.

Chapter 3

Model Driven Engineering

3.1 Eclipse Modeling Framework

EMF is a modeling framework for the Eclipse platform. It can be compared to the MOF specification (see chapter 3.2). The basis and simultaneously the language definition of EMF is a metamodel, called Ecore. This metamodel defines all entities that can appear in a EMF-compliant model. EMF only defines structural features and no possibilities for specifying behavior. Its strength is not a rich language, but the possibility to derive implementations that fulfill all (!) semantical features described in a model. Additionally it includes features like notification and persistence and a powerful reflective API.

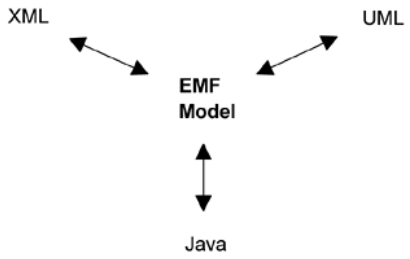


Figure 3.1: EMF as a bridge between modeling (UML/MOF), programming (Java) and persistence (XML), [BSM+03]

.

It is important to note that the Ecore metamodel again is an EMF model. The metaclasses `EClass`, `EDatatype`, `EReference` etc. cannot only be interpreted as, but in fact *are* classes of an EMF core model. This is of great importance for the presented

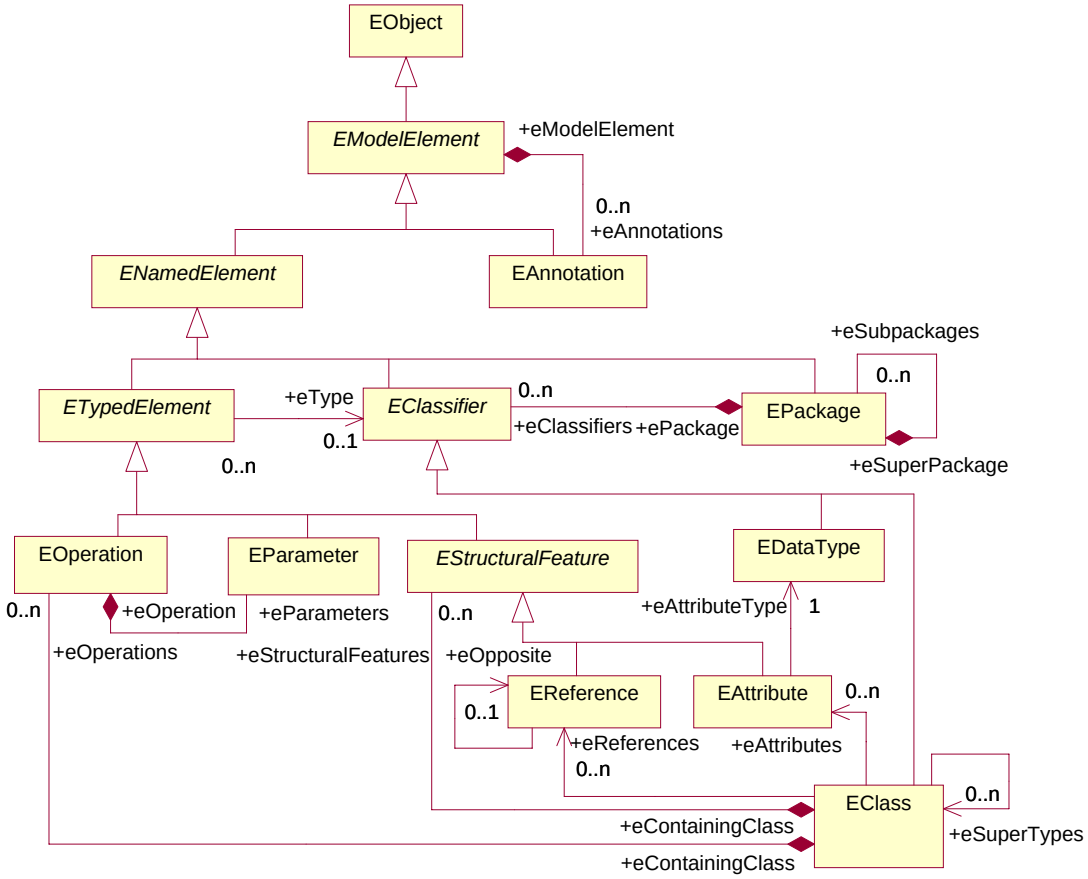


Figure 3.2: Ecore metamodel: Kernel

approach, since through that it is possible to use native EMF notions (elements of the metamodel) for the definition of transformation rules. These notions are interpreted in terms of graphs, so that the basis for the transformation engine are formal graph transformations systems.

3.1.1 Ecore Metamodel

The Ecore metamodel defines a language for structural data models. This language is meaningful enough to describe itself. For that it includes the most common entities found in almost every (meta-) modeling language, which come from the paradigm of object-oriented programming languages:

- Classes (metaclass `EClass`)
- Associations (metaclass `EReference`[1])
- Attributes (metaclass `EAttribute`)
- Datatypes (metaclass `EDatatype`)

Classes together with associations are the most important concept of EMF and are explicitly discussed in section 3.2.1. As shown in Figure 3.2 a set of classes can be organized into a package (metaclass `EPackage`), which can be further organized hierarchical in the way that a package can have an arbitrary number of subpackages. The abstract metaclass `EClassifier` summarizes the common properties of classes and data-types. Further there are the abstract metaclasses `ETypedElement`, `ENamedElement`, `EModelElement` and `EStructuralFeature`, which are used in the same way to avoid duplicate properties of classes. Attributes (metaclass `EAttribute`) and methods (metaclass `EOperation`) are also important features of EMF classes and classes in object-oriented languages in general.

The metaclass `EObject` plays a special role in EMF, since all other metaclasses inherit from `EObject`. It defines a number of methods, which can be used to compute derived properties of objects, e.g. its type. This is the starting point for the reflection mechanisms of EMF in general (see chapter 3.1.3) and the bootstrapping of EMF in particular.

3.1.2 Code Generation

EMF models can be directly translated to Java code. This code can be seen as a run-time data model of the structure defined in the class diagrams. Moreover the code generation provides a complete implementation, that manages the life cycle of objects (create, delete, set attributes etc.), while ensuring multiplicity and containment constraints. Further, a persistence API is provided, implementing load / save operations for model instances. The standard format for EMF models / model instances is XML / XMI. The code generated by EMF can be extended at any point. It is even possible to add, modify or remove features in the code and to automatically update the corresponding EMF model (reverse engineering).

An EMF model is translated into a `EPackage` during code generation. Such a package contains all information defined in the model. EMF Classes are translated to

[1] `EReference` represents association *ends*.

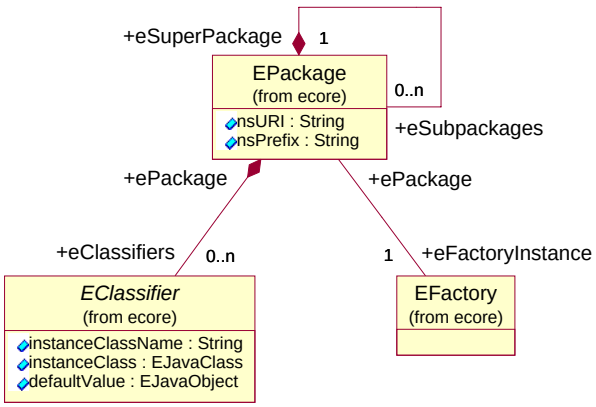


Figure 3.3: Ecore metamodel: Packages and factories

Java classes, that contain all attributes and references defined in the model. For each of these properties, getters and setters are generated, which ensure all constraints defined in the model (multiplicities, containments).

To each package there is a run-time factory, which is used for creating objects. Further an `EPackage` is identified using a unique namespace URI, also defined in the model. However, at run-time there can be more than one implementation of a package, e.g. the generated classes and a dynamic version of the model (see 3.1.3). For this matter there exists a package registry which can be used to ensure that a certain implementation is used. The registry basically maps a package namespace URI to a corresponding implementation.

3.1.3 Reflective API

EMF provides a complete reflective API, which can be used to determine all features defined in an EMF model. Given an arbitrary EMF object (instance of `EObject`) it is possible to derive its corresponding class using the method `eClass()`. An `EClass` can be further analyzed through a couple of methods. All references, containments, attributes and even methods (operations) can be computed with all meta information, like the type of such a feature for instance.

Due to the fact that all metaclasses of EMF inherit from `EObject`, it is possible to cope with the Ecore metamodel in the same way, so that the EMF metamodel in fact can be dealt with like a usual EMF model.

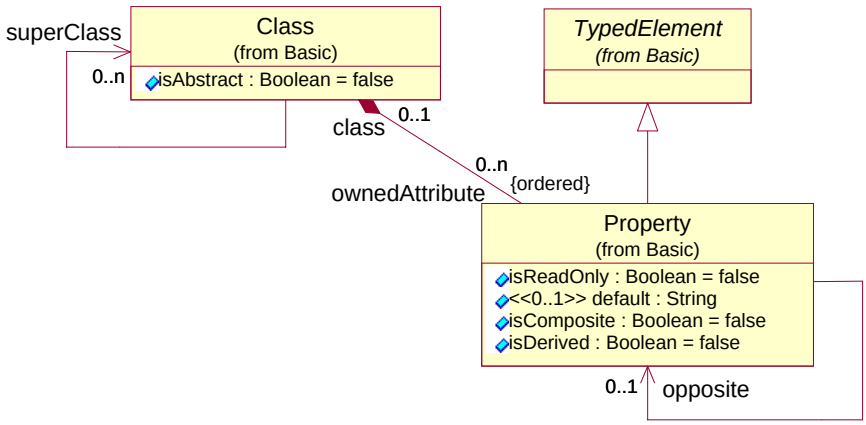


Figure 3.4: MOF metamodel: Classes and properties

3.2 MOF and EMF

MetaObject Facility (MOF) is an ISO standard initiated by the Object Management Group (OMG)[2]. It is the basis for a number of language specifications including CORBA IDL and UML 2. Like EMF, it consists of a metamodel and a number of so called technology mappings. These technology mappings are basically model transformation rules where the target model is usually source code or any other kind of structured textfiles (e.g. XML). Metamodels and there technology mappings are the essential parts of a Model Driven Architecture as described by the OMG.

3.2.1 Metamodels

The MOF metamodel is divided into two parts, Essential MOF (EMOF) and Complete MOF (CMOF). The underlying metamodel for EMF is called Ecore and is very similar to the EMOF specification. It includes its own versions of the most common datatypes (e.g. integers, string etc.) and further entities like classes, associations and packages.

One of the most important concepts described in the MOF metamodel as well as in the EMF metamodel are classes. Figure 3.4 shows that in MOF a class can inherit from a number of superclasses and have properties. These properties are always typed

[2] `http://www.omg.org/`

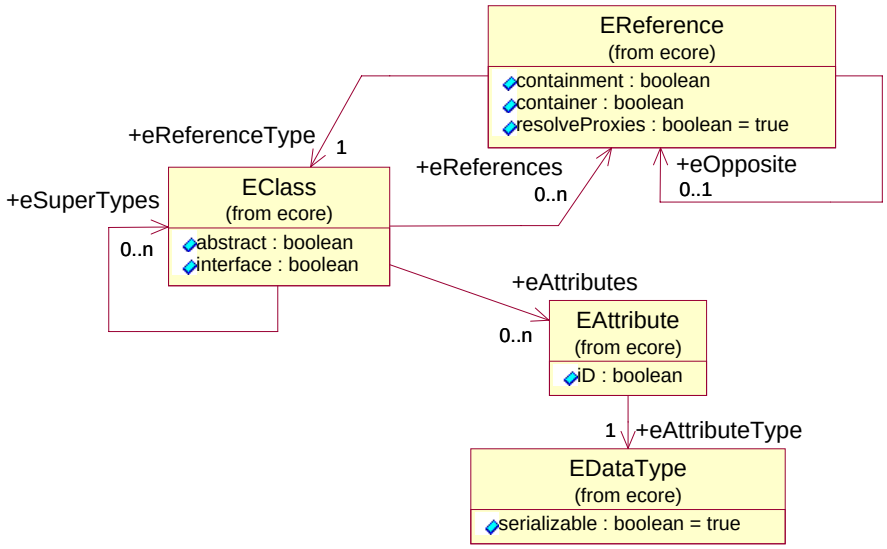


Figure 3.5: Ecore metamodel: Classes, attributes and references

because they are specializations of the abstract metaclass `TypedElement`. Further they can be either seen as simple primitive-valued attributes or as ends of a relation between two classes. If such a relation is navigable on both sides the `opposite`-link points to the other end.

The Ecore model on the other hand includes a metaclass `EClass` (Figure 3.5). The correspondence to its counterpart in MOF is obvious. Though there is a difference in the way how attributes are handled. While in MOF all kinds of attributes are modeled as instances of `Property`, there are two concepts for that in Ecore. The first one is `EAttribute` which is only used for primitive types. Therefore it has a typing link to an instance of `EDataType`. Complex properties of classes are modeled using `EReferences`, which contain like properties in MOF a link to the opposite end of the relation.

Associations can be marked as *containment* (see `EReference`) or in the case of MOF as *composite* (see `Property`). These special kinds of associations express that one object is the owner of another object. These semantics imply a couple of constraints that should always be fulfilled, e.g. an object should not be owned by itself. These constraints are formalized in chapter 4.

3.2.2 Technology mappings

Both, EMF and MOF define technology mappings to XML. Through them it is possible to generate a XML schema for a given EMF or MOF model and rules for transforming model instances into XML documents typed by the XML schema. These mappings are used for persistence in both modeling frameworks. In MOF it is called XML Metadata Interchange (XMI) which is the standard format for UML 2 models and diagrams. The EMF version is basically the same, but of course refers to the Ecore model instead of MOF. The mapping to XML is of particular importance, since it requires to translate the graph-like structure of object structures to the tree-like structure of an XML file. This is discussed in detail in chapter 4, that introduces transformations of EMF models.

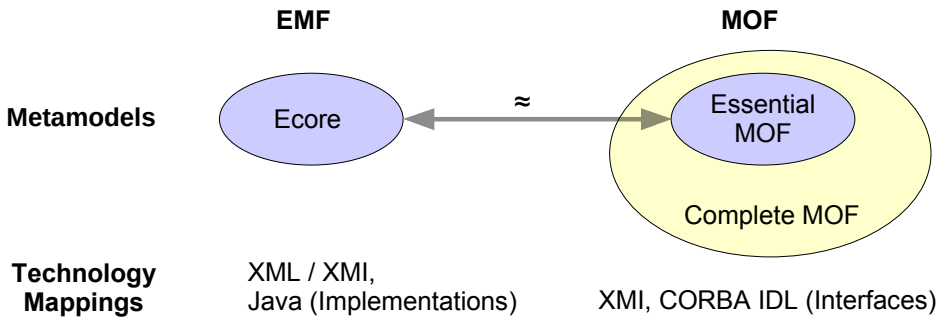


Figure 3.6: EMF and MOF metamodels and technology mappings

MOF also includes a mapping to CORBA IDL which can be used to generate code for classes and their attributes and method stubs. However, this code generation does not include any implementations. EMF on the other hand is designed for Java and the Eclipse platform and therefore includes a mapping to Java (code generation) which is not limited to interfaces, but also includes implementations for managing the life cycle of objects including validation of multiplicities and uniqueness, notification mechanisms and (de-)serialization to XML. This mapping is implemented using Java Emitter Templates (JET), which is a part of EMF.

Chapter 4

EMF Model Transformations

In chapter 2, an introduction to formal graph transformations has been given. The definitions made there are now transferred to the notions of EMF by interpreting classes as nodes and associations as edges in a typing graph with attribute definitions for nodes. In the same way, objects can also be interpreted as nodes and their links as edges of a typed attributed graph.

As described before, EMF has features that have no representation in formal graph transformation or at least cause a number of restrictions to the application of transformation rules. A complete formal interpretation of EMF notions in the terms of graphs leads to attributed typed graphs with inheritance, multiplicity and containment constraints. A formalization of attribution has been described already in a couple of publications about graph transformation systems (see [EEPT05]). It also has been shown, that multiplicities cause a restrictions in the application of transformation rules. A property that not has been analyzed before in the context of graph transformation system are containment edges, which are used in both EMF and MOF/UML.

4.1 Mapping Notions

Interpreting EMF notions in terms of formal graphs is motivated by the obvious graph-like structure of EMF models. Classes in an EMF model can be interpreted as nodes in a typing graph. Associations between classes can be seen as edges in a typing graph. Objects as instantiations of classes from an EMF model are comparable to nodes in a graph, that is typed by an EMF model, interpreted as typing graph.

As mentioned before, objects can be linked to each other by setting reference

values. These links can be interpreted as edges in a typed attributed graph. Such a typed attributed graph is basically just a set of linked objects, because the links can also be interpreted as properties of the objects. That is why such graphs are referred to as *object structures* here.

EMF notion	**Graph term**
Model	Typing graph with attribution, inheritance, multiplicities. Edges can be marked as *containments*.
Model instance	Typed, attributed graph with containment edges, inducing a tree-like structure. These graphs are referred to as *object structures*.
Class	Node in typing graph.
Object	Node in typed graph.
Association	Edge in typing graph (with possible multiplicities or containment marks).
Link	Edge in typed graph, that must not violate certain multiplicity and containment constraints.

Table 4.1: Mapping EMF notions to graph terminology.

In the following, the attribution of objects is not discussed in detail. The focus is rather set on the graph-like structure of EMF model instances. Besides the attributes of classes, EMF models further contain generalizations of classes, multiplicity constraints of association ends and containment edges as special kinds of associations.

4.2 Generalizations

Generalizations are an important property of object-oriented languages. EMF supports this concept in the way that classes can extend other classes. A subclass inherits properties of one or more superclasses. EMF allows multiple inheritance. However, Java supports only single inheritance, so that this is the usual case for EMF models, even though the metamodel allows more.

A formalization of the generalization concept can be done in various ways. Interpreting an EMF model as a typing graph, multiple inheritance can be formalized by adding a binary relation called *extends*.

Definition 4.2.1 (Typing graph with inheritance)**.** A typing graph with inheritance $T = (V, E, source, target, extends)$ is a graph together with a binary relation[1] $extends \subseteq V \times V$ with the following properties:

- $x\ extends\ y$ and $y\ extends\ z$ implies $x\ extends\ z$ (transitive).
- $(x, x) \notin extends$ for all $x \in V$ (non-reflexive).

This relation is transitive and antisymmetric, similar to the definition of partial orders. However, *extends* is not a partial order, because there is no reflexivity in the concept of inheritance. For a node type $x \in V$ this binary relation defines on one hand all super types $(x, \_)$ and on the other hand all subtypes $(\_, x)$. This definition reflects the possible usage of the concept of multiple inheritance in EMF and other modeling languages. An interesting fact about the EMF implementation is that all classes and meta classes are a specialization of `EObject`. This class provides basic reflective methods for deriving the object's type and other meta information.

A homomorphism for graphs with inheritance is a usual graph homomorphism with the additional restriction, that it must preserve the order properties of *extends*.

Definition 4.2.2. A graph homomorphism $h : G_1 \to G_2$ is also a valid homomorphism for graphs with inheritance, if it has the following property:

- $x\ extends_1\ y \ \Rightarrow\ h_V(x)\ extends_2\ h_V(y)$ (order properties are preserved).

This property can be compared to the usual definition of a monotonic function for (complete) partial orders. Note that this definition does not reflect an instantiation process, since it only defines homomorphisms between two typing graphs, but not between a typing and a typed graph.

For the graph transformation approach which is used in this thesis, inheritance is not really a restriction. Transferring this concept to transformation rules gives rise to an extended semantics of rule application in the way, that an object of type T in the LHS of a rule can be matched to an object of a subtype of T. So the definition of the term *match* must be extended for that. It is not a plain graph homomorphism anymore. This more general application scenario comes with no restrictions for the application of such rules. So there is no 'negative impact' on the transformation's semantics. Rules can be applied in the same way as before. Only the match finding is being generalized to instances of possible subclasses. For a complete formalization of inheritance for typed attributed graphs see [EEPT].

[1] As a short form for $(x, y) \in extends$ we write $x\ extends\ y$.

4.3 Multiplicities

Another important property of modeling languages are multiplicities of associations. Multiplicities constitute a real restriction to the application of transformation rules. Of course, this is only the case for finite multiplicities of association ends. In EMF, multiplicities are expressed using a upper and a lower bound of `EReferences`. These bounds are fixed integers that restrict the number of links between two objects in an object structure. A lower bound of 0 together with a negative number for the upper bound of an edge (usually -1) implies that there is no restriction for the multiplicity of an edge. In this situation there is no restriction in the application of a transformation rule. In all other cases there might appear a problem, which is a first motivation for introducing some kind of reasoning on transformation rules. There it is a goal, to analyze whether a rule might violate multiplicity constraints or not. We will later define the concept of *multiplicity consistent* rules, that provides a statical check, ensuring that transformations respect multiplicity constraints defined in an EMF model. For a complete theory about graphs with multiplicity constraints see [TR05].

4.4 Containments

Containment relations or *aggregations* / *compositions* are a special kind of unidirectional associations[2], expressing that an object is contained or owned by another object. Containment relations are an essential part of EMF models, because they define how the graph-like structure of an object structure can be mapped to a tree-like structure in XML files for example. At run-time it is always necessary to ensure that each object that is created by a transformation rule for instance has a proper container as defined in the model. Each object can have either no or exactly one container. It is not allowed that an object is (transitively) contained in itself.

At first glance, it seems valid to restrict the containment edges in a typing graph / class diagram already, in the way that there are no cycles, similar to the order properties of generalizations. This is avoided both in EMF as well as in MOF. Instead, containment constraints are formulated on the object level. In fact, it makes no sense to permit containment cycles in class diagrams as it doesn't allow to define recursive tree-like structures, e.g. a class `Tree` with two containment edges: `left, right:` `Tree` $\rightarrow$ `Tree`, modeling a binary tree. Another example is the Ecore metamodel, where the meta class `EPackage` has a containment edge to itself, representing a set of

[2]The term *unidirectional* has nothing to do with the navigability of an association in this context.

subpackages (see Figure 3.3). Nevertheless, containment constraints are important, but obviously can only be formulated on the object level. For this purpose, we now define special kinds of graphs, that have a distinguished set of so called containment edges. Like all other edges (as defined here), containment edges are directed. The source node of a containment edge is called *container* and the target is called *contained node*. Following the notation of EMF and MOF, the container ends of such edges are marked with a black diamond.

Definition 4.4.1 (Graph with containment edges)**.** A graph with containment edges is a graph $G = (V, E, source, target, C)$ with $C \subseteq E$ a set of so called *containment edges*. These containment edges induce the following transitive binary relation:

- $contains = \{(x, y) \in V \times V \mid \exists e \in C : (source(e) = x \wedge target(e) = y) \ \vee$ $\exists z \in V : (x \ contains \ z \wedge z \ contains \ y)\}$

The containment edges must have the following properties:

- $e_1, e_2 \in C : target(e_1) = target(e_2) \ \Rightarrow \ e_1 = e_2$ (at most one container).
- $(x, x) \notin contains$ for all $x \in V$ (no cycles).

This definition ensures that there are no containment cycles and that an object has at most one incoming containment edge and therefore also not more than one container. Accordingly, a homomorphism for graphs with containment edges is a usual graph homomorphism, that preserves containment edges and their order properties.

Definition 4.4.2 (Homomorphism for graphs with containment edges)**.** Given two graphs with containment edges G_1, G_2, a graph homomorphism $h : G_1 \rightarrow G_2$ is a valid homomorphism for graphs with containment edges, if it has the following property:

- $e \in C_1 \ \Rightarrow \ h_E(e) \in C_2$ (containment edges are preserved).

This property of homomorphisms is always satisfied for identities. It is also trivial to show, that the composition of these special graph homomorphisms again has this property. Due to that, graphs with containment edges and their homomorphisms form a category.

It is a goal to define EMF object structures as typed, attributed graphs with containment edges. The next step is to show, that graph transformations using the double pushout approach as defined in chapter 2, can also be applied for these special

kinds of graphs. For this, it has to be shown that the category of graphs with containment edges has pushouts.

Usual graphs have this property. Given an interface graph G_0 and two graph homomorphisms $f_1 : G_0 \to G_1$ and $f_2 : G_0 \to G_2$ a pushout graph can be constructed unique up to isomorphisms.

$$\begin{array}{ccc} G_0 & \xrightarrow{f_1} & G_1 \\ {\scriptstyle f_2}\downarrow & (PO) & \downarrow{\scriptstyle f_2'} \\ G_2 & \xrightarrow[f_1']{} & G_3 \end{array} \tag{4.1}$$

For graphs with containment edges it has to be shown that the pushout result satisfies the order properties of the containment edges and that f_1', f_2' are monotonic respectively to the *contains* relation of the graphs.

As shown in Figure 4.1, the resulting graph does not necessarily have the designated order property. Although the graphs G_0, G_1 and G_2 are valid graphs with containment edges, a containment cycle occurs in the resulting graph. The second example in Figure 4.2 shows, that also the 'at most one container for each object' - property can be violated when constructing a pushout. This gives rise to state a condition under which a pushout in the category **Graphs** is also a valid graph with containment edges, i.e. has no containment cycles and that each node has at most one container.

Definition 4.4.3 (Newly contained points)**.** For a homomorphism between graphs with containment edges $h : G_0 \to G_1$, the set of *newly contained points* $NCP \subseteq V_0$ is defined as

$$NCP_h = \{x \in V_0 \mid \forall y \in V_0 : \ (y, x) \notin \mathit{contains}_0 \ \wedge \ \exists z \in V_1 : \ (z, h_V(x)) \in \mathit{contains}_1\}$$

Definition 4.4.4 (Cyclic contained points)**.** For a span of homomorphisms between graphs with containment edges $(G_1 \xleftarrow{f_1} G_0 \xrightarrow{f_2} G_2)$, the set of *cyclic contained points* $CCP_{f_1,f_2} \subseteq NCP_{f_1}$ is defined as

$$CCP_{f_1,f_2} = \{x \in NCP_{f_1} \mid \exists y \in CCP_{f_2,f_1} : f_2(x) \ \mathit{contains}_2 \ f_2(y) \ \}$$

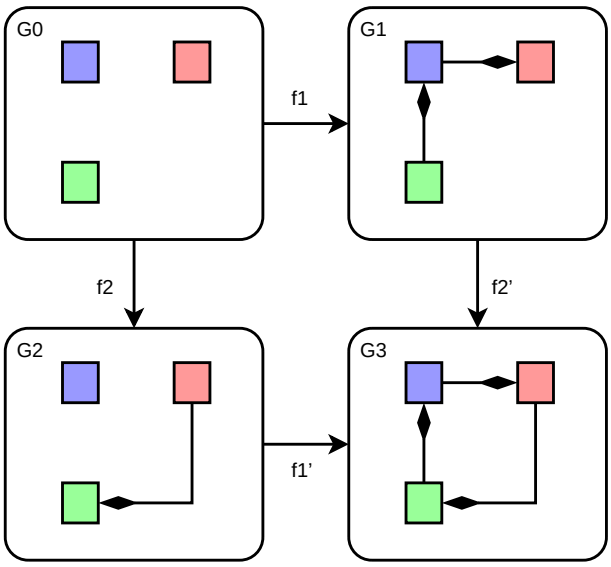


Figure 4.1: Invalid pushout of graphs with containment edges: the resulting graph G_3 has a containment cycle, which can be checked by computing the *cyclic contained points* of the span of morphisms.

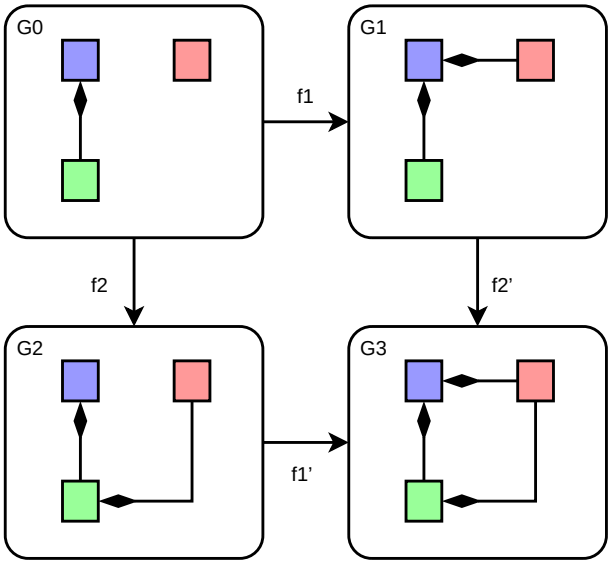


Figure 4.2: Invalid pushout of graphs with containment edges: the red node in the resulting graph G_3 has two containers. It is also the only common *newly contained point* of f_1 and f_2.

For a single homomorphism $h : G_0 \to G_1$ of graphs with containments, NCP_h is the set of nodes in G_0 that don't have a container in G_0, but do have one in G_1. A so called cyclic contained point $x \in G_0$ of a span of morphisms $(G_1 \overset{f_1}{\leftarrow} G_0 \overset{f_2}{\rightarrow} G_2)$ is a newly contained point of f_1, that corresponds to a newly contained point y of f_2 in a certain way. That is, that x becomes a (transitive) container of y in the image of f_2. The node y must also be a cyclic contained point and therefore corresponds to a newly contained point of f_1 in the same way. However, the corresponding node of y is not necessarly x again! It is also important to note that the definition of CCP_{f_1,f_2} is not symmetric, i.e. the order of f_1, f_2 matters.

These two definitions of *newly contained points* and *cyclic contained points* induce the condition under which a pushout of graphs with containment edges exists. Before stating this condition, a complex example for a span of graphs shown in Figure 4.3 is used to show how cyclic contained points are computed. We give an algorithm for that purpose, because the definition of cyclic contained points is recursive and through that it is not obvious how to compute them.

Computing cyclic contained points

1. Compute NCP_{f_1} and NCP_{f_2} by checking whether a node has a container in the image of the morphism, but not in the origin.
2. Initialize the sets of cyclic contained points: $CCP_{f_1,f_2} = NCP_{f_1}$ and $CCP_{f_2,f_1} = NCP_{f_2}$.
3. Enumerate all elements $x \in CCP_{f_1,f_2}$. Try to find a node $y \in CCP_{f_2,f_1}$ with $f_2(x)$ $contains_2$ $f_2(y)$. If no such y exists, remove x from CCP_{f_1,f_2}.
4. Do the same for CCP_{f_2,f_1}.
5. Repeat step 3 and 4 until CCP_{f_1,f_2} and CCP_{f_2,f_1} do not change.

Example in Figure 4.3

- $NCP_{f_1} := \{a, c, f, g\}$ and $NCP_{f_2} := \{b, d, h\}$.
- $CCP_{f_1,f_2} := NCP_{f_1} = \{a, c, f, g\}$ and $CCP_{f_2,f_1} := NCP_{f_2} = \{b, d, h\}$.
- Enumerating CCP_{f_1,f_2}:
    1. Node a: $f_2(a)$ $contains_2$ $f_2(b)$. Ok.

2. Node c: No matching contained node in CCP_{f_2,f_1}. Removing node c.
3. Node f: $f_2(f)$ $contains_2$ $f_2(h)$. Ok.
4. Node g: $f_2(g)$ $contains_2$ $f_2(d)$. Ok.

$\Rightarrow$ $CCP_{f_1,f_2} = \{a, f, g\}$ changed.

- Enumerating CCP_{f_2,f_1}:

1. Node b: no matching contained node in CCP_{f_1,f_2}. Removing node b.
2. Node d: $f_1(d)$ $contains_1$ $f_1(f)$. Ok.
3. Node h: $f_1(h)$ $contains_1$ $f_1(g)$. Ok.

$\Rightarrow$ $CCP_{f_2,f_1} = \{d, h\}$ changed.

- Enumerating CCP_{f_1,f_2} again:

1. Node a: no matching contained node in CCP_{f_2,f_1}. Removing node a.
2. Node f: $f_2(f)$ $contains_2$ $f_2(h)$. Ok.
3. Node g: $f_2(g)$ $contains_2$ $f_2(d)$. Ok.

$\Rightarrow$ $CCP_{f_1,f_2} = \{f, g\}$ changed.

- Enumerating CCP_{f_2,f_1} again... no modification.
- Enumerating CCP_{f_1,f_2} again... no modification.
- Done. $CCP_{f_1,f_2} = \{f, g\}$ and $CCP_{f_2,f_1} = \{d, h\}$.

Definition 4.4.5 (Containment condition)**.** Given a span of graphs with containment edges ($G_1 \xleftarrow{f_1} G_0 \xrightarrow{f_2} G_2$), where f_1 and f_2 are valid homomorphisms for graphs with containment edges, the containment condition is stated as:

- $NCP_{f_1} \cap NCP_{f_2} = \emptyset$ and
- $CCP_{f_1,f_2} = CCP_{f_2,f_1} = \emptyset$

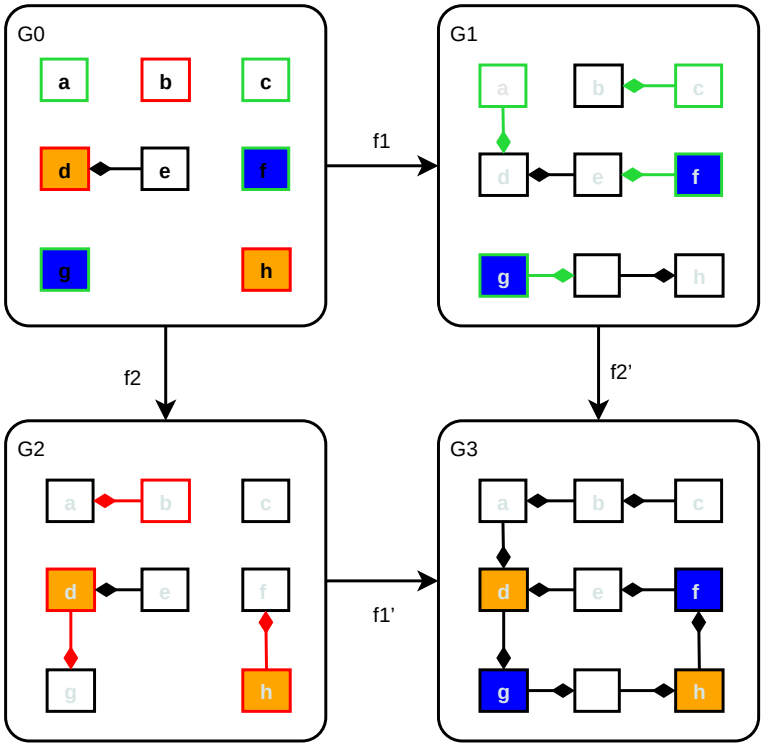


Figure 4.3: Example for a containment cycle. The newly contained points are $NCP_{f_1} = \{a, c, f, g\}$ (green border) and $NCP_{f_2} = \{b, d, h\}$ (red border) in the graph G_0. The cyclic contained points are $CCP_{f_1,f_2} = \{f, g\}$ (blue) and $CCP_{f_2,f_1} = \{d, h\}$ (orange) in the graph G_0.

The constraint, that the newly contained points of f_1 and f_2 must be disjoint, ensures that a node in the resulting graph has at most one container. The second constraint ensures that a pair of newly contained points, one from f_1 and one from f_2 do not form a containment cycle in the resulting graph, i.e. that the sets of cyclic contained points of f_1, f_2 are empty. The proof for the following theorem can be found in appendix A.1.

Theorem 4.4.1 (Pushouts of graphs with containment edges)**.** *Given a span of graphs with containment edges* $(G_1 \overset{f_1}{\leftarrow} G_0 \overset{f_2}{\rightarrow} G_2)$*, where* f_1 *and* f_2 *are valid homomorphisms for graphs with containment edges, the pushout result in the category of graphs* $(G_1 \overset{f_2'}{\rightarrow} G_3 \overset{f_1'}{\leftarrow} G_2)$ *forms also a valid pushout in the category of graphs with containment edges, if and only if the containment condition holds for the span* $(G_1 \overset{f_1}{\leftarrow} G_0 \overset{f_2}{\rightarrow} G_2)$.

4.5 Graph based Model Transformation notions

In section 4.1 an informal mapping of notions from EMF to formal graphs has been given. Now, a formal definition of object structures is given as a special kind of typed attributed graphs. Typing graphs, as defined in chapter 2 are used for defining EMF models formally. The additional features of EMF must be included in this definition. As shown in the last sections it is possible to formalize these concepts with the result of a restricted version of the transformation semantics as defined earlier for plain graphs.

Defining EMF object structures as special kinds of graphs and homomorphisms between these object structures as special graph homomorphisms leads to a possible definition of EMF transformations as a special graph transformation using the double pushout approach:

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & (PO) & {\scriptstyle c}\downarrow & (PO) & \downarrow{\scriptstyle n} \\ M & \xleftarrow{g} & C & \xrightarrow{h} & N \end{array} \tag{4.2}$$

The following definitions are given using the graph based notions introduced so far. A complete algebraic definition is omitted, since it would involve the integration of all these notions. The formalization used in the following are based on:

- plain graphs with directed edges,
- attribution of nodes using Σ-algebra,
- multiplicities of edges using an lower/upper bounds,
- node inheritance defined through a binary relation,
- containments as a special kind of edges.

Definition 4.5.1 (Model)**.** A model M is a typing graph with primitive typed attributes, edge multiplicities, inheritance and a special subset of edges, that must be containment edges in all instantiations of this typing graph. The nodes in a model are called **classes**, the edges are called **associations**.

Definition 4.5.2 (Object structure)**.** An object structure S is an attributed typed graph with containment edges as defined in section 4.4 and a typing graph homomorphism $type_S : S \to M$, where M is a model. The containment edges of S are defined through the model. Homomorphisms of object structures are homomorphisms of attributed typed graphs with containment edges. The nodes in an object structure are called **objects**, the edges are called **links**.

Transformations of these object structures can be defined using the double pushout approach as stated earlier for plain graphs. Given a transformation rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ and a match $m : L \to M$, where L, K, R and M are object structures, a context graph C together with two homomorphisms c, g can be constructed if the gluing condition holds. This is due to the fact that object structures are defined as graphs with special properties and homomorphisms between object structures as special graph homomorphisms. The attribution does not interfere with the construction. The question is, whether the constructed graph C is a valid graph with containment edges, i.e. the order properties of the *contains* relation is satisfied. The homomorphisms c and g must be monotonic respectively.

The same property, that the constructed graphs and homomorphisms respect the containment relation, must be shown for N, n and h. The context graph C is constructed by deleting objects and edges of M. In this step, containment constraints cannot be violated, because containment cycles or multiple containers for one object can only occur if containment edges are *added* to a valid object structure. This can only happen in the second step, when the transformation result N is computed. The constraints, that must be fulfilled for a valid N are stated in the containment condition defined in section 4.4.

For a valid application of an EMF transformation rule, the gluing condition for graphs must be satisfied w.r.t. to a given match. If this is the case, the context structure C can be computed by deleting objects, as defined in the rule. When C is computed, the containment condition must be verified to construct the transformation result N through a pushout of graphs with containment edges.

The definition of the containment condition shows in which cases, an EMF transformation cannot be computed as a usual graph transformation. It is not only the transformation rule itself, that can be considered as invalid. It is rather the combination of the rule and a given match. So it is always the question of *where* a transformation rule is applied. Nevertheless, it has been shown that containment problems (cycles and more than one container for a single object) only can occur, if a transformation rule adds containment edges between already existing objects or adds

new containers, which was the motivation for introducing the set of newly contained points NCP_h of a homomorphism h.

The pragmatic way of EMF, to handle these problem situations is to check the order properties induced by the containment edges every time a reference is being set. For instance, if the container of an object is being set and the object belonged to another container before, the old containment edge is simply deleted. This way, the EMF implementation always ensures at run-time, that all containment constraints are satisfied. So it is not possible to simulate a transformation step and to check then whether the result is invalid, e.g. has a cycle. Due to that, the containment condition is not just a theoretical constraint, but can be used in an actual implementation to check whether a transformation rule can be applied or not. It is this unpredictable behavior of EMF that must be avoided, especially for formal semantics. The gluing condition together with the containment condition are the basis for semantical analysis of EMF model transformation, e.g. termination and confluency (critical pair analysis).

4.6 Consistency of EMF Transformations

The order properties of the containment edges as defined above are the ones that are checked by EMF at run-time. Though, these constraints are not sufficient for a 'proper' object structure. It also must be avoided that an object has no container or that multiplicity constraints are violated.

4.6.1 Containment Consistency

For serializing an object structure, all referenced objects must be mapped to nodes of an XML document. As stated earlier, this tree-like structure is defined through the containment edges in EMF models. What might occur, is that an object is referenced by another object of a certain structure and therefore belongs to this structure. If this object is not contained in another object of the structure, a problem occurs, since the object is referenced but not contained by the structure. Besides a single root container, all referenced objects of an object structure should have a proper container and finally are transitively contained in that root container. If an object is neither a root container nor contained in another object, the structure is considered to be inconsistent. In fact, EMF forbids these situations and refuses to save such model instantiations. This gives rise to the notion of *containment consistency* of

EMF object structures and transformations.

A proper defined EMF model always includes a single root container class. A root container class has no incoming contained-edges. Instantiations of such a class are supposed to transitively contain all other objects that are referenced. When saving or loading model instantiations to/from XML, this root container class is used as the most top node in the XML-tree.

In the following, an object structure is defined as containment consistent if every object that is referenced is correctly contained in this tree-like structure, induced by the containment edges.

Definition 4.6.1 (Containment consistent object structure)**.** An object structure S, typed by a model M is called *containment consistent* if all objects in S are either transitively contained in a root container object or self are root containers. The set of root containers in S are those objects, that are typed by a class in M, that has no incoming containment-edges from a different class.

Definition 4.6.2 (Containment consistent transformation)**.** A direct transformation of object structures $S_1 \Rightarrow S_2$ is called *containment consistent transformation* if the containment consistency of S_1 implies the containment consistency of S_2.

In a containment consistent transformation $S_1 \Rightarrow S_2$, the object structure S_2 has to be containment consistent if S_1 already was containment consistent. In this case, the notation "$S_1 \Rightarrow S_2$" for a transformation can be compared to the one of an implication in logic, in the way that the transformation conserves a certain property, so that it becomes an *invariant* of the transformation.

It is a goal to transfer this property to the level of transformation rules. A transformation rule will be called containment consistent if it ensures for any match a containment consistent transformation. Therefore, it is always necessary to make the assumption that a structure was already consistent before the transformation. Otherwise (if S_1 is not consistent), it cannot be ensured that the transformation result S_2 is consistent (*ex falsum quod libet*).

The definition of containment consistent transformation also implies that if each direct transformation in a sequence $S_1 \Rightarrow S_2 \Rightarrow ... \Rightarrow S_n$ is consistent and the starting structure S_1 is too, the transformation result S_n and of course all steps S_i in the transformation are also containment consistent.

Definition 4.6.3 (RHS-containment consistent rule)**.** A transformation rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ is called *RHS-containment consistent*, if for all objects o in R at least one of the following assertions holds:

- o is a root container,
- there is another object o' in R with o' *contains* o, or
- $r^{-1}(o)$ exists and $l(r^{-1}(o))$ is not contained in another object in L.

With this definition it is ensured that an object in the RHS of a rule does not violate the containment consistency of the transformation result. This is done by excluding two cases that produce a problem: **a)** an object is being created by the rule and not contained in a container, **b)** the object was contained before, but is not contained in another object after rule application.

On the other hand, objects in the LHS of a rule also should fulfill certain constraints so that containment consistency of the result can be guaranteed. A problem might occur if a rule deletes a container object without checking for possible children. To define LHS-containment consistency it would be necessary to make rather vague assertions about the LHS and especially also about the NACs. Since it is not obvious, how this can be done in a formal way, it is avoided here. Instead, we only use the RHS-containment consistency already defined and restrict the rules in the way that they should not delete any nodes. If a rule deletes nodes, it must be ensured that it does not contain any children. This would be involved in the definition of LHS-consistency and is due to that avoided now.

Figures 4.4-4.6 show different cases of containment consistency checks, implemented in the editor. Objects that violate consistency constraints are highlighted red. For instance, Figure 4.6 shows a rule where an object `:Place` is moved to another container and its original container is being deleted. The editor highlights a problem here, because the original container might still contain other objects - e.g. another `:Place`.

The proof for the following theorem can also be found in the appendix.

Theorem 4.6.1 (Containment consistency of transformation rules)**.** *A transformation rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ produces a containment consistent transformation as defined in 4.6.2, if p does not delete any objects and is further RHS-containment consistent.*

4.6.2 Multiplicity Consistency

Multiplicity consistency is a property of a transformation rule, ensuring that it can only be applied if it does not violate any of the multiplicities defined in the EMF model.

Definition 4.6.4 (Multiplicity consistent object structure)**.** An object structure S typed by an EMF model M is called *multiplicity consistent object structure*, if all multiplicity constraints defined in M, which are given through lower and upper bounds of association ends, are fulfilled in S.

Definition 4.6.5 (Multiplicity consistent transformation)**.** A direct transformation $S_1 \Rightarrow S_2$ of object structures is called *multiplicity consistent transformation* if the multiplicity consistency of S_1 implies the multiplicity consistency of S_2.

If there are finite bounds for an association in a model, a transformation rule should explicitly check through NACs that multiplicity constraints are not violated. The implementation of the transformation editor supports static checks for this multiplicity problems that might occur. Therefore, the rules are analyzed by checking all edges with finite multiplicities that are created or deleted during rule application. If an edge is found that is restricted in that way, all NACs of the rule are analyzed whether they forbid the application of the rule if the upper/lower bounds of the edge are exceeded during rule application. Possible multiplicity problems are analyzed by the editor during the editing process and are highlighted.

Proposition 4.6.1. *A direct transformation induced by a given rule together with a match is multiplicity consistent if the following assertions hold:*

- *For every link, that is created by the rule and that has a finite upper bound there is a NAC, that explicitly checks if the upper bound is reached already.*
- *For every link, that is deleted by the rule and has a finite lower bound $\neq$ 0 there is a NAC, that explicitly checks if the lower bound is reached already.*

It is obvious, that this definition ensures multiplicity consistency as defined before. Another approach is to integrate these checks directly into the interpreter / compiler, so that this is automatically handled in the way that the rule is only applied if no constraints are violated. This kind of implementation is used in the AGG[3] engine.

[3] A framework for formal graph transformations, see [AGG].

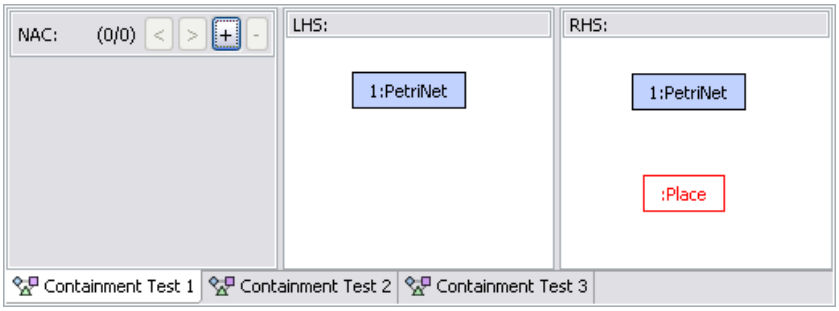


Figure 4.4: Highlighting of containment consistency constraints in the editor: An instance of `Place` is being created without adding it to a container (an instance of `PetriNet` in this case)

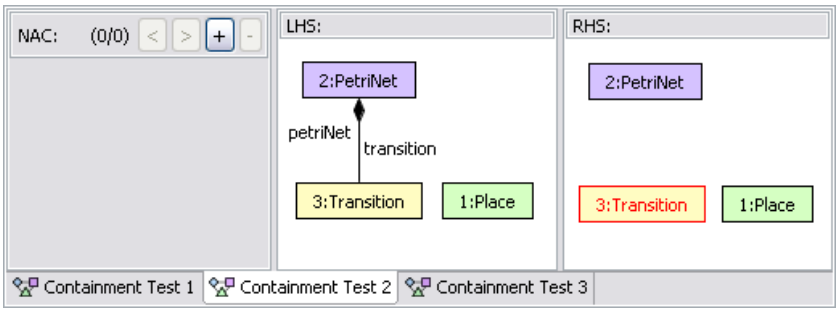


Figure 4.5: Highlighting of containment consistency constraints in the editor: The containment edge `:PetriNet` → `:Transition` is being deleted, so that the transition instance would not be contained in the Petrinet after rule application.

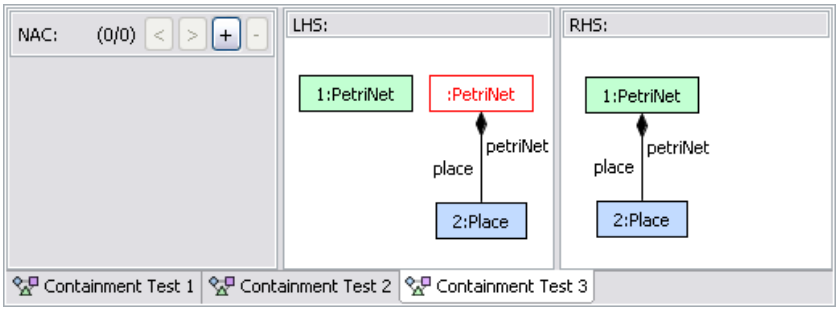


Figure 4.6: Highlighting of containment consistency constraints in the editor: A container `:PetriNet` is deleted without checking whether it still contains any children e.g. more places or transitions.

Chapter 5

Implementation

The implementation of the EMF model transformation project consists of a graphical editor, which can be used to define transformation rules, an interpreter, which computes transformations by translating the rules to AGG, and a compiler, that generates Java code. The interpreter and compiler are discussed introduced shortly in chapter 5.3. The focus here is the graphical editor in combination with the the underlying transformation model, building the foundation for all related projects.

5.1 Transformation Model

The basis for all components of the model transformation framework is a common EMF model that describes transformations of EMF model instances. As mentioned earlier, the transformation model directly uses notions of the Ecore metamodel, which is only possible because Ecore itself is an EMF model again. Since the transformation model is used to define relations *between* EMF models and uses notions of the metamodel it can be seen as an extension of the metamodel or at least somewhere between the model and metamodel level (see Figure 5.1).

The kernel of the transformation model is shown in Figure 5.2. The complete model further includes layout information (e.g. positions of nodes) used by the graphical editor. The transformation model makes extensive use of the Ecore metamodel. In Figure 5.2 this is visualized through the different coloring of classes: all classes highlighted blue are parts of Ecore and are only referenced by the rest of the actual transformation model.

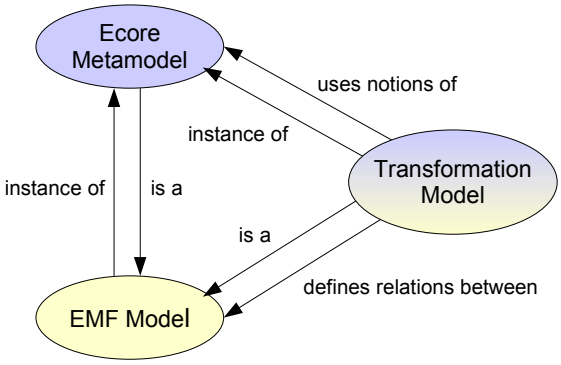


Figure 5.1: Ecore metamodel, EMF models and the Transformation model

5.1.1 The Kernel

The class `Transformation` is the root container in the transformation model. All other parts, that do not explicitly belong to another model (so called *imports* of the transformation) are transitively contained in this class. The imports of a transformation appear in the model as instances of the metaclass `EPackage` (see Figure 5.2). These packages are the actual models on which the transformation should be defined. They are loaded dynamically at run-time using the EMF concept of package registries.

Already introduced earlier, the abstract class `ObjectStructure` represents a set of possibly linked and/or attributed objects. This concept can be directly compared to an attributed typed graph. The class `ObjectStructure` has two references to the Ecore class `EObject`. All root containers in the set of objects are referenced by the link `objects`. Through that, object structures (or subclasses) become a root container for all objects in the structure. Objects which are somehow transitively contained in another object can not be directly accessed using this containment reference only. For that reason, there is a second, derived reference, called `allObjects`, which can be used to easily access all objects in a structure.

Since `ObjectStructure` is an abstract class and cannot be used directly through that, there are specializations of it, which also constitute the most important parts of a transformation rule:

- `LHS`: left-hand side of a transformation rule (one per rule),
- `RHS`: right-hand side of a transformation rule (one per rule),

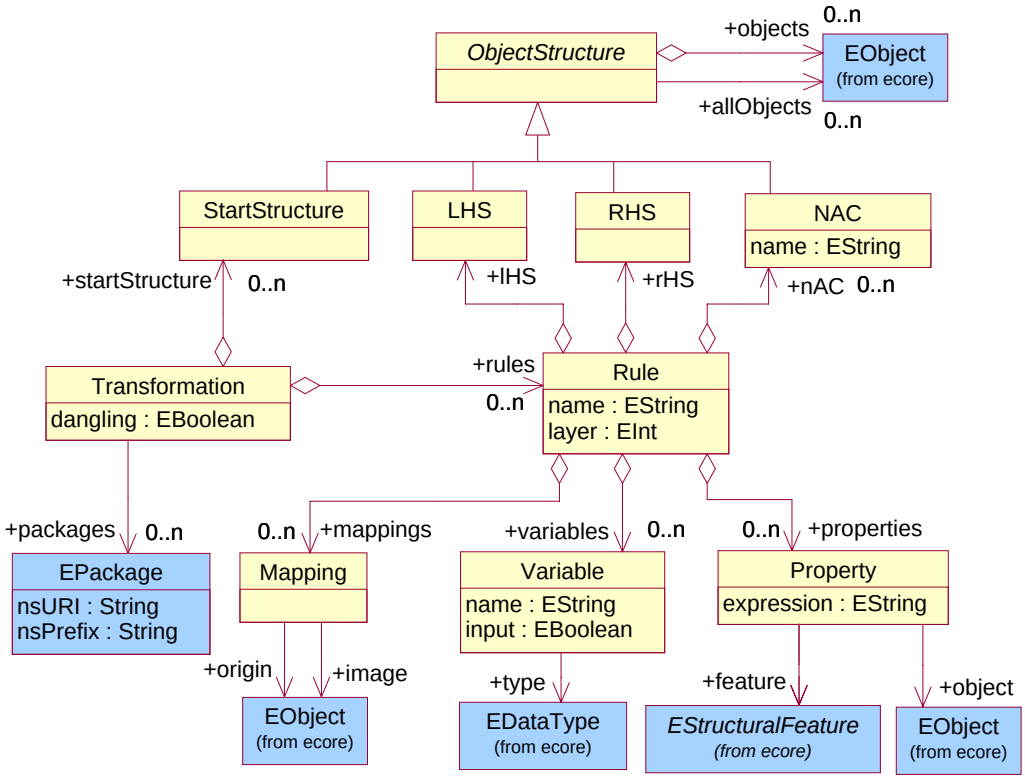


Figure 5.2: Transformation model: Kernel

- `NAC`: negative application conditions (n per rule),
- `StartStructure`: start structures of an 'EMF grammar' (n per transformation).

The class `NAC` contains an extra attribute, called `name`, which can be used to easily distinguish negative application conditions from each other. Start structures are not interesting for model transformations, but are needed for defining (visual) languages in terms of graph grammars. In the case of model transformations there are not used, because model transformations usually are applied to a given model instance, taking the place of a start structure in this case.

A rule further contains a number of 1:1 mappings between the LHS and the RHS on one side and the LHS and the NACs on the other side. A mapping is not a container for the objects, that are mapped, because they are already (transitively) contained in an object structure (LHS, RHS or NAC). An integer value can be assigned to rules

to organize rules linearly into layers.

Variables are used for calculations on the attributes of objects. A variable can be marked as an *input* variable, which means that the user has to assign values to them before the transformation rule can be applied to an instance. They are mainly used if a transformation needs more information than are given in the source model. Variables are assigned to a rule and can be used to define primitive valued expressions to object's attributes, which are evaluated at run-time. An instance of the class `Property` contains such an expression using a simple string and further must be assigned to some object in one of the object structures. To determine which attribute of an object is meant the class `Property` also contains a link to an instance of the Ecore class `EStructuralFeature`, which is used to represent a class' attribute of an imported package.

5.1.2 Layout Information

Transformation rules can be edited using a graphical editor. For this purpose layout information must be kept in order to visualize the EMT models in a diagram form. In general there are two ways to implement such *views* on a model: include the layout information directly in the model or keep all information in a separate layout model. The former case is easier to implement but can be considered as bad software design. The latter approach decouples model information and layout information, which especially is an advantage if there are multiple editor implementations, requiring different kind of layout information.

In the presented implementation, the layout information are included in the transformation model. However, this extra data is relatively decoupled in the transformation model. As shown in Figure 5.4 the visual information is organized in diagrams. These diagrams are contained in the root container of the transformation model (`Transformation`). Further, the actual layout information (positions of objects / classes / bendpoints) is modeled as a hashmap (`EObjectToPointMapEntry`), where the keys are the abstract model entities and the values are the positions of these entities. This way, the layout information keeps references to the abstract model entities. The abstract part of the transformation model doesn't know anything about the layout information.

This is a good compromise between the two approaches of how to keep layout information as described before. Even though a separation of abstract and concrete syntax (with and without layout information) is a good thing, it comes with a couple of major implementation problems. The abstract model and the layout model must

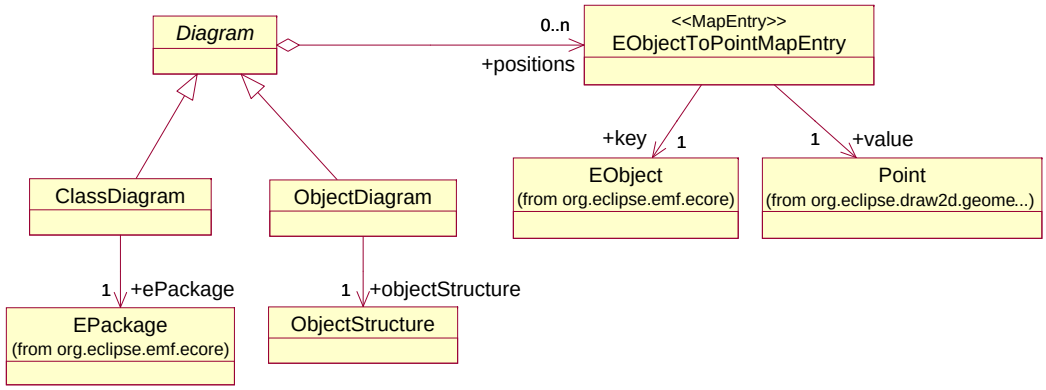


Figure 5.3: Transformationmodel: Diagrams

be kept synchronized. Further, implementing an editor with GEF means, to follow a Model-View-Controller pattern. In this MVC architecture the model must be in concrete syntax, so that the layout model must be used for this purpose. Moreover the *View* in MVC must be kept synchronized to this layout model. So in essence, it is necessary to keep the abstract and the concrete model on one hand and the concrete model and the view on the other hand synchronized. The first case alone is already not trivial to implement, because of the *behaviour* of EMF. Multiplicity and containment constraints are checked by EMF at run-time. If anything violates these constraints, EMF tries to solve the problem, which e.g. can lead to the deletion of edges that were not even directly a part of the problem. This behavior must be either reimplemented or it must always be ensured, that **all** modifications, caused not only by the editor, but also by EMF itself must be recognized and translated correctly.

These problems can be avoided using an implementation that consists of a transformation model in pure abstract syntax and a separate model in concrete syntax, including the complete abstract model with additional layout information, similar to the current implementation. The graphical editor is defined on top of the model in concrete syntax and should provide an import / export functionality to the abstract model syntax. The current editor implementation is based on such a combined model which includes layout information and the abstract entities. The last step: a separated abstract transformation model and an import / export functionality in the editor is not implemented yet. This is has been omitted so far since the layout information do not interfere with any other components of the transformation framework

(interpreter, compiler) and the fact that so far only one editor implementation exists up to now (the one described here). However, the necessary modifications are straight forward and involve only modifications in the editor and the transformation model itself.

5.2 Graphical Editor

The graphical editor for EMF model transformations is implemented on top of the transformation model (in concrete syntax). The graphical user interface includes a diagram view including separate diagrams for the LHS, RHS and NACs of a transformation rule and a palette, including tools for adding objects to the diagrams and for connecting or mapping the objects.

Transformation rules and their components are also shown in a tree-like outline view, that is connected to the diagram view and also to the property view of Eclipse, so that object can be added to diagrams using drag'n'drop and properties of objects or rules can be edited in the property view. A user guide for the editor is attached in the appendix of this paper. A more detailed description of the complete framework, including the interpreter and the compiler can be found at the EMT project homepage [EMT].

As mentioned earlier, the editor is implemented using the Graphical Editing Framework (GEF) of the Eclipse platform. GEF includes a big number of features, which are not described in detail here. The most important fact about GEF is, that it implements a special version the Model-View-Control pattern, which can also be found in a number of other middle-ware platforms, e.g. Java Swing[1] or Apache Struts[2]. The MVC pattern distinguishes between a `Model`, so called `Controllers` and `Views`. Views are visualizations of the model. In general, there can be multiple views for one model. The model and its views are decoupled, as much as they do not know about each other. For keeping the model and its views synchronized, the controller is used. For each view there must be a controller that updates the model if there are changes in the view and vice versa. To avoid such bidirectional synchronizations issues, GEF makes the restriction, that modifications can only be made in the model, so that the controller only needs to update the view accordingly. The view is in the notions of GEF a set of *figures*. The controller must be implemented using so called *editparts*. For a more detailed information on the MVC pattern in GEF see [KK05].

[1] `http://java.sun.com/products/jfc/index.jsp`
[2] `http://struts.apache.org`

5.3 Interpreter and Compiler

An interpreter and a compiler have been implemented already and is discussed in [BK06]. The interpreter translates EMF object structures into equivalent graphs in the AGG format. The actual transformation step is performed using AGG. Further it is possible to apply tools of the AGG engine for formal analyzes, e.g. critical pairs. Current work on the editor involves the implementation of a debugger tool for the graphical editor, allowing the simulation of transformation rules. For this purpose, the interpreter is being integrated into the editor, which is discussed in more detail in the last chapter.

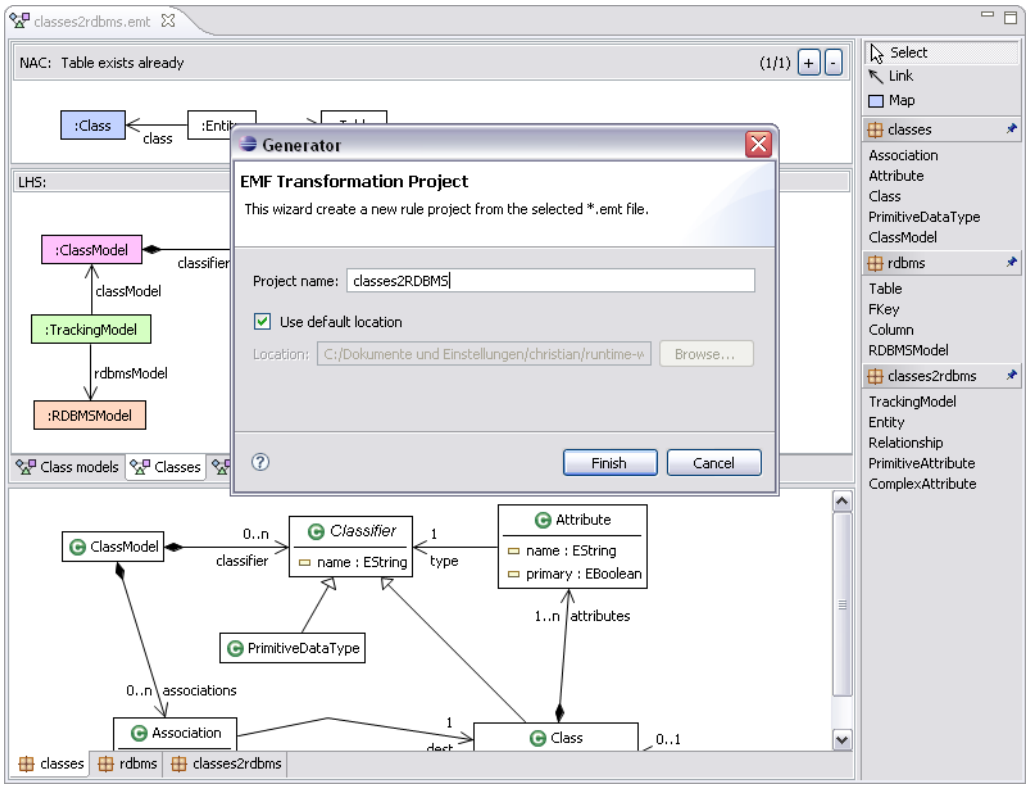


Figure 5.4: Code generation wizard for EMF model transformations.

In the compiler approach, model transformation rules are translated to Java code. A single rule can be applied or the whole transformation can be computed. The generated code does not involve the AGG engine and can easily be integrated into existing projects. The compiler is integrated into the editor in the way that it can be invoked from an entry within the context menu of the editor.

Chapter 6

Examples

In this chapter, two examples for EMF model transformations are discussed. A possible categorization of model transformations is to distinguish between *endogenous* and *exogenous* transformations. While endogenous transformations are defined on one model only, exogenous transformations define a conversion from one language two another. As an example for endogenous transformations, refactoring rules for EMF models are discussed in the following. The second example deals with simplified UML 2 Class diagrams and relational database schema, representing a exogenous model transformation.

6.1 Endogenous Transformations: EMF-Refactoring

The aim of refactoring in general is to modify the structure of a system by keeping the behavior, or in general: some semantics. The Java Editor of the Eclipse platform implements such refactoring rules for easily modifying large Java projects, where a small change can cause the need of change the source code in a lot other files, too. An example for such a refactoring is renaming or moving a class.

In the following, refactoring rules are presented for EMF itself, so that the model on which the rules are defined, is the Ecore (meta-) model. This example is also discussed in [TEB+06b].

6.1.1 Moving a class

The first example is shown in Figure 6.1. A class, which is matched through its name is being moved from one package to another. For that the containment edge

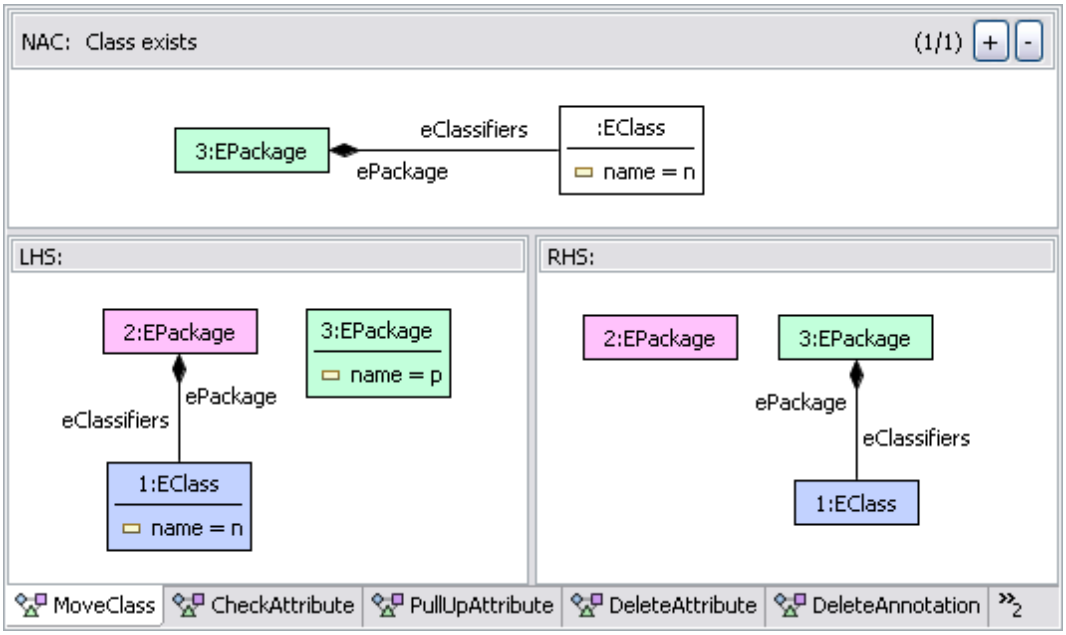


Figure 6.1: EMF-Refactoring: Rule `MoveClass`

`eClassifiers` is moved from one package to the other. The match finding relies on two input parameters: `n`: the name of the class to be moved and `p`: the name of the package, where class `c` is moved to. The negative application condition `'Class exists'` is used to ensure that there does not exist already a class with the same name in the target package.

6.1.2 Creating super classes

The refactoring rule `CreateSuperClass` in Figure 6.2 can be used to create a new class in a specified package, which automatically becomes the new super class of a given other class. As an input parameter, the name of the new super class must be given.

Further it can be interesting to connect more classes to the newly created superclass. The rule `ConnectSuperClass` in Figure 6.3 suits for this task. As a restriction, we allow only classes to be connected to the superclass, which have no attributes and no references to other classes.

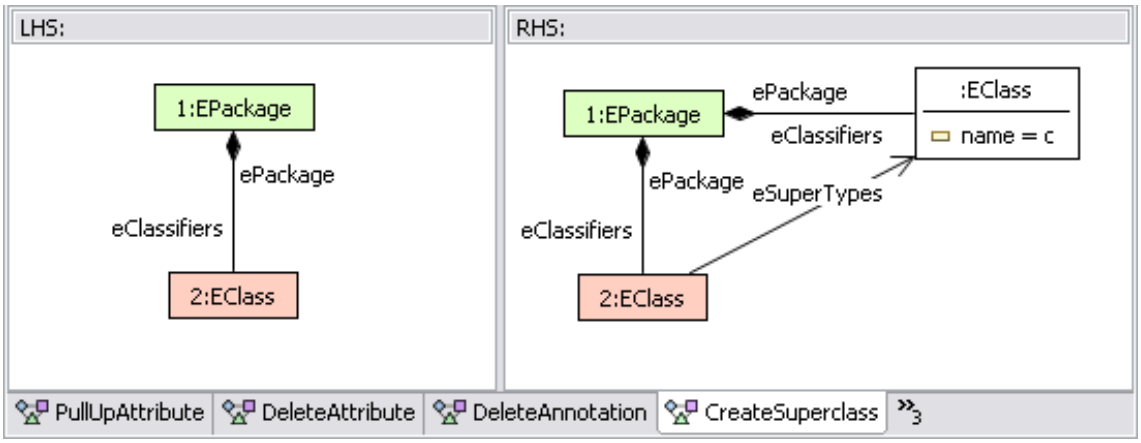


Figure 6.2: EMF-Refactoring: Rule `CreateSuperClass`

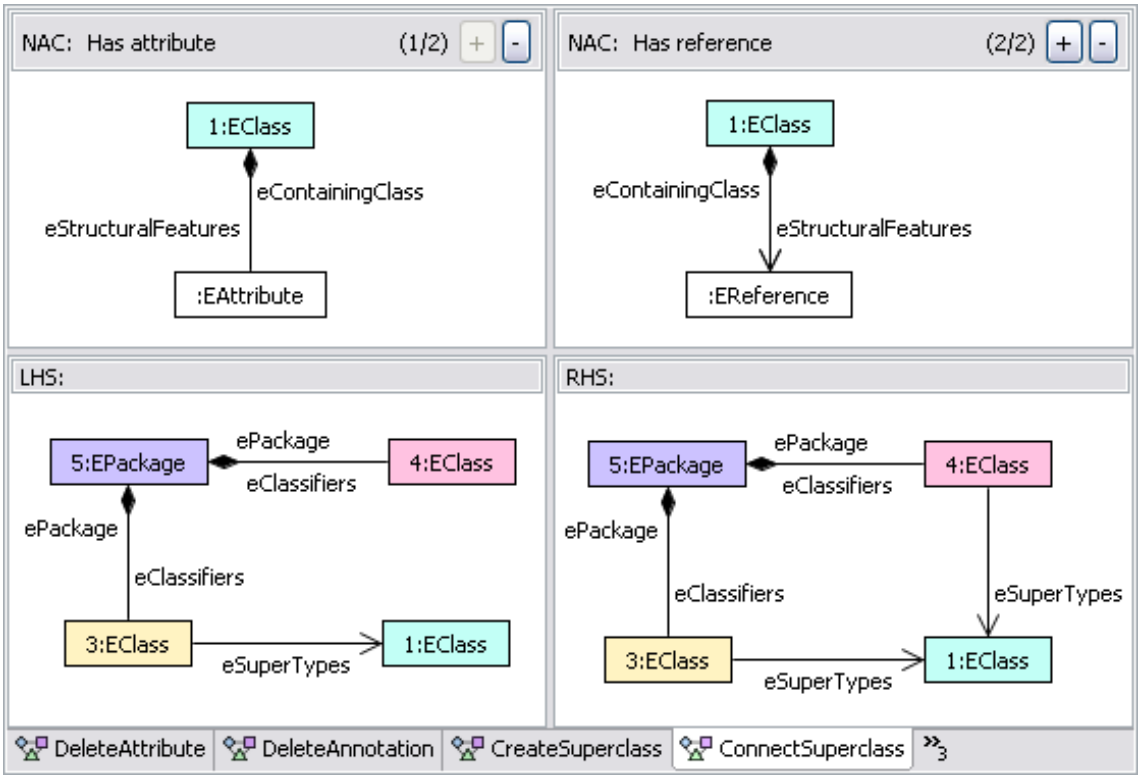


Figure 6.3: EMF-Refactoring: Rule `ConnectSuperClass`

6.1.3 Pulling up an attribute

In most cases a structural simplification of a classes can be achieved if a certain property (e.g. an attribute) is pulled up into a class' superclass. This is only possible if the attribute exists in all other subclasses as well, e.g. in the Ecore model the abstract metaclass `ENamedElement` is used as a superclass of all metaclasses that contain an attribute `name` with the type `EString`.

Realizing this kind of refactoring as an EMF model transformation is only possible with a set of four distinguished transformation rules, which have to be applied in sequence. The parameters for these rules are on one hand the name of the attribute to be pulled up (`a`) and the name of the super class (`c`), that should contain the attribute afterwards. In order to ensure that all subclasses of `c` have an attribute

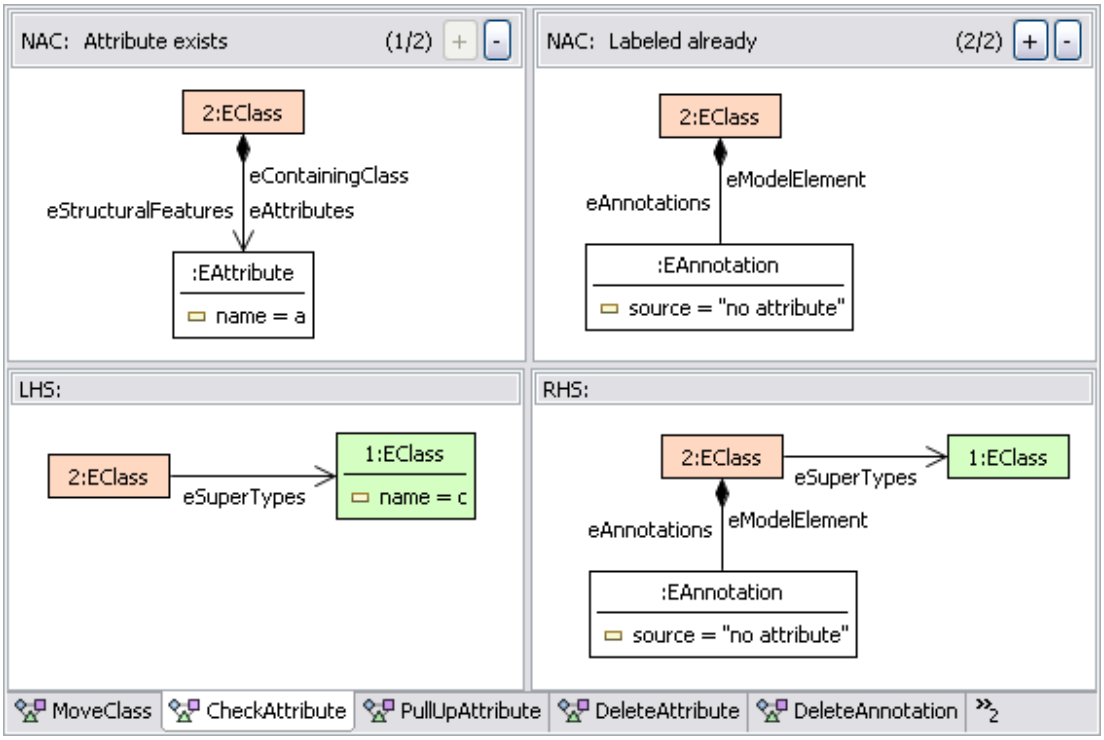


Figure 6.4: EMF-Refactoring: Rule `CheckAttribute`

`a`, the rule (`CheckAttribute` in Figure 6.4) is used. If there exists a class that is a subclass of `c` and has no such attribute, an instance of `EAnnotation` is used to label that class. This rule can only be applied to a given match, since the NAC `'Labeled already'` forbids multiple application.

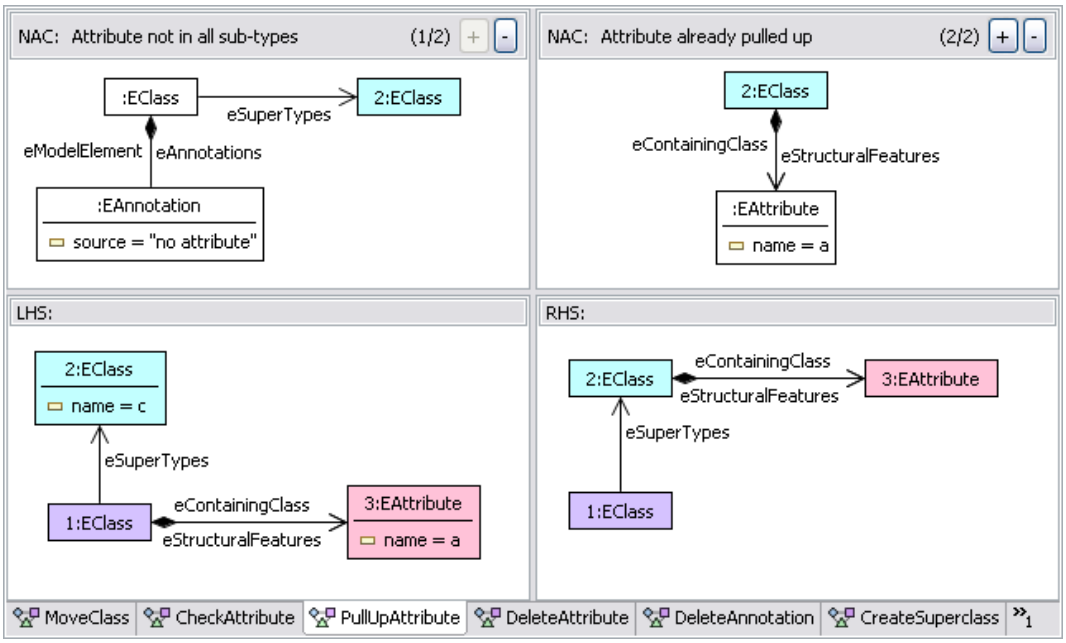


Figure 6.5: EMF-Refactoring: Rule `PullUpAttribute`

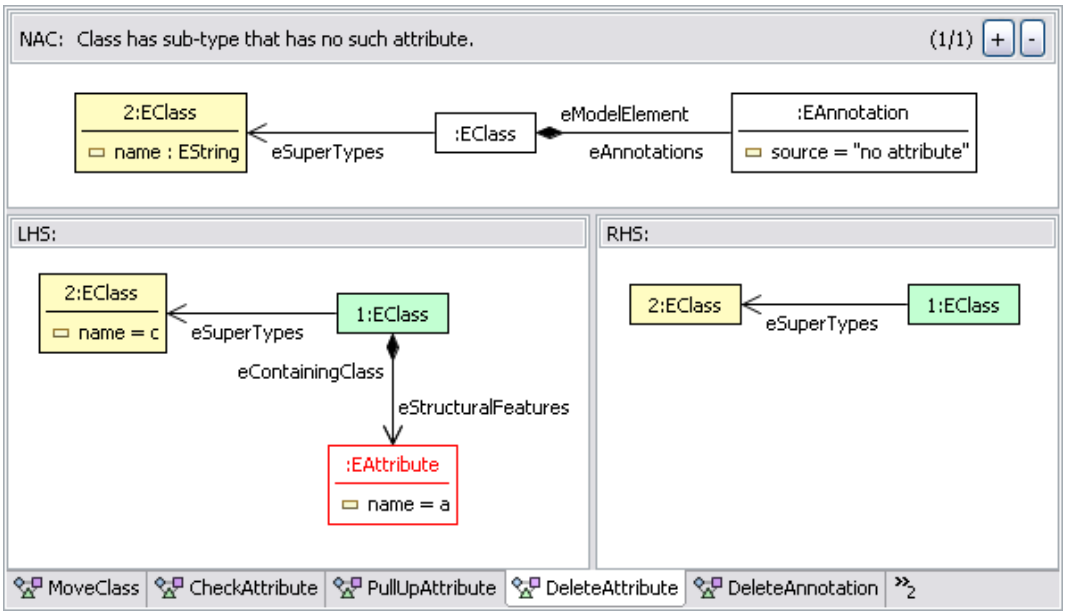


Figure 6.6: EMF-Refactoring: Rule `DeleteAttribute`

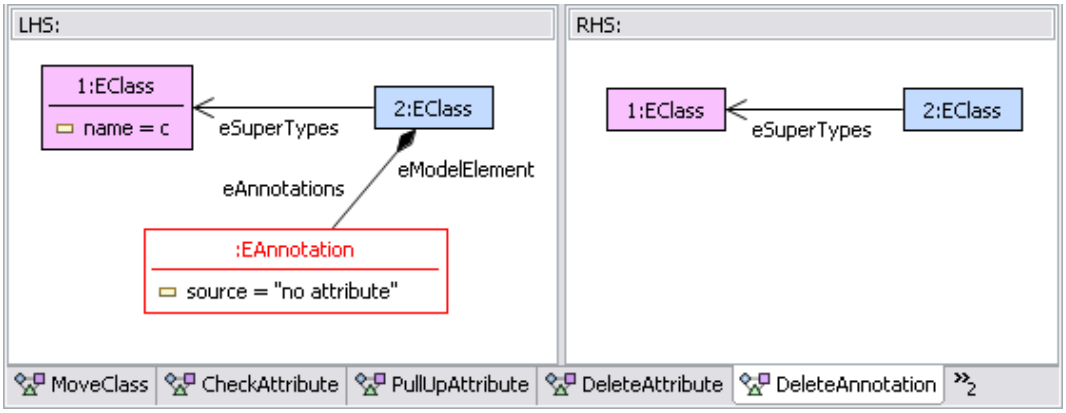


Figure 6.7: EMF-Refactoring: Rule `DeleteAnnotation`

After this first check, the actual pulling up of the attribute can be computed with the rule `PullUpAttribute` shown in Figure 6.5. If there is no subclass being labeled by `CheckAttribute` before and there is no attribute named `a` in the superclass, the attribute can be pulled up. Note that EMF would automatically update derived references like `eAllAttributes` (attributes in this and all superclasses) in the subclass. This is not shown in Figure 6.5, because its automatically done by EMF, triggered by the modification of the reference `eStructuralFeatures`.

After successfully applying the rule `PullUpAttribute` the attributes with the name `a` in all sublasses must be removed. This is done by the rule `DeleteAttribute` shown in Figure 6.6. This rule is applied as long as possible, so that the attribute is deleted in all subclasses.

In Figure 6.6 the attribute is drawn red, indicating that a containment consistency check has failed. This is due to the fact, that it is not ensured that the attribute contains no annotations that would be not contained in any resource after rule application. We assume here that such annotations do not exist so that we don't need to worry about that problem. To make sure that this problem might not occur, an NAC could be added forbidding such annotations or a rule could be added that explicitly deletes them.

As a last step, the annotations "no attribute" in all sublasses are removed. Here the same containment problem occurs, which we ignore again. This rule is applied as long as possible, so that all annotations are really deleted.

6.2 Exogenous Transformations: Class diagrams to RDBMS

Transforming simplified UML class diagrams to relational database schema (RDBMS) is a standard example for exogenous model transformations, especially in QVT-related approaches. A graph transformation based approach for this particular example has already been discussed in [TEG+06]. We refer here to the first, simplified version of [TEG+06], that has been modeled there using the AGG engine for graph transformation. The following models and transformation rules are essentially the same like the one from [TEG+06], only that the EMF approach has some more additional semantics through its containment edges and - for this particular example - more restricted multiplicities.

The (meta-) model used for the class diagrams is shown in Figure 6.8. Classes can be marked as persistent, which are mapped to tables with all its attributes and associations being translated to columns in this table. If another persistent class occurs as the type of an attribute or an association, a foreign key (`Fkey`) to this table is established.

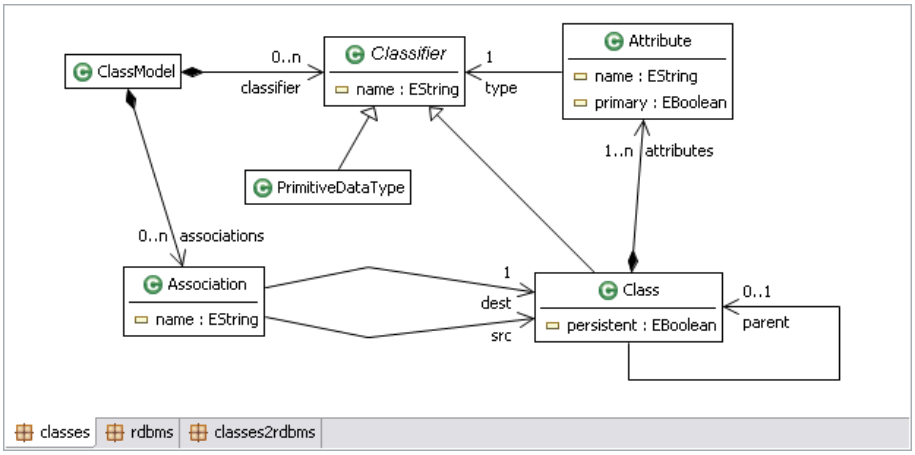


Figure 6.8: Class diagrams to RDBMS: Model for class diagrams

The original version of this example also includes a requirement that class hierarchies are handled in a special way. Only the topmost classes should be mapped to tables. Additional attributes or associations in the subclasses are translated into columns in the table representing the topmost class. To implement this behavior with

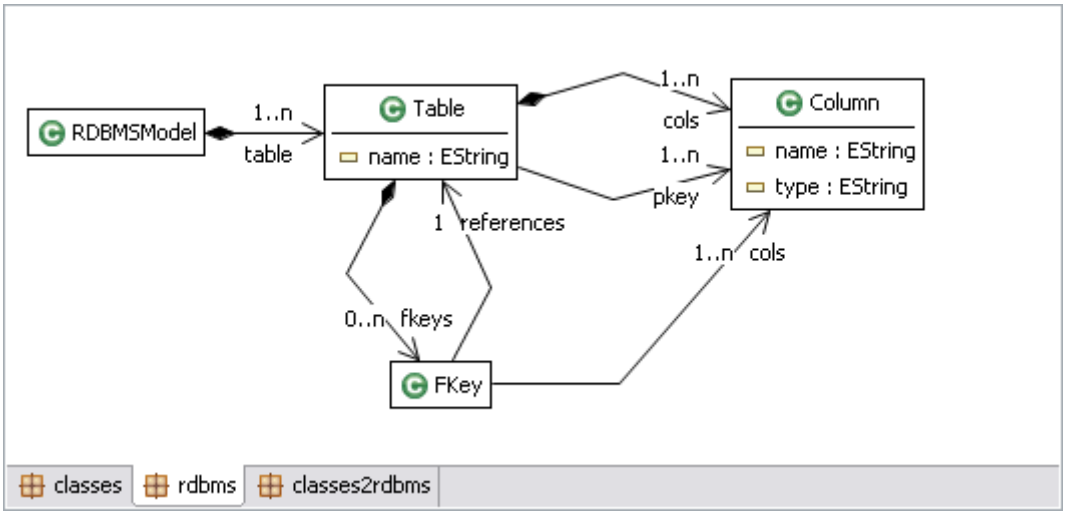


Figure 6.9: Class diagrams to RDBMS: Model for RDBMS

the EMF transformation approach it is necessary to compute the topmost class and to somehow establish a direct connection to all subclasses so that associations and attributes of the subclasses can be recognized. This is omitted here and discussed in detail in [TEG+06].

Non-persistent classes are not mapped to tables. Instead, their attributes and associations should be distributed in the tables which represent the persistent classes that access the non-persistent classes. The (meta-) model for class diagrams shown in 6.8 distinguishes between primitive typed attributes and associations.

Figures 6.8-6.10 show the Ecore models, that are used in this example. While the model for simple class diagrams is the source and the RDBMS model is the target, the so called *tracking model* in Figure 6.10 is the bridge between the source and the target model.

6.2.1 Models and Classes

The first rules are quiet simple. In Figure 6.11 a root container `:RDBMSModel` is created for an existing `:ClassModel`. These objects contain all entities being translated. The rule `Classes` translates a persistent class into a table. Whenever an RDBMS object is being created, a link to its counterpart in the class model is established as defined in the tracking model.

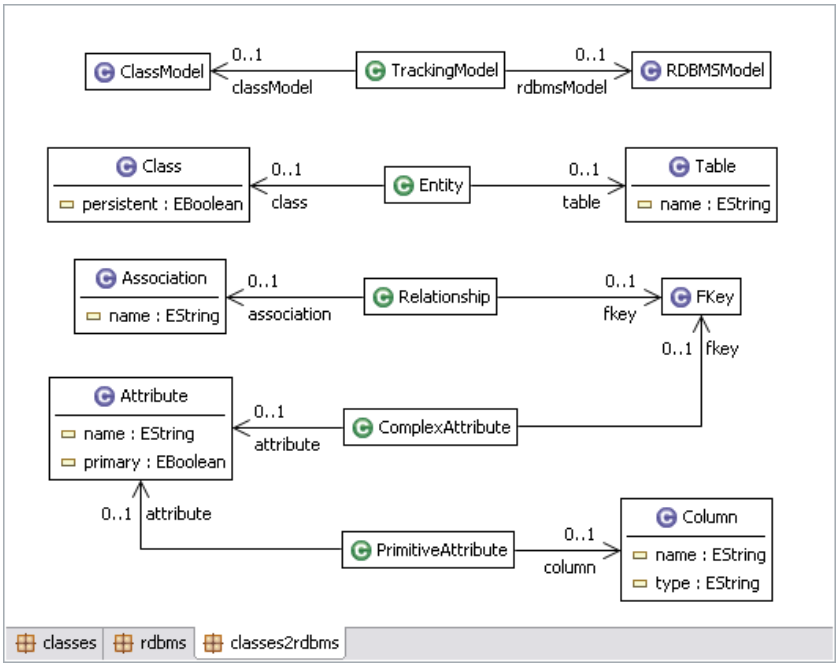


Figure 6.10: Class diagrams to RDBMS: Tracking model

6.2.2 Attributes and Associations

In Figure 6.13 it is shown how primitive typed attributes are translated to columns of a table corresponding to the class that is the owner of the attribute. The column has the same name and type as the attribute. Complex attribute (not primitive typed) are translated by creating a foreign key, as shown in Figure 6.14. A new column is created, where the name is a combination of the class' name and the attribute's name[1].

Rule `Set primary key` translates establishes an extra edge for primary columns. Figure 6.15 is a little misleading, since the edges are overlapping. What this rule does, is that it creates an edge `pkey`, it does not delete the containment edge `cols`. The last rule translates associations into foreign keys. This is essentially the same as rule `Complex attributes`.

An advanced version of this model transformation where also generalisations are considered, can be found in [TEG+06].

[1]In Figure 6.14 the name of the column is used, but this is actually the same as the class' name (see rule `Primitive attributes`)

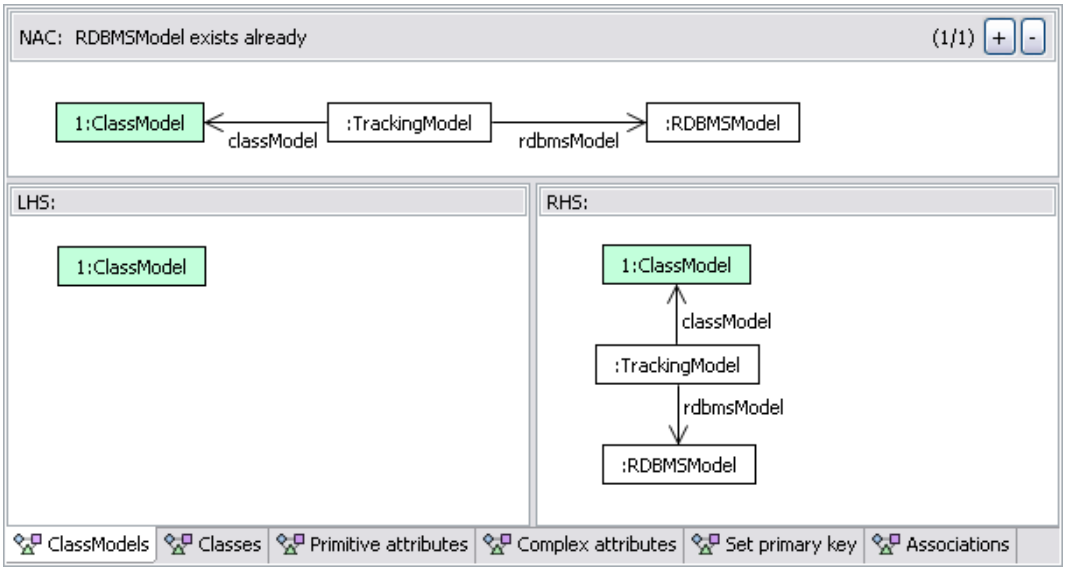


Figure 6.11: Class diagrams to RDBMS: Rule `Class models`

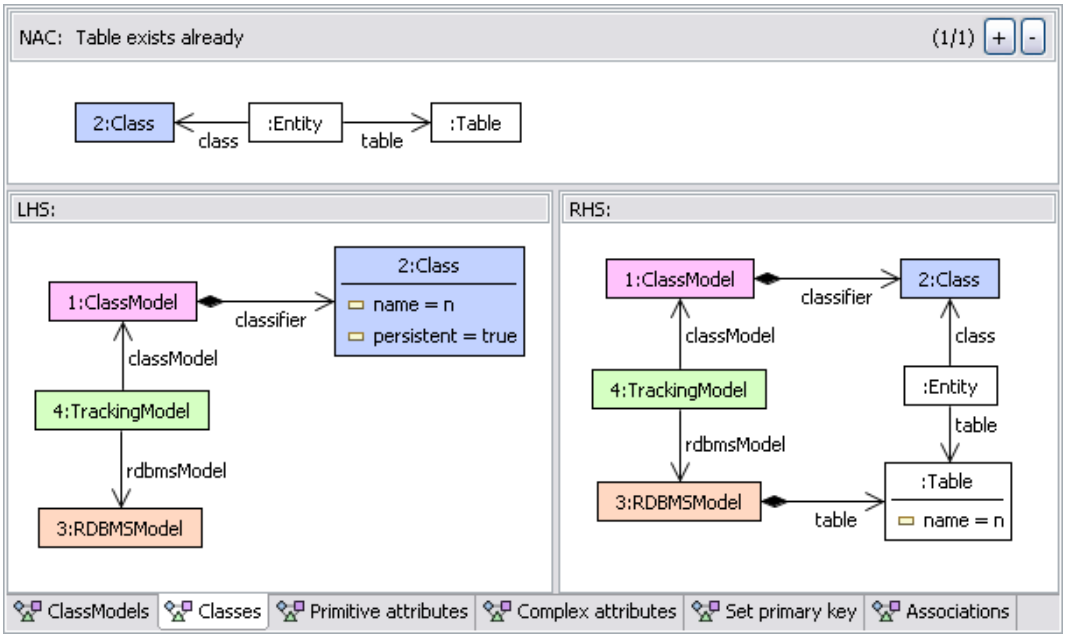


Figure 6.12: Class diagrams to RDBMS: Rule `Classes`

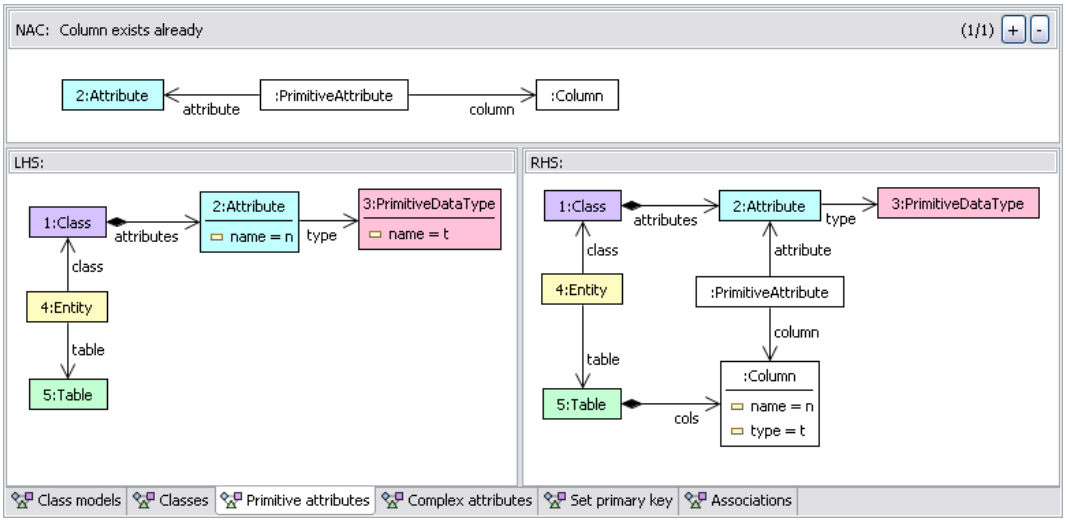


Figure 6.13: Class diagrams to RDBMS: Rule `Primitive attributes`

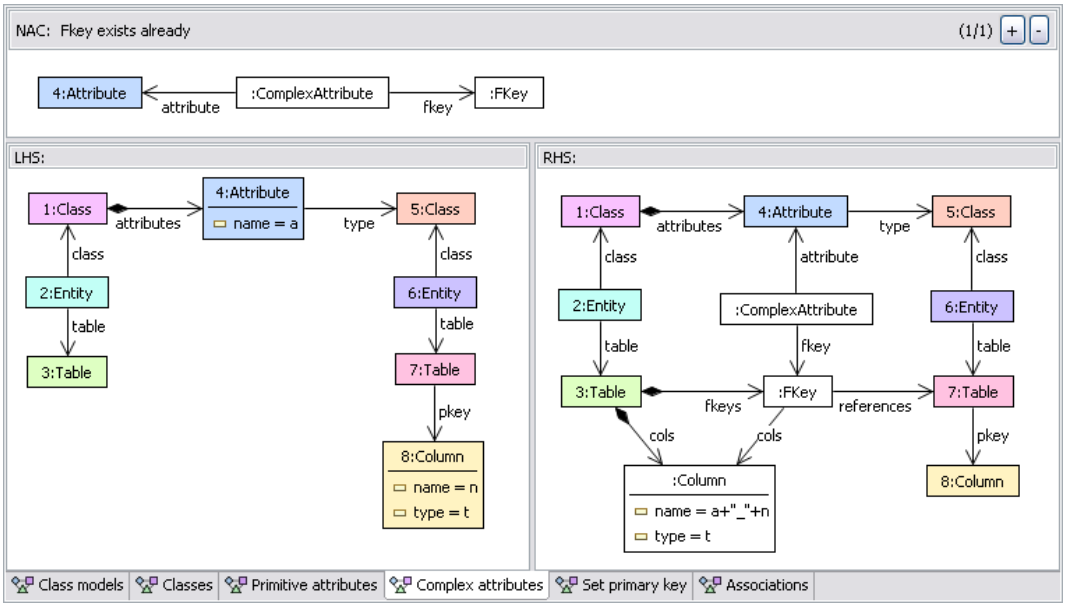


Figure 6.14: Class diagrams to RDBMS: Rule `Complex attributes`

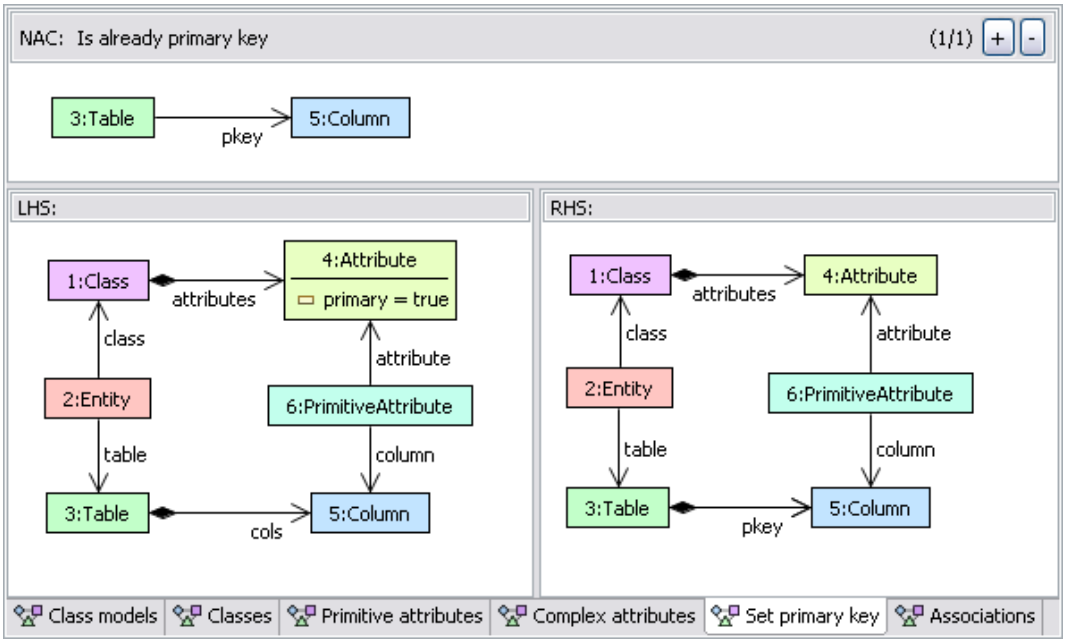


Figure 6.15: Class diagrams to RDBMS: Rule `Set primary key`

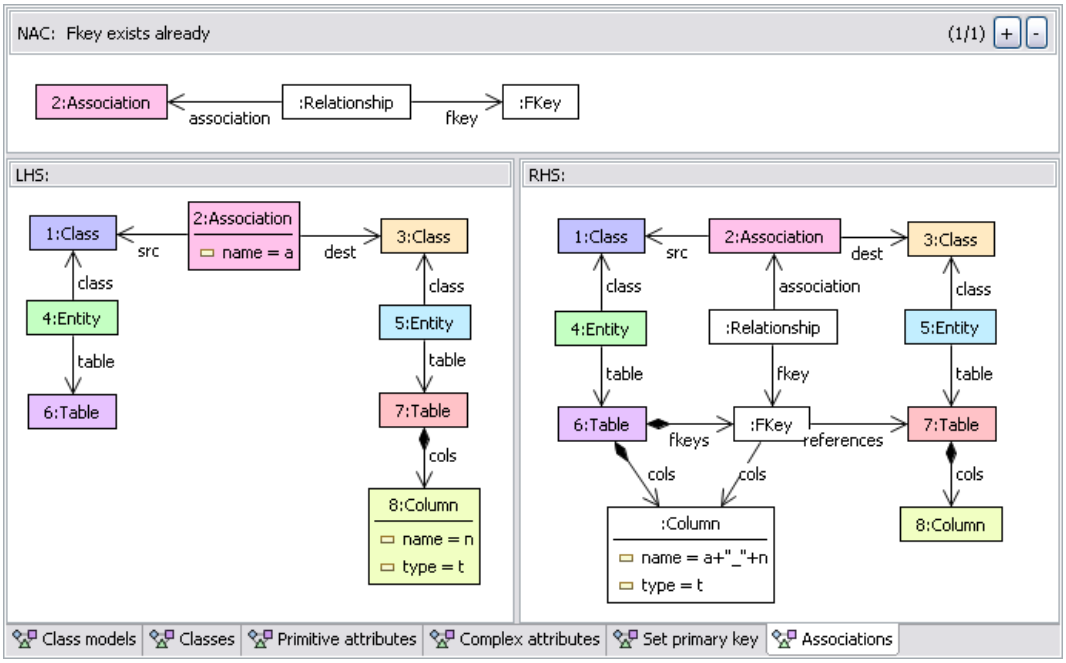


Figure 6.16: Class diagrams to RDBMS: Rule `Associations`

Chapter 7

Conclusion

Future work in the EMF Model Transformation project will involve technical developments as well as theoretical investigations. Further it would be interesting to apply the presented approach to a wider range of examples, especially in more complex scenarios as the ones described here.

7.1 Separation of Layout Information

From an engineering point of view, the layout information in the transformation model should be moved to a separate model, which keeps only references to the actual transformation model.

The decision to include the layout information in the transformation model has been made, because it simplifies the implementation of the graphical editor on one side and does not interfere with the other existing implementations, which involve the interpreter and the compiler up to now. To achieve this separation of concerns, some technical difficulties must be solved, but for all future implementations this reorganization of the underlying components would pay off.

7.2 Visual Debugger

Current work focuses on the integration of the interpreter implementation into the editor, allowing the stepwise application of rules visualized in a diagram. Different scenarios are interesting in this context, mostly known from conventional code debuggers. This includes the possibility of defining breakpoints, e.g. in the form of a graph pattern.

7.3 Method Calls and Code Integration

The EMF model transformation approach is indeed very generic and has a lot of possible applications. As described before it tries to be a bridge between formal graph transformation and technologies like EMF, Java and XML which form the basis for the model transformation language presented here. Even though the approach is very generic, it is restricted to the pattern based creation / deletion of objects / references and primitive valued calculations on the attributes, expressed in Java syntax. What is not considered yet is the possibility of defining methods in EMF models, allowing to include behavioral aspects to the structural models, by implementing them in Java. Although an EMF model itself does not include a description of the behavior of a class, it can be used to define its functionality (name, type, parameters of methods).

These functional aspects of EMF models can and properly should be also considered in the transformation approach, since they allow the explicit usage of behavioral features in EMF models or more precise, in implementations of EMF models. This extension would also allow to bootstrap the EMF model transformation approach, i.e. to define its semantics with itself. It is already possible to define transformations on the transformation model. However, it is necessary to not only be able to access attributes or references of a given object (which must be typed to do that), but also to invoke methods and work with the resulting objects. As an example one can think of a set of transformation rules defining the semantics of the interpreter described earlier. There could be a transformation rule, which states *how a transformation rule is applied* with respect to a given match. Also the editing operations that are available within the GUI can be bootstrapped. Figure 7.1 shows a rule that creates a mapping between two objects, one in the LHS, the other one in the RHS of a rule. The same way it could be possible to define the interpreter's semantics with its own language. However, for defining a rule that captures the case of object creation for instance, it is at first necessary to retrieve the type (the corresponding `EClass`) of an arbitrary `:EObject` and then to instantiate this class using a package factory. Just for retrieving the type of such an object it is necessary to call the reflective EMF method `eClass()` of the class `EObject`.

Investigations on how such method calls could be integrated into the current approach are still in a very early stage. Nevertheless, it seems important to have that kind of feature, allowing a real extension of the current application scenarios.

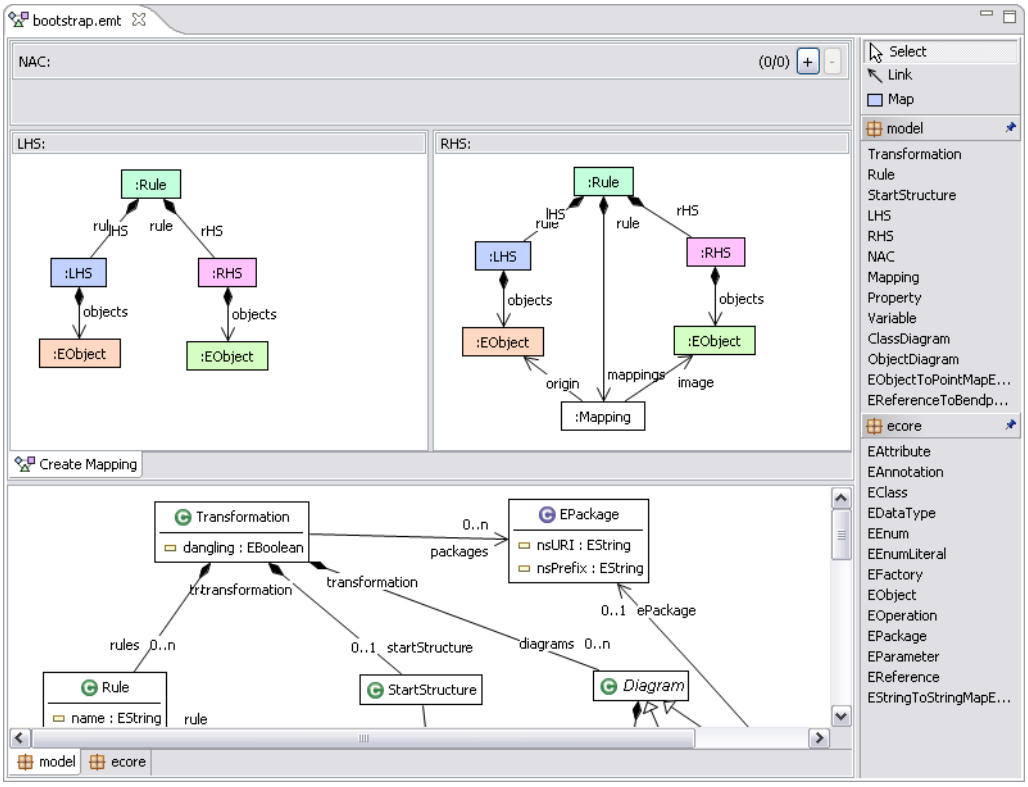


Figure 7.1: Transformations on the transformation model. The screenshot shows a rule that illustrates an editing operation of the graphical editor: a mapping between the LHS and the RHS of a rule is created.

7.4 Bidirectional Model Transformations

The QVT request for proposals includes requirements for high-level definitions of bidirectional 1:1 translations of models / languages. This includes on one hand the complete generation of a target model instance given an arbitrary source model instance (and vice versa) and the ability of translating only parts of a given instance on the other hand (*incremental updates*). For the latter case it is necessary to keep some sort of tracking references between the source and the target model[1].

At a first glance, the presented model transformation approach for EMF is not restricted to exactly two or any other number of models. Therefore it is correct to say that bidirectional transformations are only a special case of model transformation

[1]The differentiation between source and target model is misleading in as much as both models are equitable.

that can be expressed. In fact, in section 6.1 it was shown how the framework can be used for endogenous model transformations, where there is only one imported model. So instances of that language are being transformed to instances of the same language.

For the same reason there is no explicit entity for tracking references in the transformation model, which are usually needed in bidirectional transformations. Also this is not a restriction of our approach, since bidirectional transformations are only a special case of transformations and tracking references are simply additional entities, which can be defined in a separate model, which forms the bridge between the source and the target model (in both directions).

Further it is not possible to simply revert arbitrary transformation rules as requested in the QVT approach. This is due to the fact, that attribute calculations cannot be inverted in general and because negative application conditions must be considered, too. The graph transformation based approach is unidirectional, because transformations are defined in a constructive way. That means, it is specified *how* to transform a given model instance, not only how the transformation result should look like. Therefore, each source-target model transformation must be inverted manually at this point of development.

Again, it is important to point out that these are not restrictions of the transformation approach, but only side-effects that occur with the reason that the approach can be applied to any kind of model transformation and not only to bidirectional transformations.

Nevertheless, bidirectional and endogenous transformations are the most important ones in practical applications. While endogenous transformations already can be modeled smoothly with the presented approach, there are some more high-level features missing for the definition of bidirectional 1:1 transformations. This scenario requires additional information in transformation rules, so that they can be also applied reversely. Further it seems beneficial to include explicit entities describing tracking references between a source and a target model instance. These are necessary for incremental updates as required for the QVT request for proposals.

Since bidirectional transformations are only a special case of model transformations that can be defined using the EMF transformer, it should be possible to define a special bidirectional transformation model, that is either an extension of the existing model or is defined on top of it, as an high-level layer for this special application case. This new transformation model would introduce a new *vocabulary* for defining

bidirectional transformations and simultaneously restrict the possible model transformations.

Triple graph grammars as discussed in [KS06] are a graph based approach for such incremental, bidirectional transformations. They involve a lot of extra theoretical assertions that must first be translated to EMF. It is not clear, whether this approach is really suitable for EMF transformations. This could be the start for further theoretical work and implementations on top of the existing ones.

Appendix A

Proofs

A.1 Containment Theorem

Theorem A.1.1 (Pushouts of graphs with containment edges)**.** *Given a span of graphs with containment edges* $(G_1 \overset{f_1}{\leftarrow} G_0 \overset{f_2}{\rightarrow} G_2)$*, where* f_1 *and* f_2 *are valid homomorphisms for graphs with containment edges, the pushout result in the category of graphs* $(G_1 \overset{f_2'}{\rightarrow} G_3 \overset{f_1'}{\leftarrow} G_2)$ *forms also a valid pushout in the category of graphs with containment edges, if and only if the containment condition holds for the span* $(G_1 \overset{f_1}{\leftarrow} G_0 \overset{f_2}{\rightarrow} G_2)$*.*

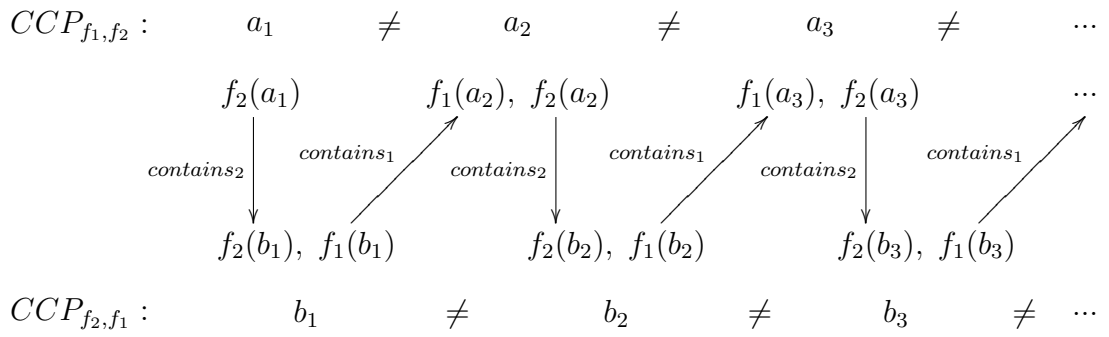
Proof. **At most one container.** Let $a \in NCP_{f_1} \cap NCP_{f_2}$. Then $f_1(a) \in V_1$ has an incoming containment edge $e_1 \in C_1$ and in the same way $f_2(a) \in V_2$ has an incoming containment edge $e_2 \in C_2$. Both, e_1 and e_2 cannot have an origin in G_0 (by definition 4.4.3). The Pushout construction for graphs glues the two node images together to a new node $b := f_2' \circ f_1(a) = f_1' \circ f_2(a) \in V_3$. Since all morphisms must be valid homomorphisms for graphs with containment edges, the node $b \in V_3$ has two incoming containment edges: $f_2'(e_1)$ and $f_1'(e_2)$, which is not allowed.

Conversely, if there is a node $b \in V_3$ with two incoming containment edges, these two edges can be written as $f_2'(e_1)$ and $f_1'(e_2)$ with $e_1 \in C_1$ and $e_2 \in C_2$, w.l.o.g. If both edges had an origin in G_1 for instance, then G_1 was not a valid graph with containment edges, because it would have a node with two incoming containment edges. Since these two edges come from different graphs, the node $b \in V_3$ must have been constructed by gluing two nodes from G_1 and G_2 together, having a common node origin in the interface graph: $a := (f_2' \circ f_1)^{-1}(b) = (f_1' \circ f_2)^{-1}(b) \in V_0$. If the node $a \in V_0$ had an incoming containment edge, this edge would have an image in G_1

and G_2 and these two edges would have been glued together to a *single* containment edge in G_3. So a cannot have an incoming containment edge in G_0 and therefore is a newly contained point, both of f_1 and f_2.

No containment cycles. Let $CCP_{f_1,f_2} \neq \emptyset$. Then $CCP_{f_2,f_1} \neq \emptyset$ by definition of cyclic contained points. The morphisms $f_1' : G_2 \to G_3$ and $f_2' : G_1 \to G_3$ must preserve the containment edges from G_1 and G_2. So if x, y are nodes in G_0 and $f_1(x)$ $contains_1$ $f_1(y)$ and conversely $f_2(y)$ $contains_2$ $f_2(x)$, then the images of x, y in G_3 form a containment cycle. The important fact is, that the order properties of both, $contains_1$ and $contains_2$ must be preserved in $contains_3$.

Assume that the images of $CCP_{f_1,f_2} \cup CCP_{f_2,f_1}$ in G_3 do not form a containment cycle. Now, let a_i be the nodes in CCP_{f_1,f_2} and b_j the nodes in CCP_{f_2,f_1}. Choose an arbitrary $a_1 \in CCP_{f_1,f_2}$ and $b_1 \in CCP_{f_2,f_1}$ so that $f_2(a_1)$ $contains_2$ $f_2(b_1)$, like the definition of cyclic contained points states. Since b_1 should not form a cycle with a_1 in G_3 and the fact that b_1 is also a cyclic contained point, there must be $a_2 \in CCP_{f_1,f_2}$ with $f_1(b_1)$ $contains_1$ $f_1(a_2)$ and $a_1 \neq a_2$. If there is no cycle in G_3 this step must be repeated ad infinitum:



This works only if CCP_{f_1,f_2} and CCP_{f_2,f_1} are either empty or infinite sets, which contradicts with our assumptions. So they must form a containment cycle in G_3.

Conversely, if there is a containment cycle in G_3, it cannot completely come from either G_1 or G_2, because then one of these graphs would already violate the order properties of the containment edges. So one part of the cycle comes from G_1 and the other one from G_2, where each of these two parts does not form a containment cycle on its own. Further, these two parts must have been glued together somehow, otherwise there wouldn't occur a cycle in G_3. The smallest cycle that can appear consists of two nodes, one containing the other and vice versa. A cycle with one node only is not possible, because then the cycle must have been already in G_1 or G_2. Of course, the cycle can consist of more than two nodes.

Choose nodes x_3, y_3 in G_3 with the following properties:

1. The nodes x_3, y_3 belong to the containment cycle in G_3, so that x_3 $contains_3$ y_3 and y_3 $contains_3$ x_3.

2. Node x_3 has a origin $x_1 := f_2'^{-1} \in V_1$ and node y_3 has a origin $y_2 := f_1'^{-1} \in V_2$. This must be possible because G_1 must contain one non-empty part of the cycle and G_2 the other one.

3. Since the two parts from G_1, G_2 must have been glued together at at least two points, we can even say that x_1 and y_2 must have origins $x_0 := f_1^{-1}(x_1) \in V_0$ and $y_0 := f_2^{-1}(x_2) \in V_0$ with $x_0 \neq y_0$.

   Lets take the example in Figure 4.3. We choose $x_0 := e$ and $y_0 := f$.

4. Let x_0 and y_0 have no container in G_0. This is also no problem. If x_0 or y_0 is contained in another node in G_0, than choose this container (or the container of this node etc.). The container must also have an image in G_1 / G_2 and then also in G_3. The containment relation between these two nodes must also be preserved from G_0 to G_3, so that the image also must be a part of the cycle in G_3.

   In our example we have to choose $x_0 := d$ instead of e. The node $y_0 = f$ has no container.

5. Let them further be newly contained points: $x_0 \in NCP_{f_1,f_2}$ and $y_0 \in NCP_{f_2,f_1}$. So the image $f_1(x_0)$ and $f_2(y_0)$ should have a container. Neither of $f_1(x_0)$, $f_2(y_0)$ can be isolated nodes without any incoming or outgoing containement edges, because then they couldn't be a part of the cycle in G_3. If they are contained in another node in G_1 / G_2, then they are already NCPs. Otherwise they can only be containers in G_1 / G_2. Lets assume $f_1(x_0)$ is a transitive container for one or more nodes in G_1. At least one of these nodes must have no children in G_1 and have an image in G_3 that is part of the cycle, because the containment order must be preserved from G_1 to G_3. Take the origin of this node in G_0 instead of x_0.

   We had $x_0 = d$ and $y_0 = f$ in our example. We have to choose $x_0 = f$ and $y_0 = h$ instead.

6. Now we have $x_0 \in NCP_{f_1}$, $y_0 \in NCP_{f_2}$ and the fact that their images form a cycle in G_3. Since $x_0 \in NCP_{f_1}$ the image $f_2(x_0) \in V_2$ must become a container

of either $f_2(y_0)$ or the image of a node $y_0' \in V_0$ that has the same properties as y_0 (NCP of f_2, part of the cycle in G_3). Otherwise, newly contained points cannot become part of a cycle. The same property must hold for y_0. Due to this, the two nodes are cyclic contained points: $x_0 \in CCP_{f_1,f_2}$ and $y_0 \in CCP_{f_2,f_1}$.

So if there is a containment cycle in G_3 there must be at least two cyclic contained points in G_0. □

A.2 Containment Consistency Theorem

Theorem A.2.1 (Containment consistency of transformation rules)**.** *A transformation rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ produces a containment consistent transformation as defined in 4.6.2, if p does not delete any objects and and is further RHS-containment consistent.*

Proof. Consider a RHS-containment consistent rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$, that does not delete any objects, together with a match $m : L \rightarrow S_1$, determining a direct transformation $S_1 \overset{p}{\Rightarrow} S_2$, where S_1 is a containment consistent object structure, but S_2 is not containment consistent. So there is at least one object o in S_2 that is neither a root container nor contained in another object of S_2. The object must fit into one of the following two cases:

1. o has been created during rule application. In this case there must be a match origin of o in the RHS of the rule, which has no mapping to the LHS (it is being created). Further it cannot be a root container and also not contained in another object of the RHS (violates containment consistency of S_2). This contradicts with the fact that p is RHS-containment consistent.

2. o existed already in S_1, but is contained in an object c in S_1 and not contained in any object in S_2. Its container cannot have been deleted, since the rule should not delete any objects. So only the containment edges from its container c to the object has been deleted. Deletion of a containment edge can only occur if the container object and the contained object are explicitly given through the match (like in Figure 4.5). This cannot occur because p is RHS-containment consistent, which forbids that an object is contained in the LHS, but not in the RHS of a rule.

□

Appendix B

User Guide

This user guide describes a model transformation framework for EMF. The framework consists of a graphical editor for the visual definition of model transformations, a compiler that generates Java code and an interpreter that can be used to apply transformations to EMF model instances.

B.1 Installation

The following softwaare is needed in order to install the model transformation environment:

- Java 5 or later
- Eclipse SDK 3.2
- EMF 2.2.0 or later
- GEF 3.2.0 or later

Installing the EMF Model Transformation environment is done using the Eclipse Update Manager. Click on *Help* → *Software Updates* → *Find and Install...* and choose *Search for new features to install*. After that a list of update sites pops up, were you have to manually add the EMF Model Transformation update site. Click on *New remote site* and enter the following data:

- Name: EMF Model Transformation update site
- URL: `http://tfs.cs.tu-berlin.de/emftrans/update`

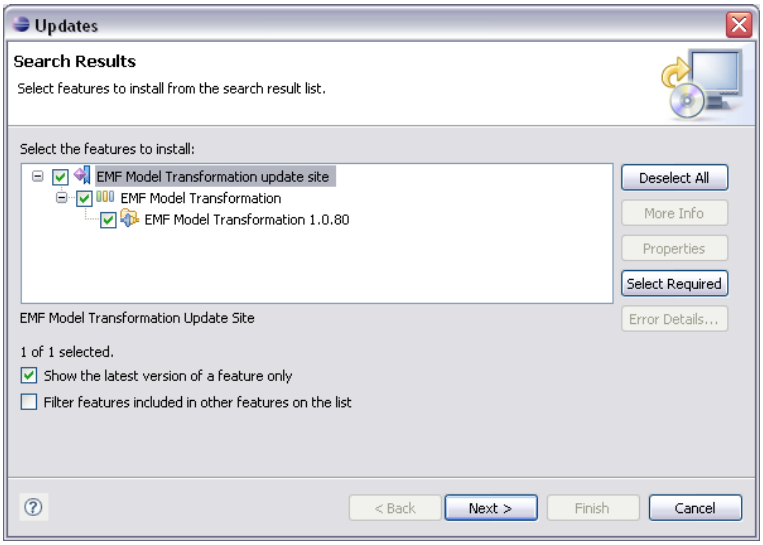


Figure B.1: Using the EMF Model Transformation update site

Now enable the update site and click on *Next*. Choose the EMF Model Transformation feature and finish the installation. After a restart of the Eclipse workbench the feature is automatically activated.

B.2 Defining Transformations

Transformations for EMF models are defined using transformation rules. These rules consist of a left-hand side (LHS), a right-hand side (RHS), possible negative application conditions (NACs) and mappings between these so called *object structures*. These object structures consist of a number of possibly linked objects typed by the EMF models. They are visulized in the editor by a diagram that contains a number of object nodes, that can be connected or attributed.

The left-hand side of a rule stands for the structural pre-conditions that must be fulfilled to apply the rule. Accordingly a right-hand side describes the result (or post-conditions) of a rule. Negative application conditions are defined in the same way and describe structural conditions that must not be fulfilled to apply the rule.

Objects in the LHS of a rule can be mapped to objects in the RHS and also to objects in the NACs. The editor visualizes mappings by coloring the mapped objects in the same way. Those objects in the LHS, which are mapped to the RHS, will be preserved during rule application. Objects in the LHS, which have no mapping to the

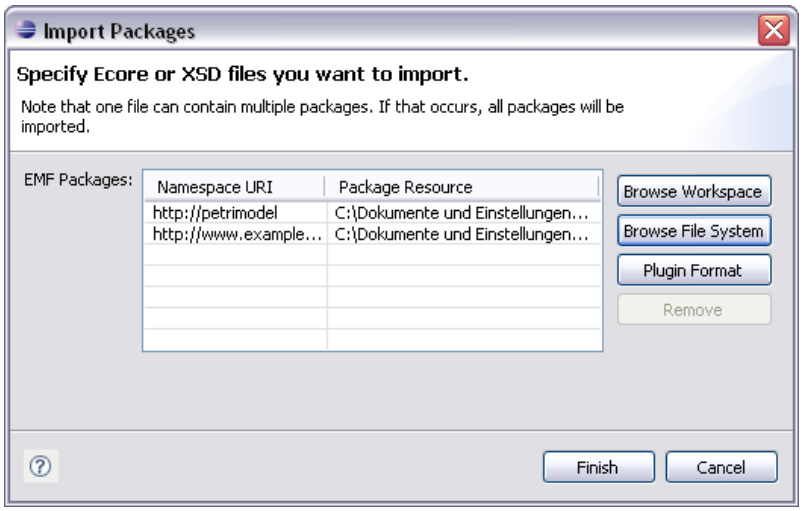


Figure B.2: Importing packages.

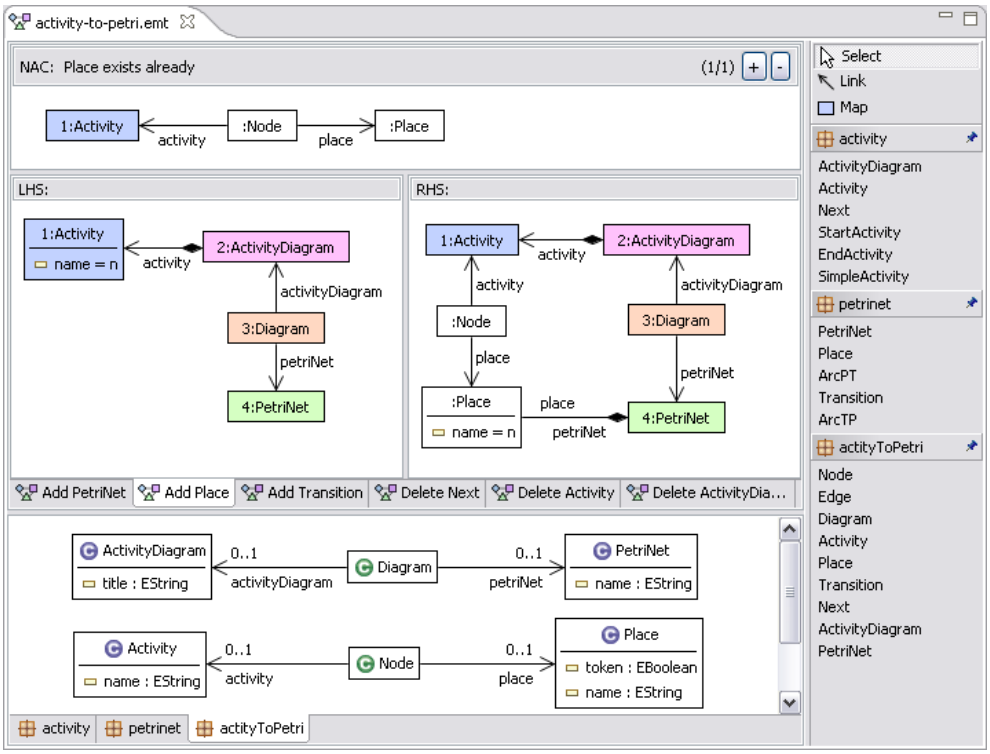


Figure B.3: A model transformation for Activity diagrams / Petri nets.

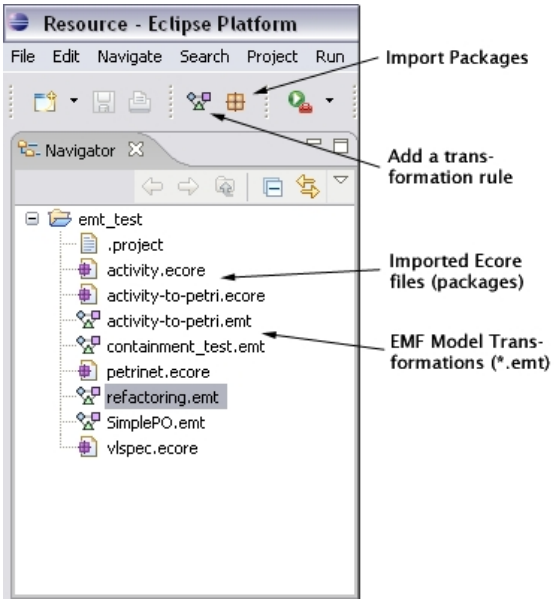


Figure B.4: Basic operations in the graphical editor.

RHS are being removed. Accordingly, objects in the RHS without a mapping will be created during rule application.

An important property of EMF classes is the possibility of defining primitive-valued attributes. These attributes are also considered in our transformation approach. In the editor, attributes of an object can be activated by assigning arbitrary java expressions, that must have the same type like the attribute as defined in the EMF class. Further typed variables can be defined for each rule and used in the expressions. Declaration of variables in the editor can be done through an entry in the context menu.

B.3 Interpreter and Compiler

Rule application is performed by a provided interpreter or using generated java classes. Code generation facilities for *.emt files is provided through a context menu entry in the resource navigator. The interpreter and the generated classes apply transformation rules by finding a match for the pattern described in the LHS into the object structure that should be transformed. These matches can be either defined explicitly or can be computed by the transformation engine.

Bibliography

[AGG] AGG homepage: `http://tfs.cs.tu-berlin.de/agg`.

[BK06] Enrico Biermann and Guenter Kuhns. Konzeption und implementierung einer regelbasierten transformationskomponente für das eclipse modeling framework. 2006.

[BSM+03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy Grose. Eclipse modeling framework: A developer's guide. 2003.

[BW99] Michael Barr and Charles Wells. Category theory lecture notes for ESSLLI: `http://www.let.uu.nl/esslli/Courses/barr-wells.html`. 1999.

[CEW93] I. Classen, H. Ehrig, and D. Wolz. Algebraic specification techniques and tools for software development: The ACT approach. *AMAST Series in Computing*, 1993.

[CH03] Krysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. *OOPSLA 03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[EB05] Hartmut Ehrig and Benjamin Braatz. Category theory for computer scientists (lecture notes): `http://tfs.cs.tu-berlin.de/lehre/SS05/KafI/`. 2005.

[EEPT] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Formal integration of inheritance with typed attributed graph transformation for efficient vl definition and model manipulation.

[EEPT05] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamentals of algebraic graph transformation. 2005.

[EMF] EMF homepage: `http://www.eclipse.org/emf`.

[EMT] EMF model transformation homepage: `http://tfs.cs.tu-berlin.de/emftrans`.

[GEF] GEF homepage: `http://www.eclipse.org/gef`.

[KK05] Krasimira Kulekova and Christian Koehler. Graphische editoren mit eclipse: `http://tfs.cs.tu-berlin.de/vila/www_ws05/folien/Vortrag-emf-gef-gmf.pdf`. 2005.

[KS06] Alexander Königs and Andy Schürr. Tool integration with triple graph grammars - a survey. *Electronic Notes in Theoretical Computer Science 148, 113-150*, 2006.

[KSW04] Jochen M. Kuester, Shane Sendall, and Michael Wahler. Comparing two model transformation approaches. 2004.

[TEB+06a] Gabriele Taentzer, Karsten Ehrig, Enrico Biermann, Guenter Kuhns, Eduard Weiss, and Christian Koehler. Graphical definition of in-place transformations in the eclipse modeling framework. *Proceedings MoDELS/UML 2006*, 2006.

[TEB+06b] Gabriele Taentzer, Karsten Ehrig, Enrico Biermann, Guenter Kuhns, Eduard Weiss, and Christian Koehler. EMF model refactoring based of graph transformation concepts. *Proc. of 3rd Workshop on Software Evolution through Transformations, EASST*, 2006.

[TEG+06] Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan de Lara, Laszlo Lengyel, Tilhamer Levendovszky, Ulrike Prange, Daniel Varro, and Szilivia Varro-Gyapay. Model transformation by graph transformation: A comapartive study. 2006.

[TR05] Gabriele Taentzer and Arend Rensink. Ensuring structural constraints in graph-based models with type inheritance. 2005.

List of Figures