A UML-Agnostic Migration Approach From UML to DSL

Hendrik Bünder
buender@itemis.de
itemis AG

Abstract

Compared to UML-based modeling, domain-specific languages (DSL) offer many advantages such as simplified tooling and lower costs. When migrating existing UML models to DSLs, however, it is difficult to guarantee that the same source code is generated afterwards. Due to the fact that existing UML models and the former generated source code may already be inconsistent. To cope with these inconsistencies the paper reports on a UML-agnostic migration approach based on the existing source code. The paper elaborates on the concept, the tool-chain and the test environment of the introduced approach.

1 Introduction

Over the last decade, open source frameworks for creating domain-specific language workbenches have improved significantly. Thereby, it has become easier to utilize model-driven software engineering based on domain-specific languages (DSL). However, many companies already adopted model-driven engineering at the beginning of the 21st century. Typically, Unified Modeling Language (UML) models are the basis for these approaches [1]. In most cases, these models are created and maintained with expensive proprietary tools such as Rational Software Architect or Enterprise Architect.

Obviously, companies willing to migrate from a UML-based to a DSL-based approach face many challenges. First, all concepts of the former approach need to be available in the domain-specific language. Second, the migration needs to be automated to handle a large number of existing UML models. Third, the migration approach must cope with inconsistencies between the models and the actual source code. Finally, the source code generated from the migrated DSL model has to equivalent to the source code used as input for the migration.

The following will elaborate on an automatic migration from UML-based to DSL-based model-driven software engineering performed under the precondition that all present concepts are available in the DSL approach. To handle inconsistencies between UML model and actual source code, the latter is the only migration input. Also, the following will describe the test process to ensure that the newly generated source code is equivalent.

2 Migration Approach

To illustrate the migration process the data layer of an application written in Java will be migrated to an instance of an Entity DSL. The language workbench for the Entity model was created using the Xtext [4]. Based on a grammar written in the Xtext-specific Extended Backus-Naur Format, the editor, parser and abstract syntax tree for the Entity DSL is generated. Besides, an Entity to Java code generator was implemented using the Xtend programming language [3].

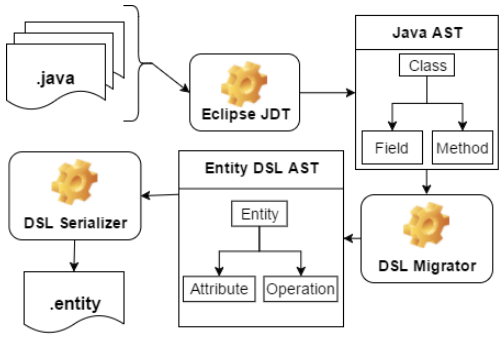

Figure 1: Migrating from Source Code.

Figure 1 shows the process of migrating existing source code written in Java to a textual representation of an Entity DSL model. Since Java input and Entity DSL are both managed within the Eclipse IDE, the migration utilizes further Eclipse plugins. First, the Java sources of a project to be migrated are parsed by the Eclipse Java Tools. Based on the instantiated Java abstract syntax tree (AST) the *DSL Migrator* creates elements of the Entity AST. For every Java class extending the `DataObject` interface, the *DSL Migrator* creates a new instance of the Entity class. Moreover, for every field or operation in such a class, a new attribute or operation is created, respectively. In addition, to the AST elements shown in the simplified example, the *DSL Migrator* also creates more detailed artifacts such as data types, parameters or exceptions. Although the migration approach relies on the source code, this source code has to abide by a particular

structure regarding available class files, implemented interfaces, etc. to be successful.

After the Entity DSLs AST has been created, the *DSL Serializer* serializes the content of the AST. The *DSL Serializer* has been inferred from the the Xtext-specific grammar and will create a text file accordingly. As Figure 1 has shown, the whole migration process can be executed using tools from the Eclipse ecosystem.

3 Testing the Migration Approach

After implementing the migration process, automated tests verified that the generated Java files are equivalent to the previous source code. The tests were implemented using the Xpect framework which is dedicated to testing Xtext domain-specific languages [2]. Besides support for parser, formatting and proposal tests, Xpect also offers functions for setting up complete Eclipse workspaces.

Figure 2: Testing the Migration Process.

Figure 2 shows the test process implemented using Xpect. In the setup phase of the test, Xpect creates an empty workspace and imports a predefined sample Java project. The first step of the actual test routine starts the migration process as shown by Figure 1. In the next step, the generator turns the migrated Entity DSL model into Java code. Finally, the test verifies that the generated source code is equivalent to the code used as input for the migration.

The test case described above is suitable for testing if from a known input to the migration process the expected output is generated. However, to test the migration approach thoroughly source code from existing software components should be used as input to the process. Although the migration is still expected to create equivalent source code, the test result should contain additional information.

4 Extending the Migration Approach Test

The following will describe how the test approach was extended to be executed for a variable number of existing software components. Moreover, the report provides detailed data on differences in the generated code, the total execution time, the number of migrated elements, and errors occurred during the migration.

Figure 3 shows the extended process to run the migration test for multiple existing software components. The elements highlighted in green were changed in comparison to the process described in Figure 2. First, the workspace setup imports multiple software components. Second, an additional step in the process creates a detailed report on the migration. Finally, the process persists every migration report per component as a file.

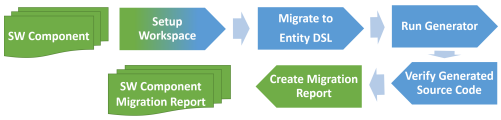

Figure 3: Testing the Migration Process.

The migration is executed on a dedicated continuous integration environment, to migrate the more that 500 software components efficiently. Depending on the software component size each migration takes between 20 seconds and several minutes. To efficiently migrate all software components, they are migrated in parallel doing 100 components per workspace. In this setup, the overall migration takes approximately 4 hours and is executed once a day. Thereby, effects of changes in the `DSL Migrator` or the software components on the migration is reported on a daily basis. A simple web page shows the migration result data per software component.

5 Conclusion

The Eclipse ecosystem offers powerful tools to implement a UML to DSL migration fully. However, the UML-agnostic approach heavily depends on the source code abiding a particular structure. The migration process mainly benefits from the JDT parser and the `DSL Serializer`. While those two handle parsing and serializing, the `DSL Migrator` can completely focus on the model-to-model transformation on the meta-model level.

Additionally, it has been shown that testing the approach with artificially created projects is an important first step of verification. However, to ensure that the migration is working, it must also be tested with existing software components. To summarize, the extended testing approach offers many benefits: first, the migration can be tested automatically for all existing software components. Second, the generated migration report is easily accessible and gives a quick overview of problems during the migration.

All in all, it has been shown that the UML-agnostic migration approach can be implemented and tested completely within the Eclipse ecosystem.

References

[1] OMG. *UML - What is UML.* URL: `http://www.uml.org/`.

[2] Xpect. *Xpect.* URL: `http://www.xpect-tests.org/`.

[3] Xtend. *Xtend.* URL: `https://www.eclipse.org/xtend/`.

[4] Xtext. *Xtext - Language Engineering for Everyone.* URL: `https://www.eclipse.org/Xtext/`.