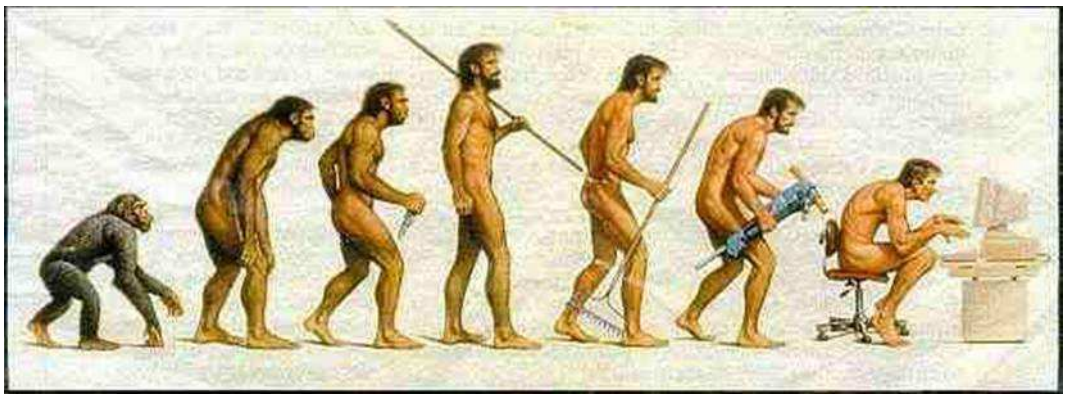
Building a framework to support Domain-Specific Language evolution

using Microsoft DSL Tools

Master's Thesis



Gerardo de Geest

Building a framework to support Domain-Specific Language evolution

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Gerardo de Geest
born in Ijsselstein, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Avanade Netherlands B.V.
Versterkerstraat 6
Almere, the Netherlands
www.avanade.nl

Cover picture: Animal Models in Human Evolution, University of Connecticut.

Building a framework to support Domain-Specific Language evolution

Author: Gerardo de Geest
Student id: 1156926
Email: gerardo.de.geest@avanade.com

Abstract

The concept of Domain-Specific Languages (DSLs) is not new. Examples of decennia old DSLs that are still popular are SQL, Tex and HTML. During their existence, these languages have evolved and several variations of them are used these days. Sometimes, for example during migration projects, translations to other variants of these DSLs are necessary. With the current software tools that make it possible to replace general purpose code by custom-built DSLs in a more convenient way, maintenance issues will become more relevant. This is especially relevant when the need to update DSLs increases over time and the number of DSLs used within a single project increases significantly. It is likely that this increase will happen in different stages of the project lifecycle, i.e., during development and maintenance, and can be caused by a variety of reasons, such as bug fixes and addition of extra features.

This masters thesis describes an approach to automatically convert models between different versions of a DSL. A DSL is introduced to define a mapping between two different versions of a DSL. Using this mapping language, a converter can be generated that converts models of an earlier version of some DSL to a newer version of the same DSL. The approach has been implemented for the Microsoft DSL-Tools infrastructure in two tools called DSLCompare and ConverterGenerator. The approach has been evaluated by means of three case studies taken from Avanade practice.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. E. Visser, Faculty EEMCS, TU Delft
Company supervisor:	drs. E.R. Jongsma, Avanade
Committee Member:	Prof. Dr. C. Witteveen, Faculty EEMCS, TU Delft
Committee Member:	drs. S.D. Vermolen, Faculty EEMCS, TU Delft

Preface

This document describes the results of my masters thesis, which I did for one part at Avanade Inc in Seattle and for the other part at Avanade Netherlands in Almere. I started my master thesis in the beginning op April 2007 and it took me 9 months to finish the research. The research evolved around Domain Specific Languages, Microsoft DSL Tools and Software Evolution. Before diving in depth, I will devote some words on my experience at Avanade.

My first contact with Avanade was at a drink at the end of the Dutch Imagine Cup final in 2006. Edwin Jongsma, Avanade's Capability Group Manager talked to me about possibilities for a Masters Thesis at Avanade. About a year later, I started my work at Avanade. I would like to thank Avanade for all the opportunities they have given me. I was able to go to Seattle for 10 weeks to talk to the best people in the field about my project. Furthermore I have visited the OOPSLA conference in Montreal to present my work at the Domain Specific Modeling workshop [15]. I have also written an article about Domain Specific Languages in the March 2008 edition of MSDN Magazine, which is read by about 100,000 developers worldwide [16].

The research topic, Domain Specific Languages was quickly selected. Edwin Jongsma has a passion for Domain Specific Languages and Avanade is interested in the development of this field. The problem of evolution of DSLs is something that Avanade experiences ofthen in practice.

There are many people that I would like to thank for their help during my project. I would like to thank Kyle Huntley, Sander Bockting, Gerben van Loon, Dennis Mulder, Antoine Savelkoul, Sander Vermolen, Arie van Deursen and Edwin Jongsma for their support and all the reviews they did.

Gerardo de Geest
Delft, the Netherlands
June 2, 2008

Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1
- 1.1 Avanade . . . 2
- 1.2 Terminology . . . 2
- 1.3 Example . . . 4
- 1.4 Research Question . . . 5

2 General Evolution Solution 9
- 2.1 Evolution Framework . . . 10

3 Representing Languages and Language Mappings 13
- 3.1 Microsoft DSL-Tools Meta-Meta-Model . . . 13
- 3.2 Transformation Language . . . 17

4 Versioning Scenarios 23
- 4.1 Removing inheritance . . . 23
- 4.2 Renaming of a domainclass . . . 26
- 4.3 Removing a domainclass . . . 27
- 4.4 Addition of a domainclass . . . 30
- 4.5 Removing recursion . . . 32

5 Automatic Detection Algorithm 37
- 5.1 General overview . . . 37
- 5.2 Representing Models . . . 38
- 5.3 Mapping Domainclasses . . . 39
- 5.4 Mapping Domainproperties . . . 40

5.5 Mapping Domainrelationships 41

6 From transformation to migration **43**
6.1 Creating a migration on M1 from a transformation on M2 43
6.2 The migration 43

7 Implementation **51**
7.1 Implementation of DSLCompare 51
7.2 Implementation of ConverterGenerator 56

8 Evaluation **59**
8.1 Experimental Design 59
8.2 Case Study 1 60
8.3 Case Study 2 64
8.4 Case Study 3 70

9 Discussion **75**
9.1 Evolution in practice 75
9.2 Changes between versions of a DSL 76
9.3 Equivalence of models 76
9.4 Applicability to other meta-meta-models 76
9.5 Transformation language not conform DSL-Tools 77
9.6 Verbose transformation language 77
9.7 Case studies 78
9.8 Scope of this thesis 80

10 Related Work **81**
10.1 Transformations on the meta-meta-model level 81
10.2 Model Comparison Engines 81
10.3 Evolution in other methodologies 82
10.4 Database Schema evolution 82
10.5 Engineering discipline for grammars 83
10.6 Document transformations 84
10.7 Language Evolution Tools 84
10.8 Graph transformations 85

11 Conclusions and Future Work **87**
11.1 Summary 87
11.2 Contributions 88
11.3 Future work 88
11.4 Conclusions 91

Bibliography **93**

A Glossary **97**

B F# Implementation of transformation on meta-models 99

List of Figures

1.1 Modeling levels . . . 3
1.2 Building a DSL using DSL-Tools . . . 4
1.3 Building a model using DSL-Tools . . . 6
1.4 Research question . . . 7

2.1 General problem of evolution . . . 9
2.2 DSL Versioning Framework . . . 10

3.1 DSL-Tools Meta-Meta Model . . . 14
3.2 isEmbedding Property . . . 15
3.3 Allow Duplicates . . . 15
3.4 DSL for adventure games . . . 17
3.5 Transformation Language . . . 19
3.6 Combination of languages . . . 22

4.1 DSL for adventure games without inheritance . . . 24
4.2 Domainclass Removal Source DSL . . . 27
4.3 Example model for the domainclass removal source DSL definition . . . 28
4.4 Entity Removal Target DSL . . . 29
4.5 Example model for the class removal target DSL definition . . . 30
4.6 Domainclass Addition Source DSL . . . 31
4.7 Entity Addition Target DSL . . . 32
4.8 Source DSL in which recursion is allowed . . . 33
4.9 Model showing nested questions . . . 34
4.10 Target DSL in which recursion is not allowed . . . 34
4.11 Model showing nested questions with Level domainproperty . . . 35

7.1 Domainclasses in DSLCompare . . . 52
7.2 Domainproperties in DSLCompare . . . 54
7.3 Domainrelationships in DSLCompare . . . 55
7.4 Example of a model converter . . . 56

8.1 Model in Beta 2 62
8.2 Datacontract model in WSSF b117 65
8.3 Datacontract model in WSSF final 65
8.4 models in WSSF b117 70
8.5 models in WSSF final 71

10.1 Ambiguity in graph transformation 85

11.1 Reusing transformations across multiple versions 90
11.2 Reusing transformations with custom DSLs 90

Chapter 1

Introduction

Today's software development projects become larger and more complex. Software engineering continuously tries to cope with this increased complexity by trying to increase the productivity of the developer. According to Software Productivity Research [44], the move from assembler to higher level programming languages increased the productivity of developers by a factor four. An approach to make the productivity of developers even higher is by creating programming languages that are specific to a certain domain. A domain could be technical, e.g. Service Oriented Architectures, or business focused, e.g. insurances. Programming languages specific to a certain domain are called Domain Specific Languages. As a consequence, programs written in such languages are more expressive in the domain they are created for and easier to understand for someone who understands the domain.

Using Domain Specific Languages (DSLs) is expected to be the most popular way of developing software in the near future [49]. Using DSLs an extra level of abstraction is added to the traditional way of developing software. These DSLs will help the developer to focus on the problem to be solved and the domain of the problem, instead of focusing on programming constructs as is the case in general purpose languages like C# and Java [17]. This results in a shorter development time and software of a higher quality standard.

Next to the shorter time to develop software using a DSL, DSLs are also promised to result in lower maintenance costs of the software developed using the DSL [49]. This can be done because of the reduced code size when a DSL is used. This is particularly important when one considers that 80% of the costs of software are usually not devoted to development, but to maintenance [32] [41].

However, DSLs come with some of the same issues that exist in traditional software development. One of these issues is the evolution of the artifact. This issue, however, becomes more complicated with DSLs, because an extra level of abstraction has been added. With traditional development, versioning issues only exist with the artifact being developed. For DSLs, versioning issues also exist for the DSLs themselves. This is especially true for DSLs focused on a business domain, because these business domains are very volatile [37].

The models conforming to a DSL definition could become syntactically invalid, because of a small change in the DSL definition. One could keep using these models with the old versions of the DSL of course. However, in practice this is not very useful, because users of the old version of the DSL could not benefit from the advantages and tools of the new

DSL. Hence, most DSLs will have to evolve over time including the programs written in these DSLs. Without adequate tool-support, DSL evolution is a complex, time-consuming, and error-prone task that severely hampers the long-term success of a DSL [42]. So DSLs can only help to reduce the overall maintenance costs of software if the maintenance costs to the DSLs themselves are also low. If this is not the case, the overall costs could turn out to be high [52].

This complicates the use of DSLs. On the one hand DSLs should be high level and close to the business to increase productivity. On the other hand, the tighter the DSL is connected to the business, the more fragile it becomes and the more often it will be subject to change [42]. This thesis proposes a solution to automatically cope with the changes that are made to the DSL.

In this chapter, some of the terminology and the research question will be presented. Chapter 2 presents the general solution approach of this thesis to the problem of DSL evolution. Chapter 3 discusses a more formal definition of the concepts used in this thesis. In chapter 4 some evolution scenarios are discussed to give the reader a feeling of what kind of problems occur when a DSL evolves. An algorithm to automatically detect the changes in a DSL definition is presented in chapter 5. Chapter 6 discusses how to get a migration on models from the changes that were detected in a DSL. An implementation of this solution is presented in chapter 7. In chapter 8 the approach of this thesis is evaluated using case studies. Chapter 9 gives a discussion on the work presented in this thesis. In chapter 10 the related work is discussed and chapter 11 presents the conclusions of this thesis and some future work.

1.1 Avanade

The research presented in this thesis has been performed for Avanade. Avanade is a joint-venture between Microsoft and Accenture. About 8000 professionals work at Avanade worldwide. They develop IT solutions for customers using all products that Microsoft has available.

Avanade is spending a lot of time doing research to ways of developing software in a better and more efficient way. At the moment, IT-projects become larger and more complicated. It is not clear how long the current methodologies of software development will be able to cope with this increasing complexity. Software Factories is one of the initiatives Avanade are looking at as an alternative to current methodologies [21].

1.2 Terminology

When using DSLs, it is important to distinguish certain levels of modeling, and how these levels are connected to each other. This is shown graphically in Fig. 1.1. The terminology used here is taken from OMG's MDA (Model Driven Architecture) [25].

The highest modeling level is the level M3: Meta-Meta-Model, in which languages are modeled. For this thesis the meta-meta-model that is implicitly defined in Microsoft DSL-Tools is used as the meta-meta-model to which the DSL definitions conform.

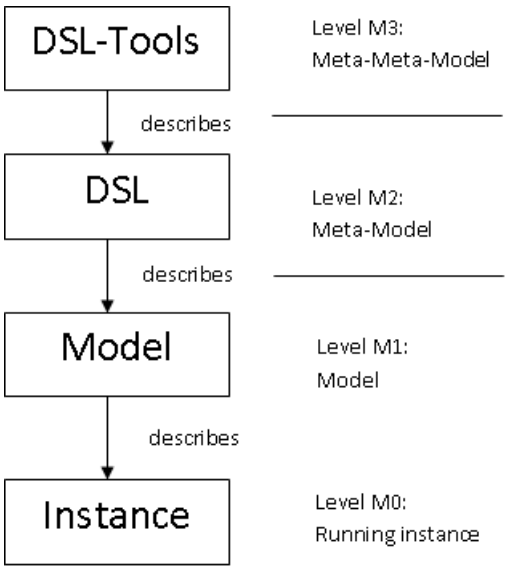


Figure 1.1: Modeling levels

The level M2: Meta-Model is the level at which the Domain-Specific Languages (DSLs) are actually modeled. Using classes that are defined at the M3 level, a programming language is created for a specific domain. This domain can either be a horizontal domain like web services, or a vertical domain like insurance policies. Domain-Specific Languages based on the Microsoft DSL-Tools are always visual languages, although they could also be represented as textual languages. However, the tools available for the Microsoft DSL-Tools models only have support for visual models.

The level M1: Model is the level at which the artifact is developed. This level represents instances of the DSL. For instance the model of an application for an insurance company is represented by this level. The DSL that is used to model applications for insurance companies is at the M2 level. The running artifact that is created from the model is represented by the level M0.

Although this comparison between OMGs MDA and Microsoft's DSL-Tools seems to fit, there are also some fundamental differences. One of them is that there is no explicit meta-meta-model in the DSL-Tools approach. In the MDA approach, the meta-meta-model is specified explicitly (Meta Object Facility, MOF), which makes it easier to reason about the meta-models that can be created using MDA. In DSL-Tools this meta-meta model is implicitly defined, so it is hard to tell which meta-models can be created using DSL-Tools and which cannot be created.

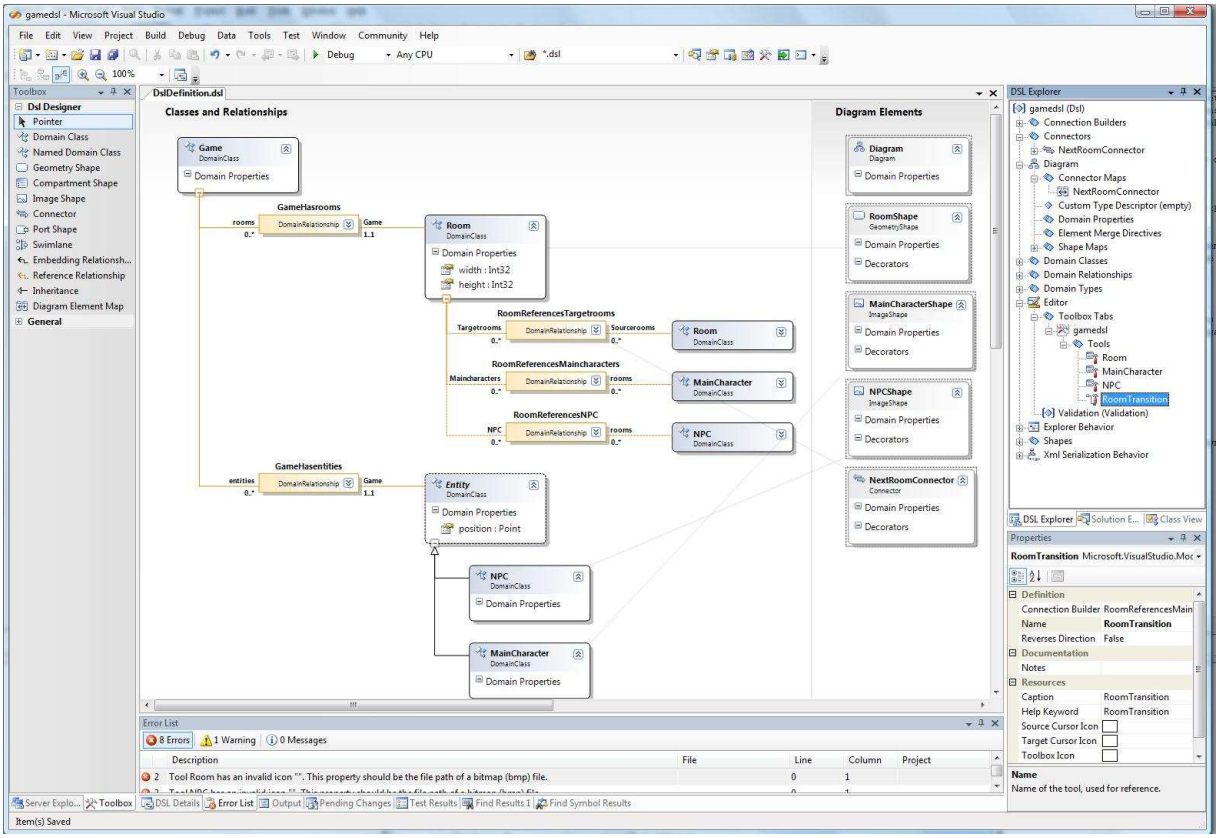


Figure 1.2: Building a DSL using DSL-Tools

1.3 Example

An example of a meta-model based on Microsoft DSL-Tools is given in Fig. 1.2. The model in the middle is the meta-model. This meta-model is defined using the meta-meta-model that is implicitly defined by DSL-Tools.

The model can be split into two parts: the part with domainclasses and domainrelationships and the part with diagram elements. In the part with domainclasses and domainrelationships, the abstract syntax of the meta-model is defined, while in the part with the diagram elements, the concrete syntax of the DSL is defined. Furthermore, on the right side of the figure, the DSL explorer can be seen. In the DSL explorer it is possible to define the toolbox for the DSL and how the models of the DSL will be serialized. On the left side, the toolbox for the meta-model can be seen. These are all the elements that can be placed on the model surface, both for the abstract syntax and the concrete syntax. On the bottom of the image, an error window is shown in which all the current problems with the meta-model are outlined. The error in this case is thrown, because an image for the toolbox of the DSL was not defined.

The meta-model in Fig. 1.2 represents a DSL for simple platform games. These games, consist of rooms and entities [19]. In the abstract syntax, five domainclasses have been defined: Game, Room, Entity, MainCharacter and NPC (Non-Playable Character). Fig. 1.2 shows that the domainclasses are connected to each other using domainrelationships. In this case, five domainrelationships have been defined: GameHasrooms, GameHasentities, RoomReferencesTargetrooms, RoomReferencesMaincharacters and RoomReferencesNPC. Furthermore, it is also possible to define an inheritance relation between the domainclasses. In this case an inheritance relation has been defined between the domainclasses Entity, MainCharacter and NPC.

The concrete syntax defined in Fig. 1.2 consists of a diagram, some geometry shapes and some image shapes. These represent the way a model is shown to a developer. An image shape is used when an instance of a domainclass is shown using an image. A geometry shape is used when the instance is represented using a square of some color. As an example, a model created in the meta-model of Fig. 1.2 is shown in Fig. 1.3. The grey squares represent instances of the Room domainclass, the monsters represent instances of the NPC domainclass and the Mario represents an instance of MainCharacter. Furthermore, the lines between the instances of Room, represents instances of the RoomReferencesTargetrooms domainrelationship.

1.4 Research Question

The research question investigated in this thesis is: *What scenarios exist considering versioning of DSLs and how can those scenarios be handled in an efficient way?*

The research question is quite common to projects using DSLs. When a project has finished, the source code is based on a certain DSL that is the newest in that time. However, two years later the customer might want some changes to the project. At that point in time, there could be a newer version of the DSL. If the project team decides to migrate the source code to the new DSL, they might run into all kinds of problems. The DSL has evolved, just like natural languages do. A sentence that was valid 10 years ago, might not be valid anymore.

The research question is shown graphically in Fig. 1.4.

In this thesis, it is assumed that the meta-meta-model does not change. It is not clear whether this is a realistic assumption, because most meta-meta-models do not exist that long already. However, it is clear that meta-meta-models evolve a lot slower than meta-models. Therefore, the focus of this thesis is on the evolution of meta-models. This means that the DSL definitions change. There can be multiple reasons for this:

- More domain knowledge. Over time, the developers of the DSL gain more knowledge about the domain. This could happen, because they are doing projects with the DSL.
- New features. Because the DSL is used in the field, people might be requesting new features that need to be included in the language definition.

When the DSL evolves, it is not clear what happens to the models conforming to that DSL. There is a chance that the model is still valid and the same code is generated from the

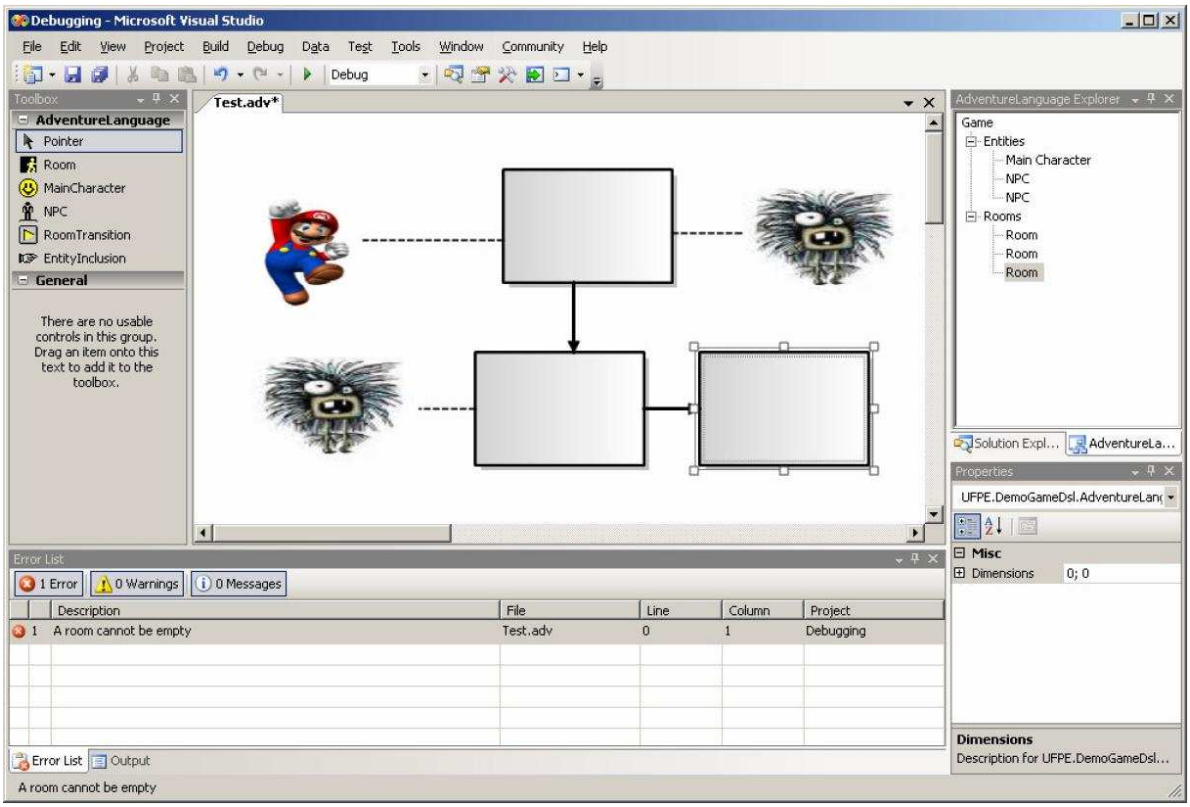


Figure 1.3: Building a model using DSL-Tools

model as was generated with the old version of the DSL. However, there is also a chance that the model cannot be read anymore and that it is impossible to generate code from the model. In this case it is necessary to have a converter to convert the old model to a new model that works with the new version of the DSL. The models that result from a conversion can be divided in four different classes:

- Syntactically correct models: The converter generates models that are syntactically correct to the new version of the DSL. This can be done quite easily, because an empty model is already syntactically correct. However, the challenge is to generate a syntactically correct new version of the model that looks similar to the old version of the model. However, nothing can be said about semantic equivalence. This class is the focus of this thesis.

- Static semantically correct models: The created models conforming to the well-formedness rules for the domain models [48]. The converter generates models that are static semantically correct to the new version of the DSL. There might be all kinds of validation rules included in the DSL that are not directly part of the syntax definition. For instance, in DSL-Tools it is possible to manually define validation rules in

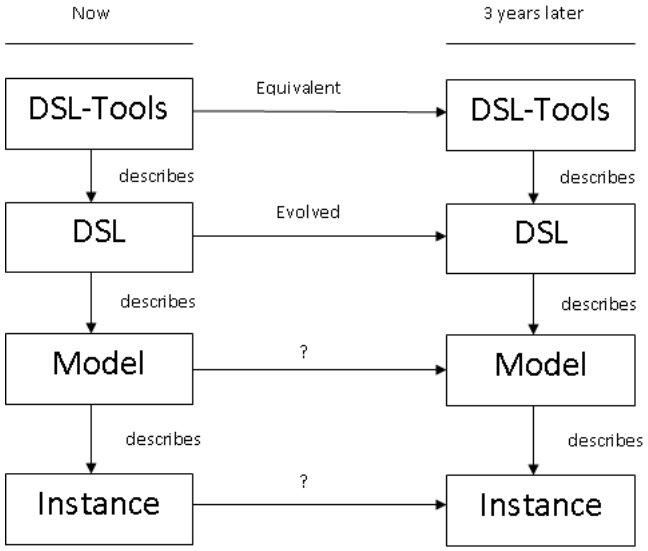


Figure 1.4: Research question

C# next to the meta-model. The converted models that are part of this class comply with all validation rules in the new DSL definition.

- Dynamic semantically correct models: The created models conforming to the same instance in the same semantic domain [48]. The converter generates models that are dynamic semantically correct to the new version of the DSL and to the old version of the model. There might have been some changes to the DSL that involve the semantics of the DSL. A model that is syntactically exactly the same and still valid in the new DSL, might have a complete different meaning in the new language definition. Converted models that belong to this class, have exactly the same semantic meaning in the source model.

- Interface equivalent models: Models are usually part of a solution that also includes custom code written in a general purpose language like C#. This custom code will interface with the code generated from the models. If the DSL changes, the code generated from the models probably changes as well. This results in the generated code not interfacing correctly anymore with the custom code. Converted models belonging to this class result in code being generated that is compatible with the custom code that was written for the old model.

The scope of this thesis is on syntactically correct models. Note that an empty model is also a syntactically correct model, and hence this requirement is not very hard to match. One could simply generate an empty model and the conversion has resulted in a syntactically correct model. However, this is not very satisfactory for the developers who are trying to convert their old models into models conforming to a new version of the DSL. So, this

thesis also focuses on converting to new models that are equivalent to the old models. The definition of equivalent here is: When the developer of the old model considers the new model as being equivalent to the old model.

Note that the research question is open about the precise meaning of versioning a DSL. The most natural interpretation is a new DSL that is an extension of an old DSL. As an alternative, the new DSL could have been developed independently of the old DSL. In chapter 8 both options will be addressed.

Chapter 2

General Evolution Solution

Fig. 2.1 shows the problem of evolution in DSLs in a more general way. DSL A is a DSL that has been developed some time in the past. DSL B is a newer DSL that has almost the same features as DSL A. The development of DSL B is based on DSL A. Many things could have changed in order to get DSL A from DSL B. Some examples of these changes are discussed in chapter 4.

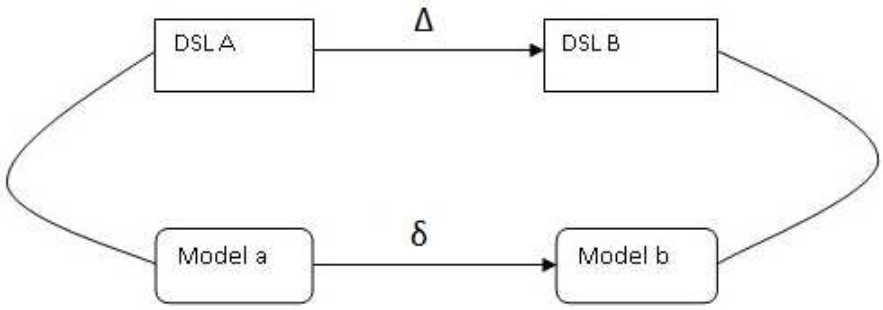


Figure 2.1: General problem of evolution

The Δ represents the transformation between DSL A and DSL B. It describes the changes that have been performed by the developers to DSL A in order to produce DSL B, such as the addition of new entities. The changes could also be a removal of entities, the change of entities, or a combination of all these operations. Examples of changes to a DSL definition are given in chapter 4.

The δ describes the migration for models conforming to DSL A to models conforming to DSL B, i.e. model a and model b. Note that the δ is on another level of abstraction compared to the DSL definitions A and B. It represents the same changes as described in Δ, but represents them on the level of models.

2.1 Evolution Framework

Using the notion of how the different DSLs and models are related to each other as described in Fig. 2.1 a framework has been developed to solve the problem of evolution in DSLs. The developers only have to describe the Δ as this is very specific to the changes they have made to a specific DSL. The δ is generated from Δ.

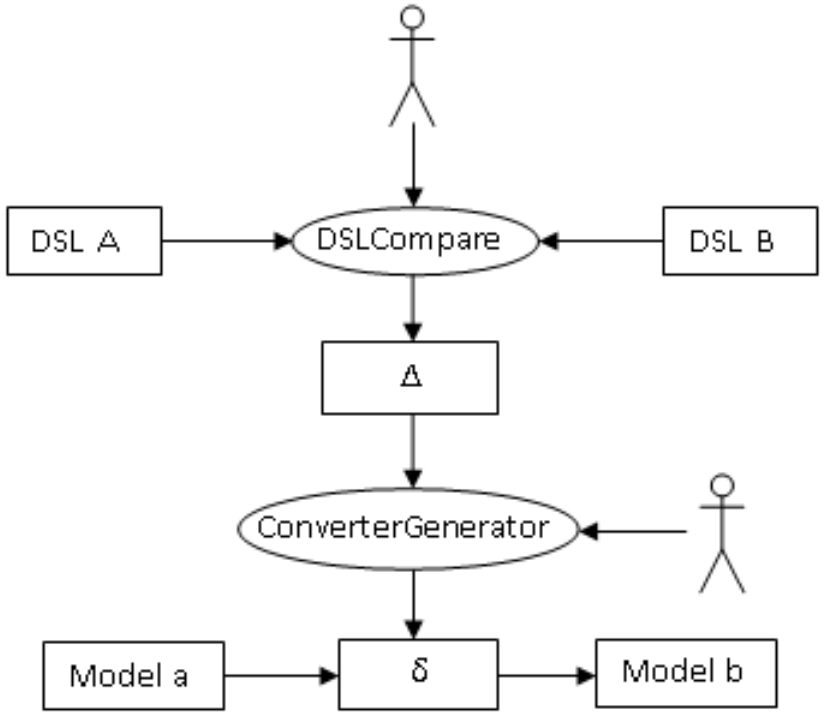


Figure 2.2: DSL Versioning Framework

Fig. 2.2 describes the framework developed for DSL versioning. Two different tools are used in this framework: The DSLCompare tool and the ConverterGenerator. Both of these tools have been developed internally within Avanade.

The inputs of this framework are the two DSL definitions, DSL A and DSL B. These DSLs are interpreted using the DSLCompare tool. This tool is able to recognize the majority of changes between the two input DSLs. Using this automatic recognition, most of the Δ is defined automatically. The automatic detection algorithm is discussed in chapter 5. This algorithm is evaluated using some case studies, which are presented in chapter 8. The implementation of the algorithm is discussed in section 7.1.

The ConverterGenerator is used to generate δ from Δ. The δ is a tool to migrate model *a* to model *b*. How the ConverterGenerator works is further described in section 6.1. Its implementation is discussed in section 7.2. The δ that is generated from the ConverterGenerator is further described in section 6.2.

Some amount of human help is usually necessary for completing and validating the Δ and the δ. There are some very specific cases that cannot be captured by the DSLCompare tool, for example the numbering issue from the scenario discussed in section 4.5. This means that Δ, the result of the DSLCompare tool, is incomplete. The developer is able to complete the function Δ in this stage and give a complete Δ to the ConverterGenerator. This human help can be added to the Δ in the DSLCompare program. The language in which Δ is expressed, is defined in section 3.2. Chapter 8 presents some case studies in which the amount of human help is estimated.

It might also be possible that the ConverterGenerator is not able to create a fully working migration (δ) for the Δ. In this case, some human help is necessary for the δ. This human help can be added by writing C# code. This is further discussed in section 7.

Note that is possible to create δ from DSLCompare. However, we have chosen to show the DSLCompare and ConverterGenerator as seperate steps, because they are conceptually different. The result of DSLCompare is Δ, which is a transformation on meta-models. The result of ConverterGenerator is δ, which is a migration on models.

Chapter 3

Representing Languages and Language Mappings

This chapter presents the languages used in this thesis. These also represent the assumptions used in the rest of the thesis. Section 3.1 presents the meta-meta-model of Microsoft DSL-Tools. This is the meta-meta-model that is used for all meta-models in this thesis. Section 3.2 presents the transformation that is introduced in this thesis. This transformation language is used to represent the transformations on meta-models. In section 3.2.3 the relationship between the two languages of sections 3.1 and 3.2 is described.

3.1 Microsoft DSL-Tools Meta-Meta-Model

Microsoft DSL-Tools is a separate package for Visual Studio (Microsoft's IDE). It can be used to define Domain-Specific Languages (DSLs). These DSLs all conform to a certain meta-meta-model that is implicitly defined within DSL-Tools.

3.1.1 Formal definition

The meta-meta-model of Microsoft DSL-Tools is proprietary to Microsoft and there is no public information available on what this meta-meta-model looks like. Meta-models are created using the visual modeling tools that the Microsoft DSL-Tools framework provides. The meta-meta-model of DSL-Tools is embedded in these tools and all meta-models created using the tools conform to the meta-meta-model.

By using DSL-Tools and trying to create different DSLs, a meta-meta-model has been inferred. Note that this is probably a subset of the meta-meta-model in DSL-Tools, because Microsoft has not made the meta-meta-model in DSL-Tools public yet. This inferred meta-meta-model is used as the basis for this thesis. Also note that the approach presented in this thesis can be used for other meta-meta-models as well, when those meta-meta-models are equivalent to the inferred meta-meta-model from DSL-Tools. However, we were not able to find such meta-meta-model.

The meta-meta-model inferred from DSL-Tools looks as follows:

$$
\begin{aligned}
DSL &\rightarrow DomainClass^{*}\ DomainRelationship^{*} \\
DomainClass &\rightarrow \text{"}class\text{"}\ Identifier(\text{":"}\ DomainClass)?\ DomainProperty^{+} \\
DomainProperty &\rightarrow \text{"}property\text{"}\ Identifier\ Type \\
DomainRelationship &\rightarrow \text{"}relation\text{"}\ Identifier\ DomainClass\ DomainClass \\
&\quad\ Boolean\ Boolean\ Identifier
\end{aligned}
$$

This meta-meta-model can also be displayed graphically using a UML class diagram. This is shown in Fig. 3.1. Thus, a DSL consists of a number of domainclasses. Each domainclass has at least one domainproperty. This domainproperty is a unique identifier for the domainclass. The name of this domainproperty is usually UUID, but could be something else as well. Furthermore, a domainclass can have a baseclass. This is a domainclass from which the domainclass inherits. A DSL also consists of DomainRelationships. A domainrelationship consists of two boolean properties: isEmbedding and allowsDuplicates. These will be explained below. The sourceName property represents an extra name for the domainrelationships that links it to the source. The rationale behind all these properties is unknown, because Microsoft has not yet released documentation about the DSL-Tools meta-meta-model. The information shown in Fig. 3.1 has been inferred by reverse engineering DSL-Tools.

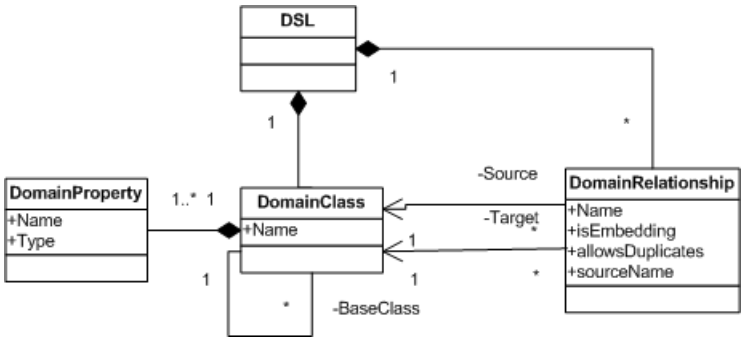


Figure 3.1: DSL-Tools Meta-Meta Model

The properties isEmbedding and allowsDuplicates are important for the type of domain-relationship. Every domainclass should be the target of exactly one domainrelationship with the isEmbbedding property set to true. This is to make sure that the Abstract Syntax Graph (ASG) of the model will contain an identifiable spanning tree. Hence, when the isEmbedding property of a domainrelationship is set to true, the domainrelationship automatically becomes a one-to-many domainrelationship. When the isEmbedding property is set to false, the domainrelationship is a many-to-many domainrelationship. Having an identifiable spanning tree is a requirement of DSL-Tools. The spanning tree is used to serialize and store the model. Furthermore, it is also used for code generation. It is not clear why a spanning

tree is necessary for serialization and code generation. The rationale behind the design of DSL-Tools is not released by Microsoft yet and hence this rationale is unknown.

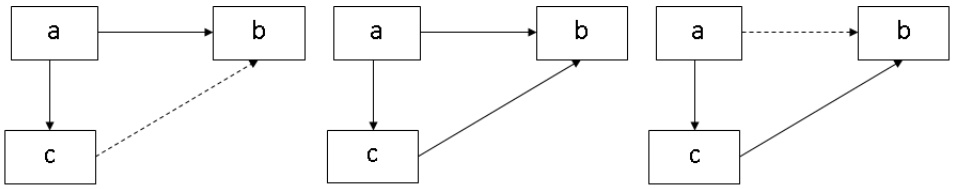


Figure 3.2: isEmbedding Property

The use of the isEmbedding property is shown in Fig. 3.2. The domainobject *a* is an instance of some domainclass *A* at the meta-model level (M2). Also the domainobject *b* is an instance of some domainclass *B* and the domainobject *c* is an instance of some domainclass *C*. The solid arrows indicate an instance of a domainrelationship with the isEmbedding property set to true. The dashed arrows indicate an instance of a domainrelationship with the isEmbedding property set to false.

Fig. 3.2 shows three different abstract syntax graphs (ASG). In the left ASG, *a* is the root of the ASG and *b* and *c* are its siblings. The second ASG does not have a clear spanning tree, because the domainrelationship between *B* and *C* also has its isEmbedding property set to true. So, the meta-model to which the second ASG conforms, is not valid in DSL-Tools. This could be made valid by either setting the isEmbedding property of the relationship between *B* and *C* to false (first ASG) or by doing that for the domainrelationship beween *A* and *B* (third ASG).

The allowsDuplicates property is only allowed to be true, when the domainrelationship is not embedding. Hence, it is always a many-to-many relation. The allowsDuplicates property determines whether it is allowed to have multiple instances of that domainrelationship connected to the same instances of domainclasses. This is shown graphically at model level (M1) in Fig. 3.3.

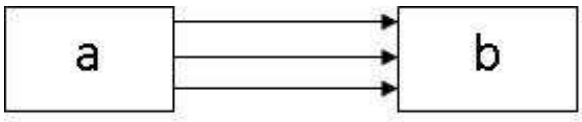


Figure 3.3: Allow Duplicates

The domainobject *a* is an instance of some domainclass *A* at meta-model level (M2). Also the domainobject *b* is an instance of some domainclass *B* at meta-model level (M2). The arrows are instances of some domainrelationship *R* which connects instances of domainclass *A* to instances of domainclass *B*. The domainrelationship *R* has the property allowDuplicates set to true. This is the reason why it is allowed to have multiple instances of

R connecting *a* and *b*. If the allowDuplicates property of domainrelationship *R* was set to false, only one instance of *R* would have been allowed between *a* and *b*.

The type property of a domainproperty in Fig. 3.1 can either be a simple type or a complex type. Simple types can be Boolean, String, Int, etc. A complex type can be defined by the developer of the DSL. This can for instance be a Point, with an x and y coordinate. Note that a complex type could also be defined by adding an extra domainclass to the meta-model and adding an embedding domainrelationship between the domainclass in which the domainproperty should be added and the newly added domainclass.

3.1.2 Serialization

The meta-models defined in DSL-Tools are serialized using an XML-based format which is proprietary to Microsoft. For each element in the meta-model, the developer can specify how this is serialized into an XML-based format. The format is similar to the XMI-format defined by OMG [36].

DSL-Tools uses a universal unique identifier (UUID), just as the XMI-format does. This UUID is useful for our purposes as well, since some parts of the automatic detection algorithm are greatly simplified by requiring that each element has a UUID. In practice, several other operations on models also use such an identifier [5]. It is assumed that the UUID of an element does not change after it has been set, and that the UUID is unique for every element. These assumptions are valid for the XMI-format defined by OMG, as well as for the XML-format that Microsoft uses to serialize meta-models.

3.1.3 Example instance of the Meta-Meta-Model

When developing a DSL conforming to the Microsoft DSL-Tools meta-meta-model, the DSL is expressed visually, instead of textually. To show this, an example of a game DSL has been created. This DSL has the purpose to develop simple games, with one player, several non-playable characters, rooms and transitions between the rooms. The abstract syntax of this DSL is shown visually in Fig. 3.4.

Note that some domainclasses in the meta-model of Fig. 3.4 are shown multiple times. This happens for instance with the Entity class and also with the Room class. It is shown this way, to improve the readability of the meta-model. By showing these classes multiple times, long connections between classes are avoided. However, note that these domainclasses are only defined once.

This language could also be written in the following form, according to the meta-meta-model language that was defined in section 3.1.1:

```
class Game
class Room
  property width Int32
  property height Int32
class Entity
  property position Point
class MainCharacter : Entity
```

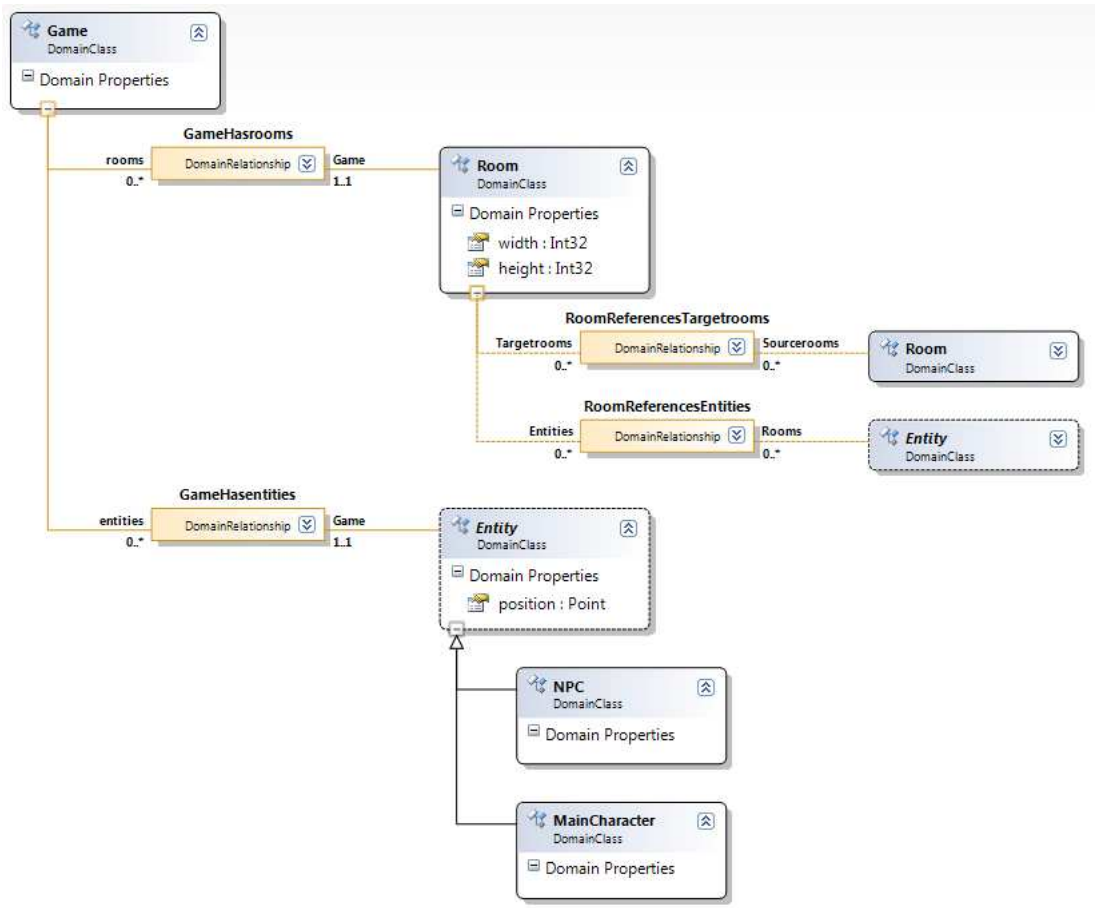


Figure 3.4: DSL for adventure games

```
class NPC : Entity
relation GameHasRooms Game Room true false Rooms
relation RoomReferencesEntities Room Entity false false Entities
relation RoomReferencesTargetRooms Room Room false false TargetRooms
relation GameHasEntities Game Entities true false Entities
```

Note that Fig. 3.4 does not show the properties isEmbedding and allowDuplicates of the domainrelationships. These properties are shown if the user clicks on a domainrelationship.

3.2 Transformation Language

During the project, a transformation language has been defined to describe the transformation between two different DSL definitions. These DSL definitions are instances of the Microsoft DSL-Tools meta-meta-model. Given a DSL definition A and transformation Δ,

the transformation Δ describes how to transform DSL definition *A* into a new DSL definition *B*. Examples of how the transformation language can be used are given in chapter 4. Different model-transformation languages exist already, of which ATL [9] and KMTL [4] are probably the most famous ones. However, these languages are developed for general model-transformations. None of these languages is specialized for transformations regarding versioning. The next sections define the transformation language used to describe Δ.

3.2.1 Syntax of the Transformation Language

The syntax of the transformation language is defined as follows:

$$Mapping \rightarrow ClassMap^{*}\ PropertyMap^{*}\ RelationMap^{*} \tag{3.1}$$

$$ClassMap \rightarrow \text{"}classmap\text{"}\ Identifier\ Identifier \tag{3.2}$$

$$PropertyMap \rightarrow \text{"}propertymap\text{"}\ Identifier\ Identifier\ Identifier\ Identifier\ Relation?\ Value? \tag{3.3}$$

$$RelationMap \rightarrow \text{"}targetrelation\text{"}\ Identifier\ Identifier\ Identifier\ Boolean\ Boolean\ Identifier\ Relation^{+} \tag{3.4}$$

$$Relation \rightarrow \text{"}relation\text{"}\ Identifier\ Identifier\ Identifier\ Boolean\ Boolean\ Identifier \tag{3.5}$$

In line 3.2 the transformation for classes is shown. The first identifier indicates the name of the class in DSL definition *A*, while the second identifier indicates the name of the class in the new DSL definition *B*.

Line 3.3 shows the transformation for properties. It takes four identifiers, one optional relation and one optional value. The first identifier indicates the name of the domainclass to which the domainproperty is connected in DSL definition *A*. The second identifier is the name of the domainproperty in DSL definition *A*. The third and the fourth identifier also indicate the name of the domainclass and the name of the domainproperty, but now in DSL definition *B*. The optional relation is used to indicate the new domainrelationship that is necessary if the property is moved to a different domainclass. The optional value gives a value to the domainproperty, in case it is a new domainproperty and needs a default value.

The transformation for domainrelationships is shown in line 3.4. It takes four identifiers, two booleans and a sequence of relations. Note that this structure is different than lines 3.2 and 3.3. The classes and properties are written in such a way that only a direct one-to-one map is supported. However, for domainrelationships it is possible to transform multiple domainrelationships of DSL definition *A* to one domainrelationship in the new DSL definition *B*. Hence, this is a many-to-one transformation.

The first identifier of line 3.4 is used for the name of the domainrelationship. The second and third identifier are used for the name of the source domainclass and the name of the target domainclass of the domainrelationship. Note that domainrelationships in DSL-Tools are always unidirectional (although it might not look that way in concrete syntax), so it is always possible to identify the source domainclass of a domainrelationship and the

target domainclass of a domainrelationship. The two booleans represent the isEmbedding and allowsDuplicates properties. The fourth identifier indicates the property of the domainrelationship in which the instance of the source is stored. At last, at least one domainrelationship must be given. This domainrelationship or sequence of domainrelationships are domainrelationships of DSL definition *A*. They describe between which two domainclasses the domainrelationship in the model based on DSL definition *B* must be drawn. This will be explained further in chapter 4.

In line 3.5 the syntax of a Relation is described. This syntax is the same as the syntax of a TargetRelation, only without the Relations. The meaning of the four identifiers and the two booleans is also the same.

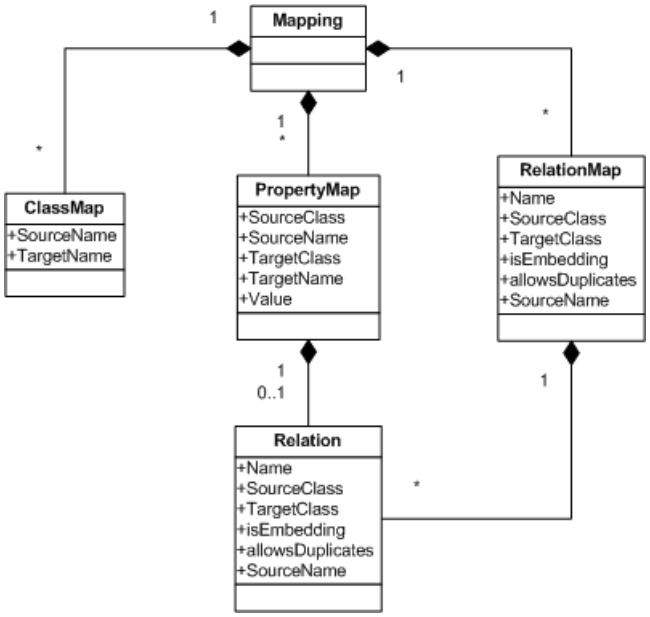


Figure 3.5: Transformation Language

The transformation language is shown visually in Fig. 3.5. The Identifiers have been given names to give the reader an overview of what each Identifier represents. Note that the Relation is connected to the PropertyMap and to the RelationMap. In practice, a Relation will not be connected to both at the same time. If the Relation is connected to a PropertyMap, the Relation represents a domainrelationship in the new version of the DSL definition. However, if the Relation is connected to a RelationMap, the Relation represents a domainrelationship in the old version of the DSL definition.

It can be noted that a lot of information is included in the transformation language, that can also be found in the DSL definition. This is done to be able to support transformations, without having the DSL definition.

3.2.2 Semantics of the Transformation Language

To define the semantics of the transformation language, the syntax of DSL-Tools and the syntax of the transformation language are used. This section gives examples of the most used transformations in this thesis. An implementation of the transformation is shown in appendix B. This is implementation is written in F#, a functional language developed by Microsoft Research. F# is a pragmatically-oriented variant of ML that shares a core language with OCaml. The programs developed in F# run on top of the .NET framework [40]. The F# code defines an apply function, which takes a DSL and a Mapping, and yields the transformed DSL. The rules given below cover the most used cases of this apply-function in this thesis.

The semantics of line 3.2 are described line 3.6.

$$(\text{"}class\text{"}\ x\ properties,\ \text{"}classmap\text{"}\ x\ y) \quad \rightarrow \quad \text{"}class\text{"}\ y\ properties \tag{3.6}$$

In line 3.6, the parameter *x* describes the name of the domainclass in the old DSL definition *A*. The parameter *y* describes the name of the domainclass in the new DSL definition *B*. The properties describe the domainproperties that a class could have. Note that these domainproperties can be transformed using the propertymap rule of line 3.3. The semantics of this rule are described in 3.7 and 3.8.

$$\begin{aligned}(\text{"}class\text{"}\ x\ \text{"}property\text{"}\ a\ type,\ \text{"}classmap\text{"}\ x\ y\ \text{"}propertymap\text{"}\ x\ a\ y\ b) \quad &\rightarrow \\ \text{"}class\text{"}\ y\ \text{"}property\text{"}\ b\ type \qquad &(3.7)\end{aligned}$$

Segment 3.7 describes a transformation in which the domainproperty stays in the same domainclass. In this case, the source model based on DSL definition *A* has a domainclass with some domainproperty associated with it. The transformation consists of a classmap and a propertymap in which both the domainclass and the domainproperty can be renamed.

$$\begin{aligned}(\text{"}class\text{"}\ x\ \text{"}property\text{"}\ a\ type,\ \text{"}classmap\text{"}\ x\ y\ \text{"}propertymap\text{"}\ x\ a\ z\ b \\ \text{"}relation\text{"}\ r\ y\ z\ isEmb\ allowDup\ s) \quad &\rightarrow \\ \text{"}class\text{"}\ y\ \text{"}class\text{"}z\ \text{"}property\text{"}\ b\ type\ \text{"}relation\text{"}\ r\ y\ z\ isEmb\ allowDup\ s \qquad &(3.8)\end{aligned}$$

Segment 3.8 shows a transformation in which a domainproperty moves to a different domainclass. In this case again, the source model has a domainclass and a domainproperty. However, the transformation is more complicated now. Because the name of the new domainclass for the domainproperty is different than the name of the domainclass in the classmap transformation, a domainrelationship between those two domainclasses needs to be defined. This also results in a model based on the new DSL Definition *B*, which has two domainclasses and a domainrelationship between them.

The semantics for the mapping of domainrelationships is described in segment 3.9. In this case only a one-to-one mapping is shown. Segment 3.10 shows the case of a two-to-one mapping. It works the same for a many-to-one mapping.

$$\begin{aligned}(&\text{"class" } w \text{ "class" } x \text{ "relation" } a\ w\ x\ isEmbOld\ allowDupOld\ s,\\ &\text{"classmap" } w\ y \text{ "classmap" } x\ z\\ &\text{"targetrelation" } b\ y\ z\ isEmb\ allowDup\ t\\ &\text{"relation" } a\ w\ x\ isEmbOld\ allowDupOld\ s) \rightarrow\\ &\text{"class" } y \text{ "class" } z \text{ "relation" } b\ y\ z\ isEmb\ allowDup\ t \end{aligned} \tag{3.9}$$

$$\begin{aligned}(&\text{"class" } u \text{ "class" } v \text{ "class" } w \text{ "relation" } r\ u\ v\ isEmb_1\ allowDup_1\ h\\ &\text{"relation" } s\ v\ w\ isEmb_2\ allowDup_2\ i,\\ &\text{"classmap" } u\ x \text{ "classmap" } v\ y \text{ "classmap" } w\ z\\ &\text{"targetrelation" } t\ x\ z\ isEmb\ allowDup\ j\\ &\text{"relation" } r\ u\ v\ isEmb_1\ allowDup_1\ h \text{ "relation" } s\ v\ w\ isEmb_2\ allowDup_2\ i) \rightarrow\\ &\text{"class" } x \text{ "class" } y \text{ "class" } z \text{ "relation" } t\ x\ z\ isEmb\ allowDup\ j \end{aligned} \tag{3.10}$$

Note that the converted DSL definition is not necessarily equal to the new DSL definition. They are, for instance, unequal if the new DSL definition contains new classes. The transformation language does not allow to define additions to the DSL definition, because they will not be reflected in δ. New classes, for instances, were not instantiated in models conforming to the old DSL definition, because these classes did not exist. Hence, when these models are migrated, these classes will not be instantiated in the converted model either. To keep the transformations written in the transformation language as small as possible, we have chosen to remove information that does not add information to δ.

Also note that the value property of a propertymap is not shown in the F# implementation. This is because the value property is not relevant to meta-models. It used as a default value for models that instantiate a particular property. Hence, it is used by the Converter-Generator.

3.2.3 Combination of languages

In sections 3.1.1 and 3.2 two languages have been presented. The transformation language has many identifiers that refer back to the DSL-Tools meta-meta-model. Fig 3.6 shows how the transformation language and the DSL-Tools meta-meta-model refer to each other.

Note that most of the identifiers shown in Fig. 3.5 have been removed in Fig. 3.6. The ClassMap had a SourceClass and TargetClass. These have been replaced by the two arrows from ClassMap to DomainClass. The same holds for the PropertyMap and DomainProperty. One arrow represents the DomainClass or DomainProperty in the old DSL definition and the other arrow represents the DomainClass or DomainProperty in the new DSL definition.

However, the RelationMap only has one arrow to DomainRelationship. The other arrow to DomainRelationship comes from Relation. The reason is that the RelationMap represents a DomainRelationship in the new DSL definition, while the Relation represents a Domain-Relationship in the old DSL definition.

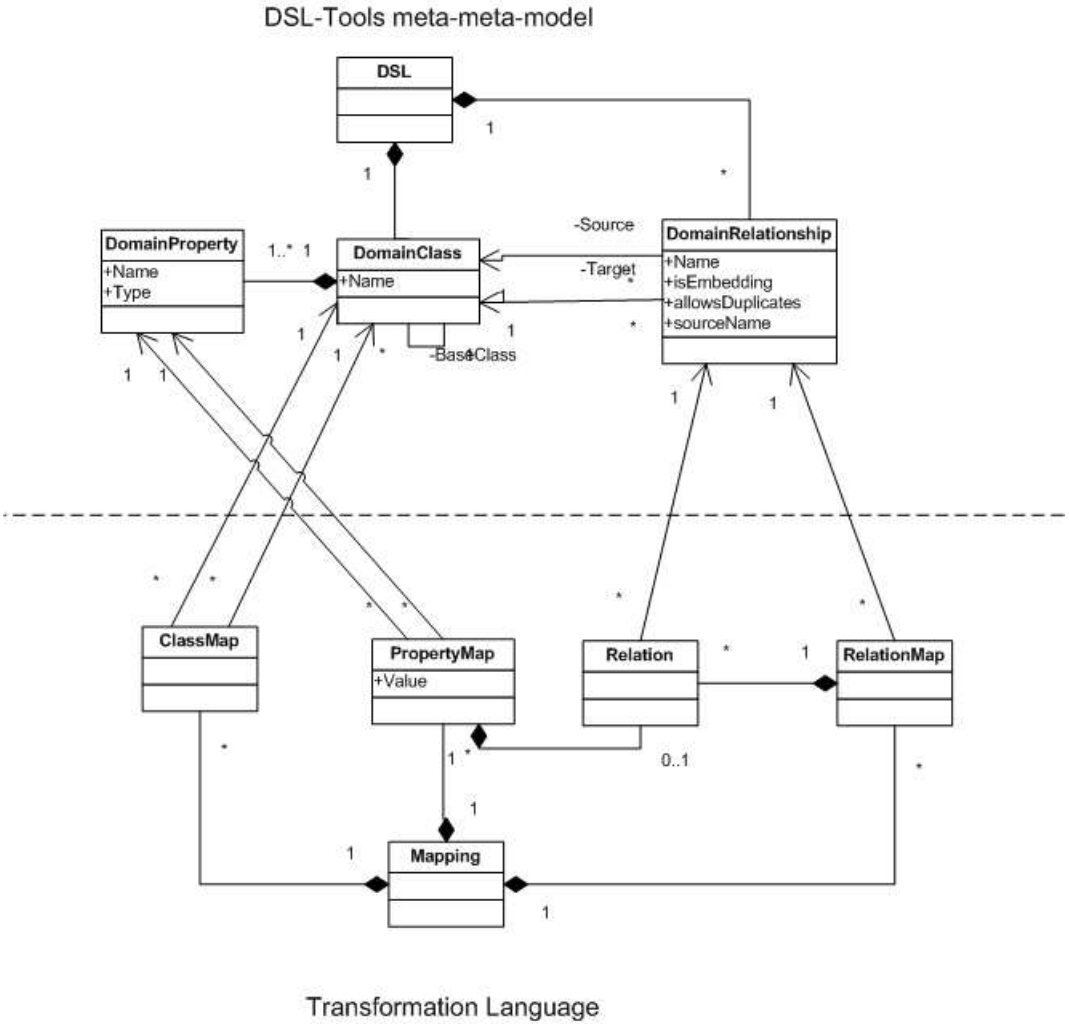


Figure 3.6: Combination of languages

Chapter 4

Versioning Scenarios

To get a better idea of what the versioning problems for DSLs are, different versioning scenarios have been identified. The scenarios in sections 4.1 and 4.2 are artificial scenarios that has have created to demonstrate what happens if inheritance is removed from a DSL definition. The scenarios in sections 4.3, 4.4 and 4.5 are cases from real-life projects and represent the removal of a domainclass, the addition of a domainclass and the removal of recursion respectively.

4.1 Removing inheritance

The DSL definition shown in Fig. 3.4 of the previous chapter has two inheritance relations. Both domainclasses MainCharacter and NPC are subdomainclasses of the domainclass Entity. It is possible to remove the inheritance relation from the DSL definition as is shown in Fig. 4.1. This figure shows exactly the same DSL definition, only the inheritance relation has been removed.

The DSL definition in Fig. 4.1 can also be written in the following form:

```
class Game
class Room
  property width Int32
  property height Int32
class MainCharacter
  property position Point
class NPC
  property position Point
relation GameHasRooms Game Room true false Rooms
relation RoomReferencesMainCharacters Room MainCharacter false false Maincharacters
relation RoomReferencesNPC Room NPC false false NPC
relation RoomReferencesTargetRooms Room Room false false TargetRooms
relation GameHasMainCharacters Game MainCharacter true false Maincharacters
relation GameHasNPC Game NPC true false NPC
```

For this scenario, it is assumed that the domainclass Entity in Fig. 3.4 is an abstract domainclass. This means that this domainclass cannot be instantiated at the model level.

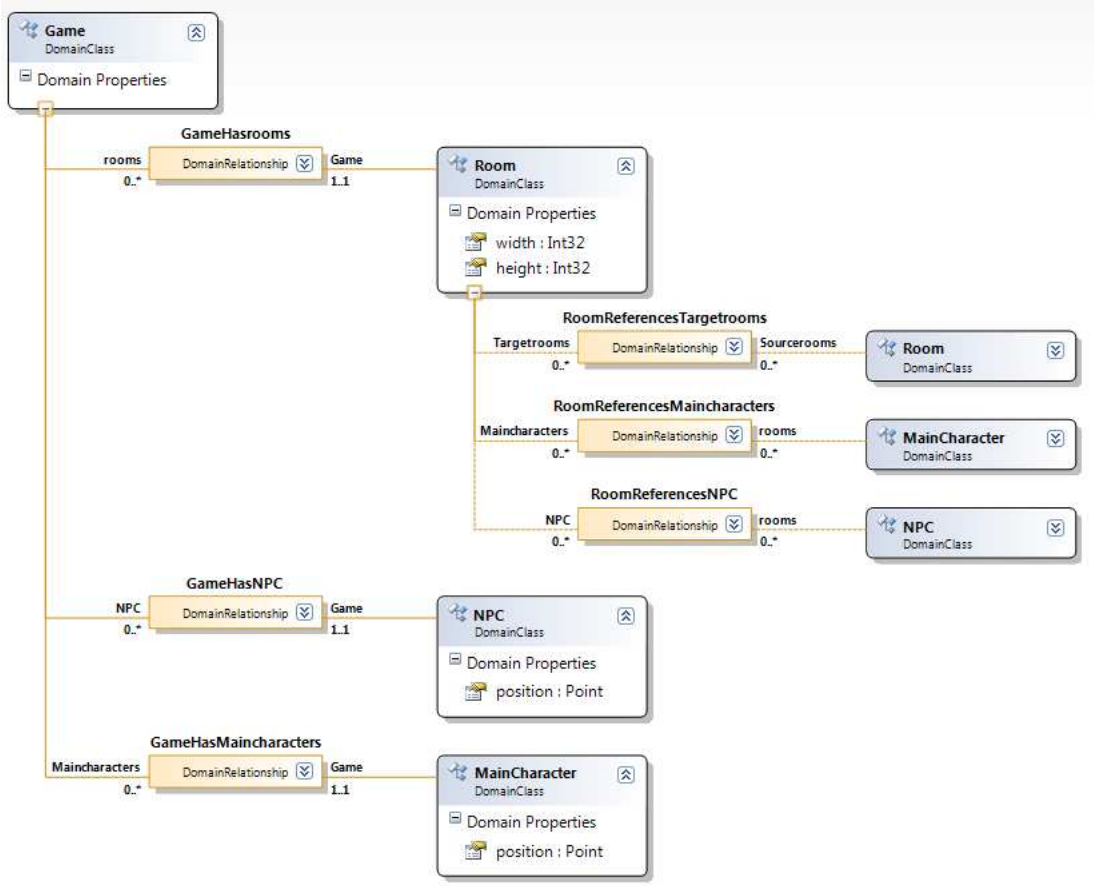


Figure 4.1: DSL for adventure games without inheritance

Hence, the domainclass Entity is removed in the DSL definition of Fig. 4.1. The functionality that was defined in the domainclass Entity has been moved to its subdomainclasses: MainCharacter and NPC.

The set of domainclasses that can be instantiated using the DSL definition in Fig. 3.4 is exactly the same as for the DSL definition in Fig. 4.1. However, this is not the case for the domainrelationships. Because the superdomainclass Entity has multiple subdomainclasses, multiple domainrelationships are necessary to cover the inheritance relation to the domainclass Entity. For instance, the domainrelationship GameHasEntities in Fig. 3.4 has been split into two domainrelationships in Fig. 4.1: GameHasMaincharacters and GameHasNPC. This was done, because Microsoft DSL-Tools does not allow a DSL definition to have two domainrelationships with the same name.

Semantically the domainrelationships GameHasMainCharacters and GameHasNPC in Fig. 4.1 are almost the same compared to the domainrelationship GameHasEntities in Fig. 3.4. The only difference is that the target of GameHasMainCharacters and GameHasNPC is re-

stricted to be an instance of respectively MainCharacter or NPC. The target of GameHasEntities can either be an instance of MainCharacter or an instance of NPC.

The transformation to transform the DSL definition of Fig. 3.4 into the DSL definition of Fig. 4.1 is given below. The transformation is described using the language given in section 3.2.

```
classmap Game Game
classmap Room Room
classmap MainCharacter MainCharacter
classmap NPC NPC
propertymap Room width Room width
propertymap Room height Room height
propertymap MainCharacter position MainCharacter position
propertymap NPC position NPC position
targetrelation GameHasRooms Game Room true false Entities
  relation GameHasRooms Game Room true false Entities
targetrelation GameHasMaincharacters Game MainCharacter true false Maincharacters
  relation GameHasEntities Game MainCharacter true false Entities
targetrelation GameHasNPC Game NPC true false NPC
  relation GameHasEntities Game NPC true false NPC
targetrelation RoomReferencesTargetRooms Room Room false false TargetRooms
  relation RoomReferencesTargetRooms Room Room false false TargetRooms
targetrelation RoomReferencesMaincharacters Room MainCharacter false false Maincharacters
  relation RoomReferencesEntities Room Entity false false Entities
targetrelation RoomReferencesNPC Room NPC false false NPC
  relation RoomReferencesEntities Room Entity false false Entities
```

Note that the domainproperty position of domainclass Entity of the DSL definition given in Fig. 3.4 is not used in the transformation. However, because of the inheritance relation, this domainproperty was also present in the domainclasses MainCharacter and NPC. This is the actual domainproperty that maps to the domainproperties of the new DSL definition.

The domainrelationships GameHasEntities and RoomReferencesEntities are both used twice in the transformation. Because the domainclass Entity has two subdomainclasses, all domainrelationships that are connected to this domainclass are split into two new domainrelationships. Note, however, that the targetdomainclass of the domainrelationships GameHasEntities and RoomReferencesEntities is not Entity in the transformation. This is because the domainrelationship is always with one of the subdomainclasses of the domainclass Entity. The domainrelationship will never be with the domainclass Entity, because we assumed Entity to be an abstract domainclass.

As can be seen in the transformation, no changes to the domainclasses or the domainproperties have been made when removing the inheritance relation from the DSL definition. The only changes are made to the domainrelationships. Hence, removing inheritance can be seen as a transformation on some domainrelationships of the DSL definition.

Note that the mapping is very verbose, because all identity mappings are also written in the transformation language. This is further discussed in section 9.6.

4.2 Renaming of a domainclass

Renaming of domainclasses in a DSL definition could happen when the developers of the language find out that a name of a domainclass is unclear to the users of the DSL. When looking back to the example of the game DSL (Fig. 3.4) one could argue that the term NPC is not clear. When this name is changed to Enemy, for instance, most models conforming to the DSL will become invalid. This new DSL is shown below:

```
class Game
class Room
  property width Int32
  property height Int32
class Entity
  property position Point
class MainCharacter : Entity
class Enemy : Entity
relation GameHasRooms Game Room true false Rooms
relation RoomReferencesEntities Room Entitiy false false Entities
relation RoomReferencesTargetRooms Room Room false false TargetRooms
relation GameHasEntities Game Entities true false Entities
```

Note that the only change here is the name of the domainclass NPC to Enemy. No other changes have been made. The set of domainclasses has changed, so all models that were instantiating the domainclass NPC have now become syntactically invalid. However, this is only a subset of all models that can be created using the DSL, because it is allowed to define models without instantiating the domainclass NPC. The mapping between the two DSL definitions is given below:

```
classmap Game Game
classmap Room Room
classmap Entity Entity
classmap MainCharacter MainCharacter
classmap NPC Enemy
propertymap Room width Room width
propertymap Room height Room height
propertymap Entity position Entity position
propertymap MainCharacter position MainCharacter position
propertymap NPC position Enemy position
targetrelation GameHasRooms Game Room true false Entities
  relation GameHasRooms Game Room true false Entities
targetrelation GameHasEntities Game Entity true false Entities
  relation GameHasEntities Game Entity true false Entities
targetrelation GameHasEntities Game MainCharacter true false Entities
  relation GameHasEntities Game MainCharacter true false Entities
targetrelation GameHasEntities Game NPC true false Entities
  relation GameHasEntities Game NPC true false Entities
targetrelation RoomReferencesTargetRooms Room Room false false TargetRooms
  relation RoomReferencesTargetRooms Room Room false false TargetRooms
targetrelation RoomReferencesEntities Room Entity false false Entities
  relation RoomReferencesEntities Room Entity false false Entities
targetrelation RoomReferencesEntities Room MainCharacter false false Entities
```

```
  relation RoomReferencesEntities Room MainCharacter false false Entities
targetrelation RoomReferencesEntities Room NPC false false Entities
  relation RoomReferencesEntities Room NPC false false Entities
```

4.3 Removing a domainclass

This scenario comes from a migration between two different DSLs that are both used for building applications based on the Service Oriented Architecture (SOA) principles. Fig. 4.2 shows the part of the source DSL that is relevant to this scenario. The Group domainclass is connected to both the Service domainclass as well as the Contract domainclass. This DSL can also be written in the following form:

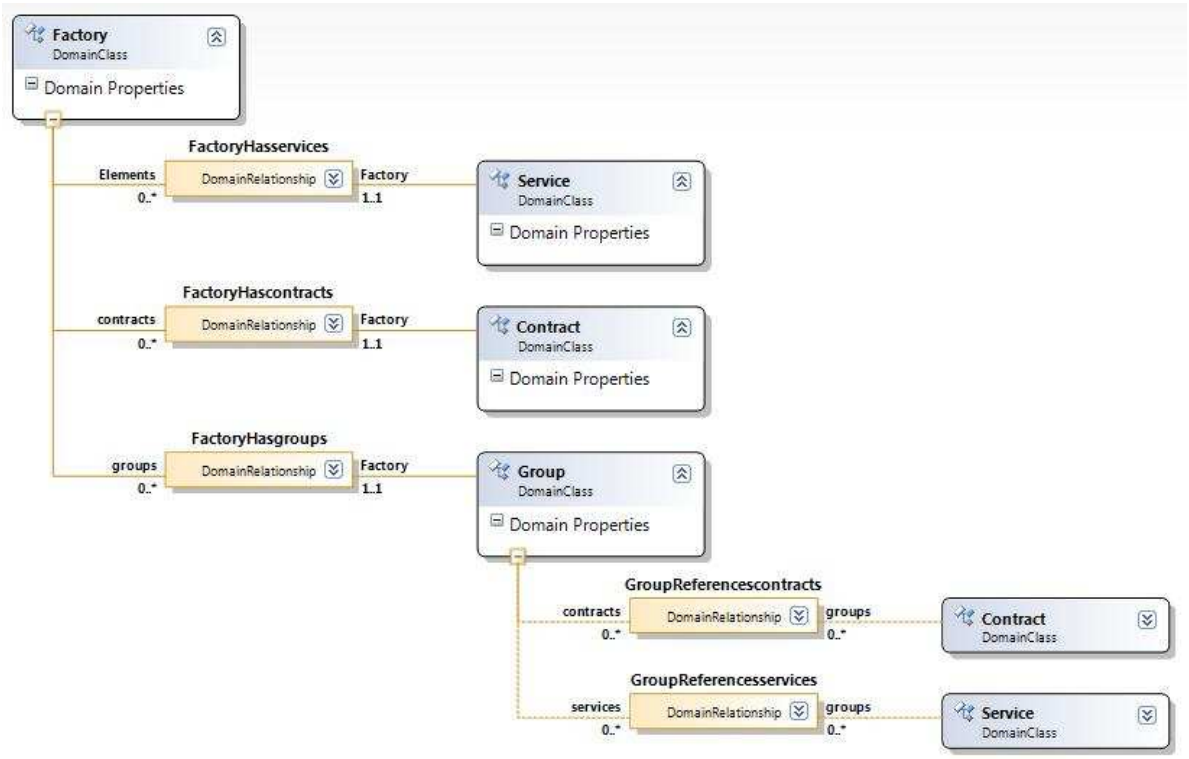


Figure 4.2: Domainclass Removal Source DSL

```
class Factory
class Service
class Contract
class Group
relation FactoryHasservices Factory Service true false Services
relation FactoryHascontracts Factory Contract true false Contracts
relation FactoryHasgroups Factory Group true false Groups
relation GroupReferencesservices Group Service false false Services
```

```
relation GroupReferencescontracts Group Contract false false Contracts
```

In the models based on the source DSL, the developer can connect an instance of Service and an instance of Contract with each other by connecting them to the same instance of Group. The semantic meaning of this is that a service consists of several contracts. For this scenario all non-embedding domainrelationships are assumed to be many-to-many domainrelationships. This means that a service can be connected to multiple groups and groups can have multiple services. The same holds for the contracts: contracts can be part of multiple groups and a group can have multiple contracts. The case of many-to-many domainrelationships is the most general case, any other type of domainrelationship is more restrictive. This is an issue that is related to the static semantics of the DSL definition and therefore out of the scope of this thesis.

An example model for the DSL shown in Fig. 4.2 can be seen in Fig. 4.3. The solid arrows are instances of embedding domainrelationships and the dashed arrows are instances of non-embedding domainrelationships. The domainobject Solution is an instance of the domainclass Factory. The domainobjects Service1 and Service2 are instances of the domainclass Service. Group1 and Group2 are instances of domainclass Group and Contract1 and Contract2 are instances of domainclass Contract.

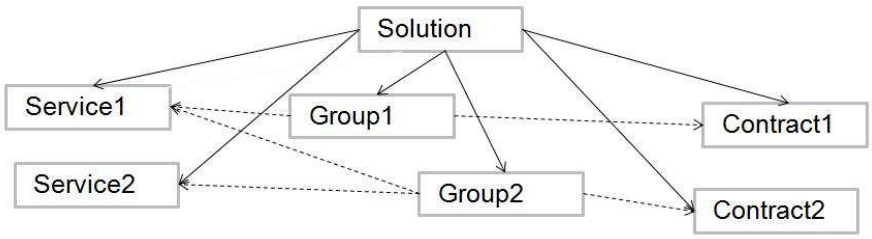


Figure 4.3: Example model for the domainclass removal source DSL definition

If, however, the domainclass Group is removed from the DSL definition, the model in Figure 4.2 would not be syntactically valid anymore. In the target DSL that is shown in Figure 4.4 the Group is removed and the Service is directly connected to the Contract.

The target DSL can also be written in the following way:

```
class Factory
class Service
class Contract
relation FactoryHasservices Factory Service true false Services
relation FactoryHascontracts Factory Contract true false Contracts
relation ServiceReferencescontracts Service Contract false false Contracts
```

In the target DSL, the contracts are directly connected to the services. So, for the conversion for each instance of a Service, the converter will have to look for all the instances of Contract that are in the same instances of Group as the service. In the source model, the direction of the domainrelationship between Service and Group is from Group to Service.

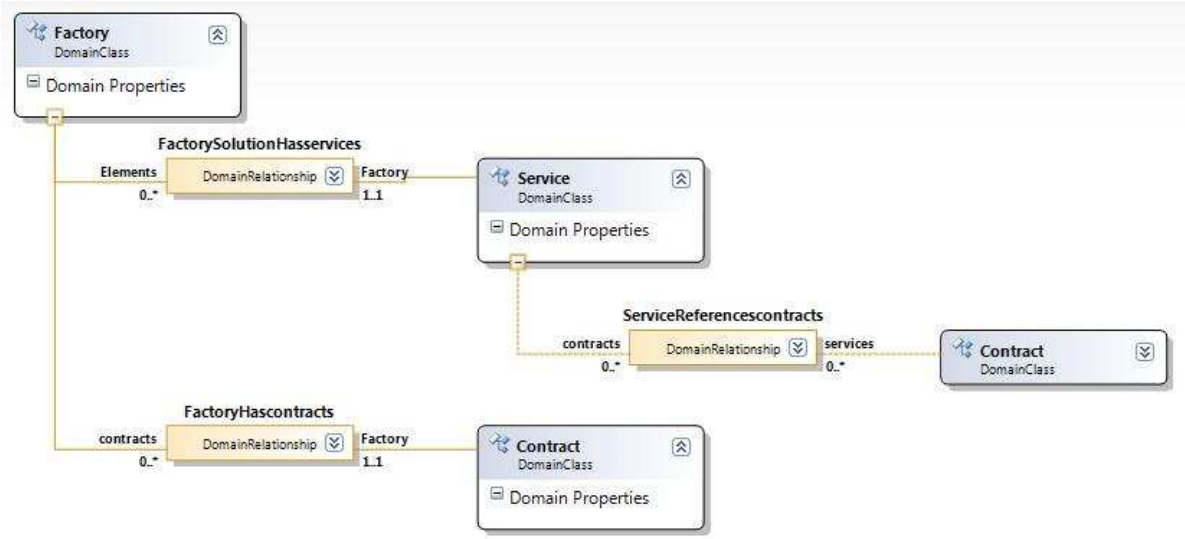


Figure 4.4: Entity Removal Target DSL

The same holds for the domainrelationship between Group and Contract: this domainrelationship is from Group to Contract. These two domainrelationships together will have to form the target domainrelationship between Service and Contract. However, the domainrelationship between Group and Service is pointing in the opposite direction. Using the transformation language defined in section 3.2, the DSL definition in Fig. 4.2 can be transformed to the DSL definition in Fig. 4.4 using the following transformation:

```
classmap Factory Factory
classmap Service Service
classmap Contract Contract
targetrelation FactoryHasservices Factory Service true false Services
  relation FactoryHasservices Factory Service true false Services
targetrelation FactoryHascontracts Factory Contract true false Contracts
  relation FactoryHascontracts Factory Contract true false Contracts
targetrelation ServiceReferencescontracts Service Contract false false Contracts
  relation GroupReferencesservices Group Service false false Services
  relation GroupReferencescontracts Group Contract false false Contracts
```

The model in Fig. 4.3 is not syntactically valid for the DSL definition of Fig. 4.4. To make this model syntactically valid, one could do several things. It is possible to remove all instances of domainrelationships that do not exist in the new DSL definition. In this case those are the domainrelationships GroupReferencesservices and GroupReferencesContracts. This makes the model syntactically correct, but not the complete transformation is reflected in this solution. It is also possible to add instances of the domainrelationships that have been added to the model. In this case the domainrelationship ServiceReferencescontracts is a new domainrelationship. As was described above, an instance of this domainre-

lationship needs to be added between every pair of Service and Contract instances that are connected to the same instance of Group. This can be seen in Fig. 4.5.

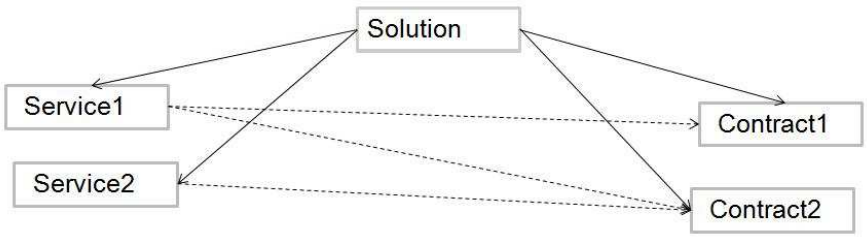


Figure 4.5: Example model for the class removal target DSL definition

For the general case of removing a domainclass, whether the conversion can be done is independent of the direction of the domainrelationship. It also does not matter how long the chain of domainclasses is between two domainclasses that have to be connected in the target model. In this scenario only the domainclass Group was in the middle, but there could be more.

4.4 Addition of a domainclass

One might think that the addition of a domainclass does not make the models of a certain DSL definition syntactically invalid. This is true for most cases, however, not for all cases. In the case in which a new domainclass is placed in between two existing domainclasses, the domainrelationships between those domainclasses might change. This is shown in this scenario.

This scenario comes from the development of a DSL for an insurance company. The source DSL offers the insurance company the feature to make questions and to define different options for this question to answer. However, the insurance company would like to reuse the list of options that is associated with a certain question. A new domainclass List is added to the DSL definition to support this requirement.

The source DSL is shown in Fig. 4.6. The QuestionModel has certain Questions and each Question has certain Options.

This language can also be written in the following form:

```
class QuestionModel
class Question
class Option
relation QuestionModelHasElements QuestionModel Question true false Elements
relation QuestionHasOptions Question Option true false Options
```

The target DSL is shown in Fig. 4.7. The domainclass List has been added to support the reuse of Options for other Questions. Every instance of Question has exactly one instance of List associated with it.

The DSL definition shown in Fig. 4.7 can also be written in the following form:

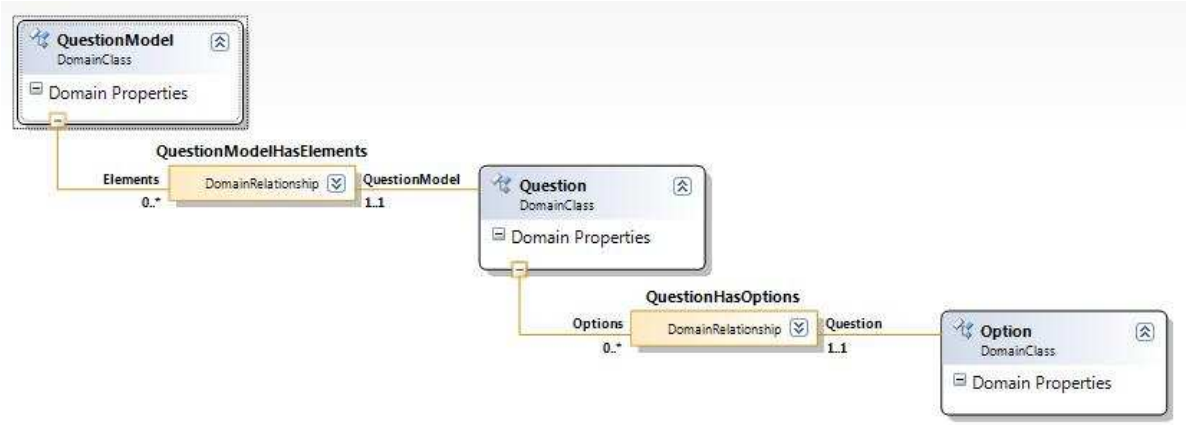


Figure 4.6: Domainclass Addition Source DSL

```
class QuestionModel
class Question
class List
class Option
relation QuestionModelHasElements QuestionModel Question true false Elements
relation QuestionModelHasLists QuestionModel List true false Lists
relation ListHasOptions List Option true false Options
relation QuestionReferenceslists Question List false false questions
```

To do the conversion of a model conforming to the source DSL, to a model conforming to the target DSL, for every instance of Question an extra instance of List needs to be created. Note that this is different from saying: for every instance of Option an extra instance of List needs to be created. However, we cannot tell which one of these two is necessary by just looking at the syntax definition of the DSL. This is something that can only be learned by looking at the semantics of a DSL. When transforming models we have to know which one to choose, so it has to be clear by looking at the transformation. This transformation is given below:

```
classmap QuestionModel QuestionModel
classmap Question Question
classmap Option Option
propertymap Question Id List Id
  relation QuestionHasLists Question List true false Lists
targetrelation QuestionModelHasElements QuestionModel Question true false Elements
  relation QuestionModelHasElements QuestionModel Question true false Elements
targetrelation QuestionModelHasLists QuestionModel List true false Lists
  relation QuestionModelHasElements QuestionModel Question true false Elements
targetrelation ListHasOptions List Option true false Options
  relation QuestionHasOptions Question Option true false Options
```

The decision on whether an instance of List needs to be created for every instance of Question or for every instance of Option is given using the propertymap. The domain-

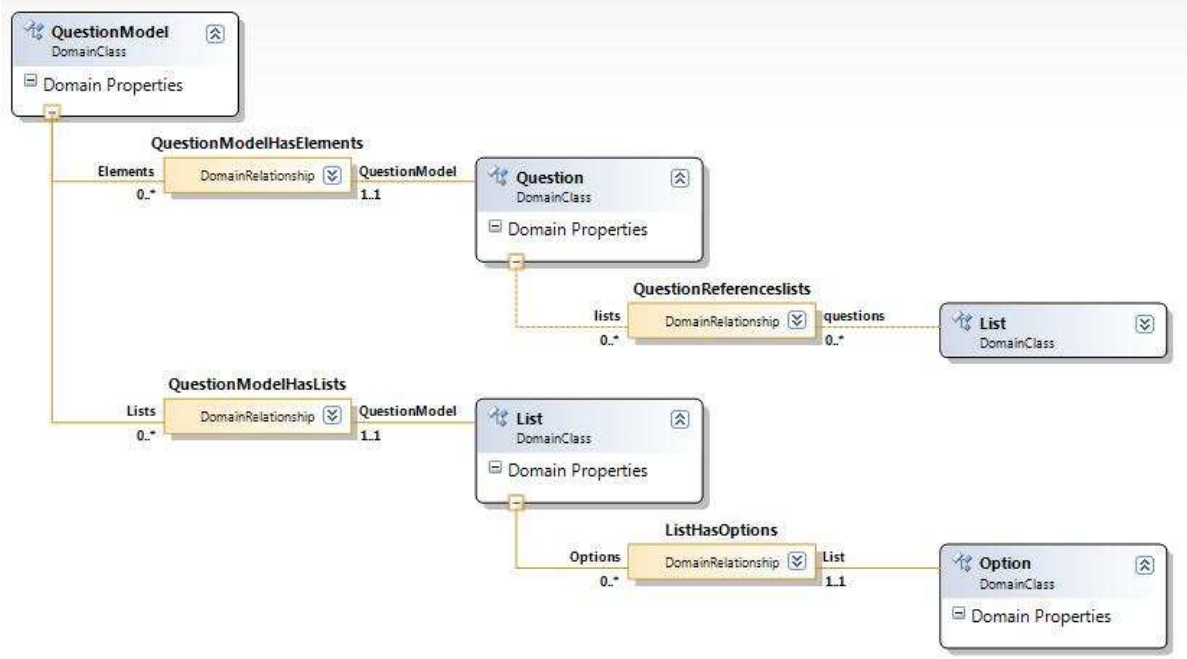


Figure 4.7: Entity Addition Target DSL

property Id is a domainproperty that every domainclass based on the DSL-Tools meta-meta Model contains. For every instance of Question the domainproperty Id is copied to a new instance of List. Furthermore, an instance of the domainrelationship QuestionHasLists is created for every time the property is copied.

Another way to see that we chose to create an instance of List for every instance of Question is by looking at the way the domainrelationships are transformed. An instance of the domainrelationship QuestionModelHasLists is created for every instance of the domainrelationship QuestionModelHasElements. The source domainclass of both domainrelationships is QuestionModel, but the target domainclass is different. This means that there should be the same amount of List instances as there are instances of the domainrelationship QuestionModelHasElements, because there is only one instance of QuestionModel allowed for every model and the domainrelationship QuestionModelHasElements does not allow duplicates.

4.5 Removing recursion

This scenario shows some limitations of the transformation language presented in section 3.2. In this scenario, the structure of the model changes. A domainrelationship is transformed into a domainproperty. So, instead of a domainproperty that maps to another domainproperty, a domainrelationship maps to a domainproperty. The developers of this

DSL decided to do this, because they thought this might be easier for the users of the DSL to understand the DSL.

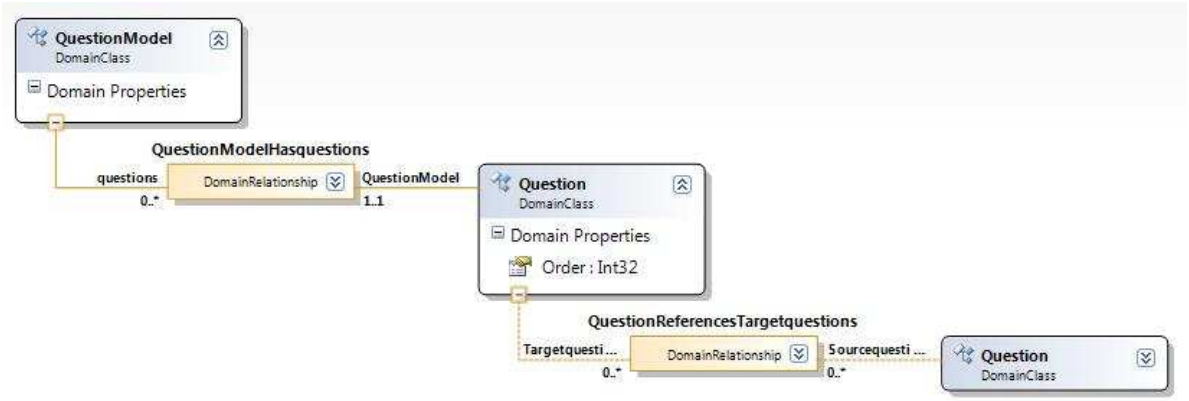


Figure 4.8: Source DSL in which recursion is allowed

The source DSL definition of this scenario is shown in Fig. 4.8. In this DSL questions can be added to a model and connected to each other. Every instance of Question has a domainproperty order which specifies in which order the questions should be asked to the user. When an instance of Question is connected to another instance of Question, it is a subquestion. This means that it will be shown in some special way when the artifact is generated. The chain of subquestions can be infinitely deep.

This DSL definition can also be written in the following form:

```
class QuestionModel
class Question
  property Order Int32
relation QuestionModelHasquestions QuestionModel Question true false questions
relation QuestionReferencesTargetquestions Question List false false Targetquestions
```

Because of the infinitely deep recursion of questions, the generator could loop quite long before it finds which question belongs to which other question. It might also be hard to figure this out for the developer, when the model becomes bigger. An example of such a model is shown in Fig. 4.9. One could imagine that this model will be hard to use once it becomes bigger.

The same DSL in which recursion is not allowed is shown in Fig. 4.10. In this DSL, the level of recursion is shown using a domainproperty called Level. The question to which another question is a subquestion, can be determined by looking at the order property of a question and the questions that have a lower order. This new DSL could also be written in the following form:

```
class QuestionModel
class Question
  property Order Int32
```

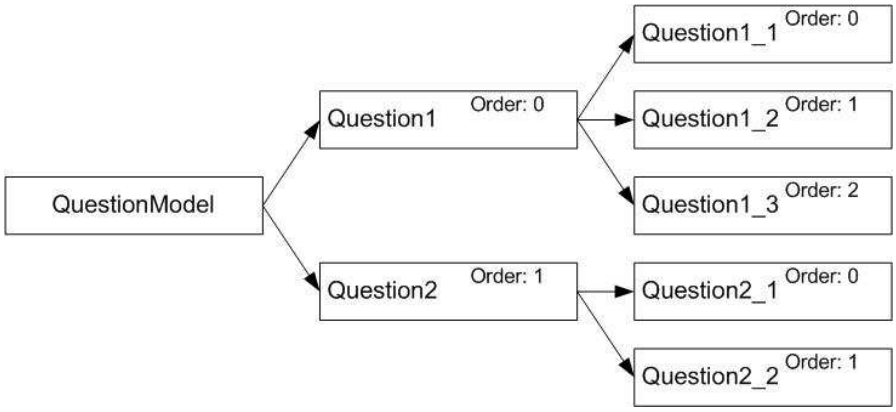


Figure 4.9: Model showing nested questions

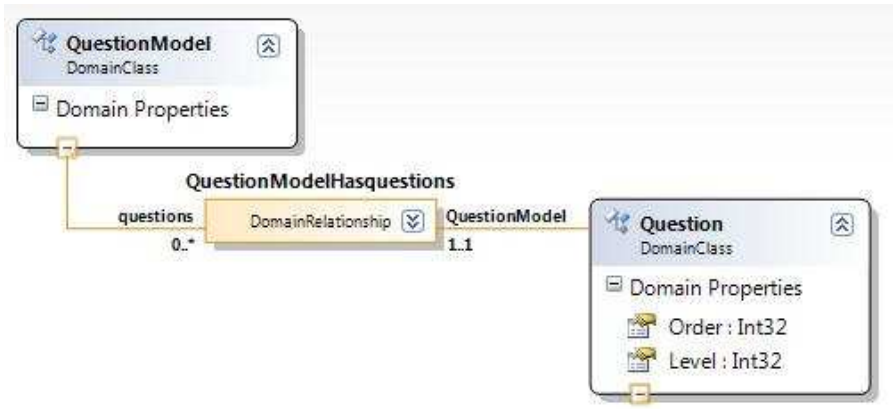


Figure 4.10: Target DSL in which recursion is not allowed

```
  property Level Int32
relation QuestionModelHasquestions QuestionModel Question true false questions
```

An example model for this DSL can be seen in Fig. 4.11. This model expresses exactly the same artifact as the model shown in Fig. 4.9. However, this model does not have the recursion as was shown in Fig. 4.9. This makes code generation easier and also improves the readability of the model for the developer.

It is not possible to create a transformation for this scenario using the transformation language presented in section 3.2. This is because this transformation is concerned with transforming domainrelationships into domainproperties. The domainproperty Level will be set to the value of the depth (how many domainrelationship instances) of an instance of a domainclass in a model. This is out of the scope of this thesis. This is further discussed in section 9.8. However, it is still possible to make a transformation for this scenario.

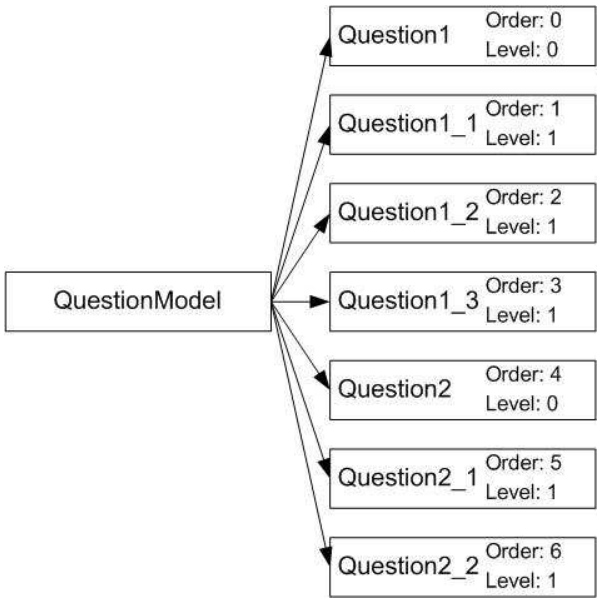


Figure 4.11: Model showing nested questions with Level domainproperty

The transformation will transform a model conforming to the source DSL definition to the target DSL definition. However, the transformation will not include enough information to transform the models in a satisfactory way for the developer. The developer will not consider the resulting models to be equivalent to the old models. The resulting models will be syntactically correct models, but the models will not include the same information as the models based on the source DSL definition.

Chapter 5

Automatic Detection Algorithm

The DSLCompare tool compares two DSL definitions (meta-models) that are different versions of each other. Therefore, it is possible to do some automatic recognition on how these DSL definitions map to each other. This is useful, because a user does not want to map all elements of two DSL definitions manually if the two DSL definitions are large. This would become a very time consuming task and also very error prone.

Usually the newer DSL definition has some features that the old DSL does not have. Furthermore, it might be the case that there are some structural changes to the new DSL definition to either integrate the new features or optimize the DSL definition. However, it might also be possible that some features have been removed. These notions form the basis of the Automatic Detection Algorithm.

A meta-model is a model and hence theories applicable to models can be applied to meta-models as well. Several approaches are already developed to automatically detect the difference between two models [5] [31]. However, these approaches are not based on the DSL-Tools meta-meta-model but on other meta-meta-models. They do not take into consideration the existence of domainclasses, domainproperties and domainrelationships. Furthermore, these approaches are only based on the universal unique identifier (UUID) of an element in the model [51]. A UUID is a unique identifier that is given to every element in the model. When changes occur to the element, the UUID stays the same. So, using the UUID, a mapping between the elements of the DSL definition can be created.

The approach presented here uses the approaches of [5] and [31], making them compatible with the DSL-Tools meta-meta-model and extending them by looking at more features of the model than just the UUID.

5.1 General overview

As was pointed out earlier, each DSL definition consists of three kinds of elements: domainclasses, domainproperties and domainrelationships. The first kind of element to look at are the domainclasses, because this kind is the only one that is always available in a DSL definition. One could make a DSL definition without any domainproperties or domainrelationships, but not without any domainclasses. Some domainclasses can be mapped

automatically, because they either have the same name, or they have the same UUID. Using the UUID, name changes in domainclasses can be detected. The domainclasses that cannot be mapped using this approach have to be mapped manually.

Domainproperties are associated with a particular domainclass. When the name or the type of the domainproperty changes, a mapping can be recognized because the UUID stays the same. However, when the domainproperty is moved to a different domainclass, it will get a different UUID as well. So this cannot be detected using the UUID of the domainproperty. By looking at the name of the domainproperty we can find out to which domainclass a domainproperty has moved. However, this is not trivial, because different domainclasses might have the same domainproperties, but with different meanings. So, we have to put some restrictions on when a moved domainproperty will be mapped automatically. These restrictions are discussed in the section 5.2. In some cases, it might turn out that these restrictions are too restrictive, however, the user is able to define a mapping manually in those cases.

The domainrelationships can change as well in a DSL definition. Domainrelationships also have a UUID, but this UUID only stays the same when the name of the domainrelationship is changed. For other changes the UUID changes and other heuristics are needed to recognize a mapping. These heuristics will be discussed in section 5.2 and are based on the results that were achieved when mapping the domainclasses and the domainproperties. For domainrelationships it is also possible to define a mapping manually in case the mapping was not recognized automatically.

5.2 Representing Models

In section 2.1 a framework for DSL versioning is introduced. Part of this framework is a tool, called DSLCompare, that can generate instances of the transformation language (described in section 3.2) automatically. As input the tool gets two different DSL definitions (M_1 and M_2) and the output of the tool is an instance of the transformation language that describes the mapping (Δ) between the two DSL definitions. To define this Δ we first need to define a model:

- A **Model** is a tuple of three sets: (Domainclasses, Domainproperties, Domainrelationships)
- The set **Domainclasses** is a set of tuples. Each tuple consists of a UUID and a name: (UUID,name)
- The set **Domainproperties** is a set of tuples. Each tuple consists of a UUID and a name: (UUID,name)
- The set **Domainrelationships** is a subset of the Cartesian product of domainclasses: Domainrelationships $\subseteq$ Domainclasses $\times$ Domainclasses . So, the set Domainrelationships is a set of tuples. Each tuple consists of two Domainclasses. The first domainclass is called the source domainclass, the second domainclass is called the target domainclass.

According to Sudarshan any change detection algorithm should consists of four steps: Map, Create, Delete and Change [13]. We will use the same four steps in this algorithm. However, this algorithm also consists of three phases: domainclasses, domainproperties and domainrelationships. So in total 12 operations are defined to define the transformation between M_1 and M_2.

Before we present the algorithm, some sets and functions need to be defined:

- C_x The set of domainclasses in M_x.
- P_x The set of domainproperties in M_x.
- R_x The set of domainrelationships in M_x.
- $uuid(\alpha)$: Gives the UUID of a certain instance in the DSL definition. This can either be an instance of a domainclass, domainproperty or domainrelationship.
- $name(\alpha)$: Gives the name of a certain instance in the DSL definition. This can either be an instance of a domainclass, domainproperty or domainrelationship.
- $class(\pi)$: Gives the domainclass to which the property π belongs.
- $source(\rho)$: Gives the source domainclass of the domainrelationship ρ.
- $target(\rho)$: Gives the target domainclass of the domainrelationship ρ.

5.3 Mapping Domainclasses

Domainclass Map This function maps all domainclasses with the same UUID. Usually these are domainclasses that have not changed at all or domainclasses of which only the name has changed. All other changes are covered in the *Domainclass Change* operation. The domainclasses c_1 represent the domainclasses of M_1 and the domainclasses c_2 represent the domainclasses from M_2.
$CM := \{(c_1, c_2) | c_1 \in C_1 \wedge c_2 \in C_2 \wedge uuid(c1) == uuid(c2)\}$

Domainclass Change This function represents all changes to domainclasses that are not represented in CM. These can be recognized by looking at whether the domainclasses have the same name. Otherwise, it is also possible to manually define a tuple (c_1, c_2). We call the set of tuples that are manually defined CChMan.
$CCh := (CCh_1 \cup CChMan) - CM$
$CCh_1 := \{(c_1, c_2) | c_1 \in C_1 \wedge c_2 \in C_2 \wedge name(c_1) == name(c_2)\}$

Domainclass Create This function represents all domainclasses that have been added to M_2. Note that there might be domainclasses that have been added to M_2 but are already in the set CCh, so they do not belong to this function. This occurs when one domainclass in M_1 maps to multiple domainclasses in M_2.
$CCr := \{c_2 | c_1 \in C_1 \wedge c_2 \in C_2 \wedge \neg\exists_{(c_1,x) \in CM \cup CCh}(x == c_2)\}$

Domainclass Delete This function represents all domainclasses that have been deleted from M_1. Note here as well that some domainclasses that are removed from M_1 are already in the set *CCh*, so they do not belong to this operation. This occurs when multiple domainclasses in M_1 map to one domainclass in M_2.
$CD := \{c | c \in C_1 : \neg\exists_{(x,c_2)\in CM\cup CCh}(x == c)\}$

5.4 Mapping Domainproperties

Domainproperty Map This function maps all domainproperties with the same UUID. Note that these domainproperties can only be domainproperties that stayed in the same domainclass, because otherwise their UUID would have changed. However, domainproperties still have the same UUID if they are in a class that is changed somehow.
$PM := \{(p_1,p_2) | p_1 \in P_1 \wedge p_2 \in P_2 \wedge uuid(p_1) == uuid(p_2) \wedge$
$\exists_{(c_1,c_2)\in CM\cup CCh}(class(p_1) == c_1 \wedge class(p_2) == c_2)\}$

Domainproperty Change This function represents all changes to domainproperties that are not represented in PM. These can be recognized by looking at whether the domainproperties have the same name. If they have the same name, they should also be part of the same domainclass. If this is not the case, we cannot be certain that the domainproperties represent exactly the same feature, because it could be that two different domainclasses have the same domainproperty, but with a different meaning. The set of tuples with domainproperties that have the same name and are in the same domainclass is called PCh_1.

However, in one case, we know that the domainproperties are the same, although they are in different domainclasses. This is when a domainclass is split into two domainclasses. Instead of having one domainclass with many domainproperties, the developer of the DSL definition might decide to split this in two domainclasses. A domainrelationship between the two domainclasses will be created as well in this case. We have chosen to only recognize this, if the domainclass in which the domainproperty resides in M_2 is a new domainclass and the domainproperty has the same name and does occur in PM. The reason to use so many contraints is because it might be the case that we recognize a mapping incorrectly. If we do not recognize a mapping automatically because of all these constraints, the user can define the mapping himself using the manual input. The domainproperties that are mapped in this way are collected in the set PCh_2.

All the domainproperties that cannot be mapped automatically, can be mapped manually by the developer in a tuple (p_1,p_2). The set of these tuples is called *PChMan*
$PCh := PCh_1 \cup PCh_2 \cup PChMan$
$PCh_1 := \{(p_1,p_2) | p_1 \in P_1 \wedge p_2 \in P_2 \wedge name(p_1) == name(p_2) \wedge$
$\exists_{(c_1,c_2)\in CM\cup CCh}(class(p_1) == c_1 \wedge class(p_2) == c_2)\}$
$PCh_2 := \{(p_1,p_2) | p_1 \in P_1 \wedge p_2 \in P_2 \wedge name(p_1) == name(p_2) \wedge$
$\exists_{r\in R_2,(c_1,c_2)\in CM\cup CCh,c_3\in CC}$
$(source(r) == c2 \wedge class(p_1) == c_1 \wedge target(r) == c_3 \wedge class(p_2) == c_3)\}$

Domainproperty Create This function represents all domainproperties that are added to M_2.
$PCr := \{p | p \in P_2 \wedge \neg\exists_{(p_1,x)\in PM\cup PCh}(x == p)\}$.

Domainproperty Delete This function represents all domainproperties that are removed from M_1.
$PD := \{p | p \in P_1 \wedge \neg\exists_{(x,p_2)\in PM\cup PCh}(x == p)\}$.

5.5 Mapping Domainrelationships

Domainrelationship Map This function maps all the domainrelationships that have the same UUID. Note that implicitely also the properties of the domainrelationships are mapped. In the first chapter, we defined that domainrelationships have properties like isEmbedding and allowsDuplicates. These properties are represented by boolean values that can change when the DSL definition evolves. Changes to these properties are mapped when the mapping between two domainrelationships is defined.

Not that the source domainrelationship is represented by a sequence with one domainrelationship, instead of just one domainrelationship. This will be explained in the *Domainrelationship Change* function.
$RM := \{([r_1], r_2) | r_1 \in R_1 \wedge r_2 \in R_2 \wedge uuid(r_1) == uuid(r_2)\}$

Domainrelationship Change This function represents all changes to domainrelationships that cannot be recognized using the UUID. For the domainclasses and the domainproperties, the only thing we could do automatically in this step was to match the names of the domainclasses and domainproperties. However, for the domainrelationships it is also possible to reuse some of the information we got in previous functions. This is the case when we know that a domainrelationship existed between two domainclasses and the domainrelationship between the domainclasses has changed in some way including both name and UUID changes. If this domainrelationship does not have the same UUID and the same name, we can recognize the mapping, because the domainrelationship is defined between two domainclasses in M_2that have a mapping with two domainclasses in M_1 between which the domainrelationship was defined.

However, the mapping is not necessarily between two domainrelationships. It might be the case that multiple source domainrelationships map in some sequence to one target domainrelationship. This happens for instance if a domainclass is deleted, like in the scenario presented in section 4.2. We have chosen not to recognize these changes automatically, because it could be very hard to be certain about this mapping. So, we have chosen to let the user specify this mapping manually in the set *RChMan*.
$RCh := (RCh_1 - RM) \cup RChMan$
$RCh_1 := \{([r_1], r_2) | r_1 \in R_1 \wedge r_2 \in R_2 \wedge$
$\exists_{(c_1,c_2),(c_3,c_4)\in CM\cup CCh}$

$(source(r_1) == c_1 \wedge source(r_2) == c_2 \wedge target(r_1) == c_3 \wedge target(r_2) == c_4)\}$

Domainrelationship Create This function represents all domainrelationships that have been created. Note that this does not include the domainrelationships that were used in the function *Domainproperty Change*. It is true that those domainrelationships were created in M_2, however they are already part of a mapping between M_1 and M_2.
$RCr := \{r | r \in R_2 \wedge \neg\exists_{(r_{1seq},x)\in RM\cup RCh}(x == r) \wedge$
$\neg\exists_{(p_1,p_2)\in PCh}(class(p_1) == source(r) \wedge class(p_2) == target(r))\}$

Domainrelationship Delete This function represents all domainrelationships that were removed from M_1. Note that removed means that the domainrelationship was not used in any of the mappings that was defined before. It does not matter whether it is the only domainrelationship in the mapping or that it is part of a sequence of domainrelationships in M_1. The set R_{1set} is defined as the set of all source domainrelationships that are involved in a mapping between M_1 and M_2.
$R_{1set} := \{r | r \in R_1 \wedge \exists_{(r_seq,r_2)\in RM\cup RCh}(\exists_{x\in r_{seq}}(x == r))\}$
$RD := \{r | r \in R_1 \wedge \neg\exists_{x\in R_{1set}}(x == r)\}$

All the functions defined above result in a set being defined. The result of all these functions is 12 sets that represent the mapping between M_1 and M_2. Note that this is not only the automatic detected part of the mapping, but also the part that was defined manually. The implementation of this automatic detection algorithm is discussed in section 7.1.

Chapter 6

From transformation to migration

This chapter describes how we get from a transformation on meta-models (Δ) to a migration on models (δ). Section 6.1 gives a short description of what the ConverterGenerator does. Section 6.2 describes two ways to implement the δ and gives the algorithm of how it was implemented in our approach.

6.1 Creating a migration on M1 from a transformation on M2

This section describes the ConverterGenerator as shown in Fig. 2.2. This tool is created to get a migration on models from a transformation on meta-models. Hence, the input of the tool is Δ and the output of the tool is δ.

Note that using this approach, it would also be possible to create an interpreter for the transformation language of section 3.2. However, an interpreter would not allow the developer to add custom code easily. The advantage of being able to add custom code is that the developer could add some code to create a transformation that also takes the semantics into account, for instance.

The ConverterGenerator generates a C# program (called δ) which does the migration on models. Note that the ConverterGenerator consists of code templates that are generic to any transformation on meta-models. How these code templates work is described in section 6.2 where the migration is discussed. How the ConverterGenerator is implemented is discussed in section 7.2.

6.2 The migration

The δ is generated from Δ. This is from a transformation on the level M2 to a migration on the level of M1 (see Fig. 1.1). This is useful for developers, because using this idea, they only have to define a transformation once. When the transformation Δ has been defined, the idea is that the transformation δ will work for any model *a* when the model *a* conforms to DSL *A*. Usually a DSL is used in multiple projects, meaning that multiple models based on the same DSL have been built. Only defining one transformation could possibly save a lot of time, compared to defining a migration for every single model.

The transformation from a transformation on M2 to a migration on M1 is based on the fact that every model created using the DSL-Tools meta-meta-model can be represented as an Abstract Syntax Graph (ASG) [14]. This ASG needs to be a connected ASG, with an explicitly defined spanning tree. In the research presented in this thesis, two approaches to create a migration on the M1 level have been considered:

- Keep the target model always a connected graph with explicitly defined spanning tree, during any stage of a model transformation.
- Allow the target model to be a disconnected graph at all stages, except for the end of the transformation.

Both approaches have turned out to work well. We have chosen the second approach in our implementation, because it gives more flexibility in case structural changes in the DSL definition have occurred.

6.2.1 Keep the target model a connected graph

The choice of the approach is important in transforming the transformation on the M2 level into a transformation on the M1 level. When the model b always needs to be a connected graph, the transformation on the M1 level should guarantee that the generated model is always connected in any state of the transformation. However, this turns out to be very restrictive to the transformation.

This approach starts at the root node and transforms every node of the graph using a depth-first search algorithm. A breadth-first search algorithm works in the same way, but is harder to implement. The algorithm starts at the root of the model. The root is always an instance of some domainclass of the DSL definition. Hence, this instance can be transformed using the transformation described in Δ. By looking at the ClassMaps, an instance of the target domainclass is created in model b that was at first the source domainclass of the model a. Also all domainproperties of the instance of this domainclass are converted to the instance of the new domainclass in model b.

The root of the model has instances of domainrelationships that connect the root to other instances of domainclasses. The instances of domainrelationships are both embedding and non-embedding. It is important to note here that it might be the case that there is a domainrelationship that is transformed from embedding to non-embedding or vice-versa. This does not matter for the model being a connected graph, however, it does matter for the explicitly defined spanning tree that should be in the model. If all non-embedding domainrelationships connected to the root of model a are converted to non-embedding, it is necessary to convert classes in an order that is determined by the transformation. Otherwise, the model will not maintain an explicitly defined spanning tree.

It is also possible that a domainrelationship is removed from the DSL definition. This could be, because two or more domainrelationships were merged into one domainrelationship as was described in section 4.3. It could also be, because some functionality has been removed from the DSL definition. In the latter case, we do not expect the new model to have the instance of this domainrelationship anymore. In the first case, however, we expect

an instance of the new domainrelationship to be present in the new model. These cases turn out to be hard to handle, because another object might have been converted already that cannot be connected to the spanning tree anymore. This object could have been in the middle of two instances of domainrelationships, as was the case in the scenario presented in section 4.3.

6.2.2 Allow the target model to be a disconnected graph at some stages

When the target model is allowed to be disconnected at some stages, it is not necessary anymore to mix the conversion of instances of domainclasses and instances of domainrelationships. The disadvantage of this approach, however, is that in the end of the algorithm the graph needs to get connected again.

The algorithm can be split into three stages:

- Transforming instances of domainclasses
- Transforming instances of domainrelationships
- Finding the root node of the new model

The ASG of every model conform to the DSL-Tools meta-meta model consists of a spanning tree. Using this spanning tree, a depth-first or breadth-first algorithm can be used to visit every instance of a domainclass exactly once. When an instance is visited, the domainclass of the instance is looked up in the ClassMap section of the transformation Δ. A new instance is created in model b of the type that is described in the ClassMap. The domainproperties are converted using the PropertyMap section of the transformation Δ. It might be possible that a new instance of another domainclass needs to be created, because the domainproperty is moved to another domainclass. This would also need the creation of an instance of a domainrelationship between these two domainclasses.

To transform the instances of the domainrelationships between instances of domainclasses, all instances of domainclasses are placed in a list. From here, the algorithm iterates over all possible domainrelations that exist in the new DSL definition. Note that the order in which the instances of domainclasses are converted, does not matter to the algorithm.

If the transformation is not very complex, one would end up with a model *b* that is a connected graph and has a spanning tree already defined. From this, it is not very hard to find the root of the spanning tree. One simply takes a random instance of a domainclass and walks up the spanning tree, until the end. This will give the root of the spanning tree.

However, it becomes harder when the graph is disconnected after all instances of domainrelationships have been converted. This happens when some structure in the DSL have become obsolete, but the domainclasses used in this structure are also used in other parts of the DSL that still exist. In this case, the roots of all disconnected parts of the graph have to be found. This can be done by finding the root for every instance of a domainclass in the model. This will result in a set of root nodes. By looking at the source model, we can see to which domainclass the root node of the sourcemodel conforms. The transformation Δ gives the transformation for this domainclass. If there is only one instance of this domainclass in

the set of root nodes, this is the root node. When there are more instances of this domainclass or no instance of this domainclass, it is not possible to tell which one is the root node with the current information. Custom code is necessary in this case.

6.2.3 The algorithm used in the migration

This section gives the basic steps of the algorithm used in δ in pseudo-code. The algorithm only performs reads on the source model, but reads and writes to the target model. This is in contrast with for instance ATL, which only does writes on the target model [9]. Before the algorithm is presented, first some definitions need to be made:

- The *sourcemodel* is the model we want to migrate
- The *targetmodel* is the model that is the result of the migration
- A model consists of *objects* (instances of domainclasses that contain domainproperties) and *relationships*
- An object has *properties* (instances of domainproperties)
- A property has a *name* and a *value*
- An object has a *class*, which is the domainclass in the meta-model that it conforms to
- A domainrelationship has a *source*, which is the object from which the domainrelationship originates
- A domainrelationship has a *target*, which is the object to which the domainrelationship points
- The *mapping* is the transformation that is defined by the developer in Δ
- A mapping consists of *classmaps*, *propertymaps* and *targetrelations*
- A classmap has a *sourceclass* which is a domainclass in the meta-model to which the sourcemodel conforms
- A classmap has a *targetclass* which is a domainclass in the meta-model to which the targetmodel conforms
- A propertymap has a *sourcename* which is the name of a domainproperty in the meta-model to which the sourcemodel conforms
- A propertymap has a *targetname* which is the name of a domainproperty in the meta-model to which the targetmodel conforms
- A propertymap has a *sourceclass* which is a class to which a domainproperty belongs. This domainclass is a domainclass in the meta-model to which the sourcemodel conforms.

- A propertymap has a *targetclass* which is a class to which a domainproperty belongs. This domainclass is a domainclass in the meta-model to which the targetmodel conforms.
- A targetrelation has *relations* which is a ordered sequence of domainrelationships.
- A targetrelation has a *sourceclass* which is a domainclass in the meta-model to which the sourcemodel conforms
- A targetrelation has a *targetclass* which is a domainclass in the meta-model to which the targetmodel conforms
- A targetrelation has a *relationcount* which is a number that tells how many domain-relationships the sequence domainrelationships contains
- A relationship has a *sourceclass* which is a domainclass in the meta-model to which the sourcemodel conforms
- A relationship has a *targetclass* which is a domainclass in the meta-model to which the targetmodel conforms
- The operation *:=* has as a left-value an object in the targetmodel and as a right-value an object in the sourcemodel. The object in the sourcemodel is used as an input. The operation looks up the migrated object in the targetmodel and assigns this to the left-value.
- The operation **=* is the equivalence relation between an object in the source model and the object that is migrated from that object in the targetmodel.

The algorithm we use in δ is the algorithm presented in section 6.2.2. The algorithm is divided into three stages:

- Convert all the objects in the model
- Convert all the relationships in the model
- Find the root of the model

Convert all the objects in the model. In this stage all objects are converted and also all properties. The case that the property is moved to a different class (section 4.4) is also handled in this stage of the algorithm.

```
1: foreach object in sourcemodel.objects
2:   find classmap with sourceclass == object.class
3:   create targetobject of type classmap.targetclass
4:   foreach propertymap in mapping with propertymap.sourceclass == object.class
5:     if propertymap.targetclass == targetobject.classname
6:       create new property in targetobject
7:       sourcename = propertymap.sourcename
8:       property.name = propertymap.targetname
```

```
9:       property.value = object.getProperty(oldname).value
10:    else
11:      if propertymap.relation!=null
12:        relationmap = propertymap.relation
13:        create new targetobject2 of type relation.targetrelation
14:        create new relation
15:        relation.source = targetobject
16:        relation.target = targetobject2
17:        create new property in targetobject2
18:        set name and value to the new property
19:      else
20:        find objects of propertymap.targetclass with relationship to object
21:        foreach obj in objects
22:        obj2 := obj
23:        create new property in obj2
24:        set name and value to the new property
```

The first line of this algorithm indicates that the algorithm is executed for all objects in the model. As was noted in chapter 3, the model can be represented as an Abstract Syntax Graph (ASG). In this ASG a spanning tree is defined. We use a breath-first algorithm to make sure that every object in the source model is visited exactly once.

Every object that is visited is an instance of some domainclass that is defined in the meta-model. Hence, we look up the ClassMap that is associated with this domainclass which is shown in the second line. This lookup can result in zero, one or multiple results. In the case of zero results, no ClassMap was defined for this domainclass, which means that the domainclass is deleted from the meta-model. Hence, no object would need to be created in the target model and line 3 can be skipped. Note that this does not necessarily mean that the domainproperties inside the domainclass are also removed. It could be the case that these are moved to other domainclasses. In the case of one result, the algorithm is continued and an object is instantiated in the targetmodel. In the case of multiple results, the first result is taken and an object is instantiated. Custom code is necessary to create some logic in order to make the right decision on which ClassMap to use.

Line 4 searches for all the domainproperties that can exist in the object that we are trying to migrate. A domainproperty could either be migrated to the object that was created in line 3 or to another object. That object might have been migrated already or might not have been migrated yet.

Line 5 until 9 represent the case in which the domainproperty stays in the same domainclass and hence is migrated to the targetobject. In this case we can create a new domainproperty in the targetobject with the new name and copy the old value into the new domainproperty.

Line 10 until 24 represent the case in which the domainproperty is moved to another domainclass. This might be a domainclass that did not exist yet or a domainclass that existed already. If the domainclass did not exist already, the PropertyMap will contain a domainrelationship that relates to the newly introduced domainclass. We can then create an instance (object) of this domainclass and add it to the target model. If the domainclass did exist al-

ready, we need to find the instances of this domainclass to which the domainproperty needs to be migrated. This is represented by lines 20 to 24. At the moment line 20 is implemented by only looking at the instances that are directly connected to the object. If another way of finding the instance is necessary, this can be created by adding custom code. In line 22 we take the converted object of the objects that were directly connected to object. So, obj is an instance in the sourcemodel, while obj2 is an instance in the targetmodel. The instance obj2 does not necessarily need to exist already. We solved this problem in the implementation by queuing the creation of these domainproperies until the obj2 is created.

Convert all the domainrelationships in the model. In this stage all domainrelationships are converted to the target model. Also the case in which multiple domainrelationships are merged in one single domainrelationship (section 4.3) is handled in this stage of the algorithm.

```
1: foreach targetrelationship in mapping
2:   foreach object in sourcemodel with targetrelationship.sourceclass *= object.class
3:     objlist = {object}
4:     for(int i=0;i<targetrelationship.relationcount; i++)
5:       sourcerelationship = targetrelationship.relationships[i]
6:       foreach obj in objlist
7:         if sourerelationship.sourceclass==obj.class
8:           objlist = obj(sourcerelationship)
9:         else
10:          objlist = find all objects with objects{sourcerelationship} == obj
11:    foreach obj in objlist
12:      create new relationship
13:      obj2 := object
14:      obj3 := obj
15:      relationship.source = obj2
16:      relationship.target = obj3
```

Note that this part of the algorithm is driven by the mapping, instead of by the sourcemodel. In the first line can be seen that we iterate over all the targetrelationships that were defined in the mapping.

The second line looks for objects in the targetmodel that might become the source of an instance of the targetrelationship. Note that it states *object in sourcemodel*, because we need the object in the sourcemodel. This object is used in lines 3 to 10. Here we move through the instances of domainrelationships in the sourcemodel until we find the objects which should be at the end of the domainrelationships. Note that this is a list of objects as indicated in line 3. It is a list, because one object can have multiple instances of domainrelationships to other objects.

Line 7 is concerned with whether the domainrelationship changes direction as is also the case in the scenario presented in section 4.3. Line 8 is executed when the domainrelationship maintains its original direction, while line 10 is executed when the domainrelationship changes direction.

The lines 11 until 16 are used to do the actual conversion of an instance of a domainrelationship. The objlist contains all objects in the sourcemodel to which the domainrelationships points that are available. The only thing that needs be done now is to lookup the

migrated objects of the objects in objlist and create instances of domainrelationships in the targetmodel.

Note that we have omitted the isEmbedding and allowsDuplicates properties of domainrelationships (see chapter 3) in the presentation of the algorithm here. However, they have been implemented in δ.

Find the root of the model. In this step we search the root of the targetmodel. Because we first migrate all objects, before we migrate all domainrelationships, it could become the case that we get a disconnected model. If this happens, we choose the model that contains the migrated object of the root of the sourcemodel. If this object does not exist, because the root is not migrated, we choose the biggest model.

Chapter 7

Implementation

This chapter discusses the implementation of the ideas presented in the previous chapters. Two tools have been presented, DSLCompare and ConverterGenerator. These tools are available to the reader upon request. Using the implementation discussed in this chapter, some case studies will be performed in chapter 8.

7.1 Implementation of DSLCompare

As was described in section 2.1 and Fig. 2.2, the DSLCompare tool takes two DSL definitions and displays these two DSLs to the user. The user is able to describe a mapping between the two DSLs. The user can do this in three different stages. First, the mapping between domainclasses is described. Then the mapping between domainproperties is described. In the end, the mapping between domainrelationships is described.

A DSL developed using Microsoft DSL-Tools is stored in an XML-format. Next to the abstract syntax, also information about concrete syntax and the tools that need to be generated for this DSL are stored in this XML file. So we have to disregard all this extra information and only look at the abstract syntax of the DSL.

The abstract syntax information of a DSL is stored in serialized form. This means that names could be different, however, this is not a problem because the serialized names have a one-to-one mapping with the names used in the abstract syntax. We use the serialized form in the transformation language as well, because this is also used when the models are stored. So, as an input to the convertergenerator, it is useful to have the serialized form, because the models that need to be converted are also stored in the serialized form. Note, that this implicates that the transformation language shows a change when the serialization of the abstract syntax has changed, not when the abstract syntax itself has changed.

The DSLCompare tool also has an implementation of the automatic detection algorithm. At every screen at which the user can describe a mapping, a button called autogenerate is added. When the user clicks this button, the automatic detection algorithm is invoked and the mappings are generated automatically.

The DSLCompare tool consists of three screens. Each screen represents one of the sets that need to defined manually in the automatic detection algorithm. These are the sets

CChMan, PChMan and RChMan, corresponding to domainclasses, domainproperties and domainrelationships as was described in chapter 5. Note that there is interaction between the screens. For instance the set RCh depends on the set CChMan, as was described in section 5.5. So, information of the first screen of DSLCompare is used in the third screen of DSLCompare.

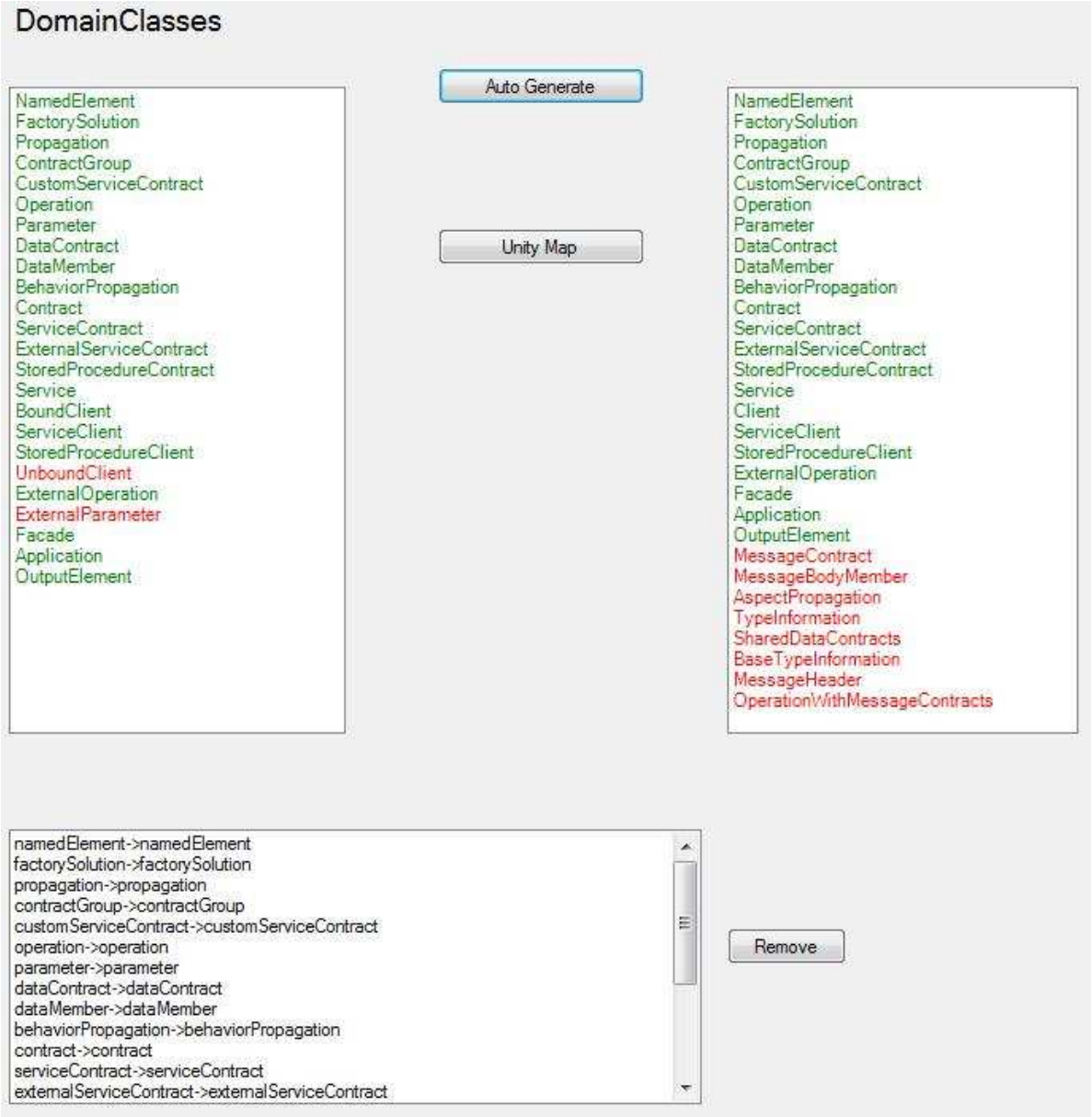


Figure 7.1: Domainclasses in DSLCompare

It is possible for a developer to see which domainclasses are already mapped and which domainclasses still need to be mapped. This is indicated to the developer by showing the different entities (domainclass, domainproperty or domainrelationship) in either a red or a green color. Green means that the entity has been mapped already. Red means that the entity has not been mapped yet. Furthermore, the current mappings can also be seen at the bottom of the screen.

7.1.1 Mapping domainclasses

The first screen of DSLCompare is shown in Fig. 7.1. This screen shows the mapping of the domainclasses. When the user presses the auto generate button, the mappings are generated automatically. This corresponds with the automatic detection for domainclasses as was discussed in section 5.3.

The set CChMan can be defined by the user with the unity map button. The user can select a domainclass from the old DSL definition and a domainclass from the new DSL definition in order to map these two domainclasses. Note that it is possible that multiple domainclasses of the source DSL definition map to the same domainclass in the target DSL definition. However, the other way around is not possible, because the migration on models (δ) would not know to which domainclass an instance of the old domainclass should be converted. If a migration has this requirement, it can be implemented with the human help on the ConverterGenerator as was discussed in section 2.1.

The remove button is used to remove a mapping on classes. This could either be a mapping that was automatically generated or a mapping that was manually added.

7.1.2 Mapping domainproperties

Fig. 7.2 shows the screen for mapping domainproperties. This screen looks similar to the screen to map the domainclasses. However, the names of the domainproperties are presented differently. First the name of the domainclass is presented after which the name of the domainproperty is presented. This helps the user to find domainproperties with the same names, but in different classes.

The auto generate button generates the mappings according the the automatic detection algorithm presented in section 5.4. Note that this automatic detection algorithm uses information from the domainclasses screen as well to create the PCh set.

It is possible to define a mapping between two domainproperties, even if the domainproperty has moved to a different domainclass. However, this is restricted to a domainproperty that is in a domainclass that is directly connected to the domainclass in which the domainproperty resided in the source DSL definition. DSLCompare checks this when the user clicks the map button.

If the domainproperty stayed in the same domainclass, DSLCompare generates a propertymap as was described in segment If the domainproperty has moved to a different domainclass, DSLCompare generates a propertymap with a relation, as was described in 3.7 of chapter 3. If the domainpropery has moved to a different domainclass, a propertymap together with a relation is generated as was described in segment 3.8.

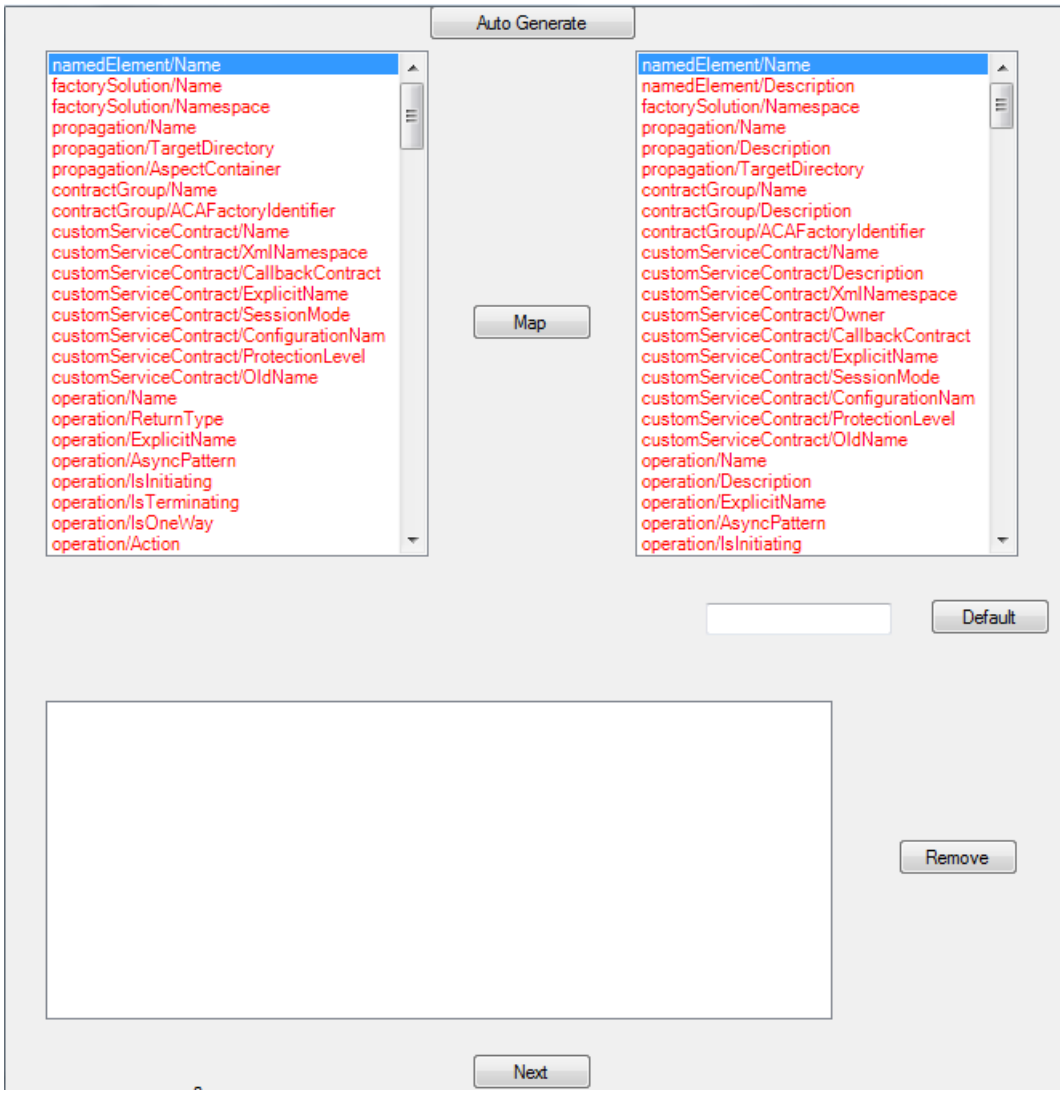


Figure 7.2: Domainproperties in DSLCompare

The button default allows the user to specify a default value for a domainproperty. This is used when the domainproperty is created inside an already existing domainclass.

7.1.3 Mapping domainrelationships

Fig. 7.3 shows the screen for mapping domainrelationships. This screen looks different compared to the previous screens. The domainrelationships are shown used the domainclass name of the source of the domainrelationship, followed by the name of the domainrelationship, followed by the domainclass name of the target domainclass of the domainrela-

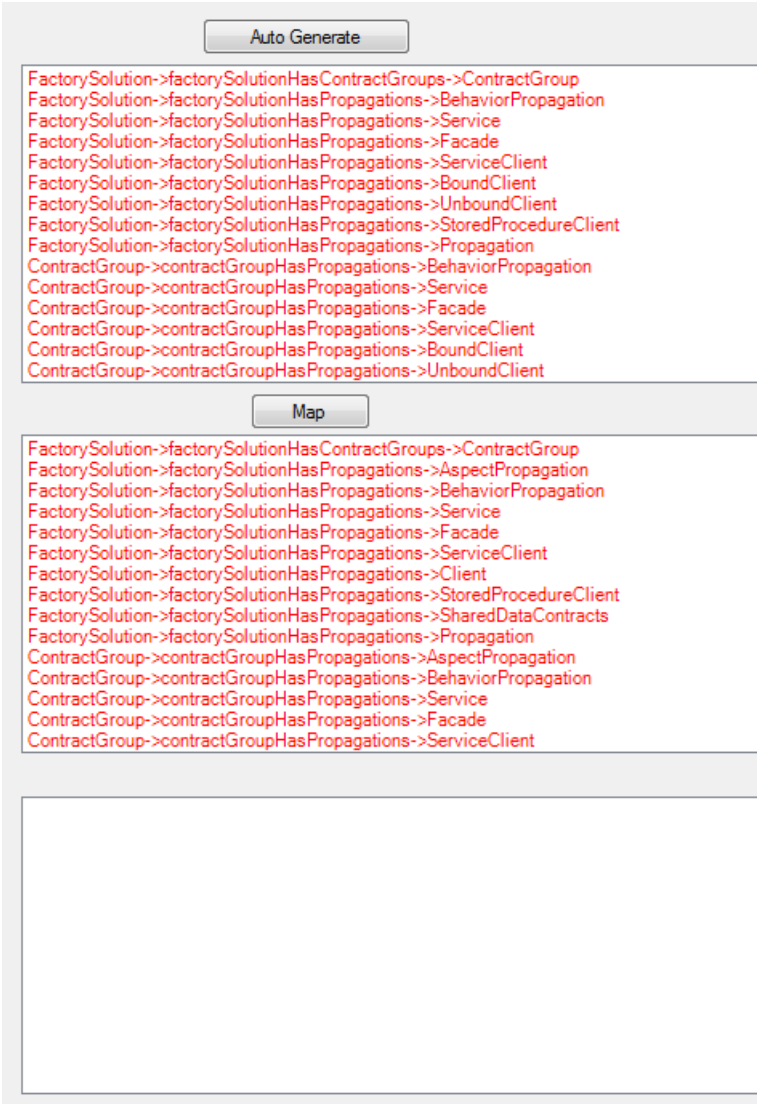


Figure 7.3: Domainrelationships in DSLCompare

tionship. Note that domainrelationships can only be mapped to each other, if they share the same source domainclass and same target domainclass.

However, it is possible to map multiple domainrelationship in the source dsl definition to one domainrelationship in the target dsl definition. The user can select multiple domainrelationships of the source dsl definition, one of the target dsl definition and click on map. DSLCompare checks whether the domainrelationships form a chain of domainrelationships. So, whether the target domainclass of one domainrelations is equal to the source domainclass of another domainrelationship. Note that it is also possible that the source domainclasses of two domainrelationships are equal, if the direction of the domainrelationship

was changed. This was illustrated in section 4.3.

7.2 Implementation of ConverterGenerator

Figure 7.4: Example of a model converter

The ConverterGenerator generates a converter on the model level from a transformation on the meta-model level. The generator takes the transformation on meta-model level as an input (Δ in Fig. 2.2). The output is a program, written in C# that is able to convert models. In Fig. 2.2 this program is called δ.

To generate the C# code for δ, a class library called stringtemplate has been used [1]. This library allows the developer to easily inject text into text files. The text files consist of C# code and are used as code templates. The injected text comes from Δ, this transformation is injected as C# code into the program.

Simply stated, the ConverterGenerator is 1468 lines of C# code that are copied to δ. The only purpose of the ConverterGenerator is to integrate the Δ into the C# code. The C# code already consists of three vectors. So, every ClassMap, PropertyMap or RelationMap as described in section 3.2 is converted into C# code.

An example of how Δ is integrated into to C# code is shown below. The transformation from section 4.1 has been taken. Note that the ClassMap and PropertyMap sections look very similar to the given transformation. The RelationMap is different, because the domainrelationships of the old DSL definition are shown before the domainrelationships in the new DSL definition. In the transformation language (section 3.2) this is shown the other way around.

```
1: Mapping result = new Mapping();
2: result.AddClassMapping(new ClassMap("Game","Game"));
3: result.AddClassMapping(new ClassMap("Room","Room"));
4: result.AddClassMapping(new ClassMap("Entity","Entity"));
5: result.AddClassMapping(new ClassMap("MainCharacter","MainCharacter"));
6: result.AddClassMapping(new ClassMap("NPC","NPC"));
7: result.AddPropertyMapping(new PropertyMap("Room","width","Room","width"));
8: result.AddPropertyMapping(new PropertyMap("Room","height","Room","height"));
9: result.AddPropertyMapping(new PropertyMap("Entity","position","Entity","position"));
```

```
10: result.AddPropertyMapping(
          new PropertyMap("MainCharacter","position","MainCharacter","position"));
11: result.AddPropertyMapping(new PropertyMap("NPC","position","NPC","position"));
12: List<Relation> sourceRelations = null;
13: sourceRelations = new List<Relation>();
14: sourceRelations.Add(new Relation("GameHasRooms","Game","Room",true,false,Entities));
15: result.AddTargetRelation(
          new Relation("GameHasRooms","Game","Room",true,false,"Entities",sourceRelations));
16: sourceRelations = new List<Relation>();
17: sourceRelations.Add(
          new Relation("GameHasEntities","Game","MainCharacter",true,false,"Entities"));
18: result.AddTargetRelation(
          new Relation("GameHasEntities","Game","MainCharacter",true,false,"Entities"));
19: sourceRelations = new List<Relation>();
20: sourceRelations.Add(
          new Relation("RoomReferencesTargetRooms","Room","Room",false,false,"TargetRooms"));
21: result.AddTargetRelation(
          new Relation("RoomReferencesTargetRooms","Room","Room",false,false,"TargetRooms"));
22: sourceRelations = new List<Relation>();
23: sourceRelations.Add(
          new Relation("RoomReferencesEntities","Room","Entity",false,false,"Entities"));
24: result.AddTargetRelation(
          new Relation("RoomReferencesEntities","Room","Entity",false,false,"Entities"));
```

The result of the ConverterGenerator is a Visual Studio solution (this is actually δ). This solution can be compiled using Visual Studio and gives the developer an executable. When starting the executable, the user is given a field in which he can select a model. Furthermore, a convert button is shown. When selecting a model and clicking *Convert* the model is converted to a model based on the new version of the DSL. A screenshot of a model converter can be seen in Fig. 7.4.

Note that generating C# code for δ gives the user the ability to add some custom C# code to the δ. This is shown as human help in Fig. 2.2. The custom C# code can be added using the double derived pattern, which is also used in Microsoft DSL-Tools [14]. This pattern uses the partial classes of the C# language. The generated code resides in a separate C# file than the custom C# code. In this way the custom code will not be overwritten when the user regenerates the converter.

Chapter 8

Evaluation

To evaluate the solution proposed in this thesis, three case studies have been performed. The first case study is about a DSL that Avanade uses internally to see whether the proposed solution works for Avanade. The second case study is about the evolution of Microsoft Web Service Software Factory Modeling Edition [2]. We use this case study to see whether the solution would also work for other DSLs. Furthermore, people have built converters between different versions of this DSL already, so we can also see whether we are able to create a converter faster using the solution proposed in this thesis. The third case study is about two different DSLs that are in the same domain. This is not a real evolution scenario, because the two versions of the DSL have been developed independently. However, this case study is very interesting, because most complex problems in DSL evolution using Microsoft DSL-Tools can be found here. Futhermore, this case study can be used to find the boundaries of the solution proposed in this thesis.

The first section of this chapter explains the experimental design that is the same for every case study. The further sections discuss each case study. Each of these sections is divided into four different parts. The first part discusses the background of the case study and the DSLs that are used. The second part will present the questions we are going to answer with the case study. The third part will present the case study itself and the fourth part analyzes the results and answers the questions asked in the second part.

8.1 Experimental Design

For every case study, we will use two versions of a DSL. We also have a real-life model conform the old DSL definition. In every case study we will perform several steps:

- Create a transformation from the old DSL to the new DSL.
- Create a migration on models from the transformation on DSLs.
- Migrate a real-life model.

Create a transformation from the old DSL to the new DSL. We will use the DSLCompare tool to create a transformation from the old DSL to the new DSL. We present a ta-

ble with information about this transformation. The table shows how many mappings are mapped automatically and how many needed to be mapped manually.

An example of such a table is shown in Table 8.1. This table is at first seperated in three sections: domainclasses, domainproperties and domainrelationships, because these are mapped seperately. For each of these sections, we look at the number of direct and indirect mappings. Direct mappings mean that nothing changed to the domainclass, domainproperty or domainrelationship. Indirect mappings mean that something changed. This could be the name, the type, or anything else. Furthermore, we also distinguish between introduced and deleted elements. The introduced field indicates how many domainclasses, domainproperties or domainrelationships have been added to the new DSL. The deleted field indicates how many domainclasses, domainproperties or domainrelationships have been removed from the old DSL to get the new DSL.

Note that domainproperties and domainrelationships are counted multiple times when multiple domainclasses inherit a domainproperty or domainrelationship from a base class. This is because the mapping needs to be described multiple times in the transformation language as well. This was shown for example in section 4.1.

Create a migration on models from the transformation on DSLs. Once we have created the transformation between the two DSLs, we will create a migration for the models conform the source DSL. This is done by using the ConverterGenerator application. This application will generate the δ as was described in chapter 6. We will look at whether we need additional C# code to support the migration.

Migrate a real-life model. For every case study we have a model conform the old DSL. This model will be migrated to be conform to the new DSL. We ask the developers of the old model to check whether the migrated model is the same as the model they developed conform the old DSL definition. We ask developers, because we do not have a formal definition of when two models are equivalent when they are conform different DSL definitions.

8.2 Case Study 1

8.2.1 Background

For this case study we use a DSL that has been created for the Service Oriented Architecture (SOA) domain. Two different versions of the DSL are used. The new version (Release 1.0) has been released about one year after the old version (Beta 2). Note that this case study has been anonymized, because of intellectual property rights. We also use a model from a real-life project as a test case for this case study. The model can be seen in Fig. 8.1, to give the reader an idea about the size of the model. The DSL can be used to simplify creating SOA architectures based on WCF (Windows Communication Foundation [38]).

The model shows several datacontracts like Address and BankAccount. When the developer would click on Address, the model will show what an Address is: a street, a number, a zip code, a city and a country. The model also shows a servicecontract called IBackendContract with serveral operations. When the developer clicks on an operation, he gets the parameters and return type of an operation. These could either be primitive types or dat-

acontracts. Furthermore, some blocks called Service and ServiceAgents are shown in the model. These determine how the created webservice will be hosted.

The two different versions of the DSL differ in many ways. Some examples are:

- Some domainclasses are merged together. For example, in Beta 2, a class BoundCliend and a class UnboundClient existed. In Release 1.0, these two classes are merged together to become a class called Client.
- Some domainproperties in domainclasses are moved to different domainclasses. In Beta 2 several domainclasses had some domainproperties about types. In Release 1.0 these domainproperties are moved out of the domainclasses and put into a single domainclass called TypeInformation. All those domainclasses have a reference domainrelationship to the domainclass TypeInformation.
- In some cases it was allowed in Beta 2 to have objects reference to other objects multiple times. For instance, an instance of the domainclass Service was allowed to be part of the same instance of ContractGroup multiple times, without any additional meaning. In Release 1.0 this was not allowed anymore.

8.2.2 Questions

We are going to answer the following questions with this case study:

- How well does the automatic detection algorithm in DSLCompare work for this particular DSL? What percentage of a DSL definition can we automatically map to another DSL definition?
- How complete is the migration that we generate from the mapping defined in DSLCompare? How many lines of custom code do we have to add to complete the migration between models?
- Does the transformation work on a real-life model that was built using this DSL?

8.2.3 The case study

The first step to build the transformation between Beta 2 and Release 1.0 is to define the transformation between the two DSL definitions. When building this transformation, we can answer the first question that we are going to answer during this case study. To measure how many mappings we can do automatically, we first count the number of domainclasses, domainproperties and domainrelationships in each DSL definition. We then look at how many domainclasses, domainproperties and domainrelationships are mapped automatically. After that, we look at why the others were not mapped automatically. This could have two reasons: the mapping is too complex to recognize for the automatic detection algorithm, or the domainclass, domainproperty or domainrelationship was deleted from the old DSL definition or introduced in the new DSL definition.

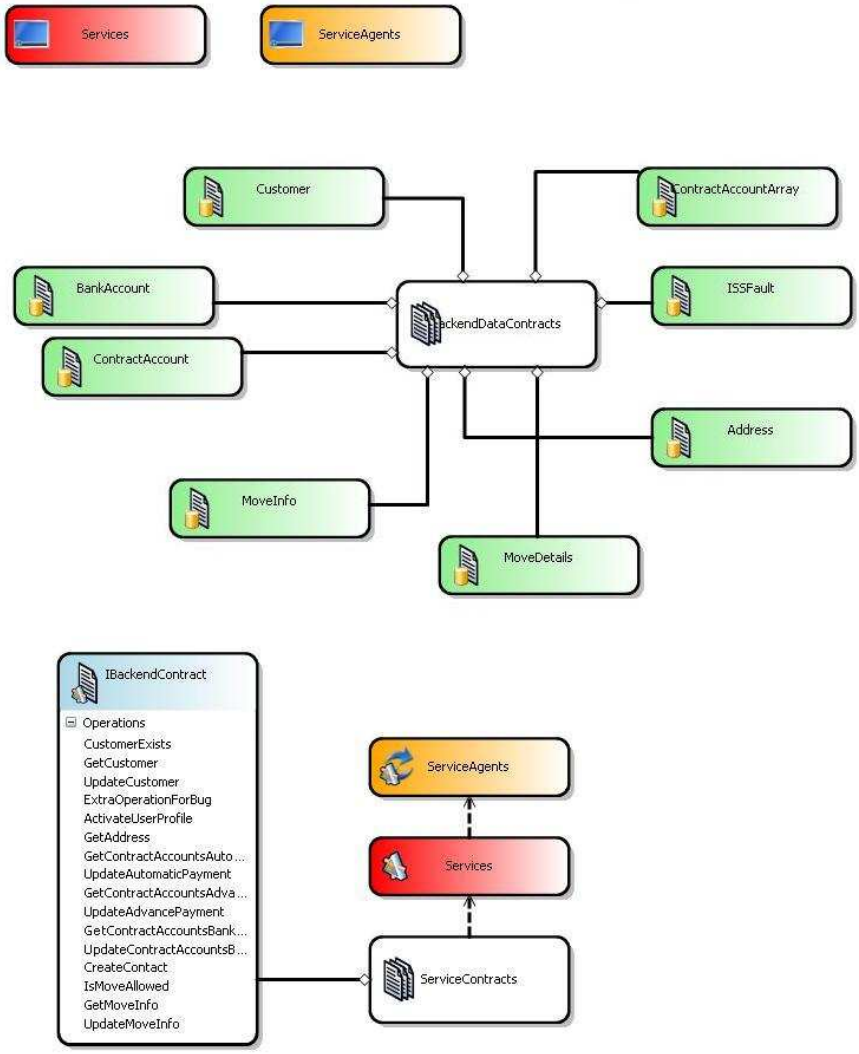


Figure 8.1: Model in Beta 2

The measurement for the mapping between Beta 2 and Release 1.0 is shown in Table 8.1 Direct mappings mean mappings between domainclasses, domainproperties or domainrelationships that have not changed (i.e. with the same UUID) as was discussed in section 8.1. The table shows that these can all be detected automatically by the DSLCompare tool.

Indirect mappings mean that the domainclasses, domainproperties or domainrelationships have changed in some way. These changes can be name changes, but it can also be merges or splits of domainclasses, domainproperties or domainrelationships. When a merge or split occurs, the number in the column Beta 2 will be different from the column Release 1.0. For instance, the domainclass merge of BoundClient and UnboundClient in Beta 2 to Client in Release 1.0 could not be recognized completely automatically by the DSLCom-

			Beta 2	Release 1.0
Domainclasses	Direct	auto	21	21
		manual	0	0
	Indirect	auto	1	1
		manual	1	0
	Deleted		1	0
	Introduced		0	8
	Total		24	30
Domainproperties	Direct	auto	128	128
		manual	0	0
	Indirect	auto	15	15
		manual	3	1
	Deleted		4	0
	Introduced		0	74
	Total		150	218
Domainrelationships	Direct	auto	80	80
		manual	0	0
	Indirect	auto	23	19
		manual	0	0
	Deleted		12	0
	Introduced		0	56
	Total		115	155

Table 8.1: Comparison and automatic mapping of DSLCompare between Beta 2 and Release 1.0

pare tool. The DSLCompare tool has recognized the mapping between BoundClient and Client, so the domainclass BoundClient is mapped automatically. This is represented by the 1 for indirect auto in Beta 2. The domainclass Client is also automatically mapped, this is represented by the 1 in indirect auto release 1.0. However, the domainclass UnboundClient is still not mapped and we had to map this domainclass manually. This is represented as the 1 in indirect manual Beta 2.

Table 8.1 shows that only one of the 24 domainclasses of Beta 2 needed to be mapped manually. The rest was deleted in Release 1.0 or was mapped automatically. For the domainproperties, only three of the 150 domainproperties needed to be mapped manually. For the domainrelationships, none of them needed to be mapped manually. So, in total only 4 entities (domainclasses, domainproperties and domainrelationships together) out of 289 entities needed to be mapped manually. This is only 1.4% of the DSL. The rest can all be done automatically.

Using the transformation defined using the DSLCompare tool as described in Table 8.1 a converter has been generated to convert models between Beta 2 and Release 1.0. The converter still needs 5 lines of custom-code (human help) to do the conversion of the type information in the right way. This custom code has been added using the doube derived

pattern as was discussed in section 7.2. We have tested this using various small test-models that were created. In these test models, all domainclasses have at least been instantiated once.

To test the converter, a model from a real-life project has been taken. This model is shown in Fig. 8.1 and was explained in section 8.2.1. Note that this model has been slightly changed to anonymize it. This model is used as an input for the converter we just created. The output of the converter is a model that can be seen in Release 1.0 and is syntactically valid to the DSL definition of Release 1.0. This shows that the approach proposed in this thesis works for a random real-life project.

The model is also semantically equivalent to the model of Beta 2. However, this is not because of our tool, but simply because the semantical meaning of the elements in the DSL definition did not change. The domain of this DSL is still the same: SOA architures. Also the meaning of data contracts, service contracts and other entities in the SOA domain did not change. Release 1.0 is simply a new version of Beta 2 that has some extra features and some refactorings.

The code generated from the model in Release 1.0 is not interface compatible with the code generated from the model in Beta 2. This means that some changes are necessary to the custom code developers made next to the model, in order to make the artifact work. This problem is out of the scope of this thesis.

8.2.4 Conclusions

Using this case study, we answered the three questions that were asked in section 8.2.2:

- We showed that the automatic detection algorithm works reasonably well for these two versions of the DSL. Only 1.4% of the DSL needed to be mapped manually. The rest has been done automatically.
- We only needed to add 5 lines of code to the code that was generated by the ConverterGenerator to get a migration that converts all our test-models.
- We tried the migration on a real-life project. The developers of that project confirmed that the migrated model was the model that they would have created if they would have used the new version of the DSL.

8.3 Case Study 2

8.3.1 Background

Microsoft worked for about a year on Web Service Software Factory Modeling Edition (WSSF), before they released the final version in November 2007. WSSF consists of three different DSLs: datacontract DSL, servicecontract DSL and host DSL. One needs at least one instance of every DSL to create a working application. Hence, for one application, one has at least three models. A more thorough description of Microsoft Web Service Software Factory Modeling Edition can be found in [16].

Even in the last two months before the final release of WSSF, radical changes have been made to the DSL definitions. Developers were using the last beta version of this software factory (b117) when the final release came out and no one was able to open his models anymore. This problem is shown graphically in Fig. 8.2 and Fig. 8.3 for the datacontract DSL.

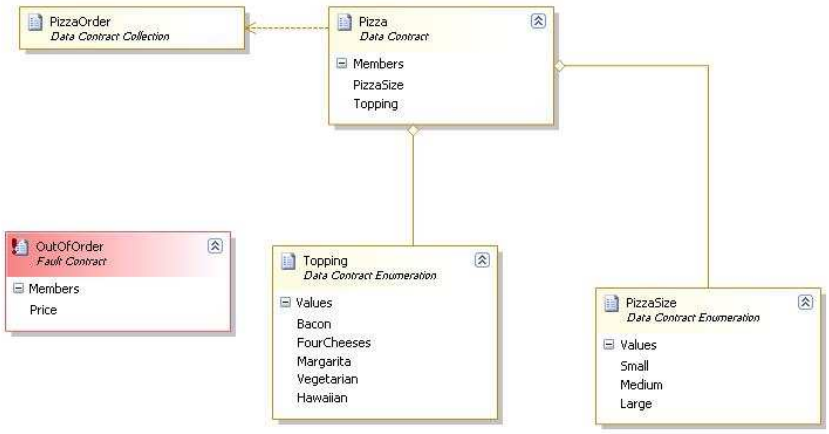


Figure 8.2: Datacontract model in WSSF b117

Figure 8.3: Datacontract model in WSSF final

As can be seen, the model is not visible in the final version of WSSF. This is very inconvenient when maintenance to the project is needed. It was also inconvenient when developing software using Web Service Software Factory at this time. Dennis Doomen of Aviva Solutions, for instance, experienced this problem. He had developed about 12 different models using the b117 version and was not able to work on them anymore using the final version of WSSF [18]. It took him about a week to analyze all the changes that were made to the three DSL definitions in Web Service Software Factory and to build a converter for his models. Also help from the developers of WSSF was necessary to create the converter.

8.3.2 Questions

We answer two questions with this case study:

- Does the solution in this thesis also work for multiple smaller DSL definitions and models conform these DSL definitions?
- How does the time needed to construct a converter by hand compare to the amount of thime needed when using our approach? In particular, can we create a converter for the models conform WSSF beta 117 to models conform WSSF final faster than in one week? In the future many DSLs will exist [49] and they will all evolve. It is very inconvient to spend a week every time to build a converter between two versions of a DSL definition.

8.3.3 The case study

WSSF consists of three different DSLs. These are the *datacontract* DSL, the *servicecontract* DSL and the *host* DSL [16]. We will define a transformation for these DSLs seperately, starting with the datacontract DSL. After that we will discuss the servicecontract DSL. At the end the host DSL will be discussed.

The first step is to compare both versions of the datacontract DSL (version b117 and final version). This is done using the DSLCompare tool.

The number of domainclasses in the both versions of the WSSF datacontract DSL are the same. For instance, the domainclass *DataElement* in the old DSL definition is called *DataMember* in the new DSL definition. When the transformation is created by the developer who also developed the DSL, this developer will know what element matches what other element. However, in this case the DSL was developed by Microsoft and Microsoft did not deliver a converter with the product. So, we do not know which class matches which other class. This is not a problem, because the automatic detection feature of DSLCompare analyzes this for us. We did not need to map any class ourselves, because the automatic detection feature was able to find all mappings automatically. After checking with Microsoft, these mappings turned out to be correct. This was also the case for the properties and the relationships.

The results of how well the automatic detection algortithm works for the datacontract DSL are shown in Table 8.2. As can be seen, a lot of changes occured to this DSL, because the number of indirect changes is much higher compared to the number of direct changes. However, no human help has been necessary to create the transformation between the two DSL definitions. All domainclasses, comainproperties and domainrelationships were mapped automatically. In the case of this DSL, most changes were rename changes. The structure of the DSL stayed the same, which helped the automatic detection.

Now that the transformation between the two versions of the datacontract DSL has been defined, the converter for the models can be generated. Because the changes to the DSL were mostly rename changes, no custom code is necessary for this converter. When the model of Fig. 8.2 is used as an input of this converter, the output model can be read in WSSF final and is also syntactically valid to the DSL definition of the datacontract DSL in WSSF final.

			b117	Final
Domainclasses	Direct	auto	3	3
		manual	0	0
	Indirect	auto	11	11
		manual	0	0
	Deleted		0	0
	Introduced		0	0
	Total		14	14
Domainproperties	Direct	auto	8	8
		manual	0	0
	Indirect	auto	40	40
		manual	0	0
	Deleted		0	0
	Introduced		0	1
	Total		48	49
Domainrelationships	Direct	auto	1	1
		manual	0	0
	Indirect	auto	68	68
		manual	0	0
	Deleted		0	0
	Introduced		0	0
	Total		69	69

Table 8.2: Comparison and automatic mapping of DSLCompare between the datacontract DSLs of WSSF b117 and WSSF Final

The next step is to create a converter for the servicecontract DSL. This works in exactly the same way as for the datacontract DSL. However, in this DSL also domainclasses, domainproperties and domainrelationships were introduced and deleted between the two different versions. However, fewer rename changes occured. Numbers on how well the DSLCompare tool was able to map the two different versions of the servicecontract DSL to each other, can be seen in Table 8.3.

As can be seen in Table 8.3 the mapping for the servicecontract DSL was done completely automatically as well. This DSL changed less compared to the datacontract DSL, but had more domainclasses, domainproperties and domainrelationships that were introduced or deleted.

The last step is to define the transformation for the host DSL. Most changes to the host DSL were done in the concrete syntax of the model. However, the evolution of the concrete syntax has no influence on the abstract syntax. How well the DSLCompare tool was able to map the two versions of the host DSL is shown in Table 8.4.

Table 8.4 shows that one manual mapping was necessary for the domainclasses. This was because two domainclasses were merged into one comainclass, which could not be recognized automatically for one of the two domainclasses. The domainclasses *ServiceType-*

			b117	Final
Domainclasses	Direct	auto	8	8
		manual	0	0
	Indirect	auto	3	3
		manual	0	0
	Deleted		2	0
	Introduced		0	2
	Total		13	13
Domainproperties	Direct	auto	21	21
		manual	0	0
	Indirect	auto	9	9
		manual	0	0
	Deleted		6	0
	Introduced		0	13
	Total		36	43
Domainrelationships	Direct	auto	17	17
		manual	0	0
	Indirect	auto	7	7
		manual	0	0
	Deleted		4	0
	Introduced		0	2
	Total		28	26

Table 8.3: Comparison and automatic mapping of DSLCompare between the servicecontract DSLs of WSSF b117 and WSSF Final

Reference and *ServiceMELReference* were merged into one domainclass *ServiceReference*. The mapping between *ServiceMELReference* and *ServiceReference* was recognized automatically. The mapping between *ServiceTypeReference* and *ServiceReference* needed to be defined manually. However, for the domainproperties and the domainrelationships, this was recognized automatically.

Converters have been generated for all three DSL definitions. For none of these converters custom code was necessary. The converters were tested on a real-life model and the models defined in [16]. The three models defined in [16] for the DSLs of version b117 are shown in Fig. 8.4. The left model is the datacontract model, the model in the middle is the servicecontract model and the model on the right is the host model. When these models are converted using the converters we have generated, we get the models in Fig. 8.5. The host model looks completely different, because the concrete syntax of the model has changed. The other models look exactly the same. However, as can be seen in Tables 8.2, 8.3 and 8.4, the DSL definitions of these models are different.

It is hard to say whether the models in Fig. 8.4 and Fig. 8.5 are interface compatible with each other. This is because it was not possible to generate code for all models in version b117 of WSSF. The code generated for the datacontract model, however, is exactly the same

			b117	Final
Domainclasses	Direct	auto	7	7
		manual	0	0
	Indirect	auto	1	1
		manual	1	0
	Deleted		0	0
	Introduced		0	0
	Total		9	8
Domainproperties	Direct	auto	10	10
		manual	0	0
	Indirect	auto	6	3
		manual	0	0
	Deleted		0	0
	Introduced		0	4
	Total		16	17
Domainrelationships	Direct	auto	6	6
		manual	0	0
	Indirect	auto	4	2
		manual	0	0
	Deleted		0	0
	Introduced		0	0
	Total		10	8

Table 8.4: Comparison and automatic mapping of DSLCompare between the host DSLs of WSSF b117 and WSSF Final

in version b117, compared to the final version of WSSF.

8.3.4 Conclusions

The first question we wanted answer in this case study was whether the solution proposed in this thesis would work as well for other DSLs as it worked for the DSL presented in the first case study. For all three DSLs, only one out of 243 entities needed to be mapped manually. This is only 0.4%, which is less than the 1.4% needed in the first case study. Also in terms of custom code the solution proposed in this thesis performed better in this case study, compared to the first case study. We did not have to write a single line of code to create the converters.

The second question we wanted answer in this case study was whether the solution proposed in this thesis would create a converter for all three models in less time than a week. Most time spent in this case study was finding the right source files in WSSF for the DSL definitions. Once these were found, it was a matter of starting the DSLCompare tool and creating a transformation using the automatic detection available in DSLCompare. Generating the model converters from these mappings was also very straightforward. In

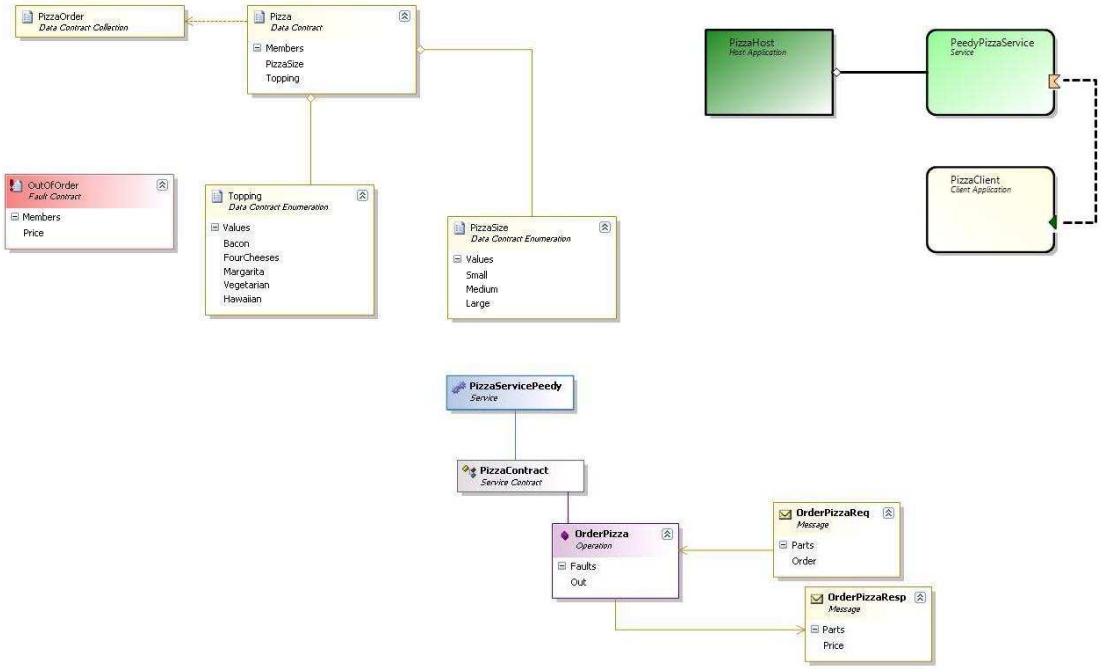


Figure 8.4: models in WSSF b117

total, the three converters can be created in half a day, which is much less than the one week [18].

8.4 Case Study 3

8.4.1 Background

This case study is not about evolution of a DSL, because the software factories used in this case study were created independently of each other. This case study is used to find the boundaries of the solution proposed in this thesis. The solution proposed in this thesis is aimed at DSL definitons that evolve, but can it also be used to migrate a model conform one DSL defintion to a model conform to a different DSL definition? This could be interesting for companies so they do not depend on the DSL of one vendor, but can convert their models to be conform to the DSL of another vendor.

For this case study we use the Release 1.0 DSL definition from the first case study and the WSSF final DSL definition from the second case study. Release 1.0 has all its features in one DSL, while WSSF has its features in three different DSLs. However, these software factories are both software factories for the same domain: Service Oriented Architecture.

In WSSF a method is created to link the models conform one of the three DSLs together. This method is not available in DSL-Tools and hence is not support by the tools presented

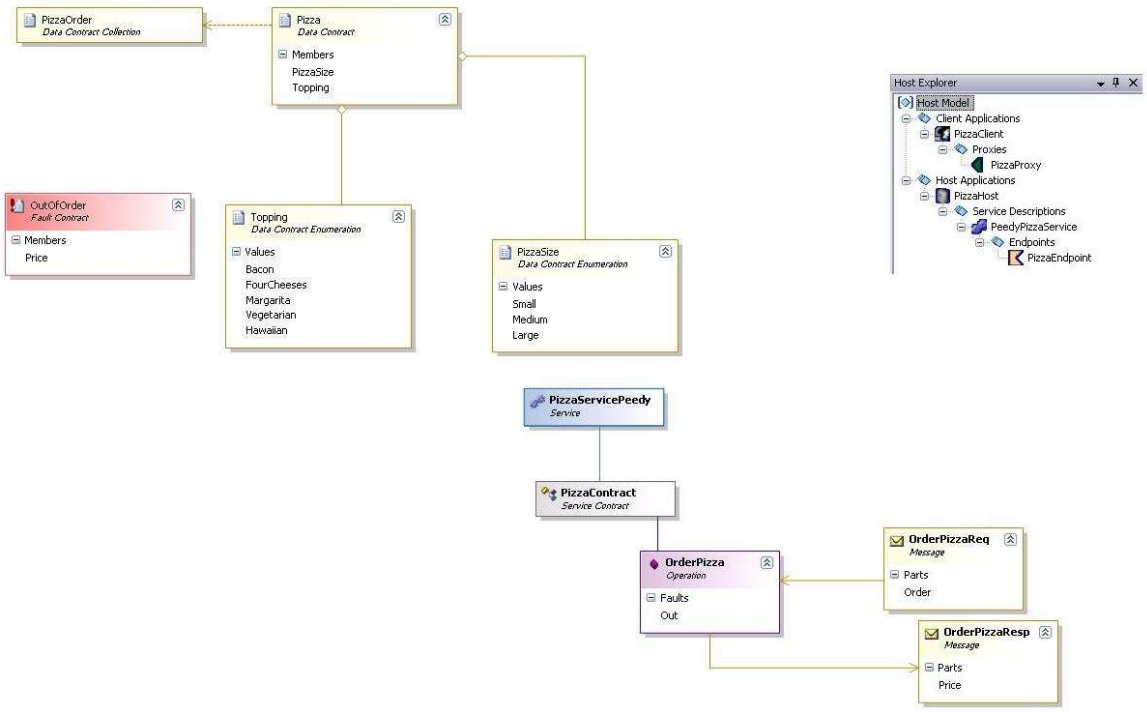


Figure 8.5: models in WSSF final

in this thesis. It will be necessary to manually develop code to support this in the code that was generated by the converter generator. Otherwise we will end up with three different models that are useless, because they are not connected to each other.

8.4.2 Questions

We answer one question with this case study:

- Can the solution proposed in this thesis be used to convert models conform a DSL definition to models conform another DSL definition, when the two DSL definitions have been developed independently?

8.4.3 The case study

We performed the case study in different stages:

- Create three transformations for the DSL definitions using the DSLCompare tool. This could either be done automatically or manually.
- Use the defined transformations to create a migration on models. This is done using the ConverterGenerator tool.

			Release 1.0	WSSF DataContract
Domainclasses	Direct	auto	0	0
		manual	0	0
	Indirect	auto	3	3
		manual	1	1
	Deleted		26	0
	Introduced		0	10
	Total		30	14
Domainproperties	Direct	auto	0	0
		manual	0	0
	Indirect	auto	3	3
		manual	4	4
	Deleted		211	0
	Introduced		0	42
	Total		218	49
Domainrelationships	Direct	auto	0	0
		manual	0	0
	Indirect	auto	3	3
		manual	0	0
	Deleted		152	0
	Introduced		0	66
	Total		155	69

Table 8.5: Comparison and automatic mapping of DSLCompare between Release 1.0 and WSSF datacontract

- Write custom code to connect the three separate models for WSSF to each other.

The first step is to create a mapping from Release 1.0 to the datacontract DSL. Table 8.5 shows the results of the mapping between the Release 1.0 DSL and the WSSF datacontract DSL. Note that for the domainclasses, many domainclasses are removed from Release 1.0. This was expected to happen, because many of the removed domainclasses are used in the other two DSL definitons of WSSF. However, what was not expected is that many domainclasses were introduced in the WSSF datacontract DSL. This shows that the domains of Release 1.0 and WSSF are more different than was expected at first.

The reason that so many domainclasses were introduced in WSSF is that WSSF has support for some features in SOA that Release 1.0 does not have support for. These features are enumerations, collections and fault contracts.

The same can be seen for the domainproperties. Many domainproperties are removed and introduced. This happens for the same reasons as for the domainclasses. Futhermore, WSSF has support for multiple target technologies (ASMX and WCF), so it consists of domainproperties for the other target technology as well. Release 1.0 only supports WCF. So WSSF, next to the same reasons as the domainclasses, also has extra domainproperties for ASMX support.

The same is also true for the domainrelationships. Again the great a of deleted and introduced domainrelationships is very large. This is for the same reasons as the domain-properties and the domainrelationships.

Next, we create the transformation to the servicecontract DSL and the host DSL. For brevity, the results for the servicecontract DSL and host DSL are omitted. They give simalar results as the conversion to the datacontract DSL. This is for the same reasons as the data-contract DSL conversion.

The second step is to create a migration on models from the transformation on DSL definitions. This is done using the ConverterGenerator tool. The result is three separate migrations that are able to convert a model conform Release 1.0 to either a datacontract model, a servicecontract model or a host model. So, if one wants to convert a single model conform Release 1.0, one needs to run three separate converters to create three seperate models.

The third step is to add custom code to the three converters in order to make the three models connect to each other. Only if the models are connected, WSSF lets the developer generate code from these models. This custom code is not very complicated, because it only adds the filename of the other models to a model.

To test the migration, we used the same test cases as the ones used for the first case study. However, it turned out that a lot of extra custom code was necessary to make the migration work correctly. For instance, Release 1.0 only supports data contracts, while WSSF distinguishes data contracts and fault contracts. This causes problems, because using Release 1.0 one would use a data contract when one means to use a fault contract. So, depending on how the data contract is used in the model conform Release 1.0, it could either become a data contract or a fault contract in the model conform WSSF. This is a semantical problem, however, and out of scope of this thesis.

After looking at the real-life project that was also used during the first case study, it becomes clear that many lines of custom code are needed to create a conversion which works satisfactory for developers. This is mostly because the set of features supported in Release 1.0 is much more different from the feature set of WSSF than one would think at first. Futhermore, WSSF uses some extensions that are not supported in DSL-Tools. Examples of this are the connections between models and the support for multiple target technologies.

8.4.4 Conclusions

In section 8.4.2 the question was asked whether the proposed approach in this thesis would also work for two DSL definitions that were created independent of eachother. It turns out that very few elements of a DSL definition can be mapped to eachother, although the DSL definitions seem to cover the same domain at first. From this a transformation between the DSL definitions can be created, without much manual interference. However, the total number of mappings is very small, so it is hard to draw any conclusions from this.

It is also possible to create a migration on models from the transformation on the DSL definitions. The migrated models are syntactically valid and some custom code is necessary

to add support for linking models to eachother. However, this was expected, because this is not a feature that is supported inside DSL-Tools.

However, it turns out that one is not able to generate code from the migrated models. This is, because the semantic meaning of many elements is different, so the models are not static semantically valid. This was discussed in chapter 1. Hence, much more custom code is necessary to support this as well.

To conclude: The approach proposed in this thesis works for this case study in which DSL definitions were used that are developed independently of eachother. The migrated models are syntactically valid. However, one could argue whether the approach is very useful, because a lot of custom code is necessary before code can be generated from the migrated models.

Chapter 9

Discussion

The chapter discusses the work presented in this thesis. Some of the challenges with the solution in this thesis are outlined in this chapter.

9.1 Evolution in practice

It is hard to tell how often a DSL definition changes in practice. For DSLs that are internally developed within Avanade, this is once in two years. For the development of Web Service Software Factory, a new version was released every 2 weeks. This makes it harder to tell whether the tool presented in this thesis will be useful in the future. If a DSL definition changes often, one would like to have a tool to do some of the work automatically. However, if the DSL definition does not change very often, it might not be a problem to do everything by hand.

We have shown in one of the case studies that the tools developed in this thesis give a faster development time for a converter to developers. In the one of the case studies presented, this was from one week to a couple of hours. When this happens once a year, it does not seem to be a big improvement. However, one such a converter needs to be developed once a week or maybe every day, this tool might save developers a lot of time. We hope that this research opens the possibility of a new way of working, in which DSL evolution can be encouraged, rather than avoided. This will result in DSLs that remains as close as possible to the underlying application domain, even when this application domain evolves over time.

One could argue that our approach only works for the case studies discussed in this thesis, because we were not able to perform many case studies. This is mostly because we were not able to find many DSL definitions based on the DSL-Tools meta-meta-model. Avanade has one factory that has been used on projects for customers. Also the DSL definitions in Microsoft's Web Service Software Factory are the first DSL definitions Microsoft has created using DSL-Tools. Furthermore, very few projects were done using these DSL definitions, so we were also not able to find many real-life projects either.

It is expected that the amount of DSLs and projects done using DSLs will increase in the future [49]. From this we hope that we will be able to find more DSL definitions and

models conforming to the DSL-Tools meta-meta-model. This will give us more material to test our approach with and will also give us more ideas to improve the approach.

9.2 Changes between versions of a DSL

Because every language defined using the DSL-Tools meta-meta-model consists of domainclasses, domainproperties and domainrelationships, changes can only occur in these three parts. When a DSL evolves, usually features are added to the DSL. This implicates addition of domainclasses, domainproperties and domainrelationships. One might expect that addition is not a concern to the syntacical validity of models conforming to a certain DSL definition. However, this turned out to be not true.

While the addition of entities in a DSL might seem easy at first, it turns out that this can be complex. When a domainclass is added, for instance, it might be the case that some domainproperties have to move from old domainclasses to this new domainclass. It might also be the case that because of this addition of a domainclass, domainrelationships change in some way.

The same conclusion holds for removing of inheritance. One would expect this to be very easy, because the domainclass names do not change. However, it turns out to be complex, because the names of domainrelationships will change. Also new domainrelationships will be introduced, because the inheritance is removed.

9.3 Equivalence of models

It is hard to tell when a model is equivalent to another model, when the models are conforming to different meta-models. However, this notion of equivalence is important to this thesis, because the developers want to have the same model after they converted their old model. For the case studies, we have used the opinion of the developers. The developers looked at whether they thought all features of the model based on the old version of the meta-model were represented in the model based on the new version of the meta-model.

It would be best to use a formal definition of when two models are equivalent if they are based on two different meta-models. However, this seems to be very hard, because the different meta-models do not need to support the same features. So it might be possible that some features of a model cannot be expressed in a different meta-model.

The formal description of when a model is equivalent to another model is left out of the scope of this thesis. For Avanade it is important that their developers can work with the tool, therefore the view of the developers has been taken as the most important factor in the case studies.

9.4 Applicability to other meta-meta-models

The research presented in this thesis has been performed in the context of Microsoft DSL-Tools. This meta-meta-model is based on the idea of Software Factories. The idea of

Software Factories outlines the principles of economies of scale and economies of scope [21]. However, not all meta-meta-models are based on these ideas.

For instance, if we look at the meta-meta-model of MetaCase [33], we see that the research of this thesis is not so interesting for their meta-meta-model. They aim their meta-meta-model at DSLs of which only one instance is created. These DSLs are used to build software that is very complex. Nokia, for instance, uses their meta-meta-model to create the operating system of their mobile phones [34].

In this case, usually only one model is created using a particular DSL. So, defining a transformation on the meta-model level and generating a transformation on the model level from this is kind of complicated. In this case, it might be better to simply define the transformation on the model level. The transformation only has to work for one particular model, so it is not very useful to generate a transformation for all models that could be created using the DSL.

Next to the meta-meta-model of MetaCase, one could also look at EMF (Eclipse Modeling Framework). Bezivin et al have proposed a bridge between Microsoft DSL-Tools and EMF [10]. Using this bridge, one would be able to use this approach for EMF as well. Unfortunately we did not have the time to implement and evaluate this.

9.5 Transformation language not conform DSL-Tools

The transformation language described in this thesis does not conform to the meta-meta-model of DSL-Tools. One might expect this language to be conform to the DSL-Tools meta-meta-model, because all other languages in this thesis conform to the DSL-Tools meta-meta-model. However, one of our main goals was to have this language as simple as possible. It is not hard to make the transformation language conform the DSL-Tools meta-meta-model. The only thing that needs to be done is that the relationships of the language need to be defined explicitly, which would make the language more verbose.

Also note that the transformation language is a textual language. However, the languages created using DSL-Tools are visual languages. Again, it is not hard to make the transformation language a visual language. However, we think that it is not useful to have a visual language in this case. The instances of the language could become quite large, especially when the DSL definitions become larger. However, models created using visual languages should stay small [21], which is not the case for the models conform our transformation languages. Hence, we have chosen to make this a textual language.

9.6 Verbose transformation language

The transformation language discussed in this thesis results in very verbose mappings. For instance, in section 4.1 the inheritance relation is removed from the DSL definition. This results in a mapping of 20 lines. A mapping of a real-life project is even worse. For the case study presented in section 8.2, 404 lines were necessary to do the mapping. Note that many of these lines represent identity mappings. One could argue whether it is necessary to

include the identity mappings in the mapping or that only the changes should be included in the mapping.

We have chosen to include the identity mappings, because the alternative would be to include the DSL definition into δ. However, this DSL definition is not always available due to intellectual property rights.

Another alternative might be to include the removal of entities explicitly in the transformation language. We have not explored this alternative well enough to say whether this would make the mappings less verbose. It would be interesting to do some case studies to this in the future.

9.7 Case studies

Some case studies have been performed in order to test the proposed method in practice. Robert Yin distinguishes three types of case studies in his book: explorative, explanatory and descriptive [53].

Explorative case studies are used when the investigator tries to answer the *why* and *how* questions and the investigator has little control over events and the focus is on a contemporary phenomenon. This is the type of case study we have been using in chapter 8. We have focussed on *how* our approach functions in real-life cases.

Explanatory case studies are used to explain events that occured in the past. An example of an explanatory case study is the book about the Cuban missle crisis by Graham Allison [6]. This book explains the course of events during the Cuban missle crisis and generalizes this to other governmental affairs.

Descriptive case studies are used to describe certain events. One could think about going to a neighborhood and describing the events that happen over time.

Regardless of the type of case study used, one could argue whether these case studies are representative. Robert Yin describes in his book some criteria for judging case study designs [53]. Four kinds of threats to validity are distinguished:

- Construct validity
- Internal validity
- External validity
- Reliability

Construct validity deals with the critics that researchers fail to develop a sufficiently operational set of measures and use subjective judgments to collect the data. According to Yin, two conditions must be met to cover construct validity: select the specific types of change that are to be studied and demonstrate that the selected measures of these changes reflect the specific type of change that has been selected [53]. However, these conditions are focused on changes from a sociological point of view, for instance, when one would try to find changes in the population of neighborhoods. For the case studies presented in this thesis we have constructed case studies with DSL definitions that are evolved DSL definitions of

each other. We have created a table (e.g. table 8.1) that we have used throughout all case studies. This table captures the changes that occured to the DSL definition and also captures how many elements our tool mapped automatically.

The table only captures the changes to the abstract syntax of the DSL definition and not to the semantics of the DSL definition. This matches the scope of this thesis, which is on the syntax of the DSL definition, as was discussed in section 1.4. However, we have tried to cover the semantics by looking at the custom code necessary to create a migration satisfactory for the developers of models conforming to the DSL definition.

Internal validity deals with whether causal relationships can be drawn from the case study results. This is not applicable to the case studies performed in this thesis, because these case studies are not concerned with how different events within the system relate to eachother. Internal validity is, for example, a concern when one observes two different events inside one neighborhood and causally relates these two events to eachother.

External validity deals with the problem of knowing whether a study's findings are generalizable beyond the immediate case study. One way to do this is to conduct multiple case studies and show that the results are similar. Multiple cases is often considered more compelling, and the overall study is therefore regarded as being more robust [22]. In total we have used four different DSL definitions for our case studies. For all four DSL definitions we have shown that the results are similar. It would have been better to conduct even more case studies, but we were not able to find other DSL definitions conforming to the DSL-Tools meta-meta-model that have been used in practice on many occasions.

Furthermore, we have presented a case study in which the DSL definitions were not evolved from each other (section 8.4). We showed in other case studies that the approach in this thesis works for DSL definitions that were evolved from each other. The approach also works for DSL definitions that did not evolve from each other, but that this might not be very useful for developers.

One could argue that we used subjective judgments to collect the data. We only used DSL definitions based on the SOA domain, because we were not able to find other DSL definitions conforming to the DSL-Tools meta-meta-model that have been used often in practice. We hope this will change in the future, so the proposed approach can be tested more extensively and also improved by looking at eventual problems that might occur in other domains. However, we have no reason to believe that our approach will not work for other application domains than the SOA domain.

Another point of discussion is the size of the DSL definitions used in the case studies. In section 8.2 we have presented a DSL definition with 403 entities (domainclasses, domainproperties and domainrelationships), which is the largest DSL definition we have used. The smallest DSL definitions are presented in section 8.3, where a DSL definition only has 35 entities. This shows that our approach works within this spectrum of size. We have no reason to believe that our approach would not work for DSL definitions larger than 403 entities, however we were not able to find larger DSL definitions that have been used in real-life scenarios.

However, one could also argue that the size of the difference between the two DSL definitions depends on whether our approach works. We believe this is true and that the case study presented in section 8.4 shows this threat. When there are too many changes,

DSLCompare is not able anymore to find a correct mapping. We have shown that this is true when the DSL definitions are developed seperately. However, more case studies will be necessary to also find cases in which the DSL definitions are real versions of each other, but have too many changes to be recognized automatically.

Reliability deals with whether the case studies can be reproduced. If a later investigator followed exactly the same procedures as described by an earlier investigator and conducted the same case study, the later investigator should come to the same conclusions as the earlier investigator [53]. This is not possible for the first case study conducted in this thesis, because the DSL definitions used are not public. However, the second case study can be reproduced. All six DSL definitions used can be downloaded from [2]. The reader is able to implement the automatic detection algorithm by using the definitions as defined in chapter 5. The manual mappings are described in section 8.2.3. To create a migration from this transformation, the reader could use the algorithm as outlined in chapter 6. However, the real-life models we used to test the migration are not available to the reader. The reader, however, can create the models defined in [16] and test the migration using these models.

9.8 Scope of this thesis

The scope of this thesis is to migrate models to be syntactically correct conforming to a new DSL definition as was discussed in section 1.4. Next to this, we wanted to make the models equivalent according to the opinion of the developers. This scope seemed to be fine for small DSL definitions as described in section 8.3. However, larger DSL definitions like the ones in section 8.2 also needed some custom code in order to handle some of the semantical changes.

More research is necessary to find out what kind of semantical changes can be expected in the evolution of DSL definitions. An approach to a solution for semantical changes could be found in the model comparison engines that exist and are able to the a semantical mapping on models. This is discussed in section 10.2.

Chapter 10

Related Work

Model transformations is a well explored domain in computer science. This chapter discusses technologies related to model transformations. It is divided into transformations on the meta-meta-model level, transformations on the meta-model level and transformations on the model level. However, not much work has been found on the case of evolution on the meta-model level and how this relates to the model level.

10.1 Transformations on the meta-meta-model level

Several meta-meta-models exist like Generic Modeling Environment (GME)[30], MS DSL-Tools [14], Eclipse Modeling Framework (EMF) [12] and many others. Some people have already tried to make transformations between all these different approaches to Model Driven Engineering (MDE). One of these approaches is described in [10] where a bridge between EMF and MS DSL-Tools is described.

Bezivin proposes to introduce an extra level called M4 (meta-meta-meta-model) to cope with transformations on the (M3) meta-meta-model level [10]. This would allow to reason about a transformation on the M3 level. They use their own AMMA framework for this now.

The paper only provides a theoretical approach to the problem of transformations on the meta-meta-model level. The conclusion is that this is a very important, but also a very hard problem. Some general overview ideas are presented on how the problem can be solved. The framework for evolution presented in this thesis is partly based on this framework.

10.2 Model Comparison Engines

Different model comparison engines exist that are based on the fact that elements in a model contain a unique identifier [5] [31]. This approach is also used in this thesis and forms the basis for the automatic detection algorithm. However, this approach was not sufficient. When removing an element from a meta-model and then adding it again, the unique identifier has changed. Therefore, we have made additions to this approach, which are presented in the automatic detection algorithm chapter.

A different approach is used in the EMF Compare tool [51]. This tool compares models that were built using EMF. Using the principles of information theory [45] EMF Compare is able to compare two models and match the elements with the same semantical meaning.

10.3 Evolution in other methodologies

The problem of evolution of meta-models does not only occur with Microsoft DSL-Tools as the methodology. Recently a lot of research is being done to meta-model evolution and the corresponding models for the ecore methodology. Ecore is the meta-model in the EMF framework [12]. In Munich, Jurgens and Hermannsdorfer have developed an approach called COPE (COuPled Evolution) [23] [24] in cooperation with BMW Car IT. They define coupling as *the activity of attaching a model migration to a sequence of metamodel changes*. Two different approaches of coupled evolution are considered: open coupled evolution and closed coupled evolution.

Open coupled evolution means that the developer defines a migration for the models manually, by looking at the evolution of the meta-model. The models can then be migrated automatically by using the manually defined migration.

Closed coupled evolution means that parts of a migration are being reused. When the developer has already specified a migration for models conforming the evolution of some meta-model, the developer might want to reuse parts of that migration on models conforming a different meta-model. This is the case when the evolution of the second meta-model is similar to the evolution of the first meta-model.

In this thesis we are using both open coupled evolution and closed coupled evolution. Open coupled evolution is represented by the human help to the ConverterGenerator in Fig. 2.2. Closed coupled evolution is in the ConverterGenerator itself. Our approach does not allow the developer to reuse his own migrations, but instead the migrations are predefined in the ConverterGenerator according to the scenarios presented in chapter 4.

10.4 Database Schema evolution

When one thinks of a database schema as the meta-model and the information contained as the model, the problem of database schema evolution becomes similar to the problem that is addressed in this thesis. In the case of database schema evolution, one wants to migrate the data to be compatible with the new database schema. Databases have existed for decades already and a lot of research has been done in this field.

In 1979 Sockut and Goldberg [46] described two levels of evolution: infological reorganization and string reorganization. Infological reorganization is about changes to the schema. For instance when one adds an attribute to a schema, infological reorganization would be the level of evolution. String reorganization is about changes to the relationships. For instance when evolving from a one-to-one relationship to a one-to-many relationship, string reorganization would be the level of evolution. This classification has inspired us to do the same separation between classes, properties and relationships. Classes and properties

are handled in the same way as infological reorganization. Relationships are handled in the same way as string reorganization.

The work presented by Sockut and Goldberg [46] is related to relational databases. However, object-oriented databases also form an interesting base for the work performed in this thesis. Different approaches have been undertaken to the evolution problem of databases like Orion [7], GemStone [39] and O_2 [8]. These systems are used to cope with the problem that when a class definition changes, all objects of that class definition should change as well. O_2 also supports more complex transformations, like moving an attribute from one class to another. This has inspired us to think about a similar scenario in which an attribute of a class in the DSL definition moves to another class in the DSL definition. This scenario is also described in the chapter about scenario's.

Pons and Keller describe a categorization for schema evolution in object databases [43]. They describe different catogories: primitive, composite and complex. A primitive operation is either an addition, a removal or a change. These are the same three operations as the ones that are possible in the transformation language described in this thesis. However, in section 3.2 the transformation language only seems to support change. Removal and addition are implicitly supported in the case the classes, properties or relationships are not mentioned. The composite category inspired us to think about the different scenarios that can occur when a DSL definition evolves. Combining different scenarios for the evolution of a DSL definition makes it possible to do a complex evolution transformation. These are described in chapter 8.

10.5 Engineering discipline for grammars

Klint et al. [26] argue that the engineering of grammars in not sufficiently understood, although grammars play an important role in software engineering. A research agenda with challenges in grammar engineering is proposed. One of the challenges proposed is the development of a grammar transformation framework. This framework could be used for the evolution of programs based on grammars that evolve. The ideas for solutions in this paper offer a language in which transformations on grammars can be described. From these description, compilers for DSLs can be generated from one DSL definition to another DSL definition.

Laemmel proposes a set of operations that are performed to evolve context free grammars [28]. He also proposes an implementation in [29]. However, these operations were defined for LLL (Lightweight LL parsing, a simple EBNF-based grammar format) and SDF (Syntax Definition Formalism, a language for constructing grammars) and not for DSL-Tools. DSL-Tools did not exist at the time of writing of that article.

Although the work presented by Laemmel is not focused on DSL-Tools, the ideas proposed can still be used. They have given us the idea to create a separate language to describe the transformation between two different DSLs. Although the transformations that can be described in this language are different from the transformations in our language, it has given us a valuable starting point to create our own transformation language.

10.6 Document transformations

In [35] Nix describes a framework that is able to edit text according to some examples of how the text should be edited. Using some pattern recognition, the examples provided by the user are analyzed and used to edit the rest of the text. However, the kinds of transformations that can be done using this approach are very limited and are not powerful enough to transform models. However, the idea to give some examples on how something should be transformed and generate a generic transformer from this has inspired us to do something similar. We are also generating a generic transform, however this is not from examples on the language, but from a higher level transformation.

The idea to generate a transformer from a higher level description, comes from the ideas on XML and its DTD. A DTD is a higher level description on how an XML document should be formatted. In [27] it is shown that one can usually represent the evolution of a DTD as a set of atomic operations. From these atomic operations, we have derived the scenario's for the evolution of meta-models in DSL-Tools. Furthermore, it is shown in [27] that from these atomic operations one can induce a transformation on the XML-document level, which can be seen as instances of the DTD.

In [50] also proposes a set of operations to perform evolution transformations on a DTD. However, they also claim that the resulting XML-documents will conform to the new DTD. It would be very interesting if we could prove the same for the solution proposed in this thesis. However, the set of operations in [50] is very limited, so the solution would lose a lot of usability in practical cases then.

10.7 Language Evolution Tools

Some tools exist that support the transformation of all kinds of languages as long as they are structured in some way. TransformGem [20] and Lever [42] are examples of such tools.

TransformGen [20] is a tool that generates converters based on the changes in a language specification. The TransformGen engine is given an old grammar and a transformation on that grammar. It produces the new grammar and a converter. The converter takes an instance of the old grammar and transforms it to an instance of the new grammar. TransformGen only supports grammars of which the instances can be represented as a tree. This is not the case for the grammars based on DSL-Tools, because these result in graphs [21]. However, the general environment in which TransformGen operates is similar to the environment in which this thesis works. The grammars evolve and from this we want the instances of grammars to evolve as well. From this approach we have learned the concepts on how to create a converter for instances from a transformation on grammars. It needed some changes, because we work with graphs and TransformGen works with trees.

Lever [42] is a tool that has recently been developed by the Technische Universitat Munchen. The tool is quite similar to TransformGen, however, this tool was build for the evolution of DSL's. It is not specified on which meta-meta-model these DSL's are defined. The tool can take a simple textual DSL definition as an input together with some transformation rules as an input and can create from this a new DSL definition and a converter for the models. However, when a structural change to the DSL occurs (i.e. adding a class),

about 20 transformation rules need to be defined manually [42]. So, this approach is quite similar to the approach proposed in this thesis, but it is much more complicated for the developer. During the case studies shown in this thesis, we never needed to add 20 transformation rules, although these case studies were much bigger compared to the case studies in [42].

10.8 Graph transformations

Because the visual models in DSL-Tools can be represented as graphs, the transformation of models can be seen as a graph transformation [14]. Jonathan Sprinkle has done a lot of work on the connection between models and graphs [47] [48] and how graph rewriting can be used in model evolution.

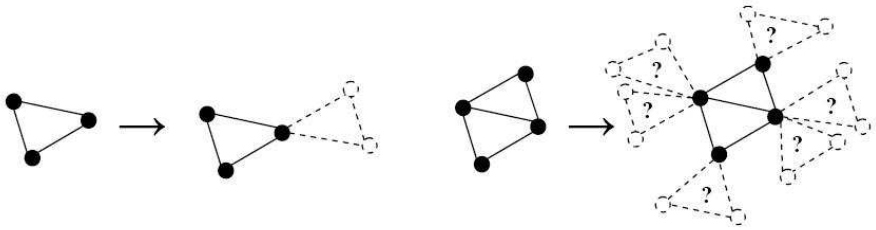


Figure 10.1: Ambiguity in graph transformation

Sprinkle describes the use of graph rewriting rules to describe model transformations. He also describes the problems that might occur when uses this technique. An example is shown in Fig. 10.1. The transformation described in the left part of Fig. 10.1 is ambiguous, because in the right part it is not clear which of the six options is the right one. Or whether multiple options are right.

Graph rewriting is used in this thesis in Δ and in δ. These describe the transformations between meta-models and models respectively. They can both be considered as a set of graph rewriting rules to transform one model into another model.

Especially for the transformation δ the work of Sprinkle has been an inspiration on which approach to choose to perform the δ transformation. Also implementations of graph rewriting on model transformations like PROGRES [11] and GReAT [3] have helped to understand how to make a transformation between two models.

Chapter 11

Conclusions and Future Work

This chapter summarizes the work of this thesis and gives an overview of the project's contributions. Finally, some ideas for future work will be discussed.

11.1 Summary

The research question investigated in this thesis is: *What scenarios exist considering versioning of DSLs and how can those scenarios be handled in an efficient way?* The research has been performed within Avanade (a global IT company) to help Avanade coping with the evolution of their DSLs conforming to the Microsoft DSL-Tools meta-meta-model.

To answer the research question, different scenarios related to versioning of DSLs have identified and presented in chapter 4. Some of these scenarios are artificial, while others are taken from DSL definitions used within Avanade.

A framework to cope with DSL evolution has been developed in order to cope with the identified scenarios. The framework is presented in chapter 2 and consists of two tools: DSLCompare and ConverterGenerator.

DSLCompare compares two DSL definitions and together with some human help creates a transformation between the two DSL definitions, which is called Δ. To compare the two DSL definitions and automatically generate a transformation, an automatic detection algorithm has been developed. This automatic detection algorithm was presented in chapter 5. The Δ is described using a transformation language that has been developed in this project as well.

ConverterGenerator takes the Δ as an input and generates a tool called δ to migrate models conforming to one of the DSL definitions used for DSLCompare to models conforming to the other DSL definition used as an input for DSLCompare. The tool δ is a program written in C# to which custom C# code can be added for the migration of models.

The two tools (DSLCompare and ConverterGenerator) have been implemented and evaluated by means of case studies. We have discussed three different case studies and have used four different software factories. In total eight different DSL definitions have been used for the case studies. The case studies have shown that the developed approach works satisfactory for the DSL definitions used within Avanade.

11.2 Contributions

We were able to find very few papers about Microsoft DSL-Tools, the toolset, methodology and meta-meta-model that are used in this thesis. To start with the research on evolution of DSL definitions that were created conforming to the DSL-Tools meta-meta-model, we have first created a formal definition of the DSL-Tools meta-meta-model by reverse engineering. This definition tells us which meta-models can be created conform to DSL-Tools. The results in this thesis are based on the formal definition of the DSL-Tools meta-meta-model.

A framework to cope with DSL evolution. We analyzed DSL definitions used within Avanade and how they evolved over time. These are DSL definition used within Avanade internally and DSL definitions from external vendors, for instance three DSL definitions from Microsofts Web Service Software Factory. By looking at the changes that have occured over time to all these DSL definitions, we have created a summary of the changes that most occured to DSL definitions. A framework to cope with these scenarios has been proposed and an automatic detection algorithm has been developed in order to detect most changes automatically. Furthermore, a transformation language has been defined to describe the transformation between two DSL definitions.

An implementation of the proposed framework. The proposed framework has been implemented in two different tools: DSLCompare and ConverterGenerator. The tools have been built to work with the Microsoft DSL-Tools infrastructure. The tools are available to the reader upon request.

An evaluation of the proposed framework. We have used three case studies to evaluate the proposed approach. We have used different DSL definitions created by different vendors for these case studies. The case studies have shown that the automatic detection performs satisfactory in real-life scenarios. For the case of the datacontract DSL evolution in Microsofts Web Service Software Factory Modeling Edition, no human interaction was necessary to create the transformation. We were able to create a transformation in a short period of time, while manually this costs a week [18]. For the cases in which the DSLs have been developed by separate teams, the automatic detection turned out to work quite satisfactory. However, this was hard to measure, because we were not able to find two separate DSL definitions which were targeted at exactly the same domain.

11.3 Future work

This thesis did not solve the problem of evolution of Domain-Specific Languages. It was merely an introduction research into the field and a solution was proposed for the syntax part of the problem. However, evolution of Domain-Specific languages is more than only the evolution of the syntax of a DSL. It is very interesting to look at how to handle the evolution of the static semantics of a language, for instance.

11.3.1 From DSL to Software Factory

The work presented in this thesis was performed using Microsoft DSL-Tools. However, Microsoft DSL-Tools is part of the Software Factories paradigm [21]. This paradigm describes

a way in which DSLs can work together with API's and Guidance Automation. Some parts of an artifact can be modeled using DSLs while other parts are modeled using API's or Guidance Automation.

From the Software Factories point of view, it might happen that a factory evolves in a way that a feature moves from being modeled in a DSL to being modeled in Guidance Automation for instance. At this moment, it is not quite clear yet how the Software Factories paradigm will be implemented, because only the DSL part is mature enough at the moment [49]. However, once the Software Factories paradigm becomes more mature, a evolution solution is necessary for every part of a Software Factory and not only for the DSL part.

11.3.2 Using a different meta-meta-model

This thesis only focused on the MS DSL-Tools meta-meta-model. However, there are many more meta-meta-models in the world than MS DSL-Tools. All these approaches differ in some ways [49]. These differences occur in many aspects of the meta-meta-model. Mostly the differences are for historic reasons and focus, however, there are also fundamental differences in technology.

This thesis only focuses on MS DSL-Tools and uses a lot of properties that meta-models based on the MS DSL-Tools meta-meta-model have. It would be very interesting to know how important these properties are for the method presented in this thesis to work. A short summary of the properties of MS DSL-Tools used, is given below:

- Models based on MS DSL-Tools are visual models only.
- Meta-models based on MS DSL-Tools consist of classes, properties and relationships. This is further described in Fig. 3.1.
- Models based on MS DSL-Tools can be represented as Abstract Syntax Graphs. Every graph has a spanning tree, which is explicitly given in the graph.

Bridges have already been built between different modeling aproaches. An example is a bridge between Microsoft DSL-Tools and EMF as was discussed in section 9.4. This would be a good starting point to explore the usage of our approach in other meta-meta-models than Microsoft DSL-Tools.

11.3.3 Reuse of transformations

The solution presented in this thesis works nicely when one wants to migrate between two specific versions of a DSL. However, one might want to migrate two versions ahead, instead of only one. This is illustrated in Fig. 11.1.

This might seem to be a very trivial problem. When one would have a model based on Version 1 and would like to migrate to Version 3, one would simply first apply transformation a and then transformation b. This would work and would result in a syntactical valid model, however, it might not be very satisfying for the developer. It might be the case that some features have been removed in Version 2 that have been added again in Version 3.

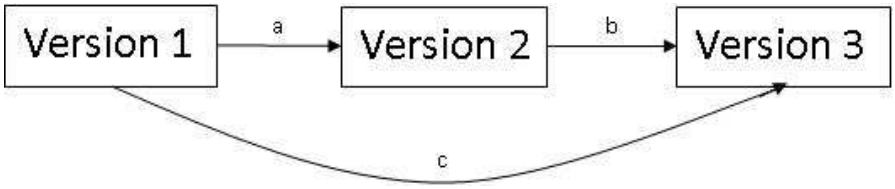


Figure 11.1: Reusing transformations across multiple versions

When one would first apply transformation a, this information will be lost. So, in migration b, this information cannot be used as an input and is therefore also lost in the model of Version 3. If, however, we would be able to create transformation c out of the transformations a and b, this information would have been available to Version 3.

In the light of economies of scale and economies of scope, DSLs are usually developed by a separate team [21]. Other developers use these DSLs as tools for their projects. During these projects, developers might find the need for some features that are not provided by the DSL. The developer then decides to modify the DSL slightly, so it supports the features that are necessary for the project. This might result in a model that is not syntactically correct anymore to the original DSL. This is illustrated in Fig. 11.2.

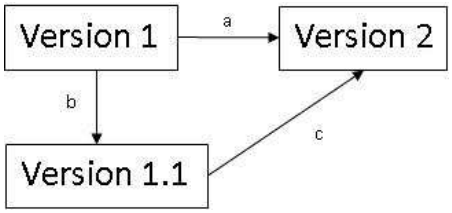


Figure 11.2: Reusing transformations with custom DSLs

Version 1 represents the version of the DSL that has been developed by some team within the organization. When some developers wants extra features in the DSL, he creates Version 1.1. The developer would be able to describe the transformation b between Version 1 and version 1.1. However, the team that developed version 1, continues working on that DSL until it becomes Version 2. They also provide developers with transformation a, that describes the transformation between Version 1 and Version 2. The developer might want to migrate his models based on Version 1.1 to Version 2 now. To do this, he would need transformation c. He could try to create transformation c by himself, however, this seems to be a waste of time as the information should already be available by combining transformations a and b in some way.

A possible approach to this challenge might be to create a new version of DSLCompare that takes three DSL definitions instead of two. Next to this, it could have an import function to import an already defined transformations for *a* and *b*. Some new automatic detection algorithm will be necessary to map the transformations and create a new transformation *c*.

11.3.4 Use within Avanade on projects

Avanade is continuing the research on DSL evolution and will use the approach presented in this thesis for real-life projects. This will allow us to improve the toolset further. Furthermore, it will give us a better overview of the boundaries of the approach presented in this thesis. When the tool is used by many consultants all over the world, consultants will come with new feature requests and new scenarios related to versioning. This will allow us to further improve the tools.

11.4 Conclusions

This thesis presented an approach to cope with the evolution of DSL definitions. The DSL definitions used within Avanade are conforming to the Microsoft DSL-Tools meta-meta-model, however, using the right bridges it might also be possible to use this approach for other meta-meta-models like EMF.

The presented approach has been implemented and evaluated. The evaluation has shown that the approach works satisfactory for DSL definitions created within Avanade. Hence, Avanade will continue using the proposed approach and we hope that this will give us more case studies and further possibilities to improve our approach.

Bibliography

[1] Stringtemplate. http://www.stringtemplate.org.

[2] Web service software factory community. http://codeplex.com/servicefactory.

[3] A. Agrawal, G. Karsai, and F. Shi. Interpreter writing using graph transformations. Technical report, ISIS Technical Report, 2003. ISIS-03-401, 2003.

[4] D. H. Akehurst, W. G. Howells, and K. D. McDonald-Maier. Kent model transformation language. *Model Transformations in Practice Workshop, part of MoDELS 2005*, 2005.

[5] M. Alanen and I. Porres. Difference and Union of Models. Technical report, TUCS, April 2003. Technical Report 527.

[6] G. Allison. *Essence of Decision: Explaining the Cuban Missile Crisis*. Little Brown, 1971.

[7] J. Banerjee, W. Kim, H.-J. Kim, and H.F. Kort. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD Rec.*, 16(3):311–322, 1987.

[8] Gilles Barbedette. Schema modifications in the LISPO2 persistent object-oriented language. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 77–96, London, UK, 1991. Springer-Verlag.

[9] J. Bézivin, E. Breton, G. Dupé, and P. Valduriez. The ATL transformation-based model management framework. Technical report, University of Nantes, 2003. TR03-08.

[10] J. Bézivin, G. Hillairet, F. Jouault, I. Kurtev, and W. Piers. Bridging the MS/DSL Tools and the Eclipse Modeling Framework. In *OOPSLA Int. Workshop on Software Factories*, 2005.

[11] D. Blostein and A. Shurr. Computing with graphs and graph rewriting. *Software - Practice and Experience*, 29(3):1–21, 1999.

[12] F. Budinsky, S.A. Brodsky, and E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[13] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 493–504, New York, NY, USA, 1996. ACM.

[14] S. Cook, G. Jones, S. Kent, and A.C. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley, 2007.

[15] G. W. de Geest, A. Savelkoul, and A. Alikoski. Building a framework to support Domain Specific Language evolution using Microsoft DSL Tools. In *Proceedings of OOPSLA 7th Workshop on Domain-Specific Modeling*, pages 60–70. University of Jyväskylä, 2007.

[16] G. W. de Geest and G. van Loon. Service Station. *MSDN Magazine*, 23(3):101–110, 2008.

[17] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[18] D. Doomen. How to: Migrating from v3b117 to v3ctp. http://www.codeplex.com/servicefactory/Thread/View.aspx?ThreadId=16626.

[19] A.W.B. Furtado and A.L.M. Santos. Using Domain-Specific Modeling towards Computer Games Development Industrialization. In *6th OOPSLA Workshop on Domain-Specific Modeling*. University of Jyväskylä, 2006.

[20] David Garlan, Charles W. Krueger, and Barbara Staudt Lerner. TransformGen: automating the maintenance of structure-oriented environments. *ACM Trans. Program. Lang. Syst.*, 16(3):727–774, 1994.

[21] J. Greenfield and K. Short. *Software Factories*. Wiley, 2004.

[22] R.E. Herriott and W.A. Firestone. Multisite qualitative policy research: Optimizing description and generalizability. *Educational Researcher*, 12:14–19.

[23] M. Herrmannsdörfer. Coupled evolution of metamodels and models. Master's thesis, Munich University of Technology, 2007.

[24] E. Jürgens. Evolutionary development of domain specific languages. Master's thesis, Munich University of Technology", 2006.

[25] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[26] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol. Vol 14 no 3*, 2005.

[27] R. Lämmel and W. Lohmann. Format evolution. *Proc. 7th international Conference on Reverse Engineering for Information Systems (RETIS 2001)*, 2001.

[28] Ralf Lämmel. Grammar Adaptation. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 550–570, London, UK, 2001. Springer-Verlag.

[29] Ralf Lämmel and Guido Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF meta-environment. In Mark van den Brand and Didier Parigot, editors, *Proceedings of the 1st Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *Electronical Notes in Theoretical Computer Science*. Elsevier Science, April 2001.

[30] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. *Proc. IEEE Int. Workshop Intelligent Signal Processing (WISP'2001)*, May 2001.

[31] K. Letkeman. Comparing and merging UML models in IBM, 2005. http://www.ibm.com/developerworks/rational/library/05/712_comp/.

[32] B.P. Lientz, P. Bennet, E.B. Swanson, and E. Burton. *Software Maintenance Management*. Addison Wesley, 1980.

[33] MetaCase. Domain-specific modeling with MetaEdit+: 10 times faster than UML. 2007.

[34] MetaCase. Nokia case study, MetaEdit+ revolutionized the way Nokia develops mobile phone software. 2007.

[35] Robert P. Nix. Editing by example. *ACM Trans. Program. Lang. Syst.*, 7(4):600–621, 1985.

[36] OMG. XML metadata interchange, version 1.2, january 2002. http://www.omg.org/.

[37] T. Panas, W. Lowe, and U. Asmann. Towards the unified recovery architecture for reverse engineering. In *Intern. Conf. on Software Engineering and Practice (SERP'03)*. CSREA Press, 2003.

[38] C. Peiris and D. Mulder. *Pro WCF: Practical Microsoft SOA Implementation*. A-Press, 2007.

[39] D. J. Penney and J. Stein. Class modification in the GemStone object-oriented DBMS. *SIGPLAN Not.*, 22(12):111–117, 1987.

[40] R. Pickering. *Foundations of F#*. A-Press, 2007.

[41] T.M. Pigoski. *Practical Software Maintenance*. Wiley Computer Publishing, 1996.

[42] M. Pizka and E.Jürgens. Automating language evolution. *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE 07)*, 2007.

[43] A. Pons and R. K. Keller. Schema evolution in object databases by catalogs. In *IDEAS '97: Proceedings of the 1997 International Symposium on Database Engineering & Applications*, page 368, Washington, DC, USA, 1997. IEEE Computer Society.

[44] Software Productivity Research. SPR programming languages table, 2005. http://www.spr.com.

[45] C. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, pages 379–423, July 1948.

[46] Gary H. Sockut and Robert P. Goldberg. Database Reorganization Principles and Practice. *ACM Comput. Surv.*, 11(4):371–395, 1979.

[47] J. Sprinkle. *Metamodel driven model migration*. PhD thesis, Graduate School of Vanderbilt University, 2003.

[48] J. Sprinkle and G. Karsai. A Domain Specific Visual Language for Domain Model Evolution. *Journal of Visual Languages and Computing*, 15(3-4):291–307, June 2004. Special Issue: Domain-Specific Modeling with Visual Languages.

[49] T. Stahl and M. Volter. *Model-Driven Software Development*. Wiley, 2003.

[50] H. Su, D. Kramer, L. Chen, K.T. Claypool, and E.A. RundenSteiner. XEM: Managing the evolution of XML documents. *RIDE-DM*, pages 103–110, 2001.

[51] A. Toulme. Presentation of EMF Compare Utility. In *10th Eclipse Modeling Symposium*. University of Nantes, 2006.

[52] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance, 10:75-92*, 1998.

[53] R.K. Yin. *Case Study Research: Design and Methods, Third Edition*. Sage Publications, December 2002.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

ACA: Avanade Connected Architectures

ASG: Abstract Syntax Graph

ATL: Atlas Transformation Language

COPE: COuPled Evolution of metamodels and models

DSL: Domain Specific Language

DTD: Document Type Definition

EMF: Eclipse Modeling Framework

HTML: HyperText Markup Language

MDA: Model Driven Architecture

MOF: Meta Object Facility

NPC: Non-Playable Character

OMG: Object Management Group

SOA: Service Oriented Architecture

SQL: Structured Query Language

UML: Unified Modeling Language

UUID: Universal Unique IDentifier

WCF: Windows Communication Foundation

WSSF: Web Service Software Factory

XML: eXtensible Markup Language

Appendix B

F# Implementation of transformation on meta-models

In this appendix we give an implementation of how transformations on meta-models are performed. As input we use a meta-model represented in the language that was represented in section 3.1. Also a mapping in the language that was presented in section 3.2 is used as an input. The output is a meta-model that is presented in the language of section 3.1.

```
// open some standard namespaces
open System

type Relation = { rName : String;
                  SourceClass : String;
                  TargetClass : String;
                  isEmbedding : bool;
                  allowsDuplicates : bool;
                  sourceName : String;
                }



type PropertyMap = { pSourceClass : String;
                     pSourceName : String;
                     pTargetClass : String;
                     pTargetName : String;
                     pRel : Relation;
                   }

type ClassMap = { SourceName : String;
                  TargetName : String;
                }


type DomainProperty = { pName : string;
                        Type : string;
                      }

type DomainClass = { cName : string;
```

```
                    properties : list<DomainProperty>
                  }

type DomainRelationship = { relName : String;
                            SourceClass : String;
                            TargetClass : String;
                            isEmbedding : bool;
                            allowsDuplicates : bool;
                            sourceName : String;
                          }

type Dsl = {  domainclasses  : list<DomainClass>;
              domainrelationships : list<DomainRelationship>;
}

type RelationMap =  { Name : String;
                      SourceClass : String;
                      TargetClass : String;
                      isEmbedding : bool;
                      allowsDuplicates : bool;
                      sourceName : String;
                      relations : list<Relation>;
                    }


type Mapping = { classmaps : list<ClassMap>;
                 propertymaps : list<PropertyMap>;
                 relationmaps : list<RelationMap> }

// [DomainProperty] ->  [PropertyMap] -> [DomainProperty]
let rec applypropsmap properties propertymap =
    match properties with
    | [] -> []
    | y::ys -> if y.pName = propertymap.pSourceName
               then [{pName=propertymap.pTargetName; Type = y.Type}] @
                     applypropsmap ys propertymap
               else applypropsmap ys propertymap

// DomainClass -> ClassMap -> [PropertyMap] -> [DomainProperty]
let rec applypropsmaps domainclass classmap propertymaps =
    match propertymaps with
    | [] -> []
    | y::ys -> if domainclass.cName = y.pSourceClass && classmap.TargetName = y.pTargetClass
               then applypropsmap domainclass.properties y @
                    applypropsmaps domainclass classmap ys
               else applypropsmaps domainclass classmap ys

// DomainClass -> [ClassMap] -> [PropertyMap] -> [DomainClass]
let rec applyclass domainclass classmaps propertymaps  =
    match classmaps with
    | [] -> [] : list<DomainClass>
    | y::ys -> if y.SourceName = domainclass.cName
               then [{cName=y.TargetName;
                      properties = applypropsmaps domainclass y propertymaps}] @
```

```
                                    applyclass domainclass ys propertymaps
                else applyclass domainclass ys propertymaps

// [DomainClass] -> [ClassMap] -> [PropertyMap] -> [DomainClass]
let rec applyclasses classes classmaps propertymaps =
    match classes with
    | [] -> []
    | y::ys -> applyclass y classmaps propertymaps @
                applyclasses ys classmaps propertymaps

// [DomainRelationship] -> Relation -> bool
let rec relexists relations rel =
    match relations with
    | [] -> false
    | y::ys -> if y.relName = rel.rName then true
                else relexists ys rel

// [DomainRelationship] -> [Relation] -> bool
let rec sourcerelsexists relations sourcerels =
    match sourcerels with
    | [] -> true
    | y::ys -> if relexists relations y then sourcerelsexists relations ys
                else false

// [DomainRelationship] -> [RelationMap] -> [DomainRelationship]
let rec applyrelations relations relmappings =
    match relmappings with
    | [] -> []
    | y::ys -> if sourcerelsexists relations y.relations
                then [{relName = y.Name;
                   SourceClass = y.SourceClass;
                   TargetClass = y.TargetClass;
                   isEmbedding = y.isEmbedding;
                   allowsDuplicates = y.allowsDuplicates;
                   sourceName = y.sourceName} ]
                else []

// String -> [ClassMap] -> String
let rec targetclassname sourcename classmaps =
    match classmaps with
    | [] -> ""
    | y::ys -> if sourcename = y.SourceName
                then y.TargetName
                else targetclassname sourcename ys

// [DomainProperty] -> String -> String
let rec resolvetype properties propertyname =
    match properties with
    | [] -> ""
    | y::ys -> if y.pName = propertyname
                then y.Type
                else resolvetype ys propertyname

// PropertyMap -> [DomainClass] -> DomainProperty
```

```
let rec createproperty propertymap domainclasses =
    match domainclasses with
    | [] -> {pName=""; Type="";}
    | y::ys -> if propertymap.pSourceClass = y.cName
               then {pName=propertymap.pTargetName;
                     Type=resolvetype y.properties propertymap.pSourceName;}
               else createproperty propertymap ys

// [DomainClass] -> [PropertyMap] -> PropertyMap -> [DomainProperty]
let rec createproperties domainclasses propertymaps propertymap =
    match propertymaps with
    | [] -> []
    | y::ys -> if y.pTargetClass = propertymap.pTargetClass
               then [createproperty y domainclasses] @ createproperties domainclasses ys propertymap
               else createproperties domainclasses ys propertymap

// [DomainClass] -> [ClassMap] -> [PropertyMap] -> [DomainClass]
let rec createclasses domainclasses classmaps propertymaps =
    match propertymaps with
    | [] -> []
    | y::ys -> if y.pTargetClass <> targetclassname y.pSourceClass classmaps
               then [{cName = y.pTargetClass;
                      properties=createproperties domainclasses propertymaps y}] @
                    createclasses domainclasses classmaps ys
               else createclasses domainclasses classmaps ys

// [ClassMap] -> [PropertyMap] -> [DomainRelationship]
let rec createrelations classmaps propertymaps =
    match propertymaps with
    | [] -> []
    | y::ys -> if y.pTargetClass <> targetclassname y.pSourceClass classmaps
               then [{relName = y.pRel.rName;
          SourceClass = y.pRel.SourceClass;
          TargetClass = y.pRel.TargetClass;
          isEmbedding = y.pRel.isEmbedding;
          allowsDuplicates = y.pRel.allowsDuplicates;
          sourceName = y.pRel.sourceName;}] @
          createrelations classmaps ys
   else createrelations classmaps ys

// [DomainClass] -> String -> [DomainClass]
let rec removeclass domainclasses name =
    match domainclasses with
    | [] -> []
    | y::ys -> if y.cName = name
               then removeclass ys name
               else [y] @ removeclass ys name

// [DomainClass] -> [DomainClass]
let rec filterclasses domainclasses =
    match domainclasses with
    | [] -> []
    | y::ys -> [y] @ filterclasses (removeclass ys y.cName)
```

```
// [DomainRelation] -> String -> [DomainRelation]
let rec removerelation relations name =
    match relations with
    | [] -> []
    | y::ys -> if y.relName = name
               then removerelation ys name
               else [y] @ removerelation ys name

// [DomainRelation] -> [DomainRelation]
let rec filterrelations relations =
    match relations with
    | [] -> []
    | y::ys -> [y] @ filterrelations (removerelation ys y.relName)

// Dsl -> Mapping -> Dsl
let rec apply dsldef map =
   {domainclasses= filterclasses (
                     applyclasses dsldef.domainclasses map.classmaps map.propertymaps @
                     createclasses dsldef.domainclasses map.classmaps map.propertymaps);
   domainrelationships=filterrelations (
                     applyrelations dsldef.domainrelationships map.relationmaps @
                     createrelations map.classmaps map.propertymaps);}
```