Change Propagation in the MDA: A Model Merging Approach

by

Alejandro Metke Jimenez

A thesis submitted to the

School of Information Technology and Electrical Engineering
The University of Queensland

for the degree of

MASTER OF INFORMATION TECHNOLOGY STUDIES

June 2005

Statement of originality

I declare that the work presented in the thesis is, to the best of my knowledge and belief, original and my own work, except as acknowledged in the text, and that the material has not been submitted, either in whole or in part, for a degree at this or any other university.

Alejandro Metke Jimenez

Acknowledgments

This thesis is dedicated to my sister Laura and my brother Ricardo.

First of all I would like to thank my supervisor Dr. Michael Lawley for his excellent guidance throughout the project. I would also like to thank Dr. Kerry Raymond and Mr. Keith Duddy, the rest of the Pegamento team, for their support.

I would like to thank the Distributed Systems Technology Center (DSTC) for the opportunity to gain work experience through the vacation project and for the use of their office space for this project.

Finally, I would like to thank the rest of my family for their unconditional support.

Abstract

Dealing with change propagation is an important issue in the Model Driven Architecture. Propagating changes in model to model transformations is a challenging task since the target models may contain important changes that shouldn't be simply overwritten. Also, the process of determining what has changed is not straightforward since MOF models have limited ability to describe conditions for object equivalence.

This thesis describes the problem of adding change propagation capabilities to Tefkat, the DSTC's transformation engine. A model merging approach is explored, in which the target model is completely regenerated when the transformation is re-run and then merged with the previous target model. The whole process involves three major steps: finding the delta between the models, gathering input from the user to guide the merge process, and merging the models.

A meta-model to represent the delta between the models is presented. A meta-model independent algorithm to find the delta is described. The algorithm deals with state-based merging in which only the two models being merged are available and relies on the trace objects created by the engine to determine which objects in the models being merged are equivalent, without having to depend on the existence of unique object identifiers. Limitations of the algorithm for certain scenarios are discussed, and possible solutions are proposed as future research. An implementation as a model transformation is presented. A strategy to gather the user input based on simple rules is shown. Finally, an algorithm that uses the information collected in the previous steps to merge the models is described and implemented also as a transformation.

Contents

List of Figures

Chapter 1

Introduction

The Model Driven Architecture is a new software engineering paradigm that looks to improve software development productivity, as well as the quality and longevity of the software [5]. Model transformations represent a key issue in the MDA. The Query, View and Transformations Request For Proposals issued by the OMG will result in a standard transformation language that will enable MOF model transformations [13]. The DSTC's Pegamento project has designed a language that meets these requirements [7] and is working on a transformation engine, Tefkat, based on the Eclipse Modeling Framework open source project.

In an MDA project, target models derived from an automatic transformation are likely to suffer a considerable amount of manual changes. Also, since change is the only constant in software engineering, source models can also suffer changes during the development life-cycle. The transformation definitions may also change if, for example, the best practices change. In re-running a transformation these changes should not be ignored and MDA tools should provide a mechanism to preserve them and resolve any conflicts that may be encountered in the process.

There are several approaches to deal with the problem of change propagation in the context of model to model transformations. This thesis explores a merging approach, in which the new target model is generated and then merged with the previous target model. In order to do so, the different types of changes that may occur need to be identified. Merging strategies used in other domains must be evaluated in order to determine an appropriate way to deal with the merge process in this context. Also, a strategy that helps direct the merge process must be derived. Finally, a solution that incorporates such a strategy and that can be easily integrated with Tefkat must be implemented.

Tefkat is designed in such a way that models cannot be updated, only created. Therefore, the engine is in charge of generating the traces that relate the

objects created in the target model with the objects in the source that they were created from. Each time a transformation is run a new trace is generated. The merge process considers changes in the source model, the target model, and the transformation definition. If the source model changes, the previous version is not available, since Tefkat does not provide versioning support. The same happens if the target model changes. This is why the merge process in this case is state based. Only the original target model, that may or may not have changed, and the new target model are available. Since the transformation is re-run then two traces exist. The first one relates the source model with the original target model and the second one relates the same source model with the new target model. If changes have occurred in the source model or the original target model then the original trace may have dangling references. The new trace doesn't have dangling references since it relates the source model with the new target model that has just been generated.

1.1 Model Driven Architecture

The Model Driven Architecture is a new framework that looks to improve the traditional software development process by raising the level of abstraction to the model layer. Development starts with a PIM (Platform Independent Model), which captures the system's requirements without considering the target platform. This model is then refined into a PSM (Platform Specific Model). Finally, a PSM is further refined into source code.

A model is a description of a system. To be able to automate some of the processes that occur in the MDA, models must be described using a well defined language that allows automated interpretation by a computer. Text based languages are usually defined using a grammar in Backus Naur Form (BNF). A text based modeling language could be defined this way, but since modeling languages are not necessarily text based (for example, UML uses graphical notation), another mechanism is needed. This mechanism is know as meta-modeling [1]. A modeling language is itself a model (that is, a model of a modeling language) and so it must also be described using a well defined language. The model that defines a modeling language is known as its meta-model. For example, the UML meta-model defines all the concepts that may exist in a UML model. A meta-model is a model as well, and it must also be described using a well defined language. This model is known as the meta-meta-model of the language.

Since theoretically there could be an infinite number of layers, the OMG has defined a four layer architecture. The layers are called M0, M1, M2 and M3. Layer M0 corresponds to model instances. Elements in this layer would be the

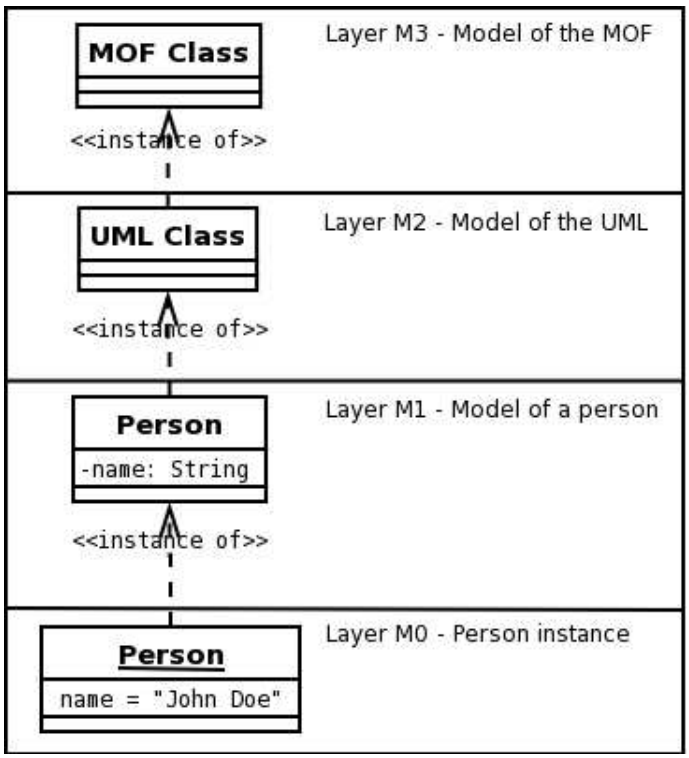


Figure 1.1: An example of the OMG's four layer architecture.

actual representations of the objects being modeled, such as "John Doe", in the case of persons. Layer M1 corresponds to the model. If UML was being used as the modeling language, the class Person would be found in this layer. Layer M2 corresponds to the meta-model. The model that defines UML would be found in this layer. Finally, layer M3 corresponds to the meta-meta-model. In order to have a common language used to define all of the modeling languages used in the MDA, the OMG developed the Meta Object Facility standard. Layer M3 contains the model that defines MOF. This model, however, is itself defined using MOF, and thus there is no need for any extra layers. Figure 1.1 shows a simple example of the four layer architecture.

It is also important to consider that the structure of a model is determined by its meta-model. However, any algorithm that operates on models in the MDA context should be able to deal with the most general form a model can take. In general, any model can be represented as a directed graph.

1.2 Model Transformations

1.2.1 Kinds of Transformations

Transformations can be classified according to the type of source and target they operate on. The following categories are relevant in the context of the MDA:

- Text to text. In this case the source and target are textual artifacts. XSLT is a popular technology used for doing this type of transformation. Other more restricted approaches on model transformations use it by operating on the XMI representation of the models [7]. A compiler is another example of a text to text transformation, turning a high level source code file into low level assembly code.
- Model to text. This type of transformation can be used at a higher level of abstraction to produce source code from models. An example is the EMF code generator, based on JET [14, 15]. This type of transformation can also be used to produce different types of textual representations of models such as the ones specified by the XMI and HUTN standards [12, 11].
- Text to model. Text to model transformations work the other way around and can be used to generate models from textual representations. A HUTN parser is an example of this type of transformation.
- Model to model. This type of transformation helps automate the refinement process between models. By defining a set of transformation rules, not only can a PIM be transformed into a PSM, but also the best practices can be captured in the transformation definition.

Note however that model transformations are not only relevant in the MDA context. A model to model transformation language can be used to transform arbitrary models, as long as they have a meta-model expressed using the MOF. For example, the Pegamento project has used Tefkat to automatically create specific HUTN parsers for different meta-models, by transforming them into ANTLR models that are then transformed into grammar files using a model to text transformation.

1.2.2 OMG's QVT

The OMG has issued a Request For Proposals that will result in a standard language for querying, creating views and transforming MOF models[13]. The language developed by the DSTC's Pegamento project focuses on model to model transformations and meets these requirements [8]. It is declarative, pattern based and allows the transformations to address semantic concepts, rather

that structural features. The basic concepts in the language are rules, patterns and traces. A transformation definition is made up of several transformation rules. These rules relate a pattern in the source model to objects that are to be created in the target model. A pattern represents a set of objects in the source model that match certain criteria and also acts as a template for objects that are to be created in the target model. Traces associate objects in the source model with objects in the target model. They are necessary in order for rules to refer to objects that might have already been created in the target model [7].

1.3 Change Propagation in the MDA

A simple example will be used to illustrate the relevance of the change propagation problem in the context of a real MDA project.

1.3.1 UML to Relational Transformation

Software systems often use relational databases to handle persistent data. If a system is specified using UML, the corresponding relational model can be produced automatically by applying a set of rules to the UML model. One possible transformation is shown in Appendix A.

1.3.2 The Online Video Rental Store

Suppose we need to build an online system to rent movies. Figure 1.2 shows a very simple model that includes three classes: a Customer class that holds the information of the video rental customers, a DVD class that holds the information of the movies being rented and a RentalOrder class that holds the information of the orders placed by the customers. When the UML to relational transformation is run, the resulting model is the relational model needed to create the database tables for the system. However, suppose a requirement of the system is to keep a log of the system's transactions through the database engine. In this case the requirement is not really part of the UML model, but nevertheless has to appear somewhere. The solution is to add a table for logging purposes in the relational model. The resulting model can be seen in Figure 1.3.

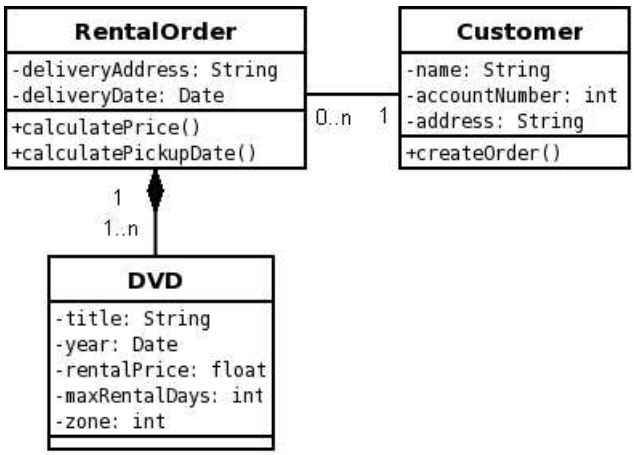


Figure 1.2: A simple UML model of the online video rental store.

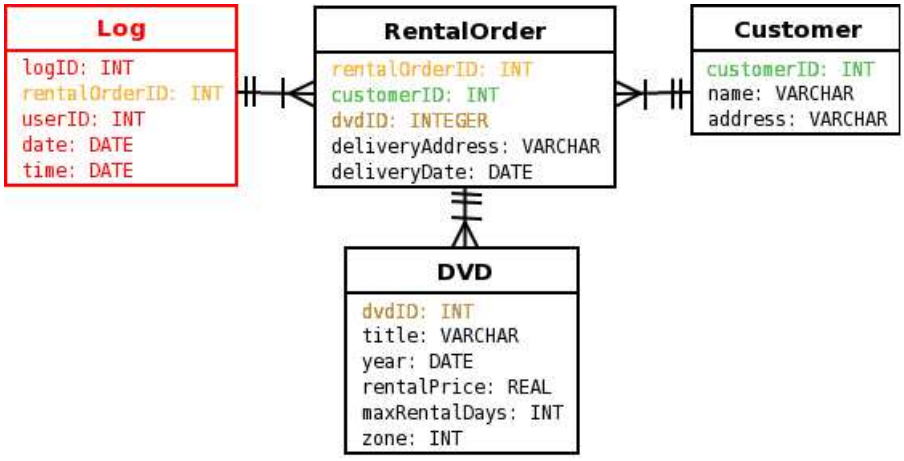


Figure 1.3: The relational model for the online video rental store. The Log table has been added manually to the model.

Suppose now that customer feedback indicates the need to include VHS tapes in the online store. Since the original UML model was created to handle only DVDs, it has to be updated to handle this new requirement. One possible solution is to introduce two classes: a Movie class and a VHS class. Most of the DVD's attributes can be moved to the Movie class and both DVD and VHS can extend Movie. Figure 1.4 shows the UML model updated in this way.

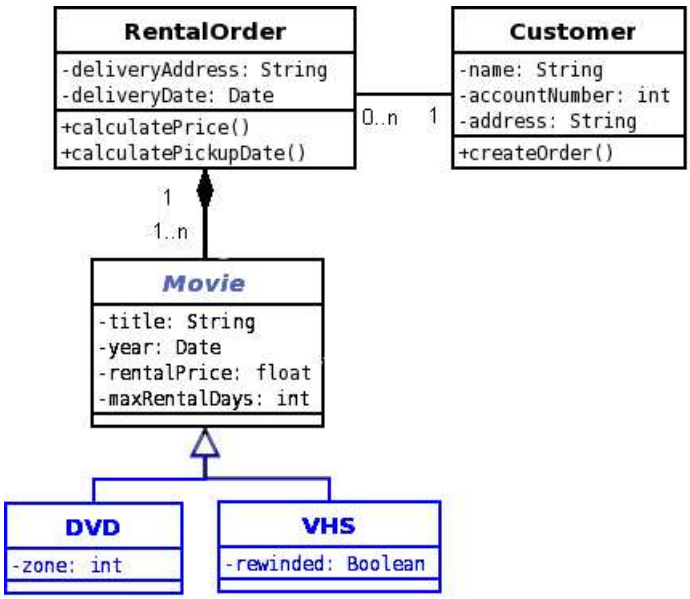


Figure 1.4: The updated UML model of the online video rental store, now able to handle VHS tapes and DVDs.

Now that the UML model has changed, the previously generated relational model will no longer work with the new system. The UML to relational transformation has to be re-run in order to propagate the changes to the relational model. However, if the transformation engine doesn't support change propagation, the manual changes introduced in the target model will be overwritten. In this case the Log table would be lost. In this simple example it is only a single table, but in a real system the manual changes could be many. Figure 1.5 shows the result of re-running the transformation on the updated UML model.

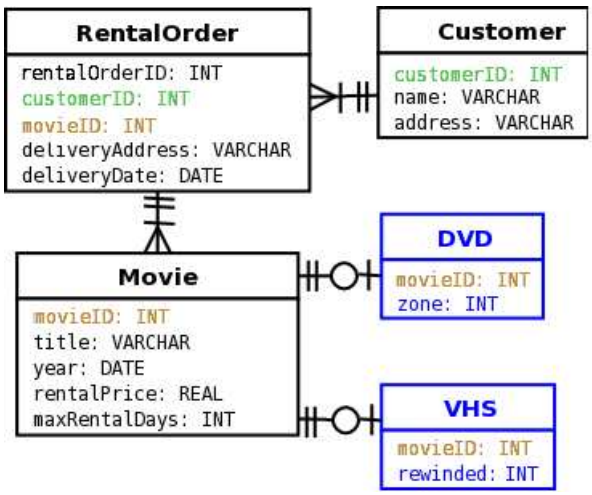


Figure 1.5: The result of re-running the transformation on the updated online video store UML model.

In order to preserve the changes using the merging approach, the new target model has to be merged with the previously generated target model. In this simple case it is obvious what the result should be: we want to keep the manual changes that were added to the old target model. However, in more complex scenarios, the different types of changes could be many. Figure 1.6 shows what the resulting merged model should look like.

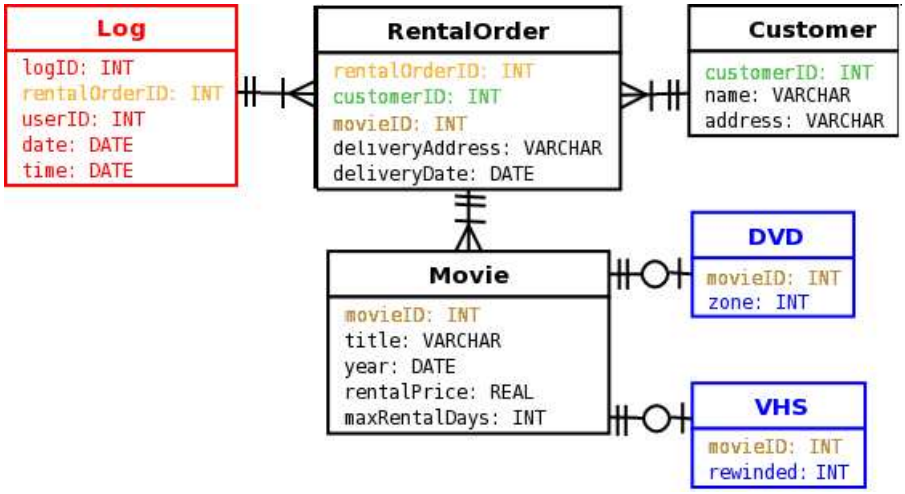


Figure 1.6: The merged online video store relational model.

1.4 Thesis Overview

Chapter 1 has provided an introduction to the Model Driven Architecture and has shown the importance of model transformations within this new software development paradigm. Also, the change propagation problem has been illustrated though a simple but meaningful example. An overview of how the model merging approach can be used to solve the problem has also been given.

The remainder of the thesis is structured as follows: Chapter 2 gives an overview of software merging techniques used in different domains and discusses the problem of merging models in the context of a model transformation engine. Chapter 3 shows how the process of detecting changes between two target models being merged can be solved using a transformation. A delta meta-model used to keep track of these changes is also introduced. Chapter 4 shows how, based on the information collected in the change detection phase and the input received from the user, the models can be merged using a transformation. Finally, Chapter 5 summarizes the work done and discusses the important issues that still need to be dealt with in the future.

Chapter 2

The Merging Problem

This thesis explores a model merging approach to deal with the problem of change propagation. Therefore, it is useful to study the different merging techniques that have been used to solve other similar problems. The next section provides an overview of software merging and the rest of the chapter describes the specific details of the model merging problem.

2.1 Software Merging

The problem of software merging has been studied extensively, mainly in the context of version control mechanisms based on optimistic locking [9]. Most merge tools employ textual merge techniques, since this approach makes them very flexible and allows them to work with any text based artifact. However, the most important limitation of this approach is that the tools know nothing about the semantics of the artifacts being merged, and thus their ability to merge conflicts automatically is also limited.

Since in the context of a version control system a common ancestor of the software artifacts being merged is always available, most tools take advantage of this and use a three-way merge technique. This type of merge is more powerful than a two-way merge, since more conflicts can be detected [9]. This technique involves detecting and representing change in data, usually known as finding the delta between two files. Textual version control tools usually use deltas to store the different versions of the files, in order to use less physical space and minimize network traffic in a distributed environment.

Change detection and representation has also been studied for hierarchically structured data. In [16], the authors describe an efficient algorithm to derive the minimum cost edit script that transforms an ordered tree into another. This involves finding a good match between the two trees, and defining a cost model

in order to determine which of the different conforming edit scripts has the minimum cost. The algorithm doesn't rely on the existence of object identifiers. An efficient algorithm for detecting changes in unordered trees has also been described in [18], in the context of XML files. Finding a good match in unordered trees without relying on object identifiers is a much harder problem. In this case the authors take advantage of the specific characteristics of XML files to achieve efficiency.

In [2] the authors study the problem of merging MOF models, in the context of a version control system. The algorithm described is similar to the ones used for change detection in hierarchically structured data, but in this case, since the structure of the data contained in a model is not constrained to a tree structure, it relies on the existence of unique object identifiers for the matching process.

Finally, the problem of change preservation has been dealt with in the context of model to text transformations. The Eclipse Modeling Framework provides a model to text transformation functionality that allows Ecore models to be transformed automatically into Java code [4]. To do so, EMF relies on the Java Emitter Templates and JMerge technologies, which provide a framework for code generation and merging [14, 15]. In order to preserve the changes, JET uses a tagging strategy based on custom javadoc tags, which are used to guide a post-generation merge process. When source code is generated automatically from a model, the generated methods and attributes are marked with the tag @generated. If the source code generation process is re-run, only the methods and attributes marked as generated will be overwritten. In order to indicate that a change must be preserved, the generated tag must be removed or changed to the tag @generated NOT. The level of granularity is determined by the artifacts being generated, in this case attributes and methods in classes.

2.2 Model Merging

Tefkat is a model transformation engine that receives one or several source models as inputs and produces one or several target models as outputs. Models cannot be modified by Tefkat, only created. Each time a transformation is run a trace model that relates the objects in the source with the objects created in the target is generated. There is no support for version control of the models built in directly, since the engine is only concerned with providing the transformation functionality. Therefore, in dealing with change propagation using the model merging approach, the type of merge that is required is two-way, which means that only the two models being merged are available. A two-way merge is always state-based. There is no way to know precisely what changes have occurred to the models since a common parent is not available, as opposed to the case

exposed in [2].

There are several steps involved in the process of merging the two models. The following sections discuss each one of them.

2.2.1 Detecting Changes

The first step in the model merging process is detecting the changes between the two target models that are going to be merged. The process of detecting the changes also involves several steps, which will be explained in the following subsections.

Finding Matching Objects

The first step in the change detection process is matching the equivalent objects in the models being merged, in order to know which objects we need to compare in order to find the differences in their attributes and references. In [2] the authors rely on unique object identifiers to find the equivalence. In this case the matching process is trivial: the objects with the same identifiers are considered equivalent. However, unique object identifiers are not always available, and this is the case with Tefkat, since it was not designed to impose such a requirement on the models it operates on.

There are several algorithms that can be used to find matches in structured data that don't rely on unique object identifiers [16]. However, these algorithms cannot be used to find the matching of the models since the structure of an arbitrary MOF model is not necessarily hierarchical. Each model can be considered as a labeled, directed graph. The problem of finding a good match is reduced to finding an isomorphic subgraph of one of the models in the other, and using this isomorphism to define the matching. The problem with this approach is that finding isomorphic subgraphs has been proved to be an NP-complete problem [10], making it an impractical solution for a real application.

Fortunately, in the case of Tefkat, there is a easier way to achieve the matching. When a transformation is run, a trace model is generated, containing trace objects that link the target objects with the source objects they were generated from. The generic trace model is shown in Figure 2.1. Each trace object is associated with exactly one target object and it may be associated with one or more source objects. When the transformation is re-run, a new target model is generated, along with a new trace model, as shown in Figure 2.2.

A model M can be considered a set of objects o. Let the old target model be $M_T$and the newly generated target model M_{NT}. Also, lets introduce two sets, D_S and D_T to represent the sets of objects that have been deleted from the source and target models respectively. The type of the objects is given by

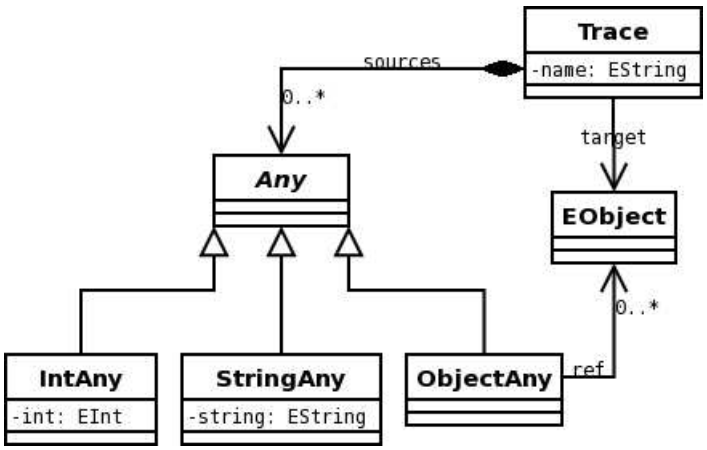


Figure 2.1: The generic trace model.

a total function $type \in M \rightarrow TYPE$, where the set $TYPE$ contains the possible object types defined in the meta-model. Formally, a trace model is a relation between objects in the source model and objects in the target model, such that $trace \in \mathbb{P}_1(M \cup D_S) \leftrightarrow M \cup D_T$. The matching between the objects in M_T and M_{NT} can be defined initially as a partial injective function

$matching = \{(o_1, o_2) \mid o_1 \in M_{NT} \wedge o_2 \in M_T \wedge T_{TR}^{-1}[\{o_2\}] = T_{NTR}^{-1}[\{o_1\}] \wedge type(o_1) = type(o_2)\}$.

By analyzing the trace models generated in each transformation run, the equivalent objects can be identified, even if their attribute or reference values have changed. The following simple algorithm shows how to do this:

1. Iterate through the objects in the original trace model (TR). For each one of them check for a dangling reference to the original target model (T).

2. If the reference is not dangling then check the reference to the source model (S).

3. If none of the references to the source model is dangling then iterate through the trace objects in the new trace model (NTR).

4. For each object in the new trace model check the reference to the source model.

5. If the set of objects referenced by the new trace object is the same as the set of objects referenced by the original trace object then check that the object pointed to by the trace object in the new target model (NT) is of the same type as the one pointed to by the trace object in the original trace model.

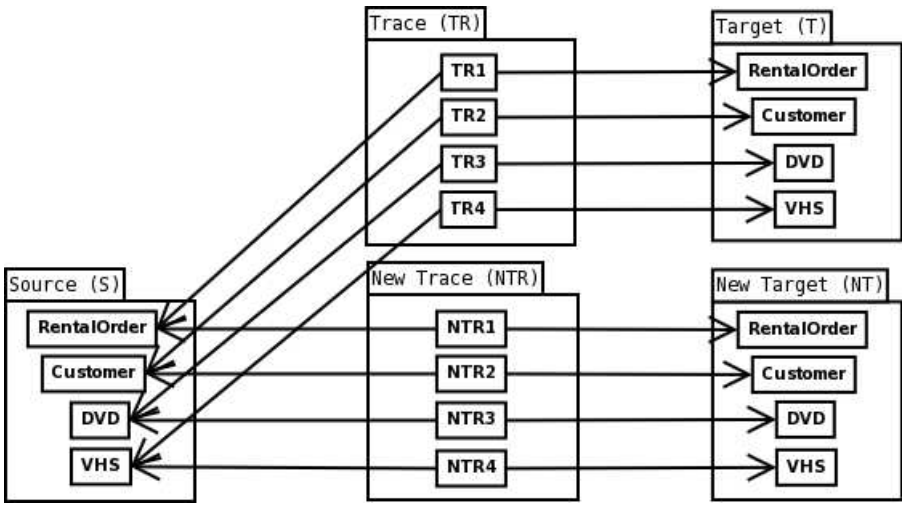


Figure 2.2: The result of re-running a transformation.

6. If the objects have the same type then a pair of matching objects has been found.

There are several important things to notice. First, there is no need to check for dangling references from the new trace model since the merge process is an atomic operation and the user cannot modify the models while the process is running. Second, an object in the target model may be related to more than one object in the source model. For example, in the UML to relational transformation the generated columns are related to a class and an attribute. Therefore, when checking for equivalence, the set of objects pointed to in the source model must be exactly the same for both trace objects being analyzed. If a reference from a trace object in the original trace model to the source model is dangling, then the target object referenced by the trace object has no match. Finally, the type of the objects has to be the same for the objects to be matched. This check is important because objects with different types may be generated from the same set of objects in the source model.

Finding Unmatched Objects

Now that the matching objects have been identified, it is simple to determine which objects in each target model have no corresponding match just by checking which ones do not belong to the set of matching objects. However, by using the information in the trace model, it is also possible to determine why the object is unmatched. There are five different possible reasons for an object to be unmatched:

1. An object was deleted from the source model. In this case, when the transformation is re-run, the target objects related with the deleted object will not be regenerated and the objects previously generated in the original target model will be unmatched. Figure 2.3 shows an example of an object being deleted from the source model, in a simple one to one transformation. In this case the VHS object in the source model is deleted and the VHS object in the original target model is now unmatched, since the VHS object is not regenerated in the new target model. Notice that the trace object in the original trace model (TR4) now has a dangling reference to the source model.

   This kind of unmatched objects will belong to the original target model.

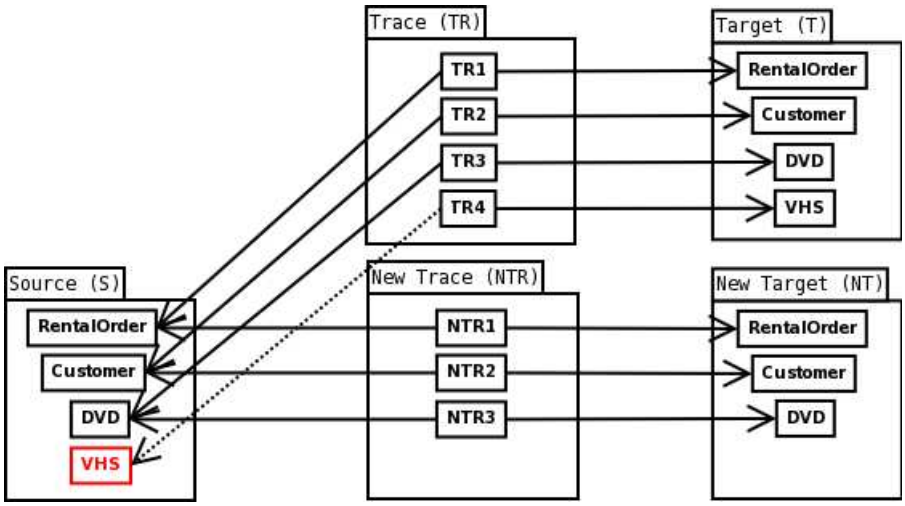


Figure 2.3: An example of an object deleted from the source model.

   Formally, the set of these objects can be defined as:
   $\{o \mid o \in M_T \wedge o \in ran(T_{TR}) \wedge \exists e.(e \in T_{TR}^{-1}[\{o\}] \wedge e \in D_S)\}$.

2. An object was added to the target model. In this case there is no trace object related to the new object since it is added manually to the generated model. Figure 2.4 shows an example of an object, Beta in this case, being added to the original target model.

   These kind of unmatched objects will also belong to the original target model. Formally, the set of these objects can be defined as:
   $\{o \mid o \in M_T \wedge o \notin ran(T_{TR})\}$

3. An object was deleted from the target model. In this case, if none of the corresponding objects in the source model are deleted, then the object will

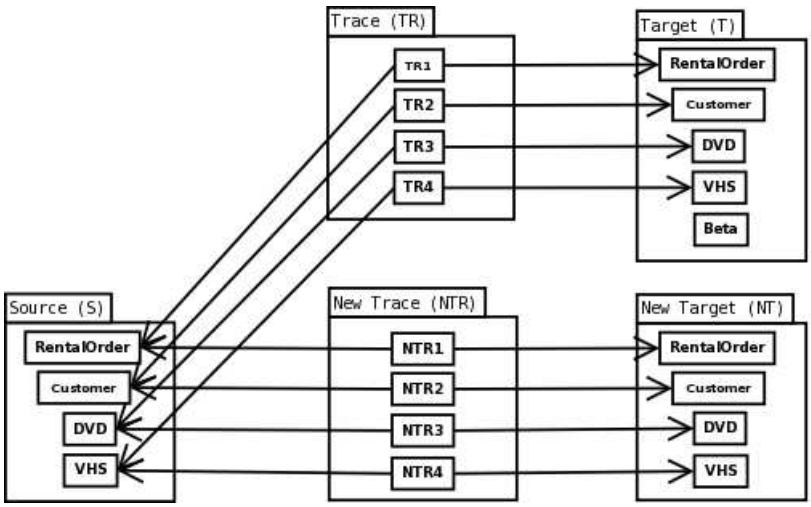


Figure 2.4: An example of an object being added to the target model.

be regenerated in the new target model and will have no match. Figure 2.5 shows an example of an object being deleted from the original target model. In this case the VHS object is deleted, and since the corresponding VHS object in the source model is still in the source model then the VHS object is regenerated in the new target model and is now unmatched. Notice that the trace object in the original trace model (TR4) now has a dangling reference to the original target model.

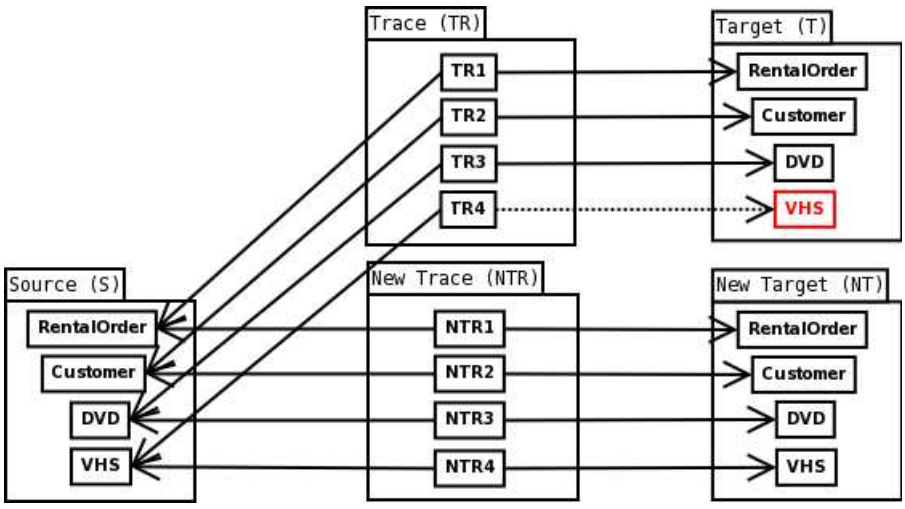


Figure 2.5: An example of an object being removed from the original target model.

These kind of unmatched objects will belong to the new target model. Formally, the set of these objects can be defined as:
$\{o \mid o \in M_{NT} \wedge T_{NTR}^{-1}[\{o\}] \in dom(T_{TR}) \wedge T_{TR}[T_{NTR}^{-1}[\{o\}]] \in D_T\}$.

4. An object was added to the source model. In this case only the trace object in the new trace model exists since the added source object didn't exist when the original transformation was run. Figure 2.6 shows an example of an object being added to the source model. In this case the Beta object is added and the corresponding Beta object is generated in the new target model.

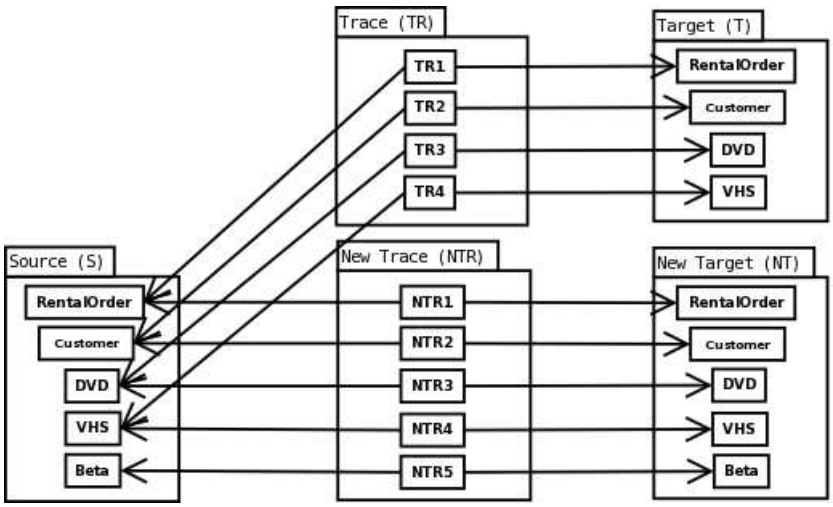


Figure 2.6: An example of an object added to the source model.

These kind of unmatched objects will belong to the new target model. Formally, the set of these objects can be defined as:
$\{o \mid o \in M_{NT} \wedge T_{NTR}^{-1}[\{o\}] \notin dom(T_{TR})\}$.

5. An unknown change has happened in the source model. All the objects that aren't matched and that don't belong to the previous categories fall into this one. A refactoring in the source model may result in an object in the original target model being unmatched, since it may be related to more than one object in the source model. Figure 2.7 shows an example of a refactoring that produces unmatched objects in both the original target model and the new target model. This example is based on a UML to relational transformation. The *name* attribute, that belongs to the Customer class, is moved to the RentalOrder class. The corresponding column in the target model is generated from both the attribute and the class that

contains the attribute. Therefore, when the transformation is re-run, the new name attribute that is generated is referenced by the same attribute as in the original transformation but a different class (the RentalOrder class in this case). Even though the traces have no dangling references, the set of objects that they point to in the source model is different. This means that even though the objects are equivalent, they will be recognized as unmatched by the algorithm. Notice that it is not possible to match these objects based only on the information provided by the trace analysis. Similar effects may be produced by changes in the transformation definition and may also lead to the impossibility to match equivalent objects.

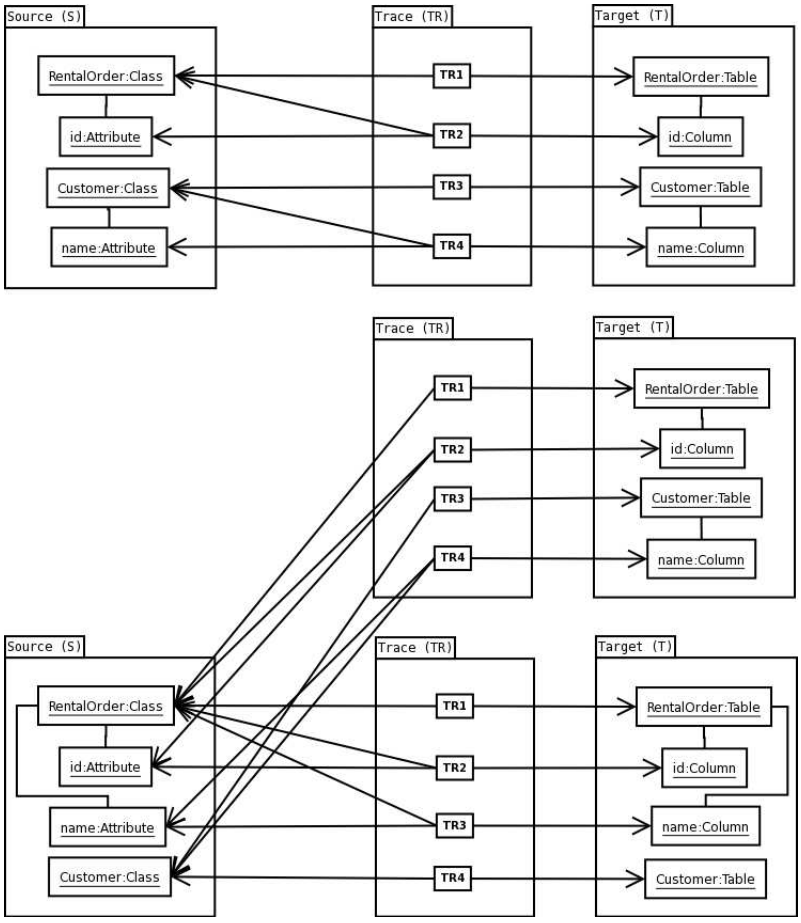


Figure 2.7: An example of a refactoring in the source model.

These kind of unmatched objects will belong to both target models. Formally, the set of these objects can be defined as:
$\{o \mid o \in M_T \wedge T_{TR}^{-1}[\{o\}] \notin dom(T_{NTR}) \wedge \forall e.(e \in T_{TR}^{-1}[\{o\}] \Rightarrow e \notin D_S)\} \cup$

$\{o \mid o \in M_{NT} \wedge T_{NTR}^{-1}[\{o\}] \notin dom(T_{TR}) \wedge \forall e.(e \in T_{NTR}^{-1}[\{o\}] \Rightarrow e \notin D_S)\}$.

Finding the Attributes' Delta

Once the matching objects have been identified, the process of finding the difference in their attribute values is straightforward. The algorithm consists of iterating through the pairs of matching objects and comparing their attribute values. When a difference is found it must be recorded by keeping track of the pair of matching objects and the attribute being compared.

Finding the References' Delta

The process of finding the references that have changed is not as simple. There are three different types of changes that may be found, depending on the type of reference:

1. If the reference is single-valued then two changes can occur: the reference is set to point to a different object or the reference is unset. A single-valued reference can be defined formally as a partial function $singleref \in NAME \times M \rightarrow M$. Let R_{T1} be the set of single-valued references in the original target model and R_{NT1} the corresponding set in the new target model. Also, lets define a partial injective function $matching \in M_T \rightarrowtail M_{NT}$ to represent the set of matching objects. The pairs of matching objects that have a change in this type of reference can be defined as: $\{(o_1, o_2) \mid o_1 \in M_T \wedge o_2 \in M_{NT} \wedge matching(o_1) = o_2 \wedge \forall r_1, r_2, n.(r_1 \in R_{T1} \wedge r_2 \in R_{NT1} \wedge n \in NAME \wedge o_1 \mapsto n \in dom(r_1) \wedge o_2 \mapsto n \in dom(r_2) \Rightarrow matching(r_1(n, o_1)) \neq r_2(n, o_2))\}$.
   Figures 2.8 and 2.9 show examples of changes in a single-valued reference.

2. If the reference multi-valued and not ordered then a change can occur if a reference is added or removed from the set of references. An unordered multi-valued reference can be defined formally as a partial function $multiref \in NAME \times M \rightarrow \mathbb{P}(M)$. Let R_{TN} be the set of unordered multi-valued references in the original target model and R_{NTN} the corresponding set in the new target model. The pairs of matching objects that have a change in this type of reference can be defined as: $\{(o_1, o_2) \mid matching(o_1) = o_2 \wedge \forall r_1, r_2, n.(r_1 \in R_{TN} \wedge r_2 \in R_{NTN} \wedge n \in NAME \wedge o_1 \mapsto n \in dom(r_1) \wedge o_2 \mapsto n \in dom(r_2) \Rightarrow \exists o_3, o_4.(o_3 \in r_1(n, o_1) \wedge o_4 \in r_2(n, o_2) \wedge matching(o_3) \neq o_4))\}$ Figure 2.10 shows an example of a change in an unordered multi-valued reference.

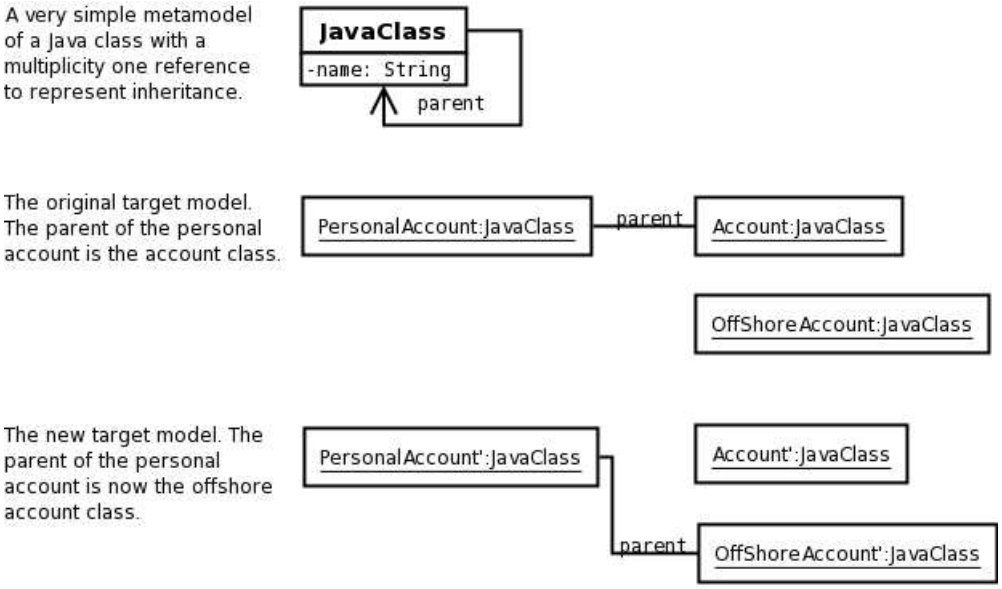


Figure 2.8: An example of a change in a single-valued reference in which a reference is set to point to another object.

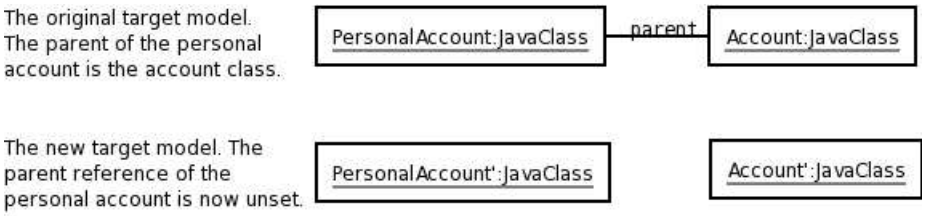
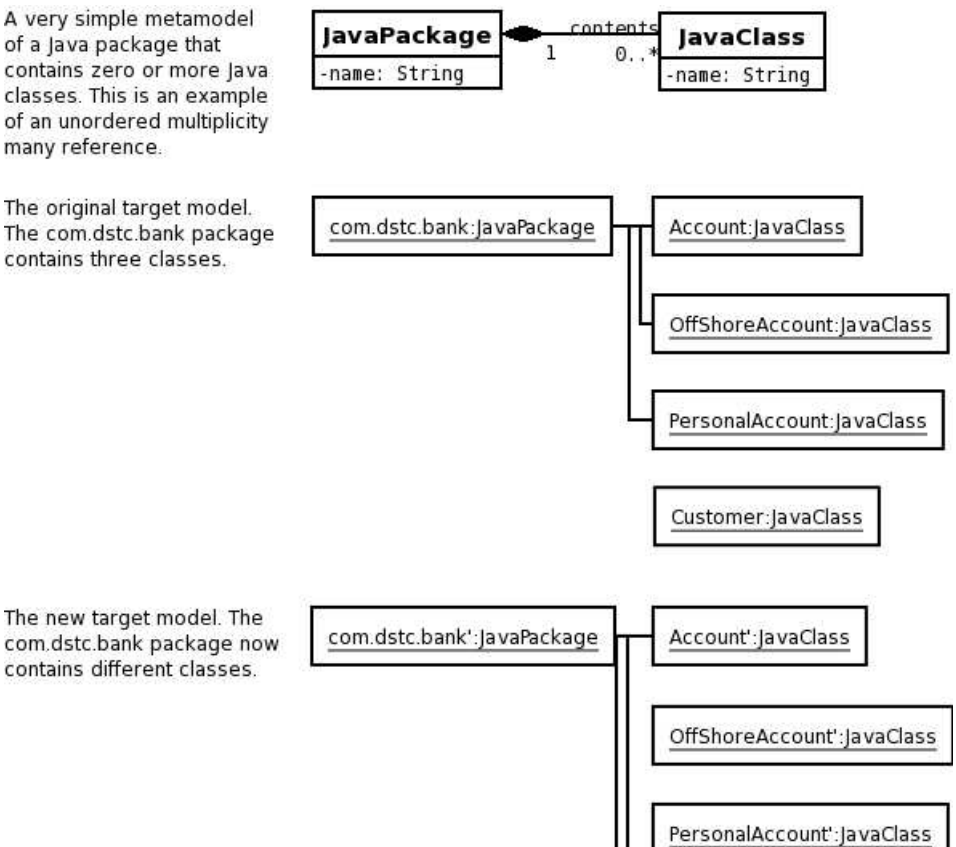


Figure 2.9: An example of a change in a single-valued reference in which a reference is unset.



3. If the reference is multi-valued and ordered then a change can occur if a reference is added or removed from the list of references or the position of one of the references changes. An ordered multi-valued reference can be defined formally as a partial function $orderedref \in NAME \times M \rightarrow iseq(M)$. Let R_{TO} be the set of ordered multi-valued references in the original target model and R_{NTO} the corresponding set in the new target model. The pairs of matching objects that have a change in this type of reference can be defined as:
$\{(o_1, o_2) \mid matching(o_1) = o_2 \wedge \forall r_1, r_2, n.(r_1 \in R_{TO} \wedge r_2 \in R_{NTO} \wedge n \in NAME \wedge o_1 \mapsto n \in dom(r_1) \wedge o_2 \mapsto n \in dom(r_2) \Rightarrow \exists o_3, o_4.(o_3 \in ran(r_1(n, o_1)) \wedge o_4 \in ran(r_2(n, o_2)) \wedge matching(o_3) \neq o_4))\} \cup$
$\{(o_1, o_2) \mid matching(o_1) = o_2 \wedge \forall r_1, r_2, n.(r_1 \in R_{TO} \wedge r_2 \in R_{NTO} \wedge n \in NAME \wedge o_1 \mapsto n \in dom(r_1) \wedge o_2 \mapsto n \in dom(r_2) \Rightarrow \exists o_3, o_4.(o_3 \in ran(r_1(n, o_1)) \wedge o_4 \in ran(r_2(n, o_2)) \wedge matching(o_3) = o_4 \wedge r_1(n, o_1)^{-1}[o_3] \neq r_2(n, o_2)^{-1}[o_4]))\}$
Figure 2.11 shows an example of a change in an ordered multi-valued reference.

2.2.2 Gathering User Input

Once the information about the differences in the target models has been collected, the actions that must be taken with respect to each change must be decided. A simple approach to deal with this problem is to specify a set of default rules for each type of change. For example, it is probably wise to keep the objects that have been added to the target model. However, it is likely that there are some cases in which a different action needs to be taken. Therefore, the user must be able to manually override the actions for each change. Once the delta model has been updated with the input gathered from the user the models can be merged.

2.2.3 Merging the Models

Now that an action has been specified for each difference in the models the process of merging them is fairly straightforward. However, the actions that need to be taken depend on the merging strategy. There are two ways in which the process can be handled: to create a new model or to merge the changes in place in the new target model. The following sections discuss each case.

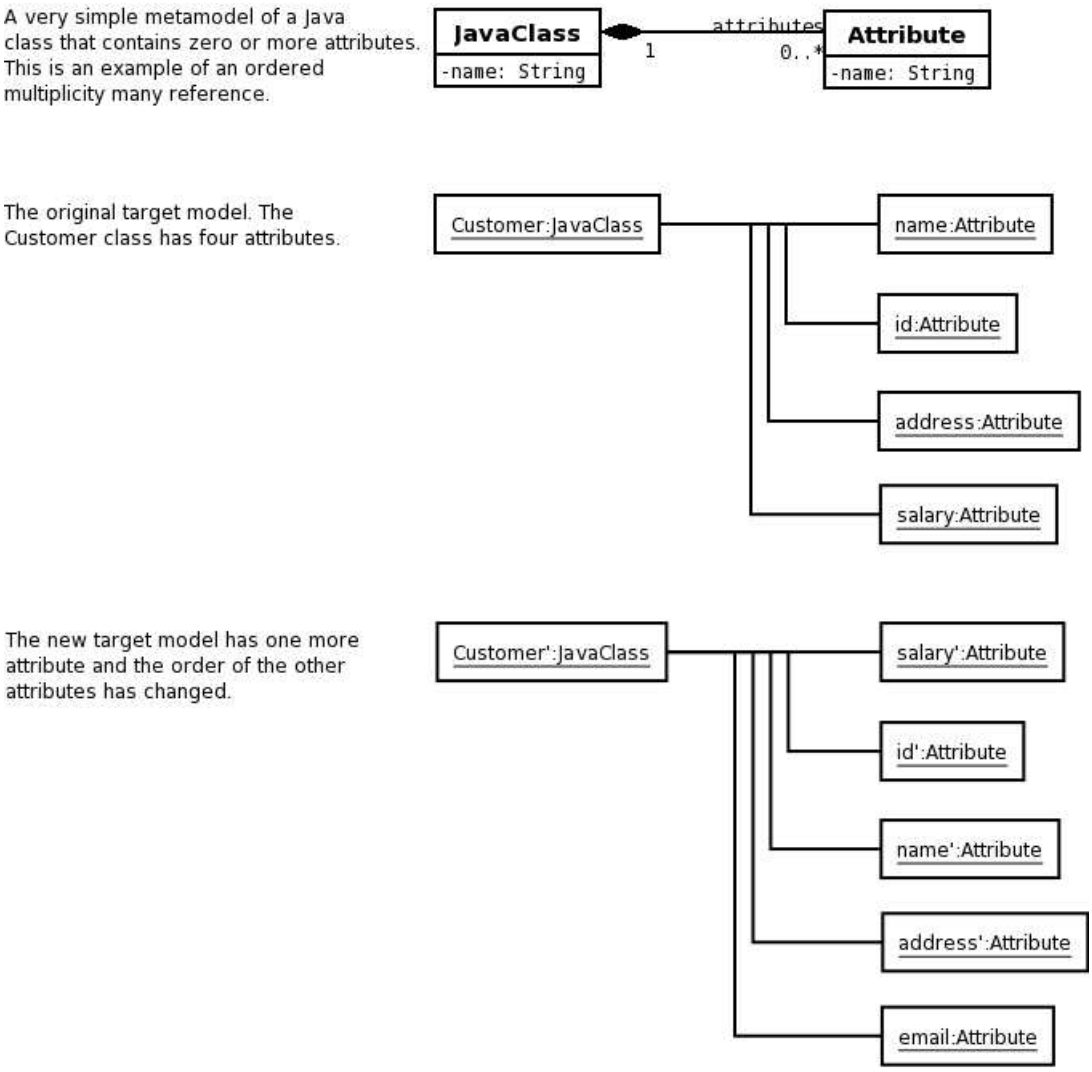


Figure 2.11: An example of a change in a multiplicity many ordered reference.

Creating a New Model

If a new model is going to be created then the actions required to merge the models into this new model are the following:

1. The matching objects and their unchanged attributes and references must be copied into the new model.
2. The unmatched objects whose associated action is to keep the object must be copied into the new model.
3. The attributes of the matching objects copied into the new model that

have changed must be set to the value specified in the associated action, which indicates if the value must be copied from the original target model or the new target model.

4. The references from the matching objects copied into the new model that have changed must be set to the value specified in the associated action. Depending on the type of reference the action will indicate if the value or relative order of the reference should be copied from the original target model or the new target model.

Modifying the New Target Model In Place

If the changes are going to be merged into the new target model then the following actions are needed:

1. The unmatched objects in the original target model that have an associated action that specifies that they must be kept in the merged model must be copied into the new target model.

2. The unmatched objects in the new target model that have an associated action that specifies that they must not be kept in the merged model must be removed.

3. The attributes that have changed and have an associated action that specifies that they must keep the value in the original target model must be set to the corresponding value.

4. The references that have changed and have an associated action that specifies that they must keep the value in the original target model must be set to the corresponding value. Depending on the type of reference the action will indicate if a reference must be added, changed or moved within the collection of references.

2.3 Merging In Tefkat

This chapter has shown an algorithm to find the delta between two target models based on the analysis of the traces that relates them to a common source model. The next chapter shows how to implement this algorithm using a transformation. An algorithm to merge the models based on this information and information gathered from the user was also presented and several ways to handle the merge process were discussed. Chapter 4 shows how to implement this algorithm also using a transformation.

Chapter 3

Change Detection

There are several ways to implement the change detection algorithm shown in Chapter 2. It could be implemented programmatically using the EMF API. However, detecting changes basically involves querying the models and traces. Therefore, it makes sense to use the QVT language to solve the problem, since it was designed specifically to query and transform models. The process of finding the delta between the models can be considered a model transformation. In this case the transformation takes the the two target models, the source model and the trace models as inputs and generates a delta model as an output. This model not only holds the information about the differences in the target models but also the actions that should be taken to produce the merged model. The default actions for each type of change can be specified in a configuration file, and the model can be modified in order to tweak the actions for particular changes.

3.1 The Delta Model

Figure 3.1 shows the MOF diagram of the delta meta-model. The following subsections explain the important meta-classes.

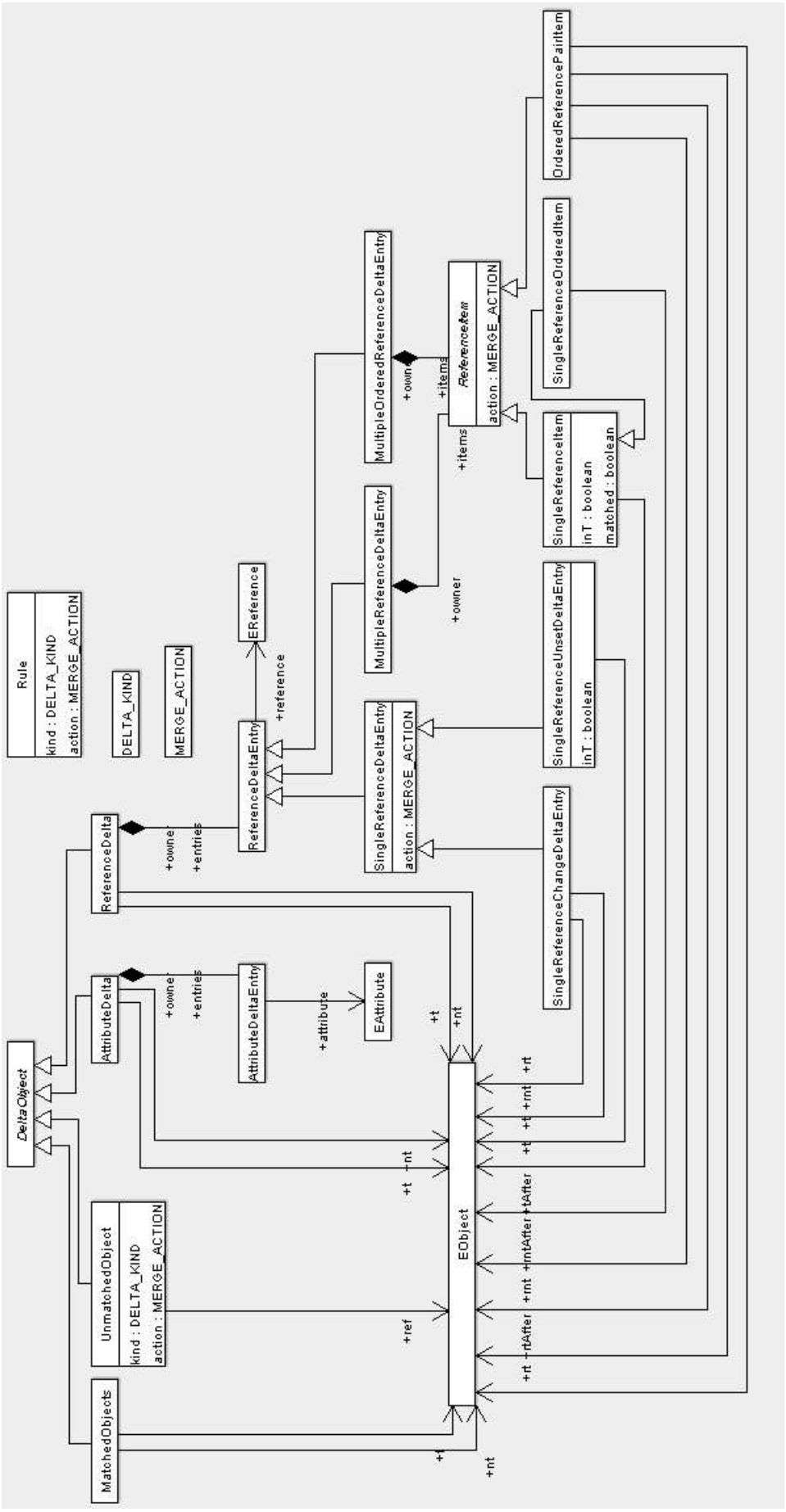


Figure 3.1: The MOF diagram of the delta meta-model.

3.1.1 Matched Objects

The MatchedObjects class is used to keep track of the objects that are identified as matching. The *t* reference points to the object in the original target model and the *nt* reference points to the corresponding object in the new target model. An instance of this class must be created for each pair of matching objects in order to copy the unchanged parts of the target model into the merged model.

3.1.2 Unmatched Objects

An instance of the UnmatchedObject class is created for each object in the target models that doesn't have a matching object. The *kind* attribute is used to indicate the reason why the object is unmatched. Section 3.3.2 discusses how to obtain this information by using the tracking objects. The *action* attribute is used to indicate what to do with this object when the models are merged. Section 3.2 shows how the default actions for each type of change can be set by creating a model containing instances of the Rule class.

3.1.3 Attributes Delta

The attribute values of the matched objects may have changed. For each pair of matching objects that have a different value in one of their attributes an instance of AttributeDelta is created. The *t* and *nt* references point to the pair of objects. For each attribute that has a different value, an instance of AttributeDeltaEntry is created and added to the *entries* reference. Each entry holds the action that must be taken when merging the models.

3.1.4 References Delta

There are several types of changes that may occur when dealing with references, depending on the type of reference. For each pair of matching objects that have a different value in one of their references an instance of ReferenceDelta is created. The *t* and *nt* references point to the pair of objects. The following entries are created depending on the type of reference and change:

- If the reference is single-valued and is now pointing to another object, then an instance of SingleReferenceChangeDeltaEntry is created. The *rt* reference points to the object referred to by *t*. The *rnt* reference points to the object that is referred to by *nt*. However if the reference was unset, then an instance of SingleReferenceUnsetDelta is created instead. These cases have to be handled with separate rules since *null* references must be dealt with using the methods provided by EMF. This class has a reference,

r, that points to the object referred to by either *t* or *nt*. The reference in the matching object is going to be unset. A boolean attribute, *inT*, is used to indicate if the object is part of the original target model or the new target model.

- If the reference is multi-valued but not ordered, then for each reference that refers to a different set of objects in either target model, an instance of MultipleReferenceDeltaEntry is created. Each object that doesn't belong to the intersection of the sets of references from the matching objects must be reported as a difference. For each one of these objects an instance of SingleReferenceItem is created. This class has a boolean attribute, *inT*, used to indicate to which model does the single reference belong to.

- If the reference is multi-valued and ordered then things get interesting. A naive approach would be to compare all the references' values with the same index and report the ones that differ between the target models. However, in this case a single change could produce a lot of delta entries if, for example, the first reference of the collection is removed. In this case all of the other references would be reported as changes since their index would no longer match. Therefore an approach that takes the relative order into consideration is used. For each referenced object without a corresponding match, a SingleReferenceOrderedItem instance is created. This class inherits from SingleReferenceItem and has an additional reference, *tAfter,* that points to the next object in the collection. This reference is used to insert the reference in the correct relative position when merging the models. For every matching referenced object that has a different relative position than its corresponding object, an instance of OrderedReferencePairItem is created. This class has four references: *rt* that points to the object pointed to by *t*, *rtAfter* that points to the next matched object after *rt*, *rnt* that points to the object pointed to by *nt* and *rntAfter* that points to the next matched object after *rnt*.

3.2 Rule Based Configuration

When calculating the delta between the two target models, different kinds of changes have to be identified. During the merge process, a certain action must be taken for each change in the delta model. For example, if an unmatched object is found in the original target model then there are two possible actions that might be taken during the merge: to keep the unmatched object or not to keep it. If the value of an attribute of a matched object has changed, then only one of the values can be kept. In order to provide a default behavior for

each one of the possible kinds of changes, a model with Rule instances must be used. The Rule class has two attributes: *kind* that specifies the kind of change and *action* that specifies the default action that must be taken for this kind of change. The transformation that calculates the delta model uses this model to set the default actions for the changes it encounters. The actions for each change in the delta model can then be modified manually, if the default value is not what the user wants.

3.3 The Change Detection Process

The following sections explain the rules used to find each kind of change.

3.3.1 Object Matching

The matching objects can be determined by the rule in Figure 3.2. In line 41 the trace objects from the original tracking model and the new tracking model are selected. The @ is used when there is more than one source model, to indicate from which one the objects must be selected. Line 42 checks that the original trace has a valid reference to the original target model, since the object it refers to may have been deleted. Line 43 checks that the possible matching objects are of the same type. Lines 44 and 45 make use of the patterns in Figure 3.3 to find the trace objects that point to the same set of objects in the source model. In order to find them, the negated inverse condition, an existential quantification, is used. The patterns are finding the trace objects that don't point to the same set of objects in the source model by looking for some object in the source model that is not pointed at by one of the trace objects. By using the NOT operator this existential quantification can be turned into the original universal quantification that was required. In line 46 a MatchedObjects instance is created and its values are set in lines 47 and 48. Finally, the matched objects are linked with a MatchingObjects instance in line 49.

```
40 RULE findMatchingObjects(str, sntr)
41 FORALL Trace@tr tr, Trace@ntr ntr
42 WHERE tr.target.eIsProxy() = false
43   AND tr.target.eClass().name = ntr.target.eClass().name
44   AND NOT (notInNTR(tr.sources, ntr))
45   AND NOT (notInTR(ntr.sources, tr))
46 MAKE MatchedObjects o2o
47 SET o2o.nt = ntr.target,
48   o2o.t = tr.target
49 LINKING MatchingObjects WITH t = o2o.t, nt = o2o.nt
50   ;
```

Figure 3.2: Rule used to match the corresponding objects in the target models.

```
12 PATTERN inNTR(ref, ntr)
13   WHERE ntrs = ntr.sources
14     AND ntrs.eClass().name = "ObjectAny"
15     AND ntrs.ref = ref
16   ;
18 PATTERN notInNTR(sources, ntr)
19   WHERE sources.eClass().name = "ObjectAny"
20     AND NOT inNTR(sources.ref, ntr)
21   ;
23 PATTERN inTR(ref, tr)
24   WHERE trs = tr.sources
25     AND trs.eClass().name = "ObjectAny"
26     AND trs.ref = ref
27   ;
29 PATTERN notInTR(sources, tr)
30   WHERE sources.eClass().name = "ObjectAny"
31     AND NOT inTR(sources.ref, tr)
32   ;
```

Figure 3.3: Patterns used to find the matching objects.

3.3.2 Objects

The following sections show the rules used to find the unmatched objects according to the reason why they are unmatched.

An Object Was Deleted from the Source Model

The rule in Figure 3.8 is used to find the objects that become unmatched because of an object being deleted from the source model. In line 53 the trace objects from the original transformation are selected. Lines 54 to 57 select only those trace objects that have a dangling reference to the source model, by checking if one of the references points to a proxy. In EMF, when a the object that a reference points to cannot be resolved, it is replaced by a proxy. This check is equivalent to checking for *null* references in standard Java. Line 58 checks that the object previously generated in the original target model hasn't been deleted. Lines 61 to 63 create the UnmatchedObject instances and set the *ref* value to the unmatched object in the original target model, the kind to the corresponding enumeration value, and the action to the default action specified in the rules. Finally, the objects are linked with an UnmatchedObjectsInT instance in line 64.

```
52 RULE findObjectsDeletedFromSource(sr, ref, en, act)
53 FORALL Trace@tr otr
54 WHERE  sr = otr.sources
55   AND sr.eClass().name = "ObjectAny"
56   AND ref = sr.ref
57   AND ref.eIsProxy() = true
58   AND otr.target.eIsProxy() = false
59   AND RuleLink LINKS name = "objectDeletedFromSource", action = act
60 MAKE UnmatchedObject umo
61 SET umo.ref = otr.target,
62   umo.kind = DELTA_KIND.OBJECT_DELETED_FROM_SOURCE,
63   umo.action = act
64 LINKING UnmatchedObjectsInT WITH t = otr.target
65   ;
```

Figure 3.4: Rule used to find objects that became unmatched because of an object being deleted from the source model.

An Object Was Added to the Target Model

The rule in Figure 3.5 is used to find the unmatched objects that were added to the original target model. Lines 68 and 69 select the objects in the original target model that have no matches. This is possible because the matching objects were already determined earlier and the results linked. Line 70 uses the isTarget pattern, shown in Figure 3.6, to ensure that the selected objects are not the target of any trace objects, since there may also be some of them that don't have a match in the new target model because the matching object in the source model was deleted. Lines 72 to 75 create an UnmatchedObject instance and set the *ref* value to the unmatched object in the original target model, the kind to the corresponding enumeration value, and the action to the default action specified in the rules. Finally, the objects are linked with an UnmatchedObjectsInT instance in line 76.

```
67 RULE findObjectsAddedToTarget(en, act)
68 FORALL _@t ot
69 WHERE NOT MatchingObjects LINKS t = ot
70   AND NOT isTarget(ot)
71   AND RuleLink LINKS name = "objectAddedToTarget", action = act
72 MAKE UnmatchedObject umo
73 SET umo.ref = ot,
74   umo.kind = DELTA_KIND.OBJECT_ADDED_TO_TARGET,
75   umo.action = act
76 LINKING UnmatchedObjectsInT WITH t = ot
77   ;
```

Figure 3.5: Rule used to find objects added to the target model.

```
 7 PATTERN isTarget(T)
 8 FORALL Trace@tr TR
 9 WHERE T = TR.target
10  ;
```

Figure 3.6: A pattern used to determine if an object is the target of an object of the original trace model.

An Object Was Deleted from the Target Model

The rule in Figure 3.7 is used to find the objects that became unmatched because of an object being deleted from the target model. Lines 80 to 82 select the pairs of trace objects that point to the same sets of objects in the source model. Line 83 filters these pairs by selecting only the trace objects in the original trace that have a dangling reference to the original target model. Lines 85 to 88 create an UnmatchedObject instance and set the *ref* value to the unmatched object in the new target model, the kind to the corresponding enumeration value, and the action to the default action specified in the rules. Finally, the objects are linked with an UnmatchedObjectsInNT instance in line 89.

```
79 RULE findObjectsDeletedFromTarget(act)
80 FORALL Trace@tr tr, Trace@ntr ntr
81 WHERE NOT (notInNTR(tr.sources, ntr))
82   AND NOT (notInTR(ntr.sources, tr))
83   AND otr.target.eIsProxy() = true
84   AND RuleLink LINKS name = "objectDeletedFromTarget", action = act
85 MAKE UnmatchedObject umo
86 SET umo.ref = ontr.target,
87   umo.kind = DELTA_KIND.OBJECT_DELETED_FROM_TARGET
88   umo.action = act
89 LINKING UnmatchedObjectsInNT WITH t = ontr.target
90  ;
```

Figure 3.7: Rule used to find objects that became unmatched because of an object being deleted from the original target model.

An Object Was Added to the Source Model

The rule in Figure 3.8 is used to find the objects that become unmatched because of an object being added to the source model. Lines 98 and 99 use the pattern in Figure 3.9 to select the trace objects in the new trace model that don't have a corresponding trace object in the original trace model that points to the same set of objects. Lines 101 to 105 create an UnmatchedObject instance and set the *ref* value to the unmatched object in the new target model, the kind to the corresponding enumeration value, and the action to the default action specified in the rules. Finally, the objects are linked with an UnmatchedObjectsInNT instance in line 105.

```
 97 RULE findObjectsAddedToSource(en, act)
 98 FORALL Trace@ntr ontr
 99 WHERE NOT hasMatchingTrace(ontr)
100   AND RuleLink LINKS name = "objectAddedToSource", action = act
101 MAKE UnmatchedObject umo
102 SET umo.ref = ontr.target,
103   umo.kind = DELTA_KIND.OBJECT_ADDED_TO_SOURCE,
104   umo.action = act
105 LINKING UnmatchedObjectsInNT WITH t = ontr.target
106   ;
```

Figure 3.8: Rule used to find objects that became unmatched because of an object being added to the source model.

```
34 PATTERN hasMatchingTrace(NTR)
35 FORALL Trace@tr TR
36 WHERE NOT (notInNTR(TR.sources, NTR))
37   AND NOT (notInTR(NTR.sources, TR))
38   ;
```

Figure 3.9: Pattern used to find the matching trace objects from the original and new trace models.

Unknown Cause

The rule in Figure 3.10 is used to find all the objects in the original target model that haven't been identified as matching or as unmatched. Lines 110 and 111 find these objects by using the linking objects created in the previous rules. Lines 113 to 116 create an UnmatchedObject instance and set the *ref* value to the unmatched object in the original target model, the kind to the corresponding enumeration value, and the action to the default action specified in the rules. The rule in Figure 3.11 finds this type of unmatched objects in the new target model.

```
108 RULE findOtherUnmatchedObjectsinT(ont)
109 FORALL _@t ot
110 WHERE NOT UnmatchedObjectsInT LINKS t = ot
111   AND NOT MatchingObjects LINKS t = ot, nt = ont
112   AND RuleLink LINKS name = "otherObjectInTarget", action = act
113 MAKE UnmatchedObject umo
114 SET umo.ref = ot,
115   umo.kind = DELTA_KIND.OTHER_OBJECT_IN_TARGET,
116   umo.action = act
117   ;
```

Figure 3.10: Rule used to find the objects in the original target model that have no known cause for being unmatched.

```
119 RULE findOtherUnmatchedObjectsinNT(ot)
120 FORALL _@nt ont
121 WHERE NOT UnmatchedObjectsInNT LINKS nt = ont
122   AND NOT MathcingObjects LINKS t = ot, nt = ont
123   AND RuleLink LINKS name = "otherObjectInNewTarget", action = act
124 MAKE UnmatchedObject umo
125 SET umo.ref = ont,
126   umo.kind = DELTA_KIND.OTHER_OBJECT_IN_NEW_TARGET,
127   umo.action = act
128   ;
```

Figure 3.11: Rule used to find the objects in the new target model that have no known cause for being unmatched.

3.3.3 Attributes

The rule in Figure 3.12 is used to find the pairs of matching objects that have some change in their attribute values. In line 131 all the objects in the original target model are selected. In line 132 the WHERE clause is used to select only the objects that have a match in the new target model, and the match is stored in the *nt* variable. Lines 133 to 138 compare all the attribute values of the pair and match only the ones that have a different value. Notice that lines 137 and 138 check for the case in which one of the attributes has been unset by using the EMF eIsSet method. Lines 139 to 141 create an AttributeDelta instance for each object that matched the WHERE clause. Finally, in line 142, the AttributeDelta instances are linked, since they have to be populated with the actual changes with another rule, shown in Figure 3.13. In this rule, in line 147 the AttributeDelta instances that had been previously created are selected. In lines 148 to 154 the attribute values are checked, to see which ones have changed. Lines 156 to 158 create an instance of AttributeDeltaEntry for each one of the differences and set the *attribute* value to the corresponding attribute and the action to the default action specified in the rules. Line 159 adds this entry to the corresponding AttributeDelta instance.

```
130 RULE findAttributeDelta(c, a, nt, act)
131 FORALL _@t t
132 WHERE MatchingObjects LINKS t = t, nt = nt
133   AND t.eClass() = c
134   AND c.eAllAttributes = a
135   AND a.changeable = true
136   AND (t.$a != nt.$a
137     OR (t.eIsSet(a) = true AND nt.eIsSet(a) = false)
138     OR (t.eIsSet(a) = false AND nt.eIsSet(a) = true))
139 MAKE AttributeDelta ad
140 SET ad.t = t,
141   ad.nt = nt
142 LINKING TToAttributeDelta WITH t = t, ad = ad
143   ;
```

Figure 3.12: Rule used to find the matching objects with differences in their attribute values.

```
145 RULE findAttributeDeltaEntries(ad, c, a, nt, act)
146 FORALL _@t t
147 WHERE TToAttributeDelta LINKS t = t, ad = ad, name = "AttDelta"
148   AND t.eClass() = c
149   AND c.eAllAttributes = a
150   AND a.changeable = true
151   AND nt = ad.nt
152   AND (t.$a != nt.$a
153     OR (t.eIsSet(a) = true AND nt.eIsSet(a) = false)
154     OR (t.eIsSet(a) = false AND nt.eIsSet(a) = true))
155   AND RuleLink LINKS name = "attributeChanged", action = act
156 MAKE AttributeDeltaEntry ade FROM ade(t, a)
157 SET ade.action = act,
158   ade.attribute = a,
159   ad.entries = ade
160   ;
```

Figure 3.13: Rule used to find the differences in the matching object's attribute values.

3.3.4 References

Several kinds of changes may occur in references depending on its type. The following sections show the rules used to find changes for each type of reference.

Single-Valued References

The rule in Figure 3.14 is used to find changes in single-valued references. The pairs of matching objects are selected in lines 163 and 164, and stored in the t and nt variables. Lines 165 to 169 select the objects' references that are single-valued. Line 170 finds the object that is referenced by the t and stores it in the rt variable. Line 171 finds the object that is referenced by nt and stores it in the rnt variable. If rt and rnt are not matching objects then a change in this type

of reference has been found. Line 172 checks this condition. Lines 174 to 178 create a SingleReferenceChangeDeltaEntry instance and set the *reference* value to the reference being checked, the *rt* value to the object referenced by *t*, the *rnt* value to the object referenced to by *nt*, and the action to the default action specified in the rules. Finally, the SingleReferenceChangeDeltaEntry instances are linked with a DeltaEntry instance in line 179.

```
162 RULE findSingleReferenceDeltaEntries(nt, rt, rnt)
163 FORALL _@t t
164 WHERE MatchingObjects LINKS t = t, nt = nt
165   AND t.eClass() = c
166   AND c.eAllReferences = r
167   AND r.changeable = true
168   AND r.many = false
169   AND t.eIsSet(r) = true
170   AND t.$r = rt
171   AND nt.$r = rnt
172   AND NOT MatchingObjects LINKS t = rt, nt = rnt
173   AND RuleLink LINKS name = "referenceChanged", action = act
174 MAKE SingleReferenceChangeDeltaEntry e FROM fsr(t, r)
175 SET e.reference = r,
176   e.rt = rt,
177   e.rnt = rnt,
178   e.action = act
179 LINKING DeltaEntry WITH t = t, nt = nt, entry = e
180   ;
```

Figure 3.14: Rule used to find changes in single-valued references.

Figure 3.15 shows the instances that are created by the transformation when dealing with the example in Figure 2.8.

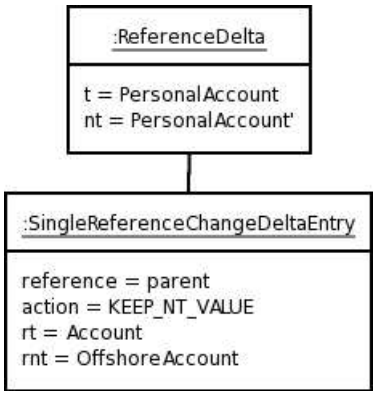


Figure 3.15: The result of running the delta transformation on the example shown in Figure 2.8.

The rules in Figures 3.16 and 3.17 are used to deal with the cases in which the single-valued references have been unset instead of being set to point to a

different object. The objects created by these rules are instances of SingleReferenceUnsetDeltaEntry.

```
182 RULE findSingleUnsetReferenceDeltaEntriesInT(nt, rnt)
183 FORALL _@t t
184 WHERE MatchingObjects LINKS t = t, nt = nt
185   AND t.eClass() = c
186   AND c.eAllReferences = r
187   AND r.changeable = true
188   AND r.many = false
189   AND t.eIsSet(r) = false
190   AND nt.eIsSet(r) = true
191   AND nt.$r = rnt
192   AND RuleLink LINKS name = "referenceChanged", action = act
193 MAKE SingleReferenceUnsetDeltaEntry e FROM fsurt(t, r)
194 SET e.reference = r,
195   e.t = rnt,
196   e.inT = true,
197   e.action = act
198 LINKING DeltaEntry WITH t = t, nt = nt, entry = e
199   ;
```

Figure 3.16: Rule used to find changes in single-valued references that have been unset in the original target model.

```
201 RULE findSingleUnsetReferenceDeltaEntriesInNT(nt, rt)
202 FORALL _@t t
203 WHERE MatchingObjects LINKS t = t, nt = nt
204   AND t.eClass() = c
205   AND c.eAllReferences = r
206   AND r.changeable = true
207   AND r.many = false
208   AND t.eIsSet(r) = true
209   AND t.$r = rt
210   AND nt.eIsSet(r) = false
211   AND RuleLink LINKS name = "referenceChanged", action = act
212 MAKE SingleReferenceUnsetDeltaEntry e FROM fsurnt(t, r)
213 SET e.reference = r,
214   e.t = rt,
215   e.inT = false,
216   e.action = act
217 LINKING DeltaEntry WITH t = t, nt = nt, entry = e
218   ;
```

Figure 3.17: Rule used to find changes in single-valued references that have been unset in the new target model.

Figure 3.18 shows the instances created by the transformation when dealing with the example shown in Figure 2.9. In this case there is only one reference to keep track of, because the other reference is unset. The *inT* attribute is used to indicate to which target model the reference that is set belongs to.

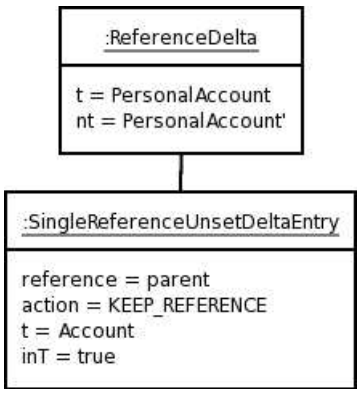


Figure 3.18: The result of running the delta transformation on the example shown in Figure 2.9, assuming the default action for this type of change is keeping the reference that is set.

Multi-Valued Unordered References

The rules used to find these changes must make use of the sets of objects that each reference points to. The rule in Figure 3.19 links these sets in the original target model and the one in Figure 3.20 links these sets in the new target model.

```
220 RULE linkUnorderedReferencesTarget(rt)
221 FORALL _@t t
222 WHERE t.eClass() = c
223   AND c.eAllReferences = r
224   AND r.changeable = true
225   AND r.ordered = false
226   AND t.eIsSet(r) = true
227   AND t.$r = rt
228 LINKING UnorderedRefT WITH t = t, rt = rt, ref = r
229   ;
```

Figure 3.19: Rule used to link the sets of objects pointed to by the multi-valued unordered references in the original target model.

```
231 RULE linkUnorderedReferencesNewTarget(rnt)
232 FORALL _@nt nt
233 WHERE nt.eClass() = c
234   AND c.eAllReferences = r
235   AND r.changeable = true
236   AND r.ordered = false
237   AND nt.eIsSet(r) = true
238   AND nt.$r = rnt
239 LINKING UnorderedRefNT WITH nt = nt, rnt = rnt, ref = r
240   ;
```

Figure 3.20: Rule used to link the sets of objects pointed to by the multi-valued unordered references in the new target model.

The objects that don't belong to both sets may or may not have a matching object. In the example shown in Figure 2.10, the set of objects referred to by the package in the original target model is {Account, OffshoreAccount, PersonalAccount}. The set in the new target model is {Account, LocalAccount, Customer}. Therefore, the objects that don't belong to both sets are OffshoreAccount, PersonalAccount, LocalAccount and Customer. From these objects only LocalAccount is unmatched. OffshoreAcount and PersonalAccount are both referred to in the original target model. Customer is referred to in the new target model.

In order to find all these changes two pairs of rules are needed. The first pair deals with the objects that are referred to in the original target model. The rule in Figure 3.21 deals with the case in which the objects are matched and the one in Figure 3.22 with the case in which the objects are unmatched. Notice that the only difference between them lies in the WHERE clause. In the first rule, the matching objects from the set of referred objects in the original target model are looked up in line 246 and then line 247 checks which ones don't belong to the set of referred objects in the new target model. In the second rule, line 261 just checks for objects in the set of referred elements in the original target model that don't have a matching object in the new target model. The result of running these rules on the example in Figure 2.10 is shown in Figure 3.23.

The second pair of rules is analogous to this one but deals with the objects referred to in the new target model. The rules are shown in Figure 3.24 and Figure 3.25. The result of running these rules on the example in Figure 2.10 is shown in Figure 3.26.

```
242 RULE findMultipleReferenceItemsFromT(nt, rt, r, act)
243 FORALL _@t t
244 WHERE MatchingObjects LINKS t = t, nt = nt
245   AND UnorderedRefT LINKS t = t, rt = rt, ref = r
246   AND (MatchingObjects LINKS t = rt, nt = rnt
247     AND NOT UnorderedRefNT LINKS nt = nt, rnt = rnt, ref = r)
248   AND RuleLink LINKS name = "singleReference", action = act
249 MAKE SingleReferenceItem i FROM fmr(t, rt)
250 SET i.t = rt,
251   i.inT = true,
252   i.action = act,
253   i.matched = true
254 LINKING SingleItem WITH t = t, item = i, ref = r
255   ;
```

Figure 3.21: Rule used to find references to matched objects that have been added to a multi-valued unordered reference in the original target model.

```
257 RULE findUnmatchedMultipleReferenceItemsFromT(nt, rt, rnt, r, act)
258 FORALL _@t t
259 WHERE MatchingObjects LINKS t = t, nt = nt
260   AND UnorderedRefT LINKS t = t, rt = rt, ref = r
261   AND NOT MatchingObjects LINKS t = rt, nt = rnt
262   AND RuleLink LINKS name = "singleReference", action = act
263 MAKE SingleReferenceItem i FROM fmr(t, rt)
264 SET i.t = rt,
265   i.inT = true,
266   i.action = act,
267   i.matched = false
268 LINKING SingleItem WITH t = t, item = i, ref = r
269   ;
```

Figure 3.22: Rule used to find references to unmatched objects that have been added to a multi-valued unordered reference in the original target model.

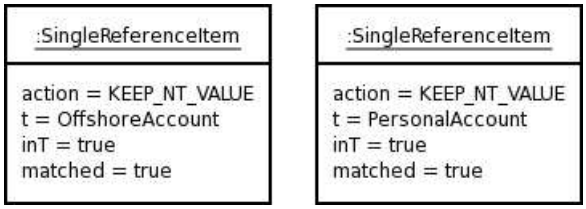


Figure 3.23: The result of running the first pair of rules on the example shown in Figure 2.10.

```
271 RULE findMultipleReferenceItemsFromNT(t, rnt, r, act)
272 FORALL _@nt nt
273 WHERE MatchingObjects LINKS t = t, t = nt
274   AND UnorderedRefNT LINKS nt = nt, rnt = rnt, ref = r
275   AND (MatchingObjects LINKS t = rt, nt = rnt
276     AND NOT UnorderedRefT LINKS t = t, rt = rt, ref = r)
277   AND RuleLink LINKS name = "singleReference", action = act
278 MAKE SingleReferenceItem i FROM fmr(nt, rnt)
279 SET i.t = rnt,
280   i.inT = false,
281   i.action = act,
282   i.matched = true
283 LINKING SingleItem WITH t = t, item = i, ref = r
284   ;
```

Figure 3.24: Rule used to find references to matched objects that have been added to a multi-valued unordered reference in the new target model.

```
286 RULE findUnmatchedMultipleReferenceItemsFromNT(t, rt, rnt, r, act)
287 FORALL _@nt nt
288 WHERE MatchingObjects LINKS t = t, nt = nt
289   AND UnorderedRefNT LINKS nt = nt, rnt = rnt, ref = r
290   AND NOT MatchingObjects LINKS t = rt, nt = rnt
291   AND RuleLink LINKS name = "singleReference", action = act
292 MAKE SingleReferenceItem i FROM fmr(nt, rnt)
293 SET i.t = rnt,
294   i.inT = false,
295   i.action = act,
296   i.matched = false
297 LINKING SingleItem WITH t = t, item = i, ref = r
298   ;
```

Figure 3.25: Rule used to find references to unmatched objects that have been added to a multi-valued unordered reference in the new target model.

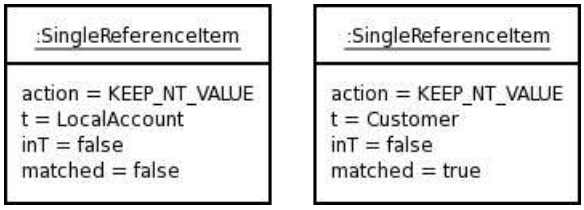


Figure 3.26: The result of running the second pair of rules on the example shown in Figure 2.10.

The rule in Figure 3.27 looks for the items created with the previous rules based on the referencing object and the reference, and groups them into a single entry. The final result of running the transformation on the example in Figure 2.10 is shown in Figure 3.28.

```
300 RULE findMultipleReferenceDeltaEntries(nt, i)
301 FORALL _@t t
302 WHERE MatchingObjects LINKS t = t, nt = nt
303   AND t.eClass() = c
304   AND c.eAllReferences = r
305   AND r.changeable = true
306   AND r.many = true
307   AND r.ordered = false
308   AND t.eIsSet(r) = true
309   AND RefLink LINKS src = t, tgt = i, ref = r, name = "singleItems"
310 MAKE MultipleReferenceDeltaEntry e FROM fsr(t, r)
311 SET e.reference = r,
312   e.items = i
313 LINKING DeltaEntry WITH t = t, nt = nt, entry = e
314   ;
```

Figure 3.27: Rule used to group unordered multi-valued reference changes.

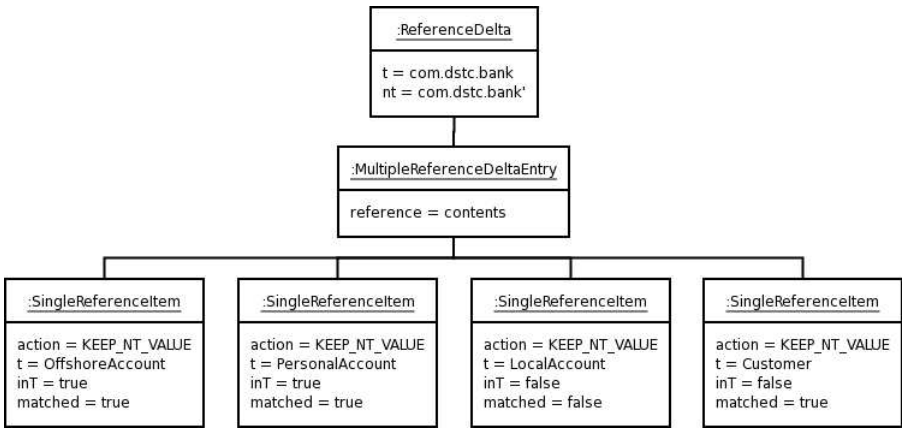


Figure 3.28: The result of running the delta transformation on the example shown in Figure 2.10.

Multi-Valued Ordered References

The rules used to find changes in ordered references must use the sets of objects that each reference points to. The rule in Figure 3.19 links these sets in the original target model and the one in Figure 3.20 links these sets in the new target model, just like in the unordered case. However, since the references are ordered, apart from linking the sets of objects, a relative order between the objects must also be established. In order to keep track of the order, instances of the RelativeOrderT and RelativeOrderNT classes must be created. These instances store the matching object that follows each object in a multi-valued ordered reference.

```
316 RULE linkOrderedReferencesTarget(rt)
317 FORALL _@t t
318 WHERE t.eClass() = c
319   AND c.eAllReferences = r
320   AND r.changeable = true
321   AND r.ordered = true
322   AND t.eIsSet(r) = true
323   AND t.$r = rt
324 LINKING OrderedRefsLinkT WITH t = t, rt = rt, ref = r
325   ;
```

Figure 3.29: Rule used to link the sets of objects pointed to by the multi-valued ordered references in the original target model.

```
327 RULE linkOrderedReferencesNewTarget(rnt)
328 FORALL _@nt nt
329 WHERE nt.eClass() = c
330   AND c.eAllReferences = r
331   AND r.changeable = true
332   AND r.ordered = true
333   AND nt.eIsSet(r) = true
334   AND nt.$r = rnt
335 LINKING OrderedRefsLinkNT WITH nt = nt, rnt = rnt, ref = r
336   ;
```

Figure 3.30: Rule used to link the sets of objects pointed to by the multi-valued ordered references in the new target model.

The patterns in Figure 3.31 are used to find this relative order. The isNextInT pattern receives a referring object (*refObj*), a reference (*ref*), a referred object (*obj*) and an object that is possibly the next matched object in the collection of ordered references (*next*) as parameters, all from the original target model. If the object is in fact the next matched object in the ordered collection then the pattern matches. Lines 346 and 347 check that both *obj* and *next* are referenced by *refObj*. Line 348 ensures that *next* is not an unmatched object. This is necessary because the relative order must not use unmatched objects as points of reference since they may not be present in the merged model. Lines 349 and 350 check that *obj* is before *next* and that there is no matching object between them. If this condition is met then *next* is in fact the next matching object that follows *obj*. A similar pattern, isNextInNT, is used to establish the relative order in the new target model. A dummy object must be linked after the last object in the multi-valued references, in order to indicate that it is the last element in the collection.

```
338 PATTERN existsMatchedBetweenInT(refObj, ref, obj, next)
339 FORALL _@t t
340 WHERE MatchingObjects LINKS t = t, nt = nt
341   AND obj BEFORE t IN refObj.$ref()
342   AND t BEFORE next IN refObj.$ref()
343   ;
345 PATTERN isNextInT(refObj, ref, obj, next)
346 WHERE refObj.$ref().indexOf(obj) != -1
347   AND refObj.$ref().indexOf(next) != -1
348   AND MatchingObjects LINKS t = next, nt = nt
349   AND obj BEFORE next IN refObj.$ref()
350   AND NOT existsMatchedBetweenInT(refObj, ref, obj, next)
351   ;
367 PATTERN existsMatchedBetweenInNT(refObj, ref, obj, next)
368 FORALL _@nt nt
369 WHERE MatchingObjects LINKS t = t, nt = nt
370   AND obj BEFORE nt IN refObj.$ref()
371   AND nt BEFORE next IN refObj.$ref()
372   ;
374 PATTERN isNextInNT(refObj, ref, obj, next)
375 WHERE refObj.$ref().indexOf(obj) != -1
376   AND refObj.$ref().indexOf(next) != -1
377   AND MatchingObjects LINKS src = t, tgt = next
378   AND obj BEFORE next IN refObj.$ref()
379   AND NOT existsMatchedBetweenInNT(refObj, ref, obj, next)
380   ;
```

Figure 3.31: Patterns used to establish the relative order in the multi-valued ordered references.

```
353 RULE linkRelativeOrderT(r, nt, rt, rnt)
354 FORALL _@t t, _@t next
355 WHERE t.eClass() = c
356   AND c.eAllReferences = r
357   AND r.changeable = true
358   AND r.many = true
359   AND r.ordered = true
360   AND t.eIsSet(r) = true
361   AND t.$r = rt
362   AND isNextInT(t, r, rt, next)
363 LINKING RelativeOrderT WITH obj = rt, next = next, referencingObject = t,
364   reference = r
365   ;
```

Figure 3.32: Rule used to link the the relative order of the multi-valued ordered references in the original target model.

```
382 RULE linkRelativeOrderNT(r, t, rt, rnt)
383 FORALL _@nt nt, _@nt next
384 WHERE nt.eClass() = c
385   AND c.eAllReferences = r
386   AND r.changeable = true
387   AND r.many = true
388   AND r.ordered = true
389   AND nt.eIsSet(r) = true
390   AND nt.$r = rnt
391   AND isNextInNT(nt, r, rnt, next)
392 LINKING RelativeOrderLinkNT WITH obj = rnt, next = next, referencingObject = nt,
393   reference = r
394   ;
```

Figure 3.33: Rule used to link the the relative order of the multi-valued ordered references in the new target model.

Once the relative order is linked, the rule in Figure 3.34 is used to find the matching objects that belong to both sets of references but that have a different relative order. Lines 397 and 398 select the matching objects and store them in the t and nt variables. Lines 399 to 403 use the relative order link to find the objects referenced by t and nt and stores them in the $objT$ and $objNT$ variables. It also finds the objects that follow $objT$ and $objNT$ in the multi-valued reference and stores them in the $nextT$ and $nextNT$ variables. Line 404 checks that $nextT$ and $nextNT$ are not matching objects, which means that a change in the relative order has occurred. The result of running these rules on the example in Figure 2.11 is shown in Figure 3.35.

```
396 RULE findOrderedReferenceItems(nt, r, objT, nextT, objNT, nextNT, act)
397 FORALL _@t t
398 WHERE MatchingObjects LINKS t = t, nt = nt
399   AND RelativeOrderLinkT LINKS obj = objT, next = nextT, referencingObject = t,
400     reference = r
401   AND RelativeOrderLinkNT LINKS obj = objNT, next = nextNT,
402     referencingObject = nt, reference = r
403   AND MatchingObjects LINKS t = objT, nt = objNT
404   AND NOT MatchingObjects LINKS t = nextT, nt = nextNT
405   AND RuleLink LINKS name = "referenceOrderChanged", action = act
406 MAKE OrderedReferencePairItem i FROM fori(t, objT, nt, objNT)
407 SET i.rt = objT,
408   i.rtAfter = nextT,
409   i.rnt = objNT,
410   i.rntAfter = nextNT,
411   i.action = act
412 LINKING OrderedItem WITH t = t, item = i, ref = r
413   ;
```

Figure 3.34: Rule used to find the changes in the order of the references in the multi-valued ordered references.

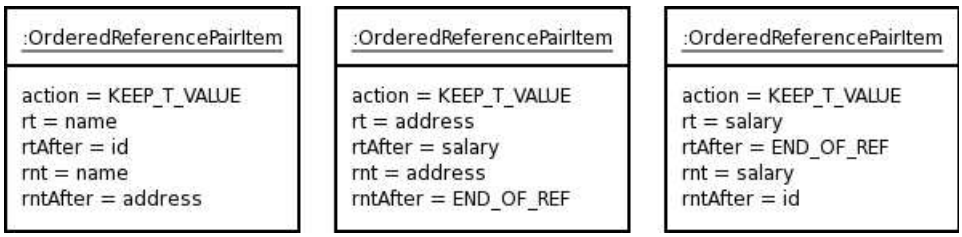


Figure 3.35: The result of running the findOrderedReferenceItems rule on the example shown in Figure 2.11.

A change in the order is not the only difference that may exist when comparing multi-valued ordered references. References to other objects may also have been added. The process of identifying these changes is similar to the one used for the multi-valued unordered references with the exception that, since in this case the order does matter, a reference to some object in the collection of references must be kept in order to know where to insert the new reference, if the user decides to keep it. Following the same pattern used for establishing the order, this reference should point to the object that goes immediately after the new referred object. In this case there are also two pairs of rules used to identify these changes, shown in Figures 3.36, 3.37, 3.38 and 3.39. The only difference with the rules used to find the changes in the unordered references is that these ones create instances of the SingleReferenceOrderedItem and set the *rtAfter* variable by using the previously established relative order. The result of running these rules on the example in Figure 2.11 is shown in Figure 3.40.

The rule in Figure 3.41 looks for the items created with the previous rules based on the referencing object and the reference, and groups them into a single entry. The final result of running the transformation on the example in Figure 2.11 is shown in Figure 3.28.

The last step in finding the delta for the references is to group all the entries into a single ReferenceDelta object. The rule in Figure 3.43 does this by using the information linked in the previous rules when creating the entries. Line 526 selects all the objects in the original target model. Line 527 uses the DeltaEntry objects, which relate the entries with their corresponding pair of matching objects, to find the pairs of matching objects with changes in their references. Lines 528 to 531 create a ReferenceDelta instance for each one of these pairs and add the corresponding entries created previously.

```
415 RULE findMatchedOrderedReferenceItemsFromT(nt, rt, rtAfter, rnt, act)
416 FORALL _@t t
417 WHERE MatchingObjects LINKS t = t, nt = nt
418   AND t.eClass() = c
419   AND c.eAllReferences = r
420   AND r.changeable = true
421   AND r.many = true
422   AND r.ordered = true
423   AND t.eIsSet(r) = true
424   AND t.$r = rt
425   AND (MatchingObjects LINKS t = rt, nt = rnt
426     AND NOT OrderedRefNT LINKS nt = nt, rnt = rnt, ref = r)
427   AND RelativeOrderLinkT LINKS obj = rt, next = rtAfter,
428     referencingObject = t, reference = r
429   AND RuleLink LINKS name = "singleReference", action = act
430 MAKE SingleReferenceOrderedItem i FROM fmr(t, rt)
431 SET i.t = rt,
432   i.inT = true,
433   i.action = act,
434   i.matched = true,
435   i.rAfter = rtAfter
436 LINKING OrderedItem WITH t = t, item = i, ref = r
437   ;
```

Figure 3.36: Rule used to find references to matched objects that have been added to a multi-valued ordered reference in the original target model.

```
439 RULE findUnmatchedOrderedReferenceItemsFromT(nt, rt, rtAfter, rnt, act)
440 FORALL _@t t
441 WHERE MatchingObjects LINKS t = t, nt = nt
442   AND t.eClass() = c
443   AND c.eAllReferences = r
444   AND r.changeable = true
445   AND r.many = true
446   AND r.ordered = true
447   AND t.eIsSet(r) = true
448   AND t.$r = rt
449   AND NOT MatchingObjects LINKS t = rt, nt = rnt
450   AND RelativeOrderLinkT LINKS obj = rt, next = rtAfter,
451     referencingObject = t, reference = r
452   AND RuleLink LINKS name = "singleReference", action = act
453 MAKE SingleReferenceItem i FROM fmr(t, rt)
454 SET i.t = rt,
455   i.inT = true,
456   i.action = act,
457   i.matched = false,
458   i.rAfter = rtAfter
459 LINKING OrderedItem WITH t = t, item = i, ref = r
460   ;
```

Figure 3.37: Rule used to find references to unmatched objects that have been added to a multi-valued ordered reference in the original target model.

```
462 RULE findMatchedOrderedReferenceItemsFromNT(nt, rt, rnt, rntAfter, act)
463 FORALL _@nt nt
464 WHERE MatchingObjects LINKS t = t, nt = nt
465   AND nt.eClass() = c
466   AND c.eAllReferences = r
467   AND r.changeable = true
468   AND r.many = true
469   AND r.ordered = true
470   AND nt.eIsSet(r) = true
471   AND nt.$r = rnt
472   AND (MatchingObjects LINKS t = rt, nt = rnt
473     AND NOT OrderedRefT LINKS t = t, rt = rt, ref = r)
474   AND RelativeOrderLinkNT LINKS obj = rnt, next = rntAfter,
475     referencingObject = t, reference = r
476   AND RuleLink LINKS name = "singleReference", action = act
477 MAKE SingleOrderedReferenceItem i FROM fmr(t, rt)
478 SET i.t = rnt,
479   i.inT = false,
480   i.action = act,
481   i.matched = true,
482   i.rAfter = rntAfter
483 LINKING OrderedItem WITH t = t, item = i, ref = r
484   ;
```

Figure 3.38: Rule used to find references to matched objects that have been added to a multi-valued ordered reference in the new target model.

```
486 RULE findUnmatchedOrderedReferenceItemsFromNT(nt, rt, rnt, rntAfter, act)
487 FORALL _@nt nt
488 WHERE MatchingObjects LINKS t = t, nt = nt
489   AND nt.eClass() = c
490   AND c.eAllReferences = r
491   AND r.changeable = true
492   AND r.many = true
493   AND r.ordered = true
494   AND nt.eIsSet(r) = true
495   AND nt.$r = rnt
496   AND NOT MatchingObjects LINKS t = rt, nt = rnt
497   AND RelativeOrderLinkNT LINKS obj = rnt, next = rntAfter,
498     referencingObject = t, reference = r
499   AND RuleLink LINKS name = "singleReference", action = act
500 MAKE SingleOrderedReferenceItem i FROM fmr(nt, rnt)
501 SET i.t = rnt,
502   i.inT = false,
503   i.action = act,
504   i.matched = false,
505   i.rAfter = rntAfter
506 LINKING OrderedItem WITH t = t, item = i, ref = r
507   ;
```

Figure 3.39: Rule used to find references to unmatched objects that have been added to a multi-valued ordered reference in the new target model.

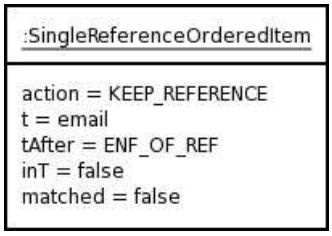


Figure 3.40: The result of running both pairs of rules on the example shown in Figure 2.11.

```
509 RULE findOrderedReferenceDeltaEntries(nt, i)
510 FORALL _@t t
511 WHERE MatchingObjects LINKS t = t, nt = nt
512   AND t.eClass() = c
513   AND c.eAllReferences = r
514   AND r.changeable = true
515   AND r.many = true
516   AND r.ordered = true
517   AND t.eIsSet(r) = true
518   AND OrderedItem LINKS t = t, item = i, ref = r
519 MAKE MultipleOrderedReferenceDeltaEntry e FROM forde(t, r)
520 SET e.reference = r,
521   e.items = i
522 LINKING DeltaEntry WITH t = t, nt = nt, entry = e
523   ;
```

Figure 3.41: Rule used to group ordered multi-valued reference changes.

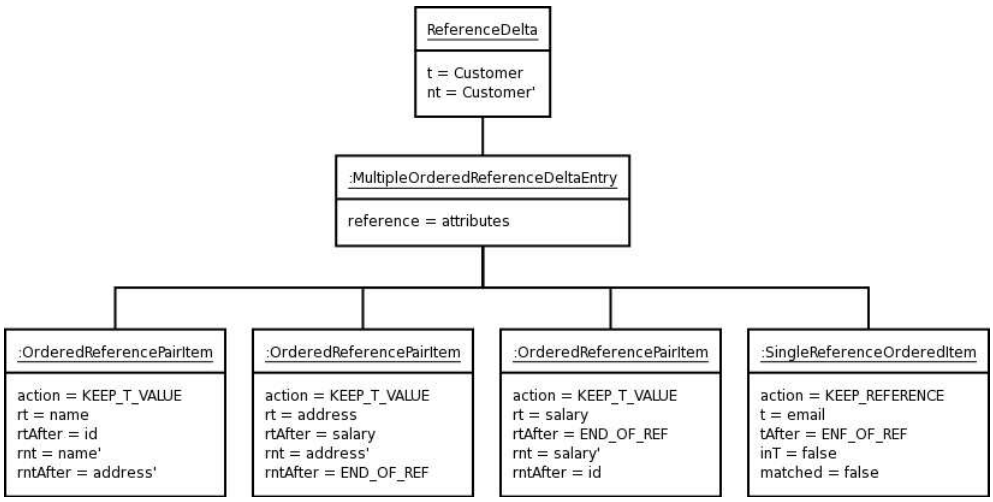


Figure 3.42: The result of running the delta transformation on the example shown in Figure 2.11.

```
525 RULE findReferenceDelta(e, nt)
526 FORALL _@t t
527 WHERE DeltaEntry LINKS t = t, nt = nt, entry = e
528 MAKE ReferenceDelta rd
529 SET rd.t = t,
530   rd.nt = nt,
531   rd.entries = e
532   ;
```

Figure 3.43: Rule used to group all the entries into a single ReferenceDelta object.

Chapter 4

Model Merging

Now that the delta between the models has been calculated and the information from the user collected, it is possible to merge the models. Recall that the merge process can be handled in several ways. The merging can be done in place in the new target model by modifying it, or a new model can be generated and the all the necessary objects from the target models copied into it. In order to use a Tefkat model transformation to handle the merge process, only the second approach can be used since Tefkat cannot modify the models it operates on. Merging the models can be considered a model transformation that receives both target models and the delta model as inputs and generates the merged model as an output.

4.1 The Merging Process

The following sections explain the rules used to merge the target models.

4.1.1 Matched Objects

The objects that were identified as matching in the change detection process should be copied into the merged model. The rule in Figure 4.1 iterates through all the objects in the original target model and the new target model and uses the pattern shown in Figure 4.2 to determine which ones are matching (lines 29 and 30). Then, in line 31, it copies each matched object from the new target model into the merged model. It is irrelevant from which target model the objects are copied since they are equivalent. Notice that the object's attributes and references are not copied. Finally, in line 32, the newly created object is linked with its matching objects in the original target model and the new target model.

```
28 RULE copyMatchedObjects(t, nt, mt)
29 FORALL _@t t, _@nt nt
30 WHERE areMatched(t, nt)
31 MAKE EXACT $nt.eClass() mt FROM cmo(nt)
32 LINKING MatchingObjects WITH t = t, nt = nt, mt = mt
33   ;
```

Figure 4.1: Rule used to copy the matched objects from the target models into the merged model.

```
10 PATTERN areMatched(T, NT)
11 FORALL MatchedObjects@delta MO
12 WHERE T = MO.t
13   AND NT = MO.nt
14   ;
```

Figure 4.2: Pattern used to determine if two objects are matched.

The new objects that have been created still have their attributes and references unset. The rule in Figure 4.3 copies the attribute values that haven't changed from the new target model. In this case it is also irrelevant from which target model they are copied since the values are the same. In order to determine which attributes haven't changed, the pattern in Figure 4.6 is used.

```
 95 RULE copyUnchangedAttributes(t, nt, mt, a)
 96 WHERE MatchingObjects LINKS nt = nt, t = t, mt = mt
 97   AND nt.eClass() = c
 98   AND c.eAllAttributes = a
 99   AND a.changeable = true
100   AND NOT attributeChanged(nt, a)
101 SET mt.$a = nt.$a
102   ;
```

Figure 4.3: Rule used to copy the matching object's attribute values that haven't changed.

```
16 PATTERN attributeChanged(S, A)
17 FORALL AttributeDeltaEntry@delta att
18 WHERE S = att.owner.nt
19   AND A = att.attribute
20   ;
```

Figure 4.4: Pattern used to determine if a matching object's attribute values have changed.

The rule in Figure 4.5 copies the references that haven't changed in the same way by using the pattern in Figure 4.6.

```
126 RULE copyUnchangedReferences(t, nt, mt, r, rt, rnt, rmt)
127 WHERE MatchingObjects LINKS t = t, nt = nt, mt = mt
128   AND nt.eClass() = c
129   AND c.eAllReferences = r
130   AND r.changeable = true
131   AND rnt = nt.$r
132   AND MatchingObjects LINKS t = rt, nt = rnt, mt = rmt
133   AND NOT referenceChanged(nt, r)
134 SET mt.$r = rmt
135   ;
```

Figure 4.5: Rule used to copy the matching object's references that haven't changed.

```
22 PATTERN referenceChanged(NT, R)
23 FORALL ReferenceDeltaEntry@delta ref
24 WHERE NT = ref.owner.nt
25   AND R = ref.reference
26   ;
```

Figure 4.6: Pattern used to determine if a matching object's references have changed.

4.1.2 Unmatched Objects

The unmatched objects in both target models that have been selected to be kept in the merged model must be copied. The rule in Figure 4.7 is used to copy the ones in the original target model. Line 36 selects all the UnmatchedObjects instances from the delta model. Lines 37 to 39 select only the ones in the original target model. Recall from section 2.2.1 that the unmatched objects that belong to the original target model are the ones that were added manually to it, the ones that remain from an object that existed previously in the source model and has now been deleted, and all other objects in it that have no match and whose cause for being unmatched is unknown.

```
35 RULE copyUnmatchedObjectsT(t, act, kind)
36 FORALL UnmatchedObject@delta umo
37 WHERE (umo.kind = DELTA_KIND.OBJECT_ADDED_TO_TARGET
38     OR umo.kind = DELTA_KIND.OBJECT_DELETED_FROM_SOURCE
39     OR umo.kind = DELTA_KIND.OTHER_OBJECT_IN_TARGET)
40   AND umo.action = MERGE_ACTION.CREATE_OBJECT
41   AND t = umo.ref
42 MAKE $t.eClass() mt
43 LINKING UnmatchedObjectsTToMT WITH t = t, mt = mt
44   ;
```

Figure 4.7: Rule used to copy the unmatched objects from the original target model into the merged model.

The rule in Figure 4.8 copies the object's attribute values. There can't be any

change associated with these since the objects only exist in the original target model. Changes in attribute values are only reported in matched objects, when the values in the pair differ. The unmatched objects in the original target model are selected in line 58 from the linking objects created previously and assigned to *t*. The corresponding objects created in the merged model are assigned to *mt*. Lines 59 to 61 select all their attributes and line 62 copies their values in *t* to *mt*.

```
57 RULE copyUnmatchedObjectAttributesT(t, mt, c, a)
58 WHERE UnmatchedObjectsTToMT LINKS t = t, mt = mt
59   AND t.eClass() = c
60   AND c.eAllAttributes = a
61   AND a.changeable = true
62 SET mt.$a = t.$a
63   ;
```

Figure 4.8: Rule used to set the attribute values of the objects created in the merged model from the unmatched objects in the original target model.

The references are copied in a similar way in the rule shown in Figure 4.9. References in an unmatched object may point to a matched object or to an unmatched object in the same target model. Lines 78 and 79 deal with these two cases. If an unmatched object is not copied to the merge model and is referred to by another unmatched object that does get copied then the reference will not be added because the referred object will not exist in the merged model.

```
73 RULE copyUnmatchedObjectReferencesT(t, mt, rt, rnt, rmt, c, r)
74 WHERE UnmatchedObjectsTToMT LINKS t = t, mt = mt
75   AND t.eClass() = c
76   AND c.eAllReferences = r
77   AND r.changeable = true
78   AND rt = t.$r
79   AND (MatchingObjects LINKS t = rt, nt = rnt, mt = rmt
80     OR UnmatchedObjectsTToMT LINKS t = rt, mt = rmt)
81 SET mt.$r = rmt
82   ;
```

Figure 4.9: Rule used to set the references of the objects created in the merged model from the unmatched objects in the original target model.

The rules in Figures 4.10, 4.11, 4.12 deal with the unmatched objects in the new target model, in the same way that the previous rules did with the objects in the original target model.

```
46 RULE copyUnmatchedObjectsNT(nt, act, kind)
47 FORALL UnmatchedObject@delta umo
48 WHERE (umo.kind = DELTA_KIND.OBJECT_DELETED_FROM_TARGET
49     OR umo.kind = DELTA_KIND.OBJECT_ADDED_TO_SOURCE
50     OR umo.kind = DELTA_KIND.OTHER_OBJECT_IN_NEW_TARGET)
51   AND umo.action = MERGE_ACTION.CREATE_OBJECT
52   AND nt = umo.ref
53 MAKE EXACT $nt.eClass() mt
54 LINKING UnmatchedObjectsNTToMT WITH nt = nt, mt = mt
55   ;
```

Figure 4.10: Rule used to copy the unmatched objects from the new target model into the merged model.

```
65 RULE copyUnmatchedObjectAttributesNT(nt, mt, c, a)
66 WHERE UnmatchedObjectsNTToMT LINKS nt = nt, mt = mt
67   AND nt.eClass() = c
68   AND c.eAllAttributes = a
69   AND a.changeable = true
70 SET mt.$a = nt.$a
71   ;
```

Figure 4.11: Rule used to set the attribute values of the objects created in the merged model from the unmatched objects in the new target model.

```
84 RULE copyUnmatchedObjectReferencesNT(nt, mt, rt, rnt, rmt, c, r)
85 WHERE UnmatchedObjectsNTToMT LINKS nt = nt, mt = mt
86   AND nt.eClass() = c
87   AND c.eAllReferences = r
88   AND r.changeable = true
89   AND rnt = nt.$r
90   AND (MatchingObjects LINKS t = rt, nt = rnt, mt = rmt
91     OR UnmatchedObjectsNTToMT LINKS nt = rnt, mt = rmt)
92 SET mt.$r = rmt
93   ;
```

Figure 4.12: Rule used to set the references of the objects created in the merged model from the unmatched objects in the new target model.

4.1.3 Attribute Changes

The rule in Figure 4.13 is used to set the matched object's attributes that have changed and must keep the value in the original target model. Line 105 selects all the AttributeDelta instances in the delta model. Lines 106 and 107 assign the pair of matching objects referred to by the changes to the *t* and *nt* variables. The entries for each pair of objects are assigned to the *ent* variable in line 108. Line 109 assigns each entry's attribute to the *a* variable. Line 110 filters the objects to include only the ones that must keep the value in the original target model. Line 111 finds the object in the merged model that was created from

the pair of matching objects and assigns it to the *mt* variable. Line 112 sets *mt*'s attribute value to the corresponding object's attribute value in the original target model.

```
104 RULE copyKeepAttributesT(t, nt, ent, a, act, mt)
105 FORALL AttributeDelta@delta att
106 WHERE att.t = t
107   AND att.nt = nt
108   AND att.entries = ent
109   AND ent.attribute = a
110   AND ent.action = MERGE_ACTION.KEEP_T_VALUE
111   AND MatchingObjects LINKS t = t, nt = nt, mt = mt
112 SET mt.$a = t.$a
113   ;
```

Figure 4.13: Rule used to set the matched object's attributes that have changed and must keep the value in the original target model.

The rule in Figure 4.14 is used to set the matched object's attributes that have changed and must keep the value in the new target model. It works in the same way as the previous rule.

```
115 RULE copyKeepAttributesNT(t, nt, ent, a, act, mt)
116 FORALL AttributeDelta@delta att
117 WHERE att.t = t
118   AND att.nt = nt
119   AND att.entries = ent
120   AND ent.attribute = a
121   AND ent.action = MERGE_ACTION.KEEP_NT_VALUE
122   AND MatchingObjects LINKS t = t, nt = nt, mt = mt
123 SET mt.$a = nt.$a
124   ;
```

Figure 4.14: Rule used to set the matched object's attributes that have changed and must keep the value in the new target model.

4.1.4 Reference Changes

Several rules are used to merge the references that have changed according to the type of reference.

Single-Valued Reference Changes

The rules in Figures 4.15 and 4.16 are used to copy the single-valued references that have changed and are now pointing to different objects. They work in the same way as the rules used to merge the changes in the attributes except that the referred object has to be resolved in the merged model. The same problem that was found when copying the unmatched object's references is found here.

An inconsistent delta model may indicate that a reference to an object that wasn't created in the merged model must be kept.

```
137 RULE copyKeepTSingleChangeReferences(t, nt, mt, r, rt, rnt, rmt)
138 FORALL SingleReferenceChangeDeltaEntry@delta ref
139 WHERE ref.action = MERGE_ACTION.KEEP_T_VALUE
140   AND MatchingObjects LINKS t = ref.owner.t, nt = ref.owner.nt, mt = mt
141   AND r = ref.reference
142   AND (MatchingObjects LINKS t = ref.rt, nt = rnt, mt = mrt
143     OR UnmatchedObjectsTToMT LINKS t = ref.rt, mt = mrt)
144 SET mt.$r = rmt
145   ;
```

Figure 4.15: Rule used to set the matched object's single-valued references that have changed and must keep the value in the original target model.

```
147 RULE copyKeepNTSingleChangeReferences(t, nt, mt, r, rt, rnt, rmt, en)
148 FORALL SingleReferenceChangeDeltaEntry@delta ref
149 WHERE ref.action = MERGE_ACTION.KEEP_NT_VALUE
150   AND MatchingObjects LINKS t = ref.owner.t, nt = ref.owner.nt, mt = mt
151   AND r = ref.reference
152   AND (MatchingObjects LINKS t = rt, nt = ref.rnt, mt = rmt
153     OR UnmatchedObjectsNTToMT LINKS nt = ref.rnt, mt = rmt)
154 SET mt.$r = rmt
155   ;
```

Figure 4.16: Rule used to set the matched object's single-valued references that have changed and must keep the value in the new target model.

The rules in Figures 4.17 and 4.18 are used to copy the single-valued references that have changed and are now unset in one of the target models.

```
157 RULE copyKeepTSingleUnsetReferences(t, nt, mt, r, rt, rnt, rmt, en)
158 FORALL SingleReferenceUnsetDeltaEntry@delta ref
159 WHERE ref.action = MERGE_ACTION.KEEP_T_VALUE
160   AND ref.inT = true
161   AND MatchingObjects LINKS t = ref.owner.t, nt = ref.owner.nt, mt = mt
162   AND r = ref.reference
163   AND MatchingObjects LINKS t = ref.t, nt = rnt, mt = rmt
164 SET mt.$r = rmt
165   ;
```

Figure 4.17: Rule used to set the matched object's single-valued references that were unset in the new target model and must be set to the value in the new target model.

```
167 RULE copyKeepNTSingleUnsetReferences(t, nt, mt, r, rt, rnt, rmt, en)
168 FORALL SingleReferenceUnsetDeltaEntry@delta ref
169 WHERE ref.action = MERGE_ACTION.KEEP_NT_VALUE
170   AND ref.inT = false
171   AND MatchingObjects LINKS t = ref.owner.t, nt = ref.owner.nt, mt = mt
172   AND r = ref.reference
173   AND MatchingObjects LINKS t = rt, nt = ref.t, mt = rmt
174 SET mt.$r = rmt
175   ;
```

Figure 4.18: Rule used to set the matched object's single-valued references that were unset in the new target model and must be set to the value in the new target model.

Multi-Valued Unordered Reference Changes

Two pairs of rules are needed in order to merge these changes. The first pair copies the multi-valued unordered references in the original target model that are reported as a change and have an associated action that indicates that they must be kept in the merged model. The rule in Figure 4.19 deals with the references that are matched and the one in Figure 4.20 with the ones that are unmatched. Notice that the first one uses the matching objects to find the reference that must be set (line 186) while the second one uses the unmatched objects that have been created in the merge model (line 212).

```
177 RULE copyKeepTUnorderedMatchedReferences(items, r, t, nt, mt, rt, rnt, rmt)
178 FORALL MultipleReferenceDeltaEntry@delta ref
179 WHERE r = ref.reference
180   AND MatchingObjects LINKS t = ref.owner.t, nt = ref.owner.nt, mt = mt
181   AND items = ref.items
182   AND items.eClass().name = "SingleReferenceItem"
183   AND items.action = MERGE_ACTION.KEEP_REFERENCE
184   AND items.inT = true
185   AND items.matched = true
186   AND MatchingObjects LINKS t = items.t, nt = rnt, mt = rmt
187 SET mt.$r = rmt
188   ;
```

Figure 4.19: Rule used to copy the multiplicity many unordered references in the original target model that are reported as a change, point to an object that has a match in the new target model, and must be kept in the merged model.

```
203 RULE copyKeepTUnorderedUnmatchedReferences(items, r, t, nt, mt, rt, rnt, rmt)
204 FORALL MultipleReferenceDeltaEntry@delta ref
205 WHERE r = ref.reference
206   AND MatchingObjects LINKS t = ref.owner.t, nt = ref.owner.nt, mt = mt
207   AND items = ref.items
208   AND items.eClass().name = "SingleReferenceItem"
209   AND items.action = MERGE_ACTION.KEEP_REFERENCE
210   AND items.inT = true
211   AND items.matched = false
212   AND UnmatchedObjectsTToMT LINKS t = items.t, mt = rmt
213 SET mt.$r = rmt
214   ;
```

Figure 4.20: Rule used to copy the multiplicity many unordered references in the original target model that are reported as a change, point to an object that doesn't have a match in the new target model, and must be kept in the merged model.

The second pair, shown in Figures 4.21 and 4.22, copies the references in the new target model.

```
190 RULE copyKeepNTUnorderedMatchedReferences(items, r, t, nt, mt, rt, rnt, rmt)
191 FORALL MultipleReferenceDeltaEntry@delta ref
192 WHERE r = ref.reference
193   AND MatchingObjects LINKS t = ref.owner.t, nt = ref.owner.nt, mt = mt
194   AND items = ref.items
195   AND items.eClass().name = "SingleReferenceItem"
196   AND items.action = MERGE_ACTION.KEEP_REFERENCE
197   AND items.inT = false
198   AND items.matched = true
199   AND MatchingObjects LINKS t = rt, nt = items.t, mt = rmt
200 SET mt.$r = rmt
201   ;
```

Figure 4.21: Rule used to copy the multiplicity many unordered references in the new target model that are reported as a change, point to an object that has a match in the original target model, and must be kept in the merged model.

```
216 RULE copyKeepNTUnorderedUnmatchedReferences(items, r, t, nt, mt, rt, rnt, rmt)
217 FORALL MultipleReferenceDeltaEntry@delta ref
218 WHERE r = ref.reference
219   AND MatchingObjects LINKS t = ref.owner.t, nt = ref.owner.nt, mt = mt
220   AND items = ref.items
221   AND items.eClass().name = "SingleReferenceItem"
222   AND items.action = MERGE_ACTION.KEEP_REFERENCE
223   AND items.inT = false
224   AND items.matched = false
225   AND UnmatchedObjectsNTToMT LINKS t = items.t, mt = rmt
226 SET mt.$r = rmt
227   ;
```

Figure 4.22: Rule used to copy the multiplicity many unordered references in the new target model that are reported as a change, point to an object that doesn't have a match in the original target model, and must be kept in the merged model.

Multi-Valued Ordered Reference Changes

Merging the ordered references is a three step process. First, the references that each pair of objects have in common must be copied into the merged model, even if there is a change reported in their order. Then, according to the changes in their order, they must be reorganized so they reflect the order selected by the user. Finally, the references that were added in either target model must be copied into the merged model in the correct relative position. Note however that a reference that was added to the set of references in the original target model may have the same relative position as a reference that was added to the set of references in the new target model. In this case there is no way to know which one must go first, since their relative position is given with respect to the same reference in the set. By default the references that come from the original delta model will be copied first and therefore will appear first in the collection of references.

The rule in Figure 4.23 copies the multi-valued ordered references that each pair of objects have in common. The pattern in Figure 4.24 is used to filter the references that appear only in the original target model, since they will be copied by another rule and only if their associated action indicates that they must be kept.

```
238 RULE copyOrderedReferences(t, nt, mt, r, rt, rnt, rmt)
239 WHERE MatchingObjects LINKS t = t, nt = nt, mt = mt
240   AND t.eClass() = c
241   AND c.eAllReferences = r
242   AND r.many = true
243   AND r.ordered = true
244   AND r.changeable = true
245   AND rt = t.$r
246   AND NOT isSingleOrderedRef(r, t, rt)
247   AND MatchingObjects LINKS t = rt, nt = rnt, mt = rmt
248 SET mt.$r = rmt
249   ;
```

Figure 4.23: Rule used to copy the multiplicity many ordered references that the pairs of matching objects have in common.

```
229 PATTERN isSingleOrderedRefT(R, T, RT)
230 FORALL SingleReferenceOrderedItem@delta item
231 WHERE item.inT = true
232   AND item.t = RT
233   AND entry = item.owner
234   AND entry.references = R
235   AND entry.owner = T
236   ;
```

Figure 4.24: Pattern used to determine if a reference that is part of a multiplicity many ordered reference in the original target model doesn't have a corresponding reference in the new target model.

The rule in Figure 4.25 is used to sort the multi-valued ordered references that must keep the relative order in the new target model. Since the references are copied from the original target model then their relative order is the same as the one in the original target model. Therefore, only the changes that indicate that the order in the new target model must be kept have to be considered.

```
251 RULE sortKeepNTOrder(items, r, rt, rtAfter, rnt, rntAfter, rmt, rmtAfter)
252 FORALL MultipleOrderedReferenceDeltaEntry@delta ref
253 WHERE r = ref.reference
254   AND MatchingObjects LINKS t = ref.owner.t, nt = ref.owner.nt, mt = mt
255   AND items = ref.items
256   AND items.eClass().name = "OrderedReferencePairItem"
257   AND items.action = MERGE_ACTION.KEEP_NT_ORDER
258   AND rnt = items.rnt
259   AND rntAfter = items.rntAfter
260   AND MatchingObjects LINKS t = rt, nt = rnt, mt = rmt
261   AND MatchingObjects LINKS t = rtAfter, nt = rntAfter, mt = rmtAfter
262 SET rmt BEFORE rmtAfter IN mt.$r()
263   ;
```

Figure 4.25: Rule used to sort the multiplicity many ordered references that must keep the relative order in the new target model.

The last step in the process is accomplished by two pairs of rules. The first pair copies the references that point to matched objects that were added in either model. The rule in Figure 4.26 copies the ones added in the original target model and the one in Figure 4.27 copies the ones added in the new target model.

```
RULE copyKeepTOrderedMatchedReferences(items, r, t, nt, ntAfter, mt, rt, rnt,
                                       rmt, rmtAfter)
FORALL MultipleOrderedReferenceDeltaEntry@delta ref
WHERE r = ref.reference
  AND MatchingObjects LINKS t = ref.owner.t, nt = ref.owner.nt, mt = mt
  AND items = ref.items
  AND items.eClass().name = "SingleOrderedReferenceItem"
  AND items.action = MERGE_ACTION.KEEP_REFERENCE
  AND items.inT = true
  AND items.matched = true
  AND MatchingObjects LINKS t = items.t, nt = rnt, mt = rmt
  AND MatchingObjects LINKS t = items.tAfter, nt = ntAfter, mt = rmtAfter
SET mt.$r = rmt,
  rmt BEFORE rmtAfter IN mt.$r
  ;
```

Figure 4.26: Rule used to copy the multiplicity many ordered references in the original target model that are reported as a change, point to an object that has a match in the new target model, and must be kept in the merged model.

```
RULE copyKeepNTOrderedMatchedReferences(items, r, t, tAfter, nt, mt, rt, rnt,
                                        rmt, rmtAfter)
FORALL MultipleOrderedReferenceDeltaEntry@delta ref
WHERE r = ref.reference
  AND MatchingObjects LINKS t = ref.owner.t, nt = ref.owner.nt, mt = mt
  AND items = ref.items
  AND items.eClass().name = "SingleOrderedReferenceItem"
  AND items.action = MERGE_ACTION.KEEP_REFERENCE
  AND items.inT = false
  AND items.matched = true
  AND MatchingObjects LINKS t = rt, nt = items.t, mt = rmt
  AND MatchingObjects LINKS t = tAfter, nt = items.tAfter, mt = rmtAfter
SET mt.$r = rmt,
  rmt BEFORE rmtAfter IN mt.$r
  ;
```

Figure 4.27: Rule used to copy the multiplicity many ordered references in the new target model that are reported as a change, point to an object that has a match in the original target model, and must be kept in the merged model.

The second pair copies the references that point to unmatched objects that were added in either model. The rule in Figure 4.28 copies the ones added in the original target model and the one in Figure 4.29 copies the ones added in the new target model.

```
297 RULE copyKeepTOrderedUnmatchedReferences(items, r, t, nt, ntAfter, mt, rt, rnt,
298                                          rmt, rmtAfter)
299 FORALL MultipleOrderedReferenceDeltaEntry@delta ref
300 WHERE r = ref.reference
301   AND MatchingObjects LINKS t = ref.owner.t, nt = ref.owner.nt, mt = mt
302   AND items = ref.items
303   AND items.eClass().name = "SingleOrderedReferenceItem"
304   AND items.action = MERGE_ACTION.KEEP_REFERENCE
305   AND items.inT = true
306   AND items.matched = false
307   AND UnmatchedObjectsTToMT LINKS t = items.t, mt = rmt
308   AND MatchingObjects LINKS t = items.tAfter, nt = ntAfter, mt = rmtAfter
309 SET mt.$r = rmt,
310   rmt BEFORE rmtAfter IN mt.$r
311   ;
```

Figure 4.28: Rule used to copy the multiplicity many ordered references in the original target model that are reported as a change, point to an object that doesn't have a match in the new target model, and must be kept in the merged model.

```
313 RULE copyKeepNTOrderedUnmatchedReferences(items, r, t, tAfter, nt, mt, rt, rnt,
314                                          rmt, rmtAfter)
315 FORALL MultipleOrderedReferenceDeltaEntry@delta ref
316 WHERE r = ref.reference
317   AND MatchingObjects LINKS t = ref.owner.t, nt = ref.owner.nt, mt = mt
318   AND items = ref.items
319   AND items.eClass().name = "SingleOrderedReferenceItem"
320   AND items.action = MERGE_ACTION.KEEP_REFERENCE
321   AND items.inT = false
322   AND items.matched = false
323   AND UnmatchedObjectsNTToMT LINKS t = items.t, mt = rmt
324   AND MatchingObjects LINKS t = tAfter, nt = items.tAfter, mt = rmtAfter
325 SET mt.$r = rmt,
326   rmt BEFORE rmtAfter IN mt.$r
327   ;
```

Figure 4.29: Rule used to copy the multiplicity many ordered references in the new target model that are reported as a change, point to an object that doesn't have a match in the original target model, and must be kept in the merged model.

Chapter 5

Conclusion

5.1 Summary of Work

The goal of this thesis was to investigate a model merging approach to the problem of change propagation in the context of model to model transformations and to provide a solution for the DSTC's transformation engine, Tefkat. The first part of the project involved researching model transformation concepts, looking into the design of Tefkat's QVT language, and studying the relevant implementation details. Then, merging techniques used in other domains were investigated and this survey was used to classify the type of merge needed in this particular case. The key steps in the process of merging models were also identified.

Based on the information gathered during the research phase a strategy to merge the models was created. This strategy deals with state-based merging in which only the two models being merged are available. Different alternatives for implementing the strategy were analyzed and a model transformation approach was chosen, in order to take advantage of all the features provided by the QVT language. Also, a strategy to gather the user input needed to guide the merge process based on simple rules was implemented.

The whole process involves three major steps: finding the delta between the models, gathering input from the user to guide the merge process, and merging the models. In order to represent the delta between the models a specific meta-model was designed. The implementation was packaged as an Eclipse plugin that exposes an API that Tefkat can use to provide the merging functionality.

The thesis has fulfilled its goal of investigating and implementing a strategy to merge EMF based models. A working prototype has also been developed.

5.2 Future Work

5.2.1 Collecting User Input

The actions that need to be taken for every change in the delta model are currently set through a set of simple rules that assign a default action for each type of change. Other ways of collecting user input are obviously necessary. This process can be handled in several ways. First, a user interface that displays the changes can be used to gather user input interactively. The changes can be display generically for every meta-model. However, the way in which the changes are displayed can be improved if user interfaces specific to certain meta-models are provided. For example, in the MDA context it would certainly make sense to provide a specific user interface to handle the merging of UML2 models.

The rules used to set the actions could also be improved. A language could be designed in order to allow complex rules based on arbitrary conditions in the models. For example, in certain cases having a rule that says "keep the objects created in the target model that don't have references to other objects" might be better than having the user set a lot of actions manually, especially if the models being merged are big and don't have a meta-model specific user interface.

Finally, it would also be useful to keep track of the choices a user has made in previous transformations. If a source model is modified several times and the transformation is also re-run several times in order to propagate the changes, it is likely that the objects added manually to the target model, for example, appear recurrently as changes. In this case it would be useful to remember the previous choice the user has made since it will most likely be the choice he will take when the transformation is re-run.

5.2.2 Delta Model Consistency

The delta model contains the information about the changes between the target models being merged and is also used to collect information from the user about the actions that must be taken regarding each change. However, if this model is not checked once the information from the user is collected it may end up in an inconsistent state. For example, suppose an object is added to the original target model and one of the references from one of the existing objects is set to point to the new object. In this case two changes will be reported in the delta model: one indicating that an object has been added to the original target model and another one indicating that a reference from one of the objects in the original target model has changed. If the user decides not to keep the new object but also decides to keep the change in the reference then the delta model will be

left in an inconsistent state because the object that the reference is supposed to point at doesn't exist in the merged model.

This is why a consistency check must be added before the merge process. This can be done programmatically or through a model transformation, since it basically involves querying the delta model. Exactly how to perform the validation of the delta model depends on how the user input is being collected. If a user interface is being used to receive information interactively then the model must be checked every time the user makes a selection that may have an impact on the consistency of the rest of the model. If a rule based approach is used then the model must be checked after the application of the rules.

5.2.3 Unidentified Matching Objects

In certain cases it may be impossible to identify a pair of matching objects just by analyzing the trace objects. These are described in section 2.2.1. Even though the trace analysis cannot determine these matches exactly, it can be used as part of a process that tries to identify possible matching objects based on other information, like having some common objects in the source model pointed at by their trace objects and having some attributes set to the same value. Information from the user could also be requested when a possible match is suspected.

Also, if extra information is available for specific meta-models and transformations, it could be used to match these objects. For example, in the example given in section 2.2.1, the impossibility of identifying the objects as matching occurs because an attribute is moved from one class to another. If this specific information is available, then it can be used to aid the matching process.

5.2.4 Change Propagation In Both Ways

This thesis deals with the change propagation problem in only one direction. It could be useful to investigate the possibility of dealing with an analogous synchronization process that propagates changes the other way around. This process could automatically update the source model with the relevant changes introduced in the target model. This feature is not so important in tools that deal with model to text transformations, but it is relevant in the context of model to model transformations, since it is common to introduce changes in the target model that need to be updated in the source model.

5.3 Research Contribution

This thesis has derived an algorithm for calculating the delta between two models in the context of a model transformation engine. The algorithm is meta-model independent and relies on the trace objects created by the engine to determine which objects in the models being merged are equivalent, without having to depend on the existence of unique object identifiers. Even though the algorithm has been implemented as a model transformation specifically for Tefkat's trace model, it could be used by other transformation engines that are state based and that generate trace objects. Limitations of the algorithm for certain scenarios have also been identified, and possible solutions have been proposed as future research.

A meta-model capable of storing the delta has also been developed, according to the different types of changes that may occur between the models. This meta-model allows changes in the models being merged to be recorded and is also independent from the algorithms derived here.

Finally, an algorithm to merge the models based on the information gathered in the delta model and the input from the user has also been derived and implemented as a model transformation.

Bibliography

[1] W. Blast A. Kleppe, J. Warmer. *MDA Explained.* Addison Wesley, 2003.

[2] M. Allanen and I. Porres. Difference and union of models. *In Proceedings of the «UML» 2003 Conference*, October 2003.

[3] Eclipse.org. Uml2 emf-based uml 2.0 metamodel implementation, May 2005. URL `http://www.eclipse.org/uml2/`.

[4] E. Merks-R. Eliersick T.J. Grose F. Budinsky, D. Steinberg. *The Eclipse Modeling Framework.* Addison Wesley, 2003.

[5] D.S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing.* John Wiley & Sons, Inc., New York, NY, USA, 2002. ISBN 0471319201.

[6] W. Premerlani F.Eddy-W. Lorensen J. Rumbaugh, M.Blaha. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[7] M.J. Lawley-K. Raymond J. Steel K. Duddy, A. Gerber. Model transformation: A declarative, resusable patterns approach. *In Proceedings 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003)*, pages 174–185, 2003.

[8] M. Lawley. Tefkat: The emf transformation engine, May 2005. URL `http://www.dstc.edu.au/tefkat/`.

[9] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28:449–462, May 2002.

[10] D.S. Johnson M.R. Garey. *Computers and Intractability: A Guide to the Theory of NPCompleteness.* Freeman, 1979.

[11] Object ManagementGroup (OMG). Human-usable textual notation. URL `http://www.omg.org/docs/ad/02-03-02.pdf`.

[12] Object ManagementGroup (OMG). Meta object facility (mof) 2.0 xmi mapping specification, 2003. URL `http://www.omg.org/docs/ptc/03-11-04.pdf`.

[13] Object ManagementGroup (OMG). Omg/rfp/qvt mof 2.0 query/views/transformations rfp, 2003. URL `http://www.omg.org/docs/ad/02-04-10.pdf`.

[14] R. Popma. Jet tutorial part 1 (introduction to jet), May 2004. URL `http://downlaod.eclipse.org/tools/emf/scripts/docs.php`.

[15] R. Popma. Jet tutorial part 2 (write code that writes code), May 2004. URL `http://downlaod.eclipse.org/tools/emf/scripts/docs.php`.

[16] H. Garcia-Molina J. Widom S. Chawathe, A. Rajarama. Change detection in hierarchically structured information. *In Proceedings of the ACM SIGMOND International Conference on Management of Data*, pages 493–504, 1996.

[17] S. Sendall S. Demathieu, C. Griffin. Model transformation with the ibm model transformation framework, May 2005. URL `http://www-128.ibm.com/developerworks/rational/library/05/503_sebas/`.

[18] D. J. DeWitt Y. Wang and J. Cai. X-diff: A fast change detection algorithm for xml documents. *In ICDE*, 2003.

Appendix A

UML To Relational Transformation

This appendix shows a simple transformation between the UML2 meta-model from the Eclipse.org UML2 project [3] and a simple relational database model, using Tefkat's QVT language. Since there is no standard meta-model for describing a relational database design, a simple ecore model taken from IBM's model transformation tutorial [17] was chosen. Figure A.1 shows a diagram of this meta-model.

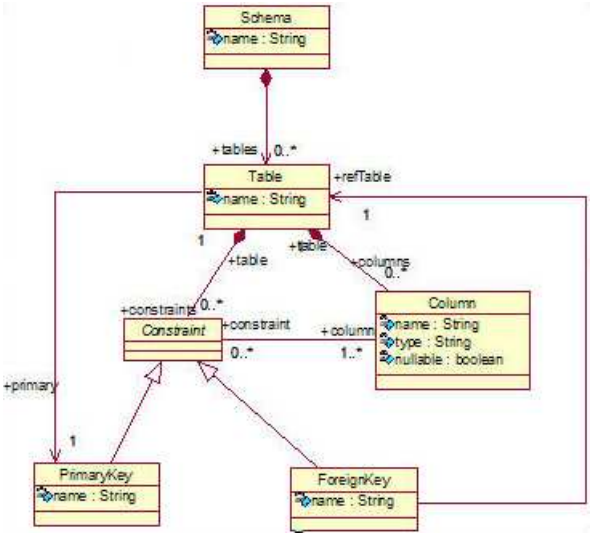


Figure A.1: Simple relational meta-model. Source: [17].

There are several ways to map an OO model into a relational database model[6]. The following rules were used in this transformation.

- Each package maps to a schema.
- Each class maps to a table.
- The superclass and each subclass map to a table.
- Each one-to-one association maps to a distinct table.
- Each one-to-many association maps to a distinct table.
- Each many-to-many association maps to a distinct table.

The transformation only considers binary associations and single inheritance. The following sections explain the rules used in the transformation.

A.1 Packages to Schemas

The rule in Figure A.2 is used to create a schema for every package in the UML model. The name of the schema is set to the name of the package.

```
 8 RULE Package2Schema(s)
 9 FORALL Package p
10 MAKE Schema s
11 SET s.name = p.name
12  ;
```

Figure A.2: Rule used to create a schema for every package.

A.2 Classes to Tables

For each class in the UML model a table is created. If the class inherits from some other class then tables are created for both the class and the superclass. These classes are related by the base class' primary key. The rule in Figure A.3 creates a table for each base class. The rule in Figure A.5 creates a table for each subclass and uses the pattern in Figure A.4 to find its topmost superclass and create the primary key based on it. Notice that the rules extend the Package2Schema rule in order to add the generated table to its corresponding schema. Also, the related classes and tables are linked with instances of Class2Table.

```
14 RULE BaseClass2Table(s, sc) EXTENDS Package2Schema(s)
15 FORALL Class c
16 WHERE NOT sc = c.superClass
17 MAKE Table t, Column col, PrimaryKey pk
18 SET t.name = c.name,
19   col.name = append(c.name, "-ID"),
20   col.type = "ID",
21   col.nullable = false,
22   pk.name = append(c.name, "-PK"),
23   t.columns = col,
24   t.primary = pk,
25   col.constraint = pk,
26   s.tables = t
27 LINKING Class2Table WITH class = c, table = t
28   ;
```

Figure A.3: Rule used to create a table for every base class.

```
30 PATTERN findBaseClass(C, BC)
31 WHERE BC = C.superClass
32   AND (NOT sc = BC.superClass OR findBaseClass(BC, BC))
33   ;
```

Figure A.4: Pattern used to find the topmost superclass of a given class.

```
35 RULE SubClass2Table(s, sc, st) EXTENDS Package2Schema(s)
36 FORALL Class c
37 WHERE sc = c.superClass
38   AND findBaseClass(c, sc)
39   AND Class2Table LINKS class = sc, table = st
40 MAKE Table t, Column col, PrimaryKey pk, ForeignKey fk
41 SET t.name = c.name,
42   col.name = st.primary.name,
43   col.type = "ID",
44   col.nullable = false,
45   pk.name = append(c.name, "-PK"),
46   fk.name = append(st.primary.name, "-FK"),
47   fk.refTable = st,
48   t.columns = col,
49   t.primary = pk,
50   t.constraints = fk,
51   col.constraint = pk,
52   col.constraint = fk,
53   s.tables = t
54 LINKING Class2Table WITH class = c, table = t
55   ;
```

Figure A.5: Rule used to create a table for every class that inherits from another.

A.3 Attributes to Columns

The rule in Figure A.6 creates a column for every attribute and inserts it in the corresponding table. The linking objects created in the BaseClass2Table and SubClass2Table rules are used to find the table that owns the attribute.

```
57 RULE Attribute2Column(c, t)
58 FORALL Property prop
59 WHERE c = prop.class_ AND Class2Table LINKS class = c, table = t
60 MAKE Column col FROM a2c(c, prop)
61 SET col.name = prop.name,
62   col.type = prop.type.name,
63   col.nullable = true,
64   t.columns = col
65   ;
```

Figure A.6: Rule used to create a column for every attribute.

A.4 Associations to Tables

All the associations are mapped to distinct tables. The rule in Figure A.8 is used to create a table for each one-to-one association. Foreign keys are created to relate the tables. The pattern in Figure A.7 is used to determine if an association end is navigable. This is necessary because the way to access the class in the association end is different depending on its navigability. The rule in Figure A.9 is used to create a table for each one-to-many association and the one in Figure A.10 to create a table for each many-to-many association. These rules work in a similar way since they all create a distinct table to represent the association.

```
67 PATTERN isNavigable(A, END)
68 WHERE NOT A.ownedElement = END
69   ;
```

Figure A.7: Pattern used to determine if an association end is navigable.

```
71 RULE OneToOneAssociation2Table(sc, end0, end1, c0, c1, ft0, ft1)
72 FORALL Association a
73 WHERE a.memberEnd{}.size() = 2
74   AND a.memberEnd{}.get(0) = end0
75   AND a.memberEnd{}.get(1) = end1
76   AND IF isNavigable(a, end0) THEN c0 = end0.class_ ELSE c0 = end0.type ENDIF
77   AND IF isNavigable(a, end1) THEN c1 = end1.class_ ELSE c1 = end1.type ENDIF
78   AND end0.upper = 1
79   AND end1.upper = 1
80   AND Class2Table LINKS class = c0, table = ft0
81   AND Class2Table LINKS class = c1, table = ft1
82 MAKE Table t, Column col0, Column col1,
83   PrimaryKey pk, ForeignKey f0, ForeignKey f1
84 SET t.name = a.name,
85   col0.name = append(ft0.name, "-ID"),
86   col0.type = "ID",
87   col0.nullable = false,
88   col1.name = append(ft1.name, "-ID"),
89   col1.type = "ID",
90   col1.nullable = false,
91   pk.name = append(a.name, "-PK"),
92   t.columns = col0,
93   t.columns = col1,
94   t.primary = pk,
95   t.constraints = f0,
96   t.constraints = f1,
97   col0.constraint = pk,
98   col1.constraint = pk,
99   f0.name = append(ft0.primary.name, "-FK"),
100  f0.refTable = ft0,
101  f1.name = append(ft1.primary.name, "-FK"),
102  f1.refTable = ft1,
103  col0.constraint = f0,
104  col1.constraint = f1
105  ;
```

Figure A.8: Rule used to create a table for each one-to-one association.

```
107 RULE OneToManyAssociation2Table(sc, end0, end1, c0, c1, ft0, ft1)
108 FORALL Association a
109 WHERE a.memberEnd{}.size() = 2
110   AND a.memberEnd{}.get(0) = end0
111   AND a.memberEnd{}.get(1) = end1
112   AND IF isNavigable(a, end0) THEN c0 = end0.class_ ELSE c0 = end0.type ENDIF
113   AND IF isNavigable(a, end1) THEN c1 = end1.class_ ELSE c1 = end1.type ENDIF
114   AND end0.upper = 1
115   AND end1.upper = -1
116   AND Class2Table LINKS class = c0, table = ft0
117   AND Class2Table LINKS class = c1, table = ft1
118 MAKE Table t, Column col0, Column col1,
119   PrimaryKey pk, ForeignKey f0, ForeignKey f1
120 SET t.name = a.name,
121   col0.name = append(ft0.name, "-ID"),
122   col0.type = "ID",
123   col0.nullable = false,
124   col1.name = append(ft1.name, "-ID"),
125   col1.type = "ID",
126   col1.nullable = false,
127   pk.name = append(a.name, "-PK"),
128   t.columns = col0,
129   t.columns = col1,
130   t.primary = pk,
131   t.constraints = f0,
132   t.constraints = f1,
133   col1.constraint = pk,
134   f0.name = append(ft0.primary.name, "-FK"),
135   f0.refTable = ft0,
136   f1.name = append(ft1.primary.name, "-FK"),
137   f1.refTable = ft1,
138   col0.constraint = f0,
139   col1.constraint = f1
140   ;
```

Figure A.9: Rule used to create a table for each one-to-many association.

```
177 RULE ManyToManyAssociation2Table(sc, end0, end1, c0, c1, ft0, ft1)
178 FORALL Association a
179 WHERE a.memberEnd{}.size() = 2
180   AND a.memberEnd{}.get(0) = end0
181   AND a.memberEnd{}.get(1) = end1
182   AND IF isNavigable(a, end0) THEN c0 = end0.class_ ELSE c0 = end0.type ENDIF
183   AND IF isNavigable(a, end1) THEN c1 = end1.class_ ELSE c1 = end1.type ENDIF
184   AND end0.upper = -1
185   AND end1.upper = -1
186   AND Class2Table LINKS class = c0, table = ft0
187   AND Class2Table LINKS class = c1, table = ft1
188 MAKE Table t, Column col0, Column col1,
189   PrimaryKey pk, ForeignKey f0, ForeignKey f1
190 SET t.name = a.name,
191   col0.name = append(ft0.name, "-ID"),
192   col0.type = "ID",
193   col0.nullable = false,
194   col1.name = append(ft1.name, "-ID"),
195   col1.type = "ID",
196   col1.nullable = false,
197   pk.name = append(a.name, "-PK"),
198   t.columns = col0,
199   t.columns = col1,
200   t.primary = pk,
201   t.constraints = f0,
202   t.constraints = f1,
203   col0.constraint = pk,
204   col1.constraint = pk,
205   f0.name = append(ft0.primary.name, "-FK"),
206   f0.refTable = ft0,
207   f1.name = append(ft1.primary.name, "-FK"),
208   f1.refTable = ft1,
209   col0.constraint = f0,
210   col1.constraint = f1
211   ;
```

Figure A.10: Rule used to create a table for each many-to-many association.