Building a framework to support Domain-Specific Language evolution using Microsoft DSL Tools

Gerardo de Geest[1], Antoine Savelkoul[1], and Aali Alikoski[2]
{gerardod,antoinesa,aalia}@avanade.com

[1] Avanade Netherlands
[2] Avanade Finland

Abstract. The concept of Domain-Specific Languages (DSLs) is nothing 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 upcoming software tools that will make it possible to replace general purpose code by custom-build DSLs in a more convenient way. Maintenance issues will become more relevant when the need to update DSLs increases over time and the number of DSLs used within a single project might increase significantly. This will likely 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 paper describes an approach to automatically convert models between different versions of a DSL. An XML-based DSL is introduced to define a mapping between two different versions of a DSL. Using this XML-based mapping language, a converter can be generated that converts models of an earlier version of some DSL to a newer version of the same DSL.

1 Introduction

In a way DSL-based software development resembles a lot of the traditional software development landscape with many of the same issues. One of these issues is the need to maintain and keep track of all the different versions of the artifacts. When compared to traditional software development, however, there are additional concerns due to the nature of DSL-based development: the DSL itself adds another level of abstraction [2] that needs versioning in itself. Moreover, as DSLs form the basis which the actual models are constructed on, we need to deal with propagating the necessary changes also to the models.

One concern related to versioning is that there are roughly two types of DSLs, vertical and horizontal ones [5]. Each has different characteristics related to versioning.

Vertical DSLs focus on non-technical problems. In general they focus on business domains such as the modeling of insurance products or railway systems. These domains can even become specific to only one organization as these DSLs might be formed by methodologies used by the domain experts and policies of the company the DSLs are made for.

Horizontal DSLs are technical DSLs that will more likely be made by the provider of the architectural framework that will be used as a base, e.g. Service Oriented Architectures

(SOA) [8]. This may also suggest that they will provide users with transformation utilities when new versions of these DSLs will be released. However, in the case of vertical DSLs, the owner and supporter of a DSL will likely not be a technology provider, but a software engineering department within the company the DSL is made for or a systems integrator, for example, companies like Avanade.

Another concern related to versioning is the frequency in which changes are introduced and who the users of the DSL will be. Technology providers typically have to be more careful when introducing new versions of their products (including DSLs), because they have a wide audience of unidentified customers. This makes it difficult if not impossible to make destructive changes. APIs for instance are hard to change [9]. Once Microsoft introduces new features, such as generics in C# 2.0, it cannot realistically be withdrawn or changed in the following version. This would require a migration in which developers need to change the code using these parts of the API as well. So, the only option for technology providers like Microsoft is to very carefully evaluate each new feature they introduce. However, for a DSL built by an insurance company, for example, there are typically no other users for the DSLs than the in-house developers. In that case it is easier to make changes in a more rapid pace and also introduce more structural changes. In-house DSL developers of course have the disadvantage of having fewer early evaluators than the technology providers do, so the likelihood of frequent changes that are also visible to the DSL users (the developers) is much higher and therefore the need to deal with versioning issues is unavoidable.

Since there are more layers of abstraction than in traditional software development scenario, there are also more levels of concepts that need to be versioned:

- *The DSL platform itself* In the case of this paper this means different versions of the DSL Tools implementation which are released by Microsoft. So far there has only been one officially released version, but multiple prerelease versions have been released as well. Moving from earlier versions was largely a manual task and was not a pleasant experience. A tool to convert DSLs to a newer version of the DSL platform was available, but was hard to use. One might expect a tool that would convert all Domain Specific Language definitions and also all models based on those definitions. However, this level of versioning is out of the scope of this paper.
- *DSLs* The actual modeling language definition, all of the concepts in the modeling language, the validation and transformation rules and other related artifacts are essential to be included in the versioning scheme. Providing transformations between different versions of the modeling language concepts are the main focus in this paper. However, only the syntactical part of a DSL will be taken into account.
- *Models* The models contain the actual implementation based on the domain model concepts. As the underlying domain model changes (a new version of the DSL definition is released) it typically needs to be possible to upgrade models so that they conform to this new version of the domain model. Thus, a new version of the models need to be created. This is the secondary focus in this paper.

The three layers of abstraction discussed above are all subject to versioning. The scope for this paper, however, is versioning of DSLs and how automatic model conversions can be performed after DSLs have evolved. In the next section, different versioning scenarios of DSLs will be presented. Section 3 presents a general framework for DSL evolution. In section 4 an XML-based DSL will be introduced to describe versioning scenarios between different DSLs and will be used to generate converters to convert models that are subject

to change because of DSL evolution. Section 5 gives a conclusion and points out some further research that still needs attention.

2 Scenarios

The problem of versioning will be introduced using three scenarios. These scenarios are taken from real projects that Avanade did either internally or together with a customer. Both horizontal and vertical DSLs will be shown. This is to show that the same kinds of problems that apply to horizontal DSLs also apply to vertical DSLs.

2.1 Scenario 1: SOA Factory vs. Web Service Software Factory

The first scenario is a comparison between Avanade's SOA Factory and Microsoft's Web Service Software Factory version 3 [1]. Both of them are considered to be Software Factories [7] , but for this case study only the DSL part of them has been used. This case is interesting for Avanade as an internal project because customers running application built using SOA Factory might like to convert to Microsoft's Web Service Software Factory version 3 in the future. Both factories are used to implement Service Oriented Architectures. Avanade's SOA Factory was developed internally at Avanade [11], while Microsoft's Web Service Software Factory is developed by Microsoft and widely available. It might be interesting for Avanade to use Microsoft Web Service Software Factory as a base for a newer version of Avanade's SOA Factory. From this perspective parts of Microsoft's web service software factory can be seen as a new version of parts of Avanade's SOA Factory. The other way around is also a possibility. Since Avanade's SOA Factory is based on ACA.NET [11], it might be interesting for a customer to migrate from Microsoft's Web Service Software Factory to Avanade's SOA Factory.

Figure 1 shows some part of the Avanade SOA Factory DSL. SOA Factory is used by Avanade to implement applications based on Service Oriented Architecture principles for customers. Microsoft DSL Tools supports two types of relations: embedding relations and reference relations [5]. The solid arrows indicate an embedding relation, while the dotted arrows indicate a reference relation.

Figure 2 shows part of a pre-release of Microsoft Web Service Software Factory version 3. The same part of the DSL is shown as was shown for Avanade SOA Factory DSL in Figure 1. There are two differences between the two models. The first difference is that FactorySolution has been given another name and is now called ServiceContractModel. The second difference is that the contractgroup has been removed. The service is now directly connected to the ServiceContract.

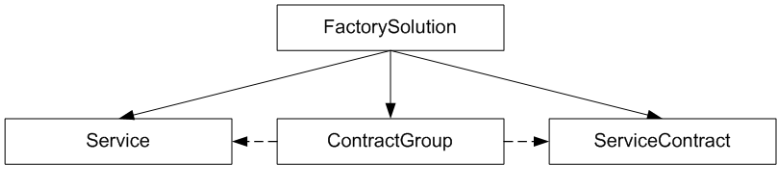

Fig. 1. Part of Avanade SOA Factory DSL

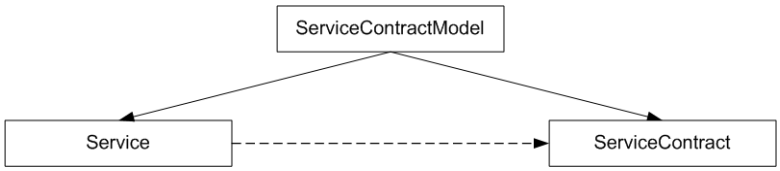

Fig. 2. Part of Microsoft Web Service Software Factory DSL

The problem that occurs is that models created using the SOA Factory DSL will not be interpreted correctly by the Web Service Software Factory DSL. In this particular example, DSL Tools does not even open the model created using the older version of the DSL. However, developers want to use the new DSL for their old models, because the new DSL most likely contains features that the developer needs in order to complete the work he/she needs to implement. Since these features are not supported by the older version of the DSL, a conversion of the model is necessary. This introduces the need for a framework that lets the developer use the old model, but using the newest version of the DSL.

This scenario introduced the problem of versioning in a horizontal DSL. The next two scenarios will give examples of versioning in vertical DSL's.

2.2 Scenario 2: Vertical DSL

The comparison of a part of Avanade's SOA Factory and a part of Microsoft Web Service Software Factory was an example of a conversion between models of different vendors. The second scenario is the result of changing requirements during the development of a vertical DSL that was developed for an insurance company using Microsoft DSL Tools. The domain of this DSL is medical statements. A medical statement is a list of health related questions like whether a person smokes or has some ailments. For each selected type of ailment, questions like "is the person still under supervision?" and "does the person still have complaints?" will be asked. For a lot of these questions, lists with possible answers that can be selected by the user, called option lists, are defined. The change request discussed here is the possibility to reuse the option lists. A simplified version of the DSL is shown in Figure 3 (before the changes are made) and Figure 4 (after the changes are made).

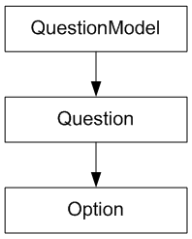

Fig. 3. Simplified Questionnaire DSL

To migrate a model that is based on the DSL definition of Figure 3 to a model based on the DSL definition of Figure 4, a list for each question needs to be created and connected using reference relations. The problem of duplicate lists, however, is not solved. The user now has the possibility to delete the duplicate lists. These actions require human intervention because the user intention in deciding which lists to use cannot be automatically

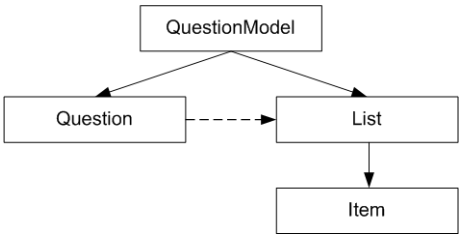

Fig. 4. Simplified Questionnaire DSL with reusable option lists

detected. An example could be questions asking whether someone is still under supervision or has complaints. Both questions could be answered with either yes or no. However, the meaning of these options is different and the developer might choose to use different lists for each of these questions.

This example introduced the versioning problem for a vertical DSL for insurance companies. The next scenario will introduce a different problem with the same DSL.

2.3 Scenario 3: Structural change

In the third scenario, the conversion method will be applied on a case where, due to complexity, a model will be transformed to a model using a structure that is closer to the structure of the target artifact. The source DSL has the possibility to define questions in a nested structure. The target artifact however will be a single two column HTML table with a row for each question. The left column will contain the question and the right column will contain the input boxes. Sub questions will be shown using an indentation in the user interface. In Figure 5, an instance of the source model is shown. Figure 6 shows the target model.

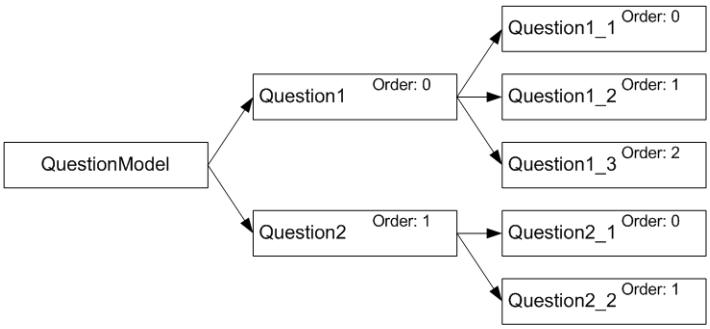

Fig. 5. Model showing nested questions

These scenarios are just examples to introduce the problem of versioning and show how many different cases there are. In the cases that we have looked at, we have seen many more very interesting scenarios. Using all these scenarios an application was created to convert models automatically. This has significantly decreased the amount of time needed to migrate an application to a newer version of the DSL.

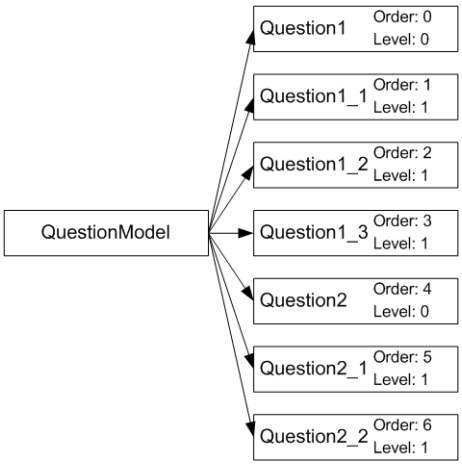

Fig. 6. Model using a level attribute on questions as alternative to nesting

3 General Solution

Figure 7 shows the problem of evolution in DSLs in a more general way. DSL A is a DSL that has been developed previously. DSL B is a newer DSL that has almost the same features as DSL A. The function F describes the changes that have been done by the developers to DSL A in order to produce DSL B. The parameter a is defined as $< C, P, R >$: a tuple of classes, properties and relations as described in the DSL definition of DSL A.

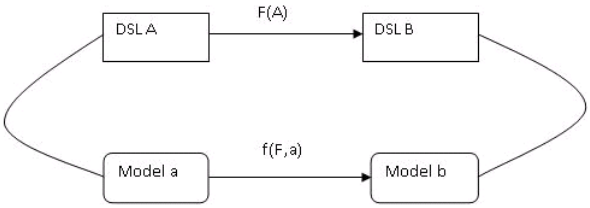

Fig. 7. General Solution

The function F is used as an input parameter for the function f(F,a). This function f describes how to convert any model a, based on a DSL A, into a model b, based on a DSL B. In other words, the function f(F,a), describes a model-to-model transformation between a *model a* based on *DSL A*, to a *model b* , based on *DSL B*.

Using the notion of how the different DSLs and models are related to each other as described in Figure 7 a framework can be developed to describe the two functions. The developers only have to describe the function F as this is very specific to the changes they have made to a specific DSL. The function f is a function that only has to be developed once and will work for every DSL and every model, because it takes the function F as one of its input parameters.

Figure 8 describes the framework used 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.

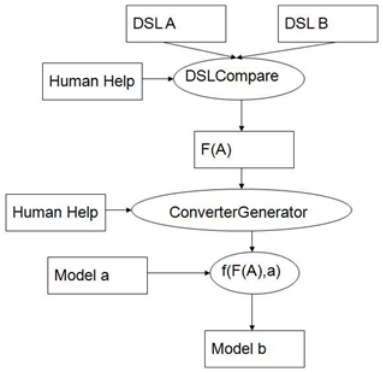

Fig. 8. DSL Versioning Framework

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 function F is already defined automatically. In the case study, number about the automatic detection are presented.

The ConverterGenerator is used to embed function F in the function f, which has F as an input parameter. It creates a tool in which the developer of model a, can load the model and convert this model to model b. Some amount of human effort is typically necessary for complementing and validating the ConverterGenerator. There are some very specific cases that cannot be captured by the DSLCompare tool, for example the numbering issue from scenario 3. This means that the function F that is the result of the DSLCompare tool, is incomplete. The developer is able to complete the function F in this stage and at the same time embed this already in the function f. Hence, a tool is generated to which only the model a has to be given to create model b.

It is questionable whether the actual conversion also needs some help from the developers. It might be the case that some properties in model b cannot be derived from the function F and the model a. We have not come across such cases yet, however.

4 Mapping DSL

The function F is described using a DSL that describes the mapping between DSL A and DSL B. This DSL is based on XML. Usually the developer does not need to write this XML by hand, because it will be generated by the DSLCompare tool. However, there might be situations in which the developer wants to edit the XML by hand, or wants to lookup a part of the mapping manually.

The Mapping DSL is a specialization of model-transformation languages like ATL [3] and KMTL [6] or an XML transformation language like XSLT. Those languages support more complicated transformations than the Mapping DSL used here. The Mapping DSL only supports transformations that are most common for DSL versioning problems. If some more advanced transformations are needed, the language developer will have to provide them by hand.

Some part of a mapping defined using the Mapping DSL is shown in Figure 9 . Note that all elements of a DSL are mentioned, even if they have not been changed. The mapping DSL consists of three main sections:

– Domain classes
– Domain properties
– Relationships

The domain classes section can be used when the name of a class has been changed, in case of the rename of FactorySolution to ServiceContractModel as shown in Figure 1 and Figure 2. However, classes that merge or split can also be specified here.

The domain properties section is used to map values of properties between the different versions of the DSL. The properties are not shown in Figure 1 or Figure 2. In DSL Tools the classes have properties, like they have in languages like C# and Java. In a model, an instance of a class can be given values for these properties. The mapping DSL supports renaming of such properties as well as moving them to other classes. However, changing the values of the properties is not supported and has to be provided through human help as shown in Figure 8.

The relationship mapping can be provided in two different ways. The first way is used as a relationship in the target model, which can be derived from a single relation, is the source model. The relation between ServiceContractModel and Service in Figure 2 is an example of such a relation. It can be derived from the relation FactoryModel to Service as shown in Figure 1.

The second way to define a relationship mapping is used when multiple relations in the source model are needed to derive a single relation in the target model. The relation between Service and ServiceContract in Figure 2 is an example of a relation that needs multiple source relations to be derived. In this case those relations are the relation between ContractGroup and Service and the relation between ContractGroup and ServiceContract.

Figure 9 shows many more properties for relations than just their name and source and target classes. This is because there are all kinds of other properties that can be given to a relation in DSL-Tools. For instance it can be specified whether the same relation can be used multiple times between the same objects in a model.

The SourceTargetClass property of a relationshipmap indicates the targetclass of a relation in the source model. Using the same convention, the properties sourcesourceclass, targetsourceclass and targettargetclass can be explained. Because all these properties of a relation can change between different versions of a DSL, they are all described in the mapping.

When the XML as shown in Figure 9 is given to the ConverterGenerator as shown in Figure 8, the ConverterGenerator adds an algorithm to the mapping. This algorithm converts the source model into a target model, without changing to source model. Hence, the source model is read only, while the target model can be read and written to as well. The reading stage of the target model is used for efficiently converting the relations between different elements of the model.

5 Case Study

A case study is performed to the functioning of the mapping DSL. For this case study, two different versions of Avanades SOA Factory have been used. In this section one of the models used for this case study is described to illustrate the case study.

Avanades SOA Factory Beta 2 and Avanades SOA Factory Release 1.0 differ in many ways. Some examples are:

```
- <Mappings>
  - <DomainClasses>
      <ClassMap SourceClass="namedElement" SourceClassKey="Id" TargetClass="namedElement" TargetClassKey="Id" />
      <ClassMap SourceClass="factorySolution" SourceClassKey="Id" TargetClass="factorySolution" TargetClassKey="Id" />
      <ClassMap SourceClass="boundClient" SourceClassKey="Id" TargetClass="client" TargetClassKey="Id" />
      <ClassMap SourceClass="unboundClient" SourceClassKey="Id" TargetClass="client" TargetClassKey="Id" />
    </DomainClasses>
  - <Properties>
      <PropertyMap TargetClass="NamedElement" TargetProperty="Name" SourceClass="NamedElement"
        SourceProperty="Name" />
      <PropertyMap TargetClass="Client" TargetProperty="Name" SourceClass="BoundClient" SourceProperty="Name" />
      <PropertyMap TargetClass="Client" TargetProperty="Name" SourceClass="UnboundClient" SourceProperty="Name" />
    </Properties>
  - <Relationships>
    - <Relation Name="FactorySolutionHasPropagations" SourceClass="factorySolution" TargetClass="client"
        SourceName="Propagations" AllowsDuplicates="False" IsEmbedding="True">
        <Relation Name="FactorySolutionHasPropagations" SourceClass="factorySolution" TargetClass="boundClient"
          SourceName="Propagations" AllowsDuplicates="False" IsEmbedding="True" />
      </Relation>
    - <Relation Name="FactorySolutionHasPropagations" SourceClass="factorySolution" TargetClass="client"
        SourceName="Propagations" AllowsDuplicates="False" IsEmbedding="True">
        <Relation Name="FactorySolutionHasPropagations" SourceClass="factorySolution"
          TargetClass="unboundClient" SourceName="Propagations" AllowsDuplicates="False" IsEmbedding="True" />
      </Relation>
    </Relationships>
</Mappings>
```

Fig. 9. Part of a mapping between SOA Factory and Web Service Software Factory

- Some classes are merged together. For example, in Avanades SOA Factory Beta 2, a class BoundCliend and a class UnboundClient existed. In Avanades SOA Factory Release 1.0, these two classes are merged together to become a class called Client.
- Some properties in classes are moved to different classes. In Avanades SOA Factory Beta 2 there were several classes that had some properties about types. In Avanades SOA Factory Release 1.0 these properties are moved out of the classes and put into a single class called TypeInformation. All classes have a reference to that class TypeInformation. This is according to Scenario 2.
- In some cases it was allowed in SOA Factory Beta 2 to have objects reference to other objects multiple times. For instance, a service was allowed to be part of the same group multiple times, without any additional meaning. In SOA Factory Release 1.0 this was not allowed anymore.

Measurements to the performance of the DSLCompare tool have been performed to multiple DSLs. The measurement between between SOA Factory Beta 2 and SOA Factory Release 1.0 is shown in Table 1 (Note that properties and relations are counted multiple times when multiple classes inherit a property or relation from a base class).

Direct mappings mean mappings between classes, properties or relations that have not changed. It is shown that these can be detected automatically by the DSLCompare tool.

Indirect mappings mean that these classes, properties or relations have changed in some way. It is shown that 35 of the 36 changes can be recognized automatically by the DSLCompare tool.

Deleted means that some entities have been removed in Release 1.0. Introduced means that some entities have been introduced. The totals give the total amount of classes, properties and relations in the DSL.

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

Table 1. Comparison and automatic mapping performance of DSLCompare between SOA Factory Beta 2 and SOA Factory Release 1.0

Using the delta defined using the DSLCompare tool as described in Table 1 a converter is generated to convert between SOA Factory Beta 2 and SOA Factory Release 1. The converter still needs 5 lines of custum-code (human help) to do the conversion of the type information in the right way.

6 Conclusion

This paper shows a possible solution for the transformation between DSLs that have evolved over time. A mapping DSL is introduced to show how mappings between different versions of a DSL can be formally described. Furthermore a framework is introduced to support this DSL (automatic recognition) and create the actual mapping. This description of a mapping is used by the framework to generate a conversion tool for models. Using the conversion tool those models are transformed to models based on an evolved DSL definition. The approach shown is applicable to DSLs that share a common domain. Cases to both horizontal and vertical domains have been presented showing that the proposed solution is applicable to a wide range of domains.

In the case studies conducted, it turned out that most of the changes of a DSL are recognized automatically by the framework. Furthermore only few lines of custom code are necessary to create a fully working converter. This saves valuable time for developers when migrating a model to a newer version of a DSL.

7 Further Research

During the case studies that have been performed within Avanade, other interesting problems related to DSL versioning came across. Versioning in DSLs has multiple issues. The one discussed above was the issue about versioning of the DSL definition. However, there are at least three more issues that need to be addressed:

– Versioning of models
– Versioning and reuse of versioning transformations
– Creating a visual mapping DSL

Versioning of models is a problem that occurs in every day development process. Just like development with a general purpose language, where versioning of code is necessary, the same holds for development using DSLs.

Versioning of transformations is versioning the mapping between two languages, because the mapping might improve in performance or reliability over time. However, more important is the need to reuse the mapping and make small changes to it. Above, the function F was defined as the mapping between DSL A and DSL B. However, in practice some developers do not exactly use DSL A. Some very small parts of DSL A are changed to get DSL A*. When the customer wants to migrate to DSL B, the function M cannot be used to create the converter, because the model was defined using DSL A*. There is only a small difference between DSL A and DSL A*, so one would like to reuse M in some way.

Creating a visual mapping may help people to better understand the mapping that is created and the concept behind it. A possible approach for this is the use of graph grammars and graph rewriting theories that have been used for decades [10] [4]. This approach has already been applied on UML models [12].

References

1. Web service software factory community. http://codeplex.com/servicefactory.
2. P. Klint A. van Deursen and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices, vol 35 no 6*, 2000.
3. J. Bezivin, E. Breton, G. Dup, and P. Valduriez. The ATL transformation-based model management framework. Technical report, University of Nantes, 2003. TR03-08.
4. D. Blostein and A. Shurr. Computing with graphs and graph rewriting. *Software - Practice and Experience, 6th Proceedings in Informatics*, 1997.
5. S. Cook, G. Jones, S. Kent, and A.C. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley, 2007.
6. D.H.Akehurst, W.G.Howells, and K.D.McDonald-Maier. Kent model transformation language. 2005.
7. J. Greenfield and K. Short. *Software Factories*. Wiley, 2004.
8. A. Johnston and S. Brown. Using service-oriented architecture and component-based development to build web service applications. *Rational-IBM white paper*, 2003.
9. J.H. Perkins. Automatically generating refactorings to support API evolution. *Program Analysis for Software Tools and Engineering*, 2005.
10. G. Rozenberg. *Handbook on Graph Grammars and Computing by Graph Transformation: Foundations*. 1997.
11. A. Savelkoul and D. Mulder. Building a DSL for an existing framework, 2007. http://msdn2.microsoft.com/en-us/library/bb381702(vs.80).aspx.
12. J. Sprinkle and G. Karsai. A domain-specific visual language for domain model evolution. *Elsevier Vol14 (3-4)*, 2004.