Towards Incremental Round-Trip Engineering Using Model Transformations

Thomas Buchmann and Bernhard Westfechtel
University of Bayreuth
Chair of Applied Computer Science I
Bayreuth, Germany
firstname.lastname@uni-bayreuth.de

Abstract—Model-driven software engineering is supported with the help of model transformations. At present, the technology for defining and executing uni-directional batch transformations seems to be well developed, while bi-directional and incremental transformations are more difficult to handle. In this paper, we present a bi-directional and incremental transformation tool for round-trip engineering between class diagrams and Java source code. Our focus lies on the model-driven construction of this tool with the help of a set of model transformation tools. Thus, our work serves as a case study of applying and composing existing model transformation tools.

Keywords—round-trip engineering; model transformations; model-driven engineering;

I. INTRODUCTION

Model-driven software engineering puts strong emphasis on the development of high-level models rather than on low-level source code. In an idealized forward engineering process, an initial requirements model is transformed in a series of steps into a working implementation. However, in practice changes may have to be propagated back, resulting in a round-trip engineering process. Furthermore, while large parts of an application may be generated from models, it may still be necessary to complement generated code with manual implementations.

In this paper, we examine a standard use case: incremental round-trip engineering between design models and source code (Figure 1). More specifically, we address the coupling between a UML class diagram and Java source code. Both model and code may be edited concurrently, and changes may be propagated back and forth to maintain consistency.

Model-driven software engineering may be applied successfully only if end users such as designers and programmers are supported by powerful tools. Unfortunately, development of tools for model-driven software engineering requires considerable effort. Thus, tool providers essentially face the same problem as end users: they need powerful meta tools to reduce their implementation effort.

In response to this need, a variety of languages and tools for model transformations have been developed and realized. Using these languages, a tool may define a model transformation at a high level of abstraction, eliminating the need for a laborious implementation in a conventional programming language.

The work presented in this paper was carried out in the context of the Valkyrie project, which is dedicated to the development of a UML-based environment for model-driven software engineering [1]. The main focus of Valkyrie lies on the provision of model editing and model transformation tools. It is important to notice that Valkyrie is built not only for, but also with model-driven engineering. To reduce the implementation effort, we reuse existing technology as far as possible.

As part of the Valkyrie environment, we built a tool for incremental round-trip engineering between UML class diagrams and Java source code. In this paper, we describe how we applied and composed existing model transformation tools to provide the round-trip engineering tool to end users of the Valkyrie environment. Unfortunately, there is no single model transformation tool available which supports incremental and bi-directional model-to-text transformations. Therefore, we had to use multiple tools and compose them in an effective way.

Thus, the work presented in this paper may serve as a case study for applying existing model transformation technology. The case study provides some insight into strengths and weaknesses of the employed model transformation tools. Furthermore, it simultaneously serves as an example of engineering with model transformations.

II. APPROACH

Our round-trip engineering tool demands for incremental bi-directional transformation between model and text. Existing model transformation approaches do not meet this requirement, as the following brief discussion demonstrates.
Template-based code generation supports the generation of source code by templates which mix static and dynamic fragments. Tools such as the EMF code generator [2] were built with this technology. Many model-to-text transformation tools have been realized with the help of template languages such as JET, XPAND, or Mof2Text [3]. With the help of templates, uni-directional model-to-text transformations may be defined. Tools such as e.g. Acceleo, which is based on the Mof2Text standard, in addition support incremental propagation of model changes to the source code.

ATL [4] is a popular tool for defining model-to-model transformations. However, ATL primarily focuses on uni-directional batch transformations. In addition, ATL allows for the definition of in-place transformations on a single model. Only recently proposals have been made for adding bi-directional transformations [5], but those approaches are just prototypical.

Query View Transformations (QVT) [6] is an OMG specification for model-to-model transformations. The specification comprises both a procedural language — QVT Operations — and a declarative language — QVT Relations. Only the latter supports incremental bi-directional model-to-model transformations: In QVT Relations, which has been implemented e.g. in the tool medini QVT1, a single set of transformation rules may be defined which may be executed in both directions (even incrementally). In QVT Operations, each of the directions would have to be realized by a separate transformation definition, and the transformation writer is responsible for specifying mutually inverse transformations.

Triple graph grammars (TGGs) [7] support incremental bi-directional model-to-model transformations, as well. The basic idea behind TGGs is to interpret both source and target models as graphs and additionally maintain a correspondence graph whose nodes reference corresponding elements from both source and target graphs, respectively. The construction of triple graphs is described with a set of production rules which are used to describe the simultaneous extension of the involved domains of the triple graph. The rules themselves do not contain any information about a transformation direction, which makes TGGs an ideal candidate for bi-directional and incremental transformation problems, like the one that is subject of this paper. Tool support in the Eclipse context is provided for example by MoTE [8], an extension to MDELab, eMoflon [9], EMofF [10], or TGG Interpreter [11].

Both QVT Relations and TGGs may be used to synchronize models; however, the Java source code is natively represented as text. Thus, a text-to-model transformation is also required for our round-trip engineering tool. For Java, this transformation is provided by the MoDisco tool, which offers a parser transforming Java source code into an instance of the MoDisco [12] Java metamodel. This metamodel provides a higher level of abstraction than the pure Java abstract syntax tree. Furthermore, it is based on Ecore which allows to use existing model transformation tools in the Eclipse world.

Using some of the model transformation tools mentioned above, we realized our round-trip engineering tool in the following way:

1) For bi-directional and incremental model synchronization, we employed the TGG Interpreter, which is used to maintain consistency between the UML model and the Java model. Alternatively, we could have selected a tool for QVT Relations; this is planned as future work.

2) For uni-directional and incremental model-to-text transformation, we selected Acceleo, which is based on the Mof2Text standard.

3) Finally, for the inverse transformation we reused and adapted the MoDisco discoverer such that it operates incrementally.

An important design issue concerns the division of labor among the components involved in the transformations. With the help of the TGG, a purely structural transformation is performed, i.e., the transformation deals only with the structure (mapping of UML to Java classes, primitive properties to fields, etc.). Behavior is addressed only by the Acceleo-based code generator. In particular, the code generator introduces the access methods for properties which are required to correctly implement the semantics of associations. In this way, the TGG rules are simplified significantly, and proven model-to-text technology may be used to generate method bodies (it would be quite awkward to do this in the TGG rules).

III. Example

In the following we sketch an example usage scenario and describe how the different models involved in the round-trip process interact. Let us start with plain Java source code. The upper left box in Figure 2 shows two simple Java classes. The classes ClassA and ClassB both contain two attributes of primitive and non-primitive type, respectively. Furthermore, ClassA also contains a method doSmth(). After the source code files have been modified, the MoDisco Java discoverer is used to create the MoDisco Java model (Step 1 in Figure 2). Please note, that we needed to adapt the MoDisco framework in order to work in an incremental mode, i.e. in case of subsequent changes to the source code, the existing Java model instance gets updated rather than created from scratch over and over again. After the Java model has been created, a TGG transformation is executed to create a corresponding UML model (Step 2). The non-primitive attributes have been transformed into an association connecting both classes. After that, the user modifies the class diagram and adds a method run() to class

1http://projects.ikv.de/qvt
ClassB. After this change, the TGG transformation is executed in the opposite direction to update the MoDisco Java model. Furthermore, the (incremental) M2T transformation is performed to update the Java source code accordingly. After that, the user implements the method doSmth(). This change triggers the MoDisco discoverer, which updates the Java model. After Step 3, the Java model now also contains accessor methods for attributes or associations such as e.g. getCounter() and the body of doSmth() (not shown). Accessor methods can be identified using the Javadoc tag @accessor as shown in Figure 2. Please notice that the UML model is not changed because the updates of the Java code and model are not relevant for the UML model. Now the user decides to modify the source code and to change the name of the method run() to execute(). Furthermore, the user adds another method named printLogFile() to ClassB. After the source code has been modified, the MoDisco Java model gets updated in Step 4. Afterwards, the TGG transformation is executed to update the UML model accordingly (Step 5). The result is shown in the bottom right box of Figure 2, which depicts the cutout of the class diagram.

IV. Model and Code Synchronization with TGGs

Our modeling tool Valkyrie uses the UML2 metamodel provided by Eclipse MDT project\(^2\). For representing Java programs, MoDisco provides an Ecore-based Java model.

The structure of both models involved in the round-trip engineering process is similar. They both use packages to group elements belonging to the same namespace. A specialized package, the so called Model is used as root object in both models. This fact helps to keep the TGG rules relatively simple in most cases. In the following, we present a sample TGG rule to match UML and Java classes respectively. Due to space restrictions, it is impossible to show all of them. Furthermore, lots of rules have the same structure, like rules for classes, interfaces, enumerations, datatypes or the inheritance hierarchies between these types.

UML class ↔ Java class: Figure 3 shows the TGG rule to match UML classes to Java classes nested in their corresponding containers — UML packages and Java packages, respectively. Each TGG rule describes a synchronous extension of the source graph (on the left), the correspondence graph (in the middle), and the target graph (on the right). Nodes are attached to the respective graphs by edges connecting them to domain nodes (on the top). The left-hand and right-hand sides of a rule are shown in a single diagram.

Nodes and edges to be created are shown in green color and are annotated with ++. Yellow boxes represent OCL attribute constraints. Here, they are used to ensure that both classes have the same name and to ensure that the visibilities of both classes match. From the synchronous rule, two directed rules are derived which create a UML class from a Java class and vice versa. Furthermore, the TGG interpreter propagates modifications such as name changes. Thus, incremental change propagations in both directions come for free without the need for an explicit specification. In the directed rules, the name constraints shown in Figure 3 are used for attribute assignment. For example, in the forward rule the UML class already exists, and its name is fixed. The name of the Java class is assigned the value specified in its attached constraint, which simultaneously satisfies the constraint of the UML class as a postcondition.

UML association ↔ Java fields: As stated above, properties referring to non-primitive types are considered as association ends. In the UML, an association must have at least two ends which are represented by properties referring to the corresponding types involved in the association. There is no equivalent for an association in Java. The corresponding association ends are expressed by fields with the respective

\(^2\)http://www.eclipse.org/modeling/mdt/?project=uml2
are changed. Since QVT relational also provides support for bi-directional incremental model transformations, we will investigate in future work how good or bad QVT-r performs compared to TGGs on this specific use case.

**REFERENCES**


