Delft University of Technology
Software Engineering Research Group
Technical Report Series

Generating Version Convertors for Domain-Specific Languages

Gerardo de Geest, Sander Vermolen,
Arie van Deursen and Eelco Visser

Report TUD-SERG-2008-037

TUD-SERG-2008-037

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Accepted for publication in the Proceedings of the Working Conference on Reverse Engineering (WCRE), 2008, IEEE Computer Society.

Generating Version Convertors for Domain-Specific Languages

Gerardo de Geest
Avanade Netherlands B.V.
The Netherlands
Gerardo.de.Geest@avanade.com

Sander Vermolen Arie van Deursen Eelco Visser
Delft University of Technology
The Netherlands
{*S.D.Vermolen, Arie.vanDeursen, E.Visser*}*@tudelft.nl*

October 2, 2008

Abstract

Domain-specific languages (DSLs) improve programmer productivity by providing high-level abstractions for the development of applications in a particular domain. However, the smaller distance to the application domain entails more frequent changes to the language. As a result, existing DSL models need to be converted to the new version. Manual conversion is tedious and error prone.

This paper presents an approach to support DSL evolution by generation of convertors between DSLs. By analyzing the differences between DSL meta-models, a mapping is reverse engineered which can be used to generate reengineering tools to automatically convert models between different versions of a 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's software development practice.

1. Introduction

Domain-specific languages (DSLs) improve programmer productivity by providing high-level abstractions for the development of applications in a particular domain [5, 14]. In addition to shorter development times, DSLs also bear the promise of lower maintenance costs [14]. First, a dramatically reduced code size (an order of magnitude reduction is not unusual) simplifies maintenance [4]. Second, automatic generation of the implementation entails that only the model needs to be maintained, not the code generated from it. This is particularly important when considering that 80% of the costs of software are usually spent in maintenance [9, 11].

While maintenance of applications developed with a DSL is simplified, the use of DSLs introduces a new type of maintenance problem: model conversion due to language evolution. The close connection of a DSL to an application domain makes it sensitive to changes in requirements and technical solutions in that domain, which will have to be reflected in changes in the language [10]. Due to such changes, existing models conforming to a DSL definition can become syntactically invalid, and cannot be used with the generator for the new language (version).

Hence, most DSLs will have to evolve over time including the models 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 [12]. Thus, DSLs can only help to reduce the overall maintenance costs of software if the cost of language evolution can also be kept low [4].

In this paper, we present an approach to support DSL evolution by the generation of convertors for migration of models. The solution that we propose is to reverse engineer the language evolution by computing the differences between DSL meta-models (language definitions). Based on these differences, we generate reengineering tools that can semi-automically migrate models conforming to one version to a next version.

The context in which we have experienced this problem, and for which we have implemented our solution, is that of Microsoft's *Visual Studio DSL Tools* [3]. It allows developers to define DSL meta-models as well as actual models in a graphical manner.

This paper is structured as follows. In Section 2 we cover related work in the area of DSL-evolution. In Section 3, we summarize the key ingredients of Microsoft's DSL Tools. Then in Sections 4–6, we describe our approach to support DSL evolution, including the identification of differences and the generation of version convertors. In Section 7 we use an implementation of our approach to evaluate our approach using an industrial case study. In Section 8 we reflect on our findings, after which we conclude with a summary of key contributions, as well as suggestions for future research.

2. Related Work

Versioning of DSLs is closely related to evolution in the context of meta-modeling environments. Several approaches have been proposed to tackle the versioning problems in these areas. In this section we discuss the most related.

Vermolen and Visser [15] propose a generic framework to support coupled evolution independent of the specific domain. They propose an automatic generation of a transformation languages (DSTLs) that is specific to a given domain such as DSLs. The generated DSTLs are based on basic

transformations rather than the direct mappings presented in this paper. The framework also supports execution of specified evolution as well as obtaining automated migration of DSL applications. It does not support automatic evolution detection as in this paper.

In the context of domain specific languages, Pizka et al. [13] discuss the evolution of DSLs and related artifacts. They recognize program evolution as a major problem and propose a language to specify DSL evolutions. In contrast to our work, this language is based on small evolution operations and DSL evolution needs to be specified manually. Their (prototypical) implementation is limited to textual DSLs and evolution of DSL compilers.

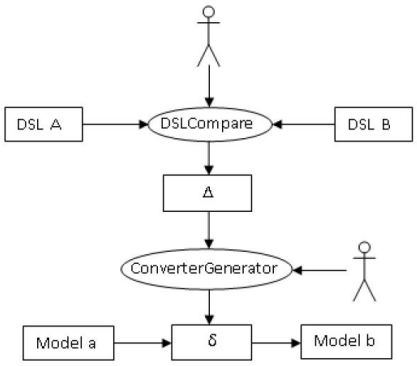

Figure 1. DSL Versioning Framework

3. DSL Tools

In our work, we have used Microsoft DSL Tools to define domain-specific languages. Microsoft DSL Tools is a separate package for Visual Studio (Microsoft's IDE). It provides a complete toolset supporting the definition of DSLs (meta-models), development of DSL applications (models) and generation of executable code. However, support for versioning DSLs is not included.

A model (DSL application) defined in DSL Tools conforms to a certain meta-model (the DSL). Similarly, the meta model conforms to a meta-meta model. This meta-meta model is implicitly defined by DSL tools and not publicly available. In order to reason about DSLs, we therefore have inferred a meta-meta model from our knowledge of the tool set. We use this model as the basis of our work, yet are aware that it is a subset of the real DSL Tools meta-meta model. Only parts relevant to our context are included. The inferred model is shown in the upper part of Figure 2.

DSL Tools uses a universal unique identifier (UUID), just as the XMI-format does to serialize models. This UUID is useful for our purposes as well, since deriving DSL transformations is greatly simplified by requiring that each element has a UUID [6]. In practice, several other operations on models also use such an identifier [2]. 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.

4. Approach

Due to a changing domain or changing requirements, DSLs need to evolve. When the DSL has been in use, DSL applications have been developed that conform to the old version of a DSL, yet may no longer be usable with a new DSL version. We have developed a framework to solve the problem of DSL evolution. It supports semi-automatic evolution detection as well as generation of automatic model migration. Its outline is shown in Figure 1.

At the top of the outline, two DSLs are input to the DSLCompare process: an old DSL version *A* and a new DSL version *B*. The DSLCompare process will compare the two DSL versions and derive a mapping Δ, which defines a transformation from DSL *A* to *B*. This derivation is semi-automatic, but, as we will see later, tending towards fully-automatic. DSLCompare and Δ are the topics of Section 5.

At the bottom-level of Figure 1, a derived Δ is used as input to the ConverterGenerator process. This will generate a model transformation δ. We dicuss the ConverterGenerator and δ in Section 6.

Some amount of human help is usually required for completing and validating Δ and δ. There are some rather specific cases that cannot be captured by the DSLCompare process or the Convertergenerator process. In the case of Δ, some user input might be required to complete the DSL transformation. In the case of the model migration δ, this human help is to support the migration of models e.g. for semantical mappings.

5. Deriving DSL transformations

The DSLCompare process derives a DSL transformation from two given DSL versions. The output of DSLCompare is a transformation written in language we developed. The language aims at DSL evolution, with simplicity as a goal. In this section we first discuss the DSL transformation language and then focus on deriving DSL transformations.

A transformation from one DSL version to another can be modeled by transformations of old components to new components. This is shown graphically in Figure 2. The top of Figure 2 shows the meta-meta model we inferred from DSL Tools. The bottom of this figure shows the transformation language and its relation to the original meta-meta model. A transformation is modeled by the 'Mapping' class. It consists of mappings between each of the possible DSL components.

A meta-model is a model and hence theories applicable

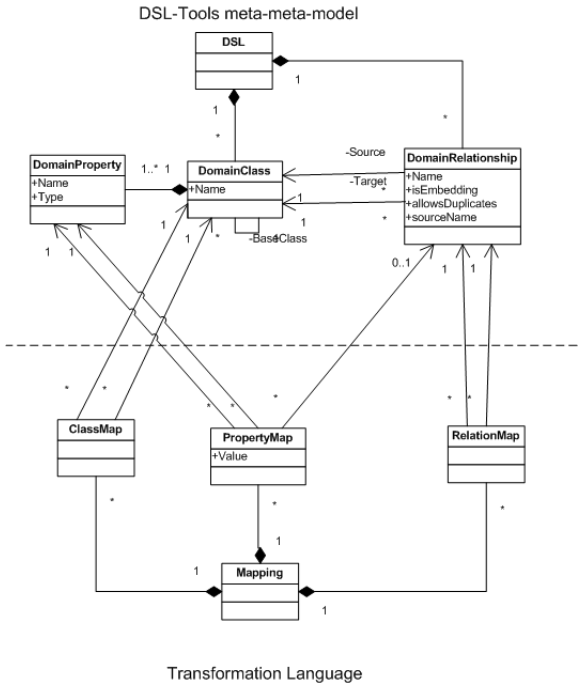

Figure 2. Combination of languages

to models can be applied to meta-models as well. Note that a DSL definition is also a meta-model. Several approaches are already developed to automatically detect the difference between two models [2] [8]. The approach presented here uses the approaches of [2] and [8], 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.

6. Model migration

The ConverterGenerator as shown in Figure 1 generates a migration on models (δ) from a transformation on meta-models (Δ). The generated migration uses a depth-first algorithm that can be split into three stages: (1) transformation of domain class instances; (2) transformation of domain relationship instances and (3) identification of the root node of the new model.

In the first stage, the instances of domain classes and their associated domain properties are migrated. This will result in the nodes of the target Abstract Syntax Graph (ASG). Next, the instances of domain relationships are migrated, which results in the edges of the target ASG. The last case considers the possibility that the ASG is not a single connected graph. In this case the user needs to select the desired graph. This occurs when domain relationships are removed from the DSL definition.

7. Evaluation

To evaluate the solution proposed in this paper, an implementation of our approach has been created. We have applied the implementation to three case studies, all within the context of actual software development projects conducted within Avanade. Details about all three cases can be found in [6]. In this paper, we elaborate on one of them, related to the evolution of Microsoft Web Service Software Factory Modeling Edition (WSSF) [1].

7.1. Questions

In addition to the evaluation of our tools which was also done in the other two case studies, this specific case study answers three additional questions: (1) Does the proposed approach work for multiple DSL definitions and models conform these DSL definitions? (2) How much human help is necessary to the DSLCompare tool and ConverterGenerator? (3) How does the time needed to construct a converter manually compare to the amount of time needed when using our approach?

7.2. Experimental Design

For each of the case studies conducted, including the one presented next, we used two DSL versions and a real-life model conforming to the old DSL definition. In each case study we performed three steps: First we derived a transformation from the old DSL to the new. Then we generated a model migration from the derived DSL transformation and finally we migrated a model.

Transformation derivation We used the DSLCompare tool to create a transformation from the old DSL to the new DSL. From this, we recorded the number of automatically derived mappings and the number of manually derived mappings for each type of component. The results are shown in Table 1. This table is at first separated in three sections: domain classes, domain properties and domain relationships. For each of these sections, we look at the number of direct and indirect mappings. Direct mappings mean that nothing changed. Indirect mappings mean that something changed, for example the name of a domain class. Furthermore, we also distinguish between introduced and deleted elements.

Migration generation Once the transformation between the two DSLs was created, we have generated a migration for models conforming to the source DSL. This is done by using the ConverterGenerator application. We checked whether we needed additional C# code to support the migration.

Migration Furthermore, for each case study we have a model conforming to the old DSL. This model will be migrated to conform to the new DSL. We have consulted the developers of the old model to check whether the migrated model is equivalent to the model they originally developed. Developers were consulted, because we do not have a formal definition of when models are equivalent when conforming to different DSLs.

			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 1. Comparison and automatic mapping of DSLCompare between the datacontract DSLs of WSSF b117 and WSSF Final

7.3. WSSF Case Study 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: data contract DSL, service contract DSL and host DSL. One needs at least one instance of every DSL to create a working application. Hence, one application consists of at least three models. A more thorough description of Microsoft Web Service Software Factory Modeling Edition can be found in [7]. In this paper we only focus on the data contract DSL.

7.4. The WSSF Case Study

The first step is to compare both versions of the data contract DSL. This is done using the DSLCompare tool. The DSL was developed by Microsoft and we are not aware of which domain class name maps to which. Fortunately, the automatic detection feature of DSLCompare is based on more than just the domain class name and does the analysis for us. We did not need to map any domain class ourselves. After checking with Microsoft, the automatic detection turned out to be correct. This was also the case for the domain properties and the domain relationships.

The results of how well the automatic detection algorithm works for the data contract DSL are shown in Table 1. As can be seen, a lot of changes occurred to this DSL, because the number of indirect changes is much higher compared to the number of direct changes. However, no human help was needed to create the transformation between the two DSL definitions.

Now that the transformation between the two versions of the data contract DSL has been defined, the converter for the models can be generated. When testing the migration on our model, the output model is syntactically valid to the DSL definition of the datacontract DSL in WSSF final.

7.5. WSSF Case Study Conclusions

The first question we asked in this case study was whether the solution proposed in this thesis would work for different DSLs. We showed that we were able to generate model converters for all three DSL definitions and that we were able to migrate models conform to these DSL definitions.

The second question we asked was how much human help is necessary. Table 1 shows that no human help was necessary to create the converter in this case study.

The third question we asked in this case study was whether the solution proposed in this paper would create a converter faster than manualy creating one. The case study 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. The converter for the data contract DSL can be created within 10 minutes.

8. Discussion

Following Yin's criteria for judging case study designs, we discuss the most important threats to construct and external validity, as well as the case study's reliability [16]. Note that the case study is explorative in nature, for which threats to internal validity do not apply.

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. A key ingredient of our case study consists of the quantitative data collected in Table 1. This table captures the changes that occured to the DSL definition and also captures how many elements our tool mapped automatically.

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. In total we have used four different DSL definitions for our case studies, of which one is described in this paper. The results for the other cases were similar to the one described here.

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 [16]. For the case presented in this paper, all details are available publicly, and the relevant DSL definitions can be downloaded from [1]. The manual mappings are described in [6]. To create a migration from this transformation, the reader could use the algorithm as outlined in [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 [7] and test the migration using these models.

9. Concluding Remarks

In this paper, we have addressed the DSL-evolution problem: The close connection of a DSL to an application domain makes it sensitive to changes in requirements and technical solutions in that domain, which will have to be reflected in changes in the language. Our key contributions to deal with this problem are as follows.

First, we offer a framework to cope with DSL-evolution. This framework consists of (1) a representation of DSL metamodels comprising domain classes domain properties, and relationships between classes; (2) a comparison process capable of computing differences between DSL definitions; and (3) a convertor generation process that can use the differences identified to produce reengineering tools;

Second, we provide an implementation this framework for the Microsoft DSL Tools. The implementation can read arbitrary DSL definitions, compute the differences between them, involve the user in dealing with complex cases, and generate C# code to support the automatic conversion between models in those two DSL versions.

Third, we offer an evaluation of the proposed approach by means of a case study based on Microsoft's publicly available Web Service Software Factory Modeling Edition. The case study shows that the automatic detection performs satisfactory in real-life scenarios.

As part of our future work, we anticipate that our tool set will be further used within various Avanade projects that are using Microsoft's DSL Tools. Furthermore, we are considering various refinements to the automatic change detection tool, in particular by also recognizing mappings between domainclasses and domainproperties or domainrelationships. In the current approach only mappings between two entities of the same kind can be recognized.

Acknowledgments This research was partially supported by NWO/JACQUARD project 638.001.610, *MoDSE: Model-Driven Software Evolution*. We also would like to thank Edwin Jongsma from Avanade for his support throughout the project.

References

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

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

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

[4] A. van Deursen and P. Klint. Little languages: little maintenance. *Journal of Software Maintenance*, 10(2):75–92, 1998.

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

[6] G. W. de Geest. Building a framework to support domain-specific language evolution. Master's thesis, Delft University of Technology, 2008.

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

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

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

[10] 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.

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

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

[13] M. Pizka and E. Jurgens. Tool Supported Multi Level Language Evolution. *To Appear*, 2007.

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

[15] S.D. Vermolen and E. Visser. Heterogeneous Coupled Evolution of Software Languages. Technical Report TUD-SERG-2008-028, Delft University of Technology, Software Engineering Research Group, 2008.

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

TUD-SERG-2008-037
ISSN 1872-5392