Handcrafting a Triple Graph Transformation System to Realize Round-trip Engineering Between UML Class Models and Java Source Code

Thomas Buchmann and Sandra Greiner
Chair of Applied Computer Science I, University of Bayreuth, Universitätsstrasse 30, 95440, Bayreuth, Germany

Keywords: Round-trip Engineering, Model-driven Development, Xtend, Triple Graph Transformation Systems.

Abstract: Model transformations are a mandatory requirement for model-driven development, a software engineering discipline, which has become more and more popular during the last decade. Over the years, the concept of unidirectional model transformations and corresponding tool support reached maturity since these kind of transformations are widely used in model-driven engineering, mainly for forward engineering and code generation. In incremental and iterative software engineering processes, forward engineering may be too restrictive since it resembles waterfall-like processes. Thus, bidirectional transformations are required, which aim to provide support for (incrementally) transforming one or more source model to one or more target model and vice versa from only one transformation description. However, they seem to be rarely used in model-driven software development as adequate tool support is missing. On the other hand, programming languages nowadays provide support for higher-level features like closures or lambda expressions which allow to describe transformation patterns in a declarative way. In this paper, we present an approach for round-trip engineering between UML class models and Java source code, which is realized with a triple graph transformation system written in the Xtend programming language.

1 INTRODUCTION

During the last few years, model-driven software engineering has become more and more popular. The main focus of model-driven software engineering is put on the development of higher-level models rather than on source code. Throughout the years, the *Unified Modeling Language (UML)* (OMG, 2015b) has been established as the standard modeling language for model-driven development.

The Object Management Group (OMG) proposes *Model-Driven Architecture (MDA)* (Mellor et al., 2004), a result of a standardization process for core concepts in model-driven software engineering, which put strong emphasis on interoperability and portability. UML serves as the standardized modeling language for MDA. To achieve interoperability and portability, the MDA approach uses both platform independent (PIM) and platform specific (PSM) models and it uses UML to describe both of them.

Model transformations constitute the core essence of model-driven software engineering and especially MDA, where they are used, e.g., to transform PIM to PSM models. Throughout the years, many languages and accompanying tools have emerged (Czarnecki and Helsen, 2006). We observe tools, which support *unidirectional batch* transformations, e.g. ATL (Jouault et al., 2008), as well as languages for *(incremental) bidirectional* transformations, as described in the QVT standard (OMG, 2015a).

Nowadays, tools for specifying and executing unidirectional batch transformations are mature and used frequently in model-driven development. However, in many scenarios transformations of this kind do not suffice: After transforming a source model into a target model, extensions and changes to the target model may still be required. As a consequence, changes to the source model need to be propagated such that manual modifications of the target model are retained. These propagations call for *incremental* rather than batch transformations. Furthermore, changes to the target model may have to be propagated to the source model; then, transformations need to be *bidirectional*.

Altogether, this results in a *round-trip engineering process* in which source and target models may be edited independently and changes need to be propagated in both directions. While several languages and tools have been proposed for bidirectional and in-

Buchmann, T. and Greiner, S.
Handcrafting a Triple Graph Transformation System to Realize Round-trip Engineering Between UML Class Models and Java Source Code.
DOI: 10.5220/0005957100270038
In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 2: ICSOFT-PT*, pages 27-38
ISBN: 978-989-758-194-6

cremental transformations, there are still a number of unresolved issues concerning both the languages for defining transformations and the respective support tools (Stevens, 2007; Westfechtel, 2016).

As stated above, many formalisms for defining model transformations have been proposed. Besides the approaches mentioned earlier, graphs may be used to represent models in a natural way. As a consequence, graph transformation rules describe modifications of graph structures in a declarative way. Furthermore, there is a comprehensive body of theories, languages, tools, and applications (Rozenberg, 1997; Ehrig et al., 1999).

Maintaining consistency between independent and evolving models requires a graph transformation system, which comprises at least the source graph and the target graph. In order to incrementally propagate changes, a correspondence graph is necessary in between for maintaining traceability links. As a result, a *triple graph transformation system (TGTS)* is required, containing rules defining source-to-target and target-to-source transformations, as well as actions for checking consistency and repairing inconsistencies (Buchmann et al., 2008).

In this paper, we provide a case study dealing with incremental round-trip engineering of UML class models and Java source code. Our solution has been implemented using a handcrafted TGTS written with the Xtend[1] programming language.

The paper is structured as follows: Section 2 provides an overview on the proposed approach. In Section 3 our approach is motivated while key aspects of the implementation are described in Section 4. The experiences gained in the case study are discussed in Section 5 and related work is presented in Section 6. Section 7 concludes the paper.

2 OVERVIEW

This section provides an overview on our approach. The round-trip engineering use case presented in this paper would require a *bidirectional incremental model-to-text transformation* since the UML model is represented as an instance of its metamodel and the Java source code is stored using a set of text files. However, up to now there is no language or tool available supporting this kind of transformations. Thus, a combination of multiple transformation languages and tools is needed to solve this issue. Figure 1 depicts the chosen architecture. The TGTS operates on a correspondence model, which is placed in between the UML model and a MoDisco (Bruneliere et al., 2010) Java model. The TGTS merely addresses structural transformations, including methods with empty bodies for user-defined operations in the UML model, and ignores accessor methods. The rules may be executed both in forward and backward direction. In the forward direction, an additional M2T transformation using Acceleo is executed, which supplies the accessor methods. Please note that Acceleo allows to protect user-defined blocks within method bodies assuring that manually supplied code fragments remain unchanged on subsequent transformations. The text-to-model transformation takes Java source code and transforms it into an instance of the Java metamodel provided by MoDisco. We supply our own incremental discoverer, as the standard discoverer works in batch mode only. The focus in this paper is on the TGTS which has been implemented using Xtend and which provides bidirectional transformations between the UML model and the MoDisco Java model. The following subsections briefly introduce Xtend and the metamodels used in the transformation.

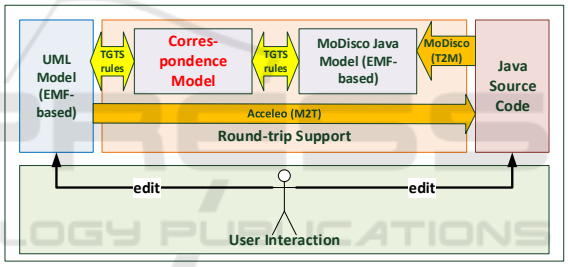

Figure 1: Round-trip support for UML CASE Tools.

2.1 Xtend

The general-purpose programming language Xtend has its syntactical and semantical roots in the Java programming language. However, it focuses on a more concise syntax and extra functionality such as type inference, extension methods and operator overloading. While Xtend primarily is an object-oriented language, it also provides declarative features known from functional programming, e.g., lambda expressions. The Xtend language is statically typed and it uses the type system of Java without modifications. Xtend code is compiled into fully executable Java code and thus, integrates seamlessly with all existing Java libraries.

2.2 Used Metamodels

As explained above, we strive for round-trip engineering support for UML class models and Java mod-

[1] http://eclipse.org/xtend/

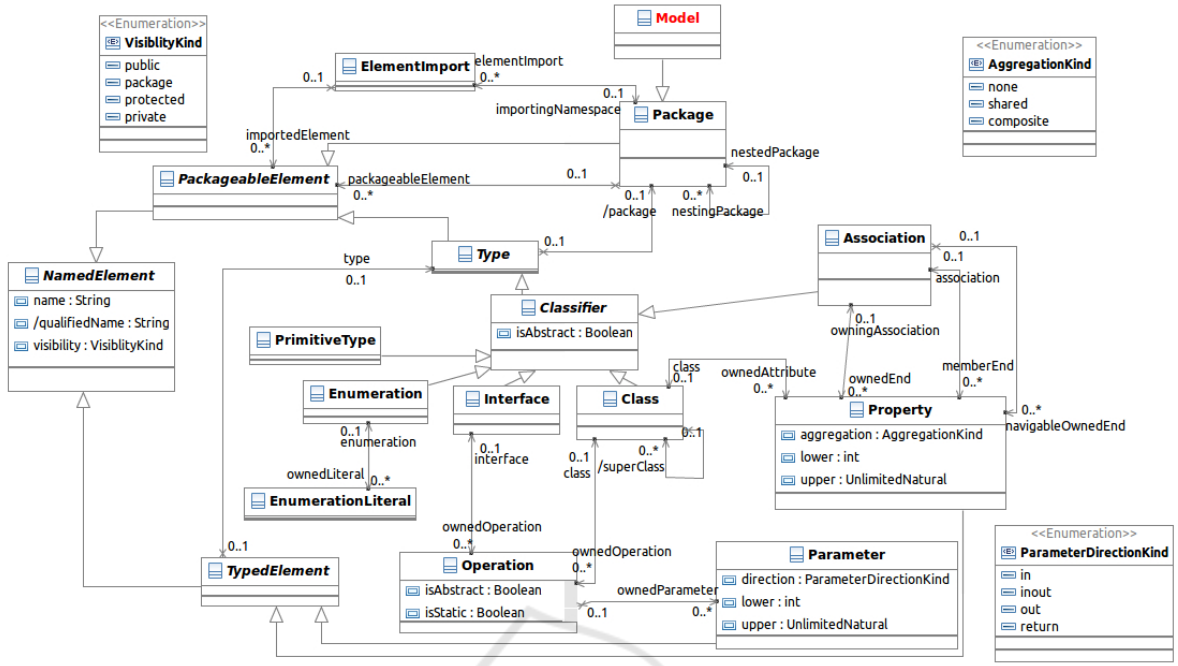

Figure 2: Some relevant parts of the Eclipse UML2 metamodel.

els representing the Java source code (by reusing the MoDisco framework). Both metamodels are based on Ecore (Steinberg et al., 2009), i.e. they share the same meta-metamodel. Since most UML diagrams lack precise and formal semantics, we restrict the case study on the structural features of both models. Behavioral aspects, like the body of method declarations, are not regarded in this implementation.

2.2.1 Eclipse UML2

Eclipse UML2 is part of the *Eclipse Modeling Project*[2]. It provides an Ecore-based implementation of the OMG UML2 specification (OMG, 2015b). Eclipse UML2 only constitutes the abstract syntax of UML2. Figure 2 depicts a simplified overview on relevant metaclasses involved in the transformation.

Packages constitute the container elements of the UML model tree, with a Model, which is a specialization of a Package, as the root element. Packages may be nested and they contain PackageableElements, e.g. Classes, Interfaces or Enumerations. Classes and interfaces may contain Operations (which may have Parameters) and Propertys.

Properties and parameters have a multiplicity and a type. Primitive types may be declared in the model or they may be imported (importedElements). Please note that in our case study we always import the pre-defined primitive types supplied with the Eclipse UML2 implementation.

An Association is a relation between two or more Classifiers[3]. The ends are contained as properties either in the association or the opposite classifier. Depending on whether both associations ends are navigable, an association is said to be bidirectional or unidirectional.

2.2.2 Java (MoDisco)

MoDisco is an extensible framework for developing model-driven tools to support use cases of software modernization. It provides an Ecore-compliant metamodel for Java which resembles the AST of the Java language. Furthermore, it provides a discovery mechanism, that allows to parse existing Java source code into instances of the supplied Java metamodel. Relevant cutouts of the MoDisco Java metamodel are shown in Figure 3.

Similar to the UML2 metamodel, a Model element constitutes the root element and contains a hierarchy of Packages. However, in the Java metamodel, the Model is no specialization of a Package. The model contains primitive types (orphanTypes), as well as ParameterizedTypes and ArrayTypes, which represent multi-valued typed elements.

Packages may be nested (ownedPackages) and may contain ownedElements, e.g. AbstractTypeDec-

[2]http://www.eclipse.org/modeling/

[3]Please note that in our approach we always assume that an association has exactly two member ends.

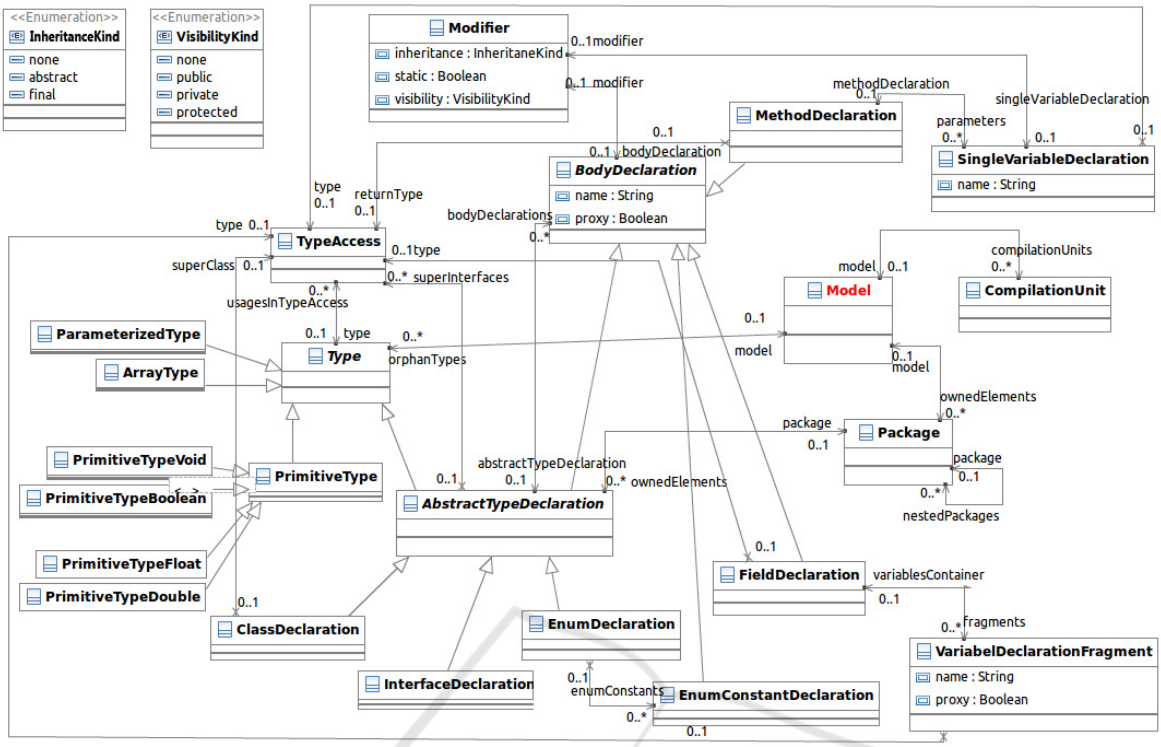

Figure 3: Some relevant aspects of the MoDisco Java metamodel.

laration, the metaclass of which is specialized among others by ClassDeclaration, InterfaceDeclaration and EnumerationDeclaration.

An AbstractTypeDeclaration contains BodyDeclarations, which can be either attributes (FieldDeclaration) or operations (MethodDeclaration). Both are typed elements where the type of the operation represents the returned type. Instead of accessing this type directly, the metaclass TypeAccess provides access to the respective type. With this layer of indirection the interrelation of types is recorded. The input parameters for a method are modeled as ownedParameters with the metaclass SingleVariableDeclaration. Moreover, the name of an attribute is placed in the VariableDeclarationFragment.

The CompilationUnits referenced in the Model constitute the abstraction of physical files which contain type declarations, like classes, interfaces or enumerations. They serve as anchor points for the M2T code generation supplied by the MoDisco framework.

Both metamodels share great similarities but also significant differences, which led to rule explosion in previous round-trip approaches. For instance, while the UML metamodel has one class for all kinds of primitive types, the Java metamodel subclasses the base class PrimitiveType for each primitive type explicitly. Furthermore, there are elements in each model which do not have a corresponding element in the other one. For example, a corresponding construct for an UML Association is missing in the Java metamodel whereas a CompilationUnit is not regarded by the UML metamodel. Furthermore, Propertys in UML are multiplicity elements, i.e. they have a lower and an upper bound whereas in Java arrays or collections have to be used to express this behavior.

3 MOTIVATION

Triple Graph Grammars (TGG) (Schürr, 1994) have been proposed to make the task of specifying intermodel consistency maintenance easier. They may be used to formalize model transformations without specifying a direction. The TGG execution environment automatically derives the respective forward or backward transformation rules. On the one hand, TGGs are highly declarative but on the other hand, some transformations are easier to describe in an imperative way. In (Buchmann and Westfechtel, 2013) the authors present an implementation of round-trip engineering UML class models and Java source code using TGGs. This case study unveils that many rules have to be copied and pasted since the used TGG approach lacks concepts, like rule inheritance and imperative language constructs.

Another implementation of round-trip engineering UML class models and Java source code using the bidirectional model transformation language QVT-R was discussed in (Greiner et al., 2016). Since there is no reference implementation for QVT-R, the only available tool for the Eclipse environment is *medini QVT*, which does not fully implement the QVT-R standard. As a consequence, the check-before-enforce semantics is not covered and it is not quite clear how when- and where-clauses are applied during transformation execution. Furthermore, it is observed that the textual syntax of QVT-R allows a concise notation of the transformation rules. Still, sometimes the rules become quite complex due to the fact that the language does not provide constructs for specifying control flow or a template mechanism.

As stated in Section 1, many languages supporting model transformations are available, e.g. ATL or QVT, which provide a declarative approach to describe model transformations in a concise way. However, these languages come with certain limitations: Our use case needs a bidirectional model transformation, which has to be performed in an incremental way, in order to update existing models after subsequent changes. ATL only supports unidirectional batch transformations and thus, can not be used here. QVT-R on the other hand, addresses incremental and bidirectional transformations. However, as discussed in (Greiner et al., 2016), tool support for QVT-R is far from being mature and there are also some flaws in the official standard. In (Buchmann and Westfechtel, 2013), it was shown that the chosen TGG approach comes with better tool support than QVT-R but lacking rule inheritance or template mechanisms lead to an explosion of transformation rules.

All in all, our previous work showed using QVT-R or TGGs might become quite tedious and also corresponding tool support has several weaknesses. Handcrafting a TGTS on the other hand, requires to specify the generation of graphs, to consider modifications and deletions for each direction as well as to specify rules for checking consistency and establishing correspondences. The contribution of this paper is to investigate whether the effort in handcrafting a TGTS (i.e. program both transformation directions explicitly) outweighs the drawbacks of the aforementioned approaches.

4 IMPLEMENTING THE TGTS

As stated in Section 1, model transformations play a key role in model-driven engineering. In principle, model transformations may be programmed in an ordinary programming language, such as Java. This approach is still followed frequently but clearly a more high-level approach is desired. In the past, many formalisms have been proposed for defining model transformations in the context of object-oriented modeling, e.g. QVT (OMG, 2015a). On the other hand, graphs may represent models in a natural way. *Graph transformations* describe modifications of graph structures in a declarative way. For maintaining consistency between interdependent and evolving models, a graph transformation system is required, which deals with at least two graphs, namely a *source graph s* and a *target graph t*. For incremental change propagation, a *correspondence graph c* is placed in between *s* and *t* for maintaining *traceability links*. This results in a *triple graph transformation system (TGTS)*, the rules of which define source-to-target and target-to-source transformations, as well as actions for checking consistency and repairing inconsistencies.

Developing a TGTS by hand seems to be a tedious and laborious task: First, in addition to the generation of graphs, modifications and deletions have to be addressed. Second, each direction of the transformation has to be specified – as well as rules for checking consistency and establishing correspondences.

4.1 Correspondence Model

In our use case, source and target models are the Eclipse UML2 model and the MoDisco Java model (c.f., Section 2.2) respectively. As stated above, for incremental operations, a correspondence model is needed. Figure 4 shows the correspondence model used in our implementation. A Transformation contains an arbitrary number of correspondence elements (Corr).

In order to establish traceability links, a correspondence element always refers to one EObject from the source model and one EObject from the target model. The String desc describes the element. Since Corr is abstract in order to allow different and complex specializations, we added the concrete class BasicElem without additional features which is sufficient

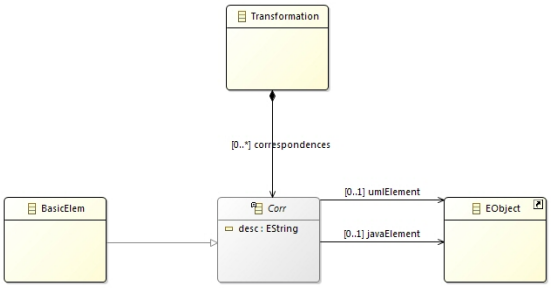

Figure 4: Correspondence model.

for our use case. If there are 1:n mappings, no additional correspondence object is created but the additional elements (2-n) are manually maintained in the rule that transforms the 1:1 mapping.

4.2 Transformation Rules

When specifying a model transformation, the software engineer needs to decide whether the model elements should be transformed bottom-up, i.e. starting with the leaf nodes of the spanning containment tree or at the root node. As declarative approaches perform a topological sort of model elements, software developers only have very limited influence on the transformation order. Typically, the developer must presume the execution behavior when declaring the rules. When hand-crafting a TGTS on the other hand, the transformation order is fully under the control of the user. We chose to strictly follow a top-down approach on the highest level (i.e. on the element level). As a consequence, when transforming child elements, e.g. attributes or methods of classes, we can always be sure that the respective containers and the referred types already exist. Therefore, our transformation starts at the respective root elements (i.e. the Model in both cases) and the primitive types. Next, the Package hierarchy is established. Once this is finished, type declarations (classes, interfaces, enumeration) and associations are transformed followed by the respective inheritance relationships. We further tackle properties and methods afterwards.

As we are hand-crafting our TGTS (unlike in previous approaches (Greiner et al., 2016; Buchmann and Westfechtel, 2013), where bidirectional model transformation formalisms were used), we need to explicitly specify both transformation directions. Furthermore, we need to address creation, deletion or update of existing elements. To reduce the specification effort, we made use of the fact that Xtend builds on the Java VM and it provides mechanisms for inheritance and method overloading. Listing 1 provides a cutout of the abstract base class for all transformation rules.

The base class consists of the three resources sourceModel, targetModel and corrModel, which are not shown in Listing 1 due to space restrictions and which are created or kept consistent during the transformations. The methods sourceToTarget() and targetToSource() are overridden in each class inheriting from Elem2Elem. The respective implementations contain the transformation rule for the corresponding elements for each transformation direction. The super rule, which is executed in the overriding methods at last (e.g., Listing 2, line 9,18), ensures that all unreferenced elements, where the umlElement or the javaElement are empty, are deleted and saves the transformation (Listing 1, line 6,11). The String given to the deletion calls (Listing 1, line 4,9) can be used to adapt the behavior for specific elements, e.g. deleting elements without a correspondence object.

Listing 1: Abstract base class for our transformation rules.

```
1  abstract class Elem2Elem {
2   ...
3   def void sourceToTarget(String s) {
4     deleteUnreferencedTargetElements(s)
5     corrModel.save(null)
6     targetModel.save(null)
7   }
8   def void targetToSource(String s) {
9     deleteUnreferencedSourceElements(s)
10    corrModel.save(null)
11    sourceModel.save(null)
12  }
13  def Corr getCorrModelElem(EObject ob) {
14   corrModel.contents.get(0).correspondences
15   .findFirst[ c | c.umlElement == ob || c.
         javaElement == ob]
16  }
17  def getOrCreateCorrModelElement(EObject ob,
        String description) {
18   var Corr c = ob.getCorrModelElem
19   if (c == null) {
20    c = corrFactory.createBasicElem => [
21     if (ob.eClass.EPackage instanceof
           UMLPackage)
22      umlElement = ob
23     if (ob.eClass.EPackage instanceof
           JavaPackage)
24      javaElement = ob
25     desc = description
26    ]
27    corrModel.contents.get(0).correspondences
          += c
28   }
29   return c
30 } }
```

The methods getOrCreateModelElement and getCorrModelElem maintain the traceability links in the correspondence model. Within the transformation rules sourceToTarget and targetToSource the method getOrCreateModelElement is called (e.g., Listing 2, line 5,14) to extract the element to be created. The call parameters are:

EObject ob The context object on which the method is called

String description The description identifying the correspondence elements

This method calls getCorrModelElem which retrieves the requested object from the correspondence

model if it is already present. If a correspondence object cannot be found for the given element, it will be created (Listing 1, line 20) and stored in the respective EReference (umlElement or javaElement). Please note that for each of the subsequent descriptions the *source* represents the UML model and the *target* indicates the Java model. Accordingly, if the method sourceToTarget is called, a transformation from the UML model to the Java model is performed and vice versa in case of calling targetToSource. We further denote the source to target as *forward* and the opposite as *backward* direction.

The redefinitions of the methods sourceToTarget and targetToSource always consist of the following basic steps: First, the resource of the specified direction is searched for the element transformed in the respective rule. Next, the correspondence element is extracted with the aforementioned method getOrCreateCorrespondenceModelElement. From the returned element the target object for the respective direction is either extracted – if it already exists – or created and added to the correspondence element as umlElement or javaElement by the methods getOrCreateSourceElem and getOrCreateTargetElem respectively (Listing 2, line 5,14). This way an already existing correspondence and target element can be altered incrementally[4]. Thereafter, the basic properties of the target element for the respective direction are updated or added.

Listing 2: Class to transform the root elements of both models.

```
1  class Model2Model extends Elem2Elem {
2   override void sourceToTarget(String s) {
3    sourceModel.contents.filter(typeof(org.
         eclipse.uml2.uml.Model))
4    .forEach[m |
5     var targ = m.getOrCreateCorrModelElement(s
          ).getOrCreateTargetElem(JavaPackage.
          eINSTANCE.model)
6     targ.name = m.name
7     targetModel.contents += targ
8    ]
9    super.sourceToTarget("model")
10  }
11  override void targetToSource(String s) {
12   targetModel.contents.filter(typeof(Model))
13   .forEach[ m |
14     var src = m.getOrCreateCorrModelElement(s
           ).getOrCreateSourceElem(UMLPackage.
           Literals.MODEL)
15     src.name = m.name
16     sourceModel.contents += src
17   ]
18   super.targetToSource("model")
19 } }
```

[4]Incremental means that the target and correspondence element will not be created anew if they already exist. Their properties, on the other hand, are overwritten. Otherwise, one would have to calculate or track the differences between the previous and the current version first.

For the Model2Model rule seen in Listing 2 this only comprises the name attribute. Afterwards, the created or updated model is added to the targetModel or sourceModel respectively. The metaclass Model is the root for the respective metamodel, i.e. there should only exist one Model element. Other references of the Models, e.g. the orphanTypes of the Java Model, are maintained in the transformation of the referenced elements.

The transformation of models (and, e.g. packages and classes) are declared fairly straightforward because the metamodels share many features and are almost 1:1 mappings. Treating UML associations, which have no corresponding concept in the Java model, is more complex, for instance. We decided to add a ClassDeclaration in the Java model for every association. This class contains both memberEnds of the association as fields and corresponding accessor methods with proper visibilities that reflect the navigability as specified in the UML model[5]. The correspondence element with the description "association" always points to an Association as umlElement and a ClassDeclaration as javaElement.

Listing 3 presents a simplification of how a class is created for an association and vice versa. The created class is added to the package of the association (Listing 3, line 7), receives the name of the association and stores all memberEnds as bodyDeclarations. As each classifier needs a CompilationUnit in the Java model, it is created or updated by reusing the method getOrCreateCompU in line 8 that supports all kinds of AbstractTypeDeclarations. The CompilationUnit is added to the Java model without creating a separate correspondence element because the UML association is already mapped to the corresponding Java class. Thus, it is necessary to manually update or delete the CompilationUnit in case the corresponding association is modified or removed. The update is performed in the method getOrCreateCompU, the deletion in the method deleteUnreferencedTargetElements (Listing 1, line 4).

Listing 3: Class to transform the UML associations to Java classes.

```
1 class Association2Class extends Elem2Elem {
2  override sourceToTarget(String s) {
3   sourceModel.allContents.filter(typeof(
        Association))
```

[5]This class serves as 1:1 mapping for the association (but not for its ends) and can be extended for more general UML concepts like, e.g. n-ary associations.

```
4     .forEach [ a |
5       var cl = a.getOrCreateCorrModelElement(s
            ).getOrCreateTargetElem(JavaPackage
            .eINSTANCE.classDeclaration)
6       cl.name = a.name.toFirstUpper()
7       a.package.corrModelElement.javaElement.
            ownedElements += cl
8       cl.originalCompilationUnit =
            getOrCreateCompU(cl)
9       if (cl.comments.size == 0)
10       cl.comments += jFact.createJavadoc
11      cl.comments.get(0).content = "/**
12       * @associationClass" + cl.name + "*/"
13    ...
14    ]
15    super.sourceToTarget(s)
16  }
17  override targetToSource(String s) {
18   targetModel.allContents.filter(typeof(
         ClassDeclaration)).filter[ cd |
19   cd.comments.size > 0 && cd.comments.get(0).
         content.contains('@associationClass')]
20   .forEach [ cd |
21    var assoc = cd.getOrCreateCorrModelElement
         (s).getOrCreateSourceElem(UMLPackage.
         eINSTANCE.association)
22    assoc.name = cd.name.toFirstLower
23    assoc.package = cd.package.
         corrModelElement.umlElement
24   ]
25   super.targetToSource(s)
26 } }
```

To distinguish between association classes and "regular" classes, we annotate them with a Javadoc comment (Listing 3, line 11f.). Thus, we can identify annotated classes in the backward direction and build up the Association correspondingly. The name is extracted from the ClassDeclaration (Listing 3, line 22) and the right package (Listing 3, line 23) is set. The memberEnds are treated like other properties with the next rules.

Listing 4 shows the forward transformation of properties of all kinds. If they are of primitive type, they will be added to the containing classifier. If they belong to an association, the method setOwnerJava (Listing 4, line 5) will add both memberEnds as bodyDeclarations to the corresponding introduced association class. Furthermore, this method applies comments whether the end is owned by the association or the class and whether it is navigable.

Listing 4: Transformation for UML properties.

```
1 override def sourceToTarget(String s) {
2  sourceModel.allContents.filter(typeof(
       Property))
3  .forEach [ p |
4    var fd = p.getOrCreateCorrModelElement(s).
         getOrCreateTargetElem(JavaPackage.
         eINSTANCE.fieldDeclaration)
5    setOwnerJava(p, fd)
6    fd.originalCompilationUnit = fd.
         abstractTypeDeclaration.
         originalCompilationUnit
7    setNameJava(p,fd)
8    setTypeJava(p, fd)
9    setVisibility(p,fd)
10 ]
11 super.sourceToTarget(s) }
```

In the backward direction depicted in Listing 5 these comments are used to construct the memberEnds. The method setOwnerUML (Listing 5, line 7) determines from the annotation whether the end is navigable or not and whether it is owned by the association. As a consequence, it will add the created property to either one of the classes participating in the association or to the association as one of the ownedEnd or the navigableOwnedEnd references. The type is determined with the method setTypeUML. In case the type of the property is primitive, the property is simply added to the UML classifier corresponding with the abstractTypeDeclaration of the Java field. If the type is a ParameterizedType or an ArrayType, the value upper of the UML Property will be set to infinity. Moreover, in the latter case the UML type cannot be retrieved from the correspondence element in a straightforward way but the typeArguments and types respectively must be used to obtain the corresponding umlElement.

Listing 5: Transformation of Java FieldDeclarations.

```
1 override targetToSource(String s) {
2  targetModel.allContents.filter(typeof(
       FieldDeclaration))
3  .forEach [ fd |
4    var p = fd.getOrCreateCorrModelElement(s).
         getOrCreateSourceElem(UMLPackage.
         Literals.PROPERTY)
5    p.name = fd.fragments.get(0).name
6    setTypeUML(fd, p)
7    setOwnerUML(fd,p)
8   ...
9  ]
10 super.targetToSource(s)  }
```

4.3 Round-trip Example

This section provides a brief example to illustrate a simple round-trip scenario in our use case.

Figure 5 depicts the single transformation steps. It provides cutouts of the UML abstract syntax as well as of the resulting Java model in the Ecore tree representation. The transformation starts with a given UML model and an empty Java model. Subsequent

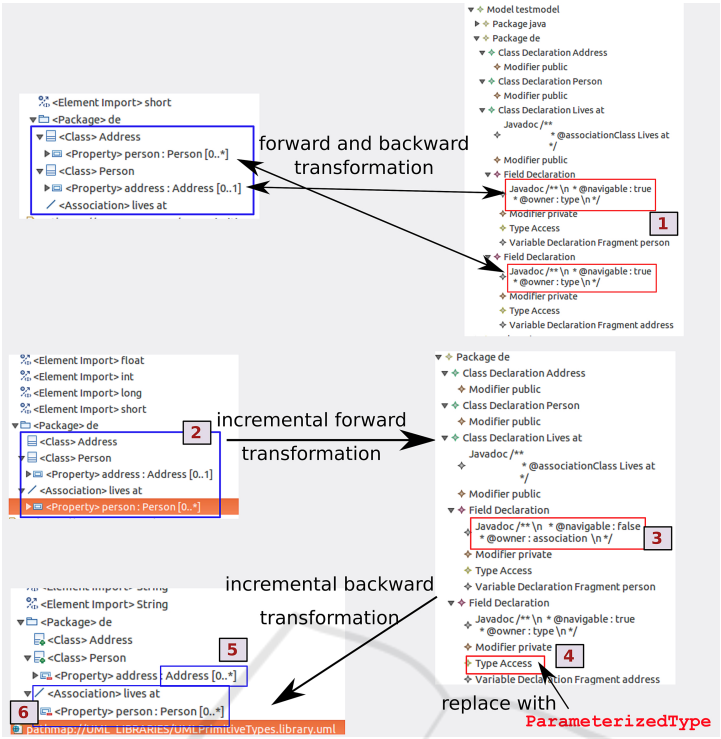

Figure 5: Example round-trip scenario.

changes are added incrementally. In the initial model many persons may live at one Address where the ends of the association lives at are contained in the respective opposite class. This indicates an association which is navigable in both directions. Transforming in the forward direction results in three classes where the ends are added as fields in the association class. However, the comment of the fields registers the classes participating in the association as owner of the fields (1) by noting a "type" instead of an "association". Transforming backward correctly results in the original model. Next, the association becomes owner of the person end in the UML instance (2). Transforming forward again adapts the comment of the corresponding field and notes that the owner is now the association (3). Thereafter, it is decided that a person might live at different addresses so the class Address is replaced by an ParameterizedType that serves as a collection for many Address instances (4). These changes are propagated back to the UML model when applying an incremental backward transformation: the property address is now multi-valued (5) and the person end is still contained in the association (6).

5 DISCUSSION

This section discusses the lessons learned from implementing a bidirectional model transformation with a hand-crafted TGTS written in Xtend. The integration of the system, on the other hand, is yet to come and the evaluation will be published separately.

Basically, we can categorize the relations between source and target model elements into 1:1 mappings (e.g., a UML package and a Java package) and 1:n (a UML class and a Java ClassDeclaration, a Java CompilatonUnit and a ParameterizedType) mappings. We found out that 1:1 mappings are straightforward and easy to implement and that additional elements in the involved models may be added without restrictions: For instance, a UML class needs a corresponding Java ClassDeclaration and the ClassDeclaration always belongs to a CompilationUnit which must be included additionally in the Java model.

In contrast to purely declarative approaches, language features, like inheritance or generics, can be used seamlessly in the Xtend language. While the lack of these features in the QVT-R or the TGG approach leads to a combinatorial explosion in the number of rules which have to be copied and pasted, e.g. in case that a rule needs to be invoked for subclasses of an (abstract) super type, we were able to express

the same behavior within a single rule in the TGTS.

Listings 4 and 5 show that complex assignments can easily be moved to operations, which may be reused from various places – a feature which is common in object-oriented programming languages but which is lacking in (declarative) model transformation languages. Please note, that QVT-R allows to specify queries which could also be reused in arbitrary places but only without side-effects, while non-top level relations can also be called from other rules and may contain side-effects.

The downside of the presented approach is that both transformation directions have to be specified explicitly. Hence, whenever the metamodels are changed in a way that requires adapting the transformation rules, these changes have to be performed at different places inside the transformation script. It is up to the software engineer to ensure a consistent modification throughout the script. Moreover, allowing only 1:1 correspondences affords to keep the 2-n other elements manually consistent in incremental transformations which results in an imminent danger of introducing inconsistencies by mistake.

The alternative of using 1:n correspondences in the trace model would require to keep track which additional elements are mapped and stored at which position. This complicates the rules significantly and requires users to know the mapping assumptions. If, for instance, a class was mapped to a class, a CompilationUnit, and a ParameterizedType, this would result in a collection of three different Java elements that must be treated separately. Thus, the user must be sure which of the mapped elements is of which type.

Another problem of the current approach are elements that have no corresponding partner in the opposite model, i.e. "0:1" mappings. In general, these elements cannot be created in advance or at the end of the transformation since they may reference other elements that are created during the transformation or they may be referenced by other elements. For instance, the collection container in the Java model has no partner element in the opposite model. Thus, the user needs to manually add and update such an element by calling a respective operation when required during the transformation.

Nevertheless, in a generic base rule the current approach is able to describe the parts dealing with the trace links between the model elements and locating changes. Consequently, the concrete rules only describe the changes using Xtend's lambda expressions. In total, it took **1121** lines of **Xtend** code to specify the transformation rules, compared to **2947** lines of **QVT-R** code. Hence, for the transformation problem in this case study, the usage of a declarative language (which should provide a more concise and compact solution) required almost three times the number of lines of code than using a modern imperative language with lambda expressions.

One note on testing our implementation: Another benefit of the approach presented in this paper is the fact that each Xtend specification can easily be tested automatically with the help of JUnit test cases. Moreover, even those test cases may be written using Xtend. We supplied an extensive set of test cases for each rule and tested model modifications for both transformation directions (creation, update and deletion of respective model elements). Being able to run automated tests is a considerable improvement over our previous TGG (Buchmann and Westfechtel, 2013) and QVT-R (Greiner et al., 2016) implementations, where all of those tests had to be carried out manually in a tedious and time consuming procedure.

Apart from this use case, our correspondence model is generic enough to be reused with other bidirectional model transformations. Having correspondence elements that map one element of a source to one element of a target model is the most natural mapping present in transformations. Non-bijective mappings always need further consideration and are more likely to be established manually since these transformations are hard to automate. For traceability and managing the transformation incrementally the resulting correspondence graph serves as persisted trace information. As a consequence, one could think about writing a DSL on top of Xtend allowing to specify transformation patterns.

6 RELATED WORK

Common approaches that are used to transform text (e.g. source code) to models are based on parsers for the specific text languages. Usually, these approaches work on the resulting parse trees and map the tree items to corresponding model elements. Typical limitations are maintenance problems when the underlying M2T templates are changed.

In (Bork et al., 2008), Bork et al. describe an approach towards model and source code round-trip engineering, which is based on reverse engineering of M2T transformation templates. The idea of this approach is to use (customizable) code generation templates as a grammar to parse the generated (and later modified) code. The benefit of this approach compared to other approaches using plain Java parsers and the resulting parse tree as a source for the code to model transformation is that changes to the templates are automatically taken into account during re-

verse engineering. While the approach described in (Bork et al., 2008) requires considerable implementation effort since a template parser, reasoner and token creator have to be implemented, our approach just required the specification of respective Xtend rules that relate two elements of the respective metamodels. Since MoDisco is able to parse source code which even contains syntax or compile errors into a corresponding Java model, our approach is also independent of the style of the generated code and it also does not depend on a (usually) fine grained parse tree. Furthermore, Javadoc tags can be used to add additional meta-information to the code. While the approach presented in (Bork et al., 2008) is able to round-trip engineer only code that has been generated with the corresponding templates, our approach is able to handle any code which complies to Java language specification version 5. In addition, the approach by Bork et al. requires bijective reversible templates. E.g. the approach will fail if an attribute name in a class contains the class name.

Angyal et al. present in (Angyal et al., 2008) an approach for model and code round-trip engineering based on differencing and merging of abstract syntax trees (AST). In this approach, the AST is regarded to be the platform-specific model (PSM) according to the taxonomy of models in MDA (Mellor et al., 2004). Nevertheless, in this approach the AST model has a very low level of abstraction because it exactly represents the code. Contrastingly, the discovered Java model which is used in our approach is on a higher level of abstraction. The round-trip engineering approach comprises two different round-trip tasks: one between PIM and PSM and one between PSM and code. The approach tries to prevent information loss during round-trip engineering by using a so called trace model which is used to synchronize the PIM and the PSM (the AST). Furthermore, the AST and the source code are updated using a fine grained bidirectional incremental merge based on three-way differencing. In our approach, information loss is prevented by using Javadoc tags as annotations. In case model and code are changed simultaneously and the changes are contradicting, one transformation direction has to be chosen, which causes that some changes might get lost.

There are also approaches that are dedicated to model-to-model round-trip engineering. This task involves synchronizing models and keeping them consistent. Antkiewicz and Czarnecki propose an approach towards round-trip engineering for *framework-specific modeling languages* (FSML) (Antkiewicz and Czarnecki, 2006). FSMLs are a special category of DSLs which are defined on top of object-oriented application frameworks. In contrast to general round-trip engineering approaches, the approach presented in (Antkiewicz and Czarnecki, 2006) does not have to deal with non-isomorphic mappings between the artifacts, as the problem domain is much smaller and only code for a specific framework is generated by the code generators of the FSML. The synchronization of the involved implementation model is based on a comparison inspired by CVS and reconciliation. In a last step, conflict resolution has to be carried out interactively by the user.

Hettel et al. (Hettel et al., 2009) propose an approach towards model round-trip engineering based on abductive logic programming. In particular, this approach does not place restrictions such as injective behavior on the underlying transformations. A reference implementation is given which can be used to reverse unidirectional transformations based on the Tefkat language. It is a general approach, which could also be applied to other model transformation languages, like QVT. However, since the source transformation does not necessarily need to be injective, ambiguities have to be solved when reversing the transformation. At the end, the "best" solution has to be picked by the user or it has to be determined using some kind of heuristics.

7 CONCLUSION AND FUTURE WORK

In this paper, we presented a solution for incremental round-trip engineering between UML class models and Java source code, using a hand-crafted *triple graph transformation system (TGTS)* written with the Xtend programming language.

While nowadays tool support for model transformation mostly covers unidirectional batch transformation, this use case demands for a bidirectional transformation, which works in an incremental manner. However, our previous research revealed that there is no adequate tool support to solve this problem.

To this end, we chose to code both transformation directions explicitly using the Xtend language. We exploited lambda expressions to describe the graph patterns of our TGTS in a declarative way and made use of the traditional imperative programming style whenever it was appropriate.

Future work comprises a quantitative and qualitative evaluation of our three implemented solutions for incremental round-trip engineering.

REFERENCES

Angyal, L., Lengyel, L., and Charaf, H. (2008). A synchronizing technique for syntactic model-code round-trip engineering. In *Proceedings of the 15th International Conference on the Engineering of Computer Based Systems (ECBS 2008)*, pages 463–472.

Antkiewicz, M. and Czarnecki, K. (2006). Framework-specific modeling languages with round-trip engineering. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, LNCS 4199, pages 692–706, Genova, Italy.

Bork, M., Geiger, L., Schneider, C., and Zündorf, A. (2008). Towards roundtrip engineering - a template-based reverse engineering approach. In Schieferdecker, I. and Hartman, A., editors, *ECMDA-FA*, volume 5095 of *Lecture Notes in Computer Science*, pages 33–47. Springer.

Bruneliere, H., Cabot, J., Jouault, F., and Madiot, F. (2010). MoDisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 173–174, New York, NY, USA. ACM.

Buchmann, T., Dotor, A., and Westfechtel, B. (2008). Triple graph grammars or triple graph transformation systems? In Chaudron, M. R. V., editor, *Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers*, volume 5421 of *Lecture Notes in Computer Science*, pages 138–150. Springer.

Buchmann, T. and Westfechtel, B. (2013). Towards incremental round-trip engineering using model transformations. In Demirors, O. and Turetken, O., editors, *Proceedings of the 39th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2013)*, pages 130–133. IEEE Conference Publishing Service.

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645.

Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G., editors (1999). *Handbook on Graph Grammars and Computing by Graph Transformation*, volume 2: Applications, Languages, and Tools. Singapore.

Greiner, S., Buchmann, T., and Westfechtel, B. (2016). Bidirectional Transformations with QVT-R: A Case Study in Round-trip Engineering UML Class Models and Java Source Code. In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016)*. accepted for publication.

Hettel, T., Lawley, M., and Raymond, K. (2009). Towards model round-trip engineering: An abductive approach. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT 2009)*, LNCS 5563, pages 100–115, Zurich, Switzerland.

Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). Atl: A model transformation tool. *Science of Computer Programming*, 72:31 – 39. Special Issue on Second issue of experimental software and toolkits (EST).

Mellor, S. J., Kendall, S., Uhl, A., and Weise, D. (2004). *MDA Distilled*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

OMG (2015a). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Needham, MA, formal/2015-02-01 edition.

OMG (2015b). *Unified Modeling Language (UML)*. Object Management Group, Needham, MA, formal/15-03-01 edition.

Rozenberg, G., editor (1997). *Handbook on Graph Grammars and Computing by Graph Transformations*, volume 1: Foundations. Singapore.

Schürr, A. (1994). Specification of Graph Translators with Triple Graph Grammars. In *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 1994)*, volume 903 of *LNCS 903*, pages 151–163, Herrsching, Germany.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF Eclipse Modeling Framework*. The Eclipse Series. Boston, MA, 2nd edition.

Stevens, P. (2007). Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007)*, LNCS 4735, pages 1–15, Nashville, USA.

Westfechtel, B. (2016). A Case Study for a Bidirectional Transformation Between Heterogeneous Metamodels in QVT Relations. In Maciaszek, L. A. and Filipe, J., editors, *Evaluation of Novel Approaches to Software Engineering*, volume 599 of *Communications in Computer and Information Science (CCIS)*, chapter 8, pages 141–161. Springer International Publishing. Revised Selected Papers from ENASE 2015.