BXtend - A Framework for (Bidirectional) Incremental Model Transformations

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

Keywords: Model Transformations, Bidirectional, Incremental, Imperative, Model-driven Development.

Abstract: Model transformations constitute the core essence of model-driven software development (MDSD) – a software engineering discipline, which gained more and more attention during the last decade. While technology for unidirectional batch transformations seems to be fairly well developed, tool support for bidirectional and incremental transformations is still restricted. Results obtained with case studies carried out with popular bidirectional approaches motivated us to set up our own light-weight framework for bidirectional and incremental model transformations based on the Xtend programming language. Our approach provides several advantages, as it reduces the cognitive complexity for transformation developers, and allows for a greater flexibility in transformation specifications by providing procedural language constructs. In addition, it provides a higher expressive power and allows for compact specifications at the same time.

1 INTRODUCTION

The motivation behind *Model-driven software development (MDSD)* (Völter et al., 2006) is to replace low-level programming with the development of high-level models. Starting from an initial model capturing the requirements, a set of models over multiple levels of abstraction is derived until finally code is generated. Modeling languages are usually defined with the help of *meta models* in the context of object-oriented modeling. In the academic world, the de-facto standard for research dedicated to model-driven engineering is the *Eclipse Modeling Framework (EMF)* (Steinberg et al., 2009). It strictly focuses on principles from object-oriented modeling and only provides core concepts for defining classes, attributes and relationships between classes. EMF is based on its meta model *Ecore*, which basically resembles *Essential MOF (EMOF)*, a subset of MOF (OMG, 2015b).

Model-driven Architecture (MDA) (Mellor et al., 2004) is the standard process for model-driven software engineering. In the MDA context, model transformations are typically chained. A *platform independent model (PIM)* is refined to a *platform specific model (PSM)* using a series of subsequent model transformations. Consequently, MDSD highly depends on model transformations. A model transformation describes how a source model s, represented as an instance of an input meta model is transformed into a target model t conforming to the output meta model. Typically, we distinguish between *model-to-text (M2T)* and *model-to-model (M2M)* transformations, depending on the representation of the target models. A wide range of formalisms have been proposed in the past for defining model transformations (Czarnecki and Helsen, 2006). They vary with respect to computational paradigms (procedural, functional, rule-based, object-oriented), directionality (unidirectional vs. bidirectional), support for in-place, out-place, incremental or batch transformations, etc. Numerous model transformation languages emerged accordingly, ranging from general ones, like ATL (Jouault et al., 2008) or QVT (OMG, 2015a), graph based languages like Henshin (Arendt et al., 2010) or eMoflon (Anjorin et al., 2012) to domain-specific languages such as, e.g., the Epsilon family of transformation languages (Rose et al., 2014) or EMG (Popoola et al., 2016). Our observations have shown that tool support for (unidirectional) batch transformations is fairly well developed. However, in many scenarios different kinds of transformations are required: After transforming a source model into a target model, additions or modifications to the target model may be necessary. Consequently, changes to the source model need to be propagated in a way which allows to retain the manual modifications of the target model. These change propagations de-

Buchmann, T.
BXtend - A Framework for (Bidirectional) Incremental Model Transformations.
DOI: 10.5220/0006563503360345
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 336-345
ISBN: 978-989-758-283-7
Copyright © 2018 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

mand for *incremental* rather than batch transformations. Additionally, changes to the target model may have to be propagated back to the source model, resulting in *bidirectional* transformations. While several languages and tools for bidirectional and incremental transformations have been proposed in the past (OMG, 2015a; Schürr, 1994), there are still a number of unresolved issues concerning both the languages for defining transformations and the respective supporting tools (Stevens, 2007; Westfechtel, 2016).

In this paper, we present BXtend, a light-weight framework for bidirectional and incremental model transformations based on the Xtend programming language. The framework helps transformation developers to easily create a hand-crafted Triple Graph Transformation System (TGTS) which will be used for a bidirectional and incremental model-to-model transformation. Moreover, it is an object-oriented and imperative language with some functional parts allowing for concise transformation specifications with a high degree of expressive power. Our results obtained from several case studies have shown that BXtend allows to reduce the cognitive complexity for the transformation developer, by explicitly specifying both transformation directions. Furthermore, the procedural character of the language allows for a greater flexibility in the transformation rules. However, the overall size of the transformation specification is outperforms purely declarative and bidirectional approaches, as QVT-R for example. The resulting transformation seamlessly integrates into Java code, which makes it easy for tool integrators to empower their tools with powerful model transformations.

2 RELATED WORK

Over the years, many approaches for model transformations have been proposed. In the following, we will focus our discussion only on the ones providing support for bidirectional and incremental model-to-model transformations. Please note that rather than comparing our framework against single tools, we tried to categorize the existing formalisms and discuss the benefits and drawbacks of the chosen approach instead. Of course we will give examples of tools belonging to the respective categories. However, our own observations and case studies have revealed, that all of the approaches listed below have limitations when conditional creations of target elements are required. In this case, procedural approaches like BXtend are more powerful.

2.1 Rule-based Approaches

Approaches belonging to this category usually are grammar-based and provide a high-level language allowing for generating all consistent pairs of source and target models. While this technique can be applied to strings, lists, or trees, the most prominent representative are Triple Graph Grammars (TGG) (Schürr, 1994). TGGs have been implemented by various tools, such as, e.g. eMoflon (Anjorin et al., 2012), TGG Interpreter (Kindler and Wagner, 2007), MoTE (Giese et al., 2014) or EMorF (Klassen and Wagner, 2012), just to name a few. The basic idea behind TGGs is to interpret both source and target models as graphs and additionally have a correspondence graph, whose nodes reference corresponding elements from both source and target graphs, respectively. The resulting model transformation language is highly declarative, as 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. Please note that within the rules, no information about a transformation direction is contained. The corresponding rules for forward and backward transformation may be derived automatically by the TGG engine. Trace information is stored in the correspondence graph, which is exploited for incremental change propagation. From the viewpoint of the transformation designer, incrementality and bidirectionality come for free and need not be specified explicitly in the transformation definition, as well as modifications and deletions, which are also handled automatically by the TGG engine.

2.2 Constraint-based Approaches

Constraint-based approaches are usually even more high-level than grammar-based approaches, as they only require a specification of the consistency relation but all details how to restore this consistency relation are left open. QVT-R (OMG, 2015a) is a language following this principle. Unfortunately, there are only very few tools which are based on this standard. QVT-R allows of a declarative specification of bidirectional transformations. The transformation developer may provide a single relational specification which defines relations between elements of source and target models respectively. This specification may be executed in different directions (forward and backward) and in different modes (checkonly and enforcement). Furthermore, QVT-R allows to propagate updates from the source model to the target model in subsequent transformation executions (incremental behavior).

2.3 Functional Programming-based Approaches

In functional programming-based approaches, the basic idea is to program a single function (forward or backward), which is used to infer the opposite function (backward or forward) and providing a pair which adheres to round tripping laws. These approaches origin from the BX community and are based on lenses, which are used to specify *view/update* problems. A different terminology is used for forward and backward directions: the forward direction is referred to as *get*, while the backward direction is called *put*. The round trip idea can be realised either by using *get* to infer *put*, or vice versa. In all cases, the underlying consistency relation is not specified explicitly. One representative of this approach is BiGUL [1]

2.4 Our Approach

The main differences of our approach compared to the approaches discussed above, is that BXtend is a procedural and object-oriented framework, which is implemented in the programming language *Xtend*[2]. Besides its procedural and object-oriented character, Xtend also provides powerful functional parts, as lambda expressions. The procedural constructs allow for a greater flexibility in transformation specifications. Our BXtend framework is a light-weight framework helping transformation developers to easily create a TGTS as an internal DSL. It is easy to use for Java developers, as Xtend directly builds upon Java and just makes it less verbose. Thus, the transformation developer does not need to learn a new programming language. Furthermore, the resulting M2M transformation may be integrated seamlessly with any other Java application without requiring additional dependencies, which makes it particularly interesting for tool integrators. Transformation directions are specified independently in an unidirectional language, which provides a greater flexibility for the transformation developer, but provides not support to check if the backward transformation matches the forward direction. Nevertheless, our case studies unvieled, that although both transformation directions need to be specified separately, the overall transformation script is still dense an compact and does not require more LOC than approaches optimized for bidirectional transformations.

[1] www.prg.nii.ac.jp/project/bigul/

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

3 BACKGROUND

3.1 Model Transformations

As stated above, model transformations constitute the core essence of model-driven software development. Given a source model s and a target model t, both conforming to their respective meta models, a model transformation describes how s is converted to t. Depending on the representation of the models s and t, we distinguish between *model-to-model (M2M)*, *model-to-text (M2T)*, *text-to-model (T2M)*, and *text-to-text (T2T)* transformations.

Throughout the years, a wide range of languages and tools for *model transformations* has been developed (Czarnecki and Helsen, 2006). These approaches differ from each other in several aspects:

Computational Paradigms: Computational paradigms of model transformation languages comprise procedural, functional, object-oriented, or rule-based languages.

Transformation Direction: While some languages only provide support for unidirectional (i.e., from the source to the target model), other languages allow to formalize and execute bidirectional transformations (i.e., from source to target models and vice versa).

Execution Modes: While *batch* transformations create the complete target model in each transformation run from scratch, *incremental* transformations may be used to retain changes made to the target model.

Source/Target Model Relationships: If source and target model refer to the same model instance, the transformation is called *in-place* transformation. *Out-place* transformations operate on different instances of source and target models.

Source/Target Language Relationships: If both source and target model are instances of the same meta model, the transformation is called *endogenous*. In an *exogenous* transformation, the meta models or source and target models are different.

Different model transformation languages exist, ranging from general ones, like ATL (Jouault et al., 2008) or QVT (OMG, 2015a), graph based languages (e.g., Henshin (Arendt et al., 2010) or eMoflon (Anjorin et al., 2012)) to domain-specific transformation languages such as the Epsilon family of transformation languages (Rose et al., 2014) or EMG (Popoola et al., 2016).

3.2 Graph Transformations

Model instances may be interpreted as graphs, as a model typically constitutes a spanning containment-tree whose elements are interconnected with cross-tree edges. As a consequence, a model transformation is regarded as a problem in the domain of *graph transformations* (Ehrig et al., 2005). In general, graph-based systems may be classified into two different categories: *Graph-rewrite systems* and *graph grammars*. For the application scenario of bidirectional model-to-model transformations, a special kind of graph grammars – triple graph grammars (Schürr, 1994) – are used typically. *Graph rewriting* implies that the graphs are transformed by applying rewrite rules, which specify the replacement of a graph pattern (left-hand side) by a subgraph to be embedded into the overall host graph. *Triple Graph Grammars (TGG)* (Schürr, 1994) interpret both source and target models as graphs and additionally a correspondence graph whose nodes reference corresponding elements from both source and target graphs, respectively, is used. TGGs allow to describe model transformations in a highly declarative way by means of production rules, which are used to describe the simultaneous extension of the involved graphs.

3.3 Triple Graph Transformation System

As stated above, graphs may be used to represent models in a natural way and graph transformations declaratively describe modifications of graph structures. In order to maintain consistency between interdependent and evolving models, a *graph transformation system* is necessary dealing with at least two graphs: a *source graph s*, and a *target graph t*. When incremental change propagation is required, an additional *correspondence graph c* is placed in between *s* and *t*, in order to maintain traceability links. Altogether, this results in a *triple graph transformation system (TGTS)* comprising rules for defining source-to-target and target-to-source transformations and actions for checking consistency and repairing inconsistencies. TGGs are able to automatically derive a corresponding TGTS from a TGG specification but, as described in (Buchmann et al., 2009) and (Buchmann and Westfechtel, 2016), the TGG approach suffers from certain limitations which have been unveiled in our case study. As a consequence, we decided to provide an alternative implementation by hand-crafting the TGTS which deals with the models involved in this use case. Please note that we intentionally decided to implement the TGTS manually rather than using graph transformation tools, like Henshin (Arendt et al., 2010), to specify the corresponding forward and backward rules since we wanted to compare the bidirectional approaches with manual implementations of model transformations in a current programming language.

4 BXtend FRAMEWORK

Developing a Triple Graph Transformation System by hand seems to be a laborious task at first glance: Each alteration of the involved graphs needs to be addressed, including generations, modifications and deletions. Furthermore, each transformation direction needs to be specified separately. In addition, rules for checking consistency and establishing correspondences are required. In the following, we present our framework BXtend, which helps transformation developers to easily create a TGTS for bidirectional and incremental model transformations. The framework takes care of the correspondence model and detecting modifications and deletions. Thus, the user can focus on specifying the forward and backward rules for the transformation.

4.1 Correspondence Model

In order to implement a TGTS, besides source and target graph, a correspondence graph is needed. It turned out, that a simple correspondence model, as depicted in Figure 1 is sufficient for all transformation problems that we encountered so far.

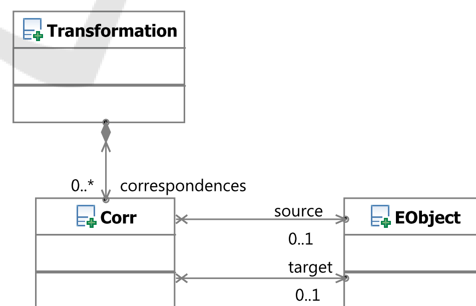

Figure 1: Correspondence metamodel.

A Transformation contains an arbitrary number of correspondence elements (Corr). In order to establish traceability links, such elements always refer to at least one EObject from the source model and one EObject from the target model. In general, the easiest kind of mapping are 1:1 mappings. Thus, for the Corr class at maximum one source and one target element are allowed. In case of 1:n mappings,

which are also possible, no additional elements are created in the correspondence model nor is the upper bound for source or target elements extended. Rather the additional elements (2-n) are maintained manually in the rule that covers the basic (1:1) mapping. Please note that the user may also extend the correspondence model by subclassing from the Corr class and add m:n relationships between source and target model elements if necessary.

The main purpose of the correspondence model is to establish traceability links between source and target elements. The framework checks the correspondence model before modifying source or target models in order to detect updates or deletions of model elements.

4.2 Transformation Rules

As stated in Section 3.3, a triple graph transformation system may be used for bidirectional and incremental model transformations. Our BXtend framework implements such a TGTS by using the correspondence model as correspondence graph *c*. Maintaining the correspondence graph is done by the framework, the transformation engineer can focus on the respective rule specifications for forward and backward transformations.

When specifying a model transformation, the software engineer needs to decide in which direction the spanning containment tree should be processed, i.e. in a bottom-up way by starting at the leaves, or top-down when the root element is considered first. As declarative approaches, like QVT-R or TGGs perform a topological sort of model elements, transformation developers have very limited influence on the transformation order. Thus, the developer must typically assume the resulting ordering of the rules (and the execution semantics), when the model transformation is specified.

Contrastingly, in our approach, the transformation order is fully under the control of the software engineer. I.e., when strictly following a top-down order the user can assume that container elements in the target model already exist, when their child elements are created.

The common behavior (including management of the correspondence model) is implemented in an abstract base class, which all user-implemented transformation rules need to inherit from. Listing 1 shows a cutout of this class with the most important methods, which handle the correspondence model getOrCreateCorrModelElement and the on-demand creation of source and target model elements getOrCreateSourceElem, getOrCreateTargetElem.

Listing 1: Cutout of the abstract base class Elem2Elem.

```
1  abstract class Elem2Elem {
2    ...
3    def getCorrModelElem(EObject obj) {
4      (corrModel.contents?.get(0) as Transformation).
           correspondences.
5        findFirst[corr |
6          corr.sourceElement == obj || corr.
               targetElement == obj]
7    }
8
9    def getOrCreateCorrModelElement(EObject obj) {
10     var Corr corr = obj.getCorrModelElem
11     if (corr == null) {
12       corr = corrFactory.createBasicElem => [
13         if (obj.eClass.EPackage instanceof
               sourcePackage)
14           sourceElement = obj
15         if (obj.eClass.EPackage instanceof
               targetPackage)
16           targetElement = obj ]
17       (corrModel.contents.get(0) as Transformation).
             correspondences += corr
18     }
19   }
20
21   def getOrCreateSourceElem(Corr corr, EClass clazz) {
22     var EObject source = corr.sourceElement
23     if (corr.sourceElement == null) {
24       source = sourceFactory.create(clazz)
25       corr.sourceElement = source
26     } return source
27   }
28
29   def getOrCreateTargetElem(Corr corr, EClass clazz) {
30     var EObject target = corr.targetElement
31     if (corr.targetElement == null) {
32       target = targetFactory.create(clazz)
33       corr.targetElement = target
34     } return target
35   }
36 }
```

Listing 2 depicts a cutout of the class which serves as an entry point for the transformation. The class provides methods sourceToTarget() and targetToSource(), which call the corresponding methods from the rule classes in the order specified by the user (please note, these methods are omitted from the listing due to space restrictions). Furthermore, this class contains methods dealing with the deletion of elements, as shown in the cutout in Listing 2.

Listing 2: Cutout of the entry class for the transformation.

```
1  class Source2TargetTransformation {
2  ...
3    def private deleteUnreferencedSourceElements() {
4      val List<EObject> deletionList = newArrayList
5      (corrModel.contents?.get(0) as Transformation).
           correspondences.
6        filter[c | c.sourceElement == null].
7        forEach[c |
8          if (c.targetElement != null) {
9            deletionList += c.targetElement
10         }
11         deletionList += c
12       ]
13       deletionList.forEach[e | EcoreUtil.delete(e,
             true)]
14   }
15 ...
```

In order to determine elements which need to be deleted, the correspondence model is checked at the end of the transformation. In case of Listing 2, the method deleteUnreferencesSourceElements

is called at the end of the sourceToTarget() transformation. If correspondence model elements exist, in which the reference sourceElement is empty, the corresponding target element is deleted. Afterwards the correspondence element is deleted, too. An analogous method is used to determine deletions in the target model.

5 EXAMPLE

To demonstrate the flexibility and feasibility of our approach, we chose the example "Families to Persons" which is well-known in the model transformation community. Furthermore, the example is part of the ATL (Jouault et al., 2008) transformation zoo. The example was selected as a use case for the 2017 Transformation Tool Contest (TTC)[3] and we contributed a solution written with the help of the BXtend framework presented in this paper and integrated it into the Benchmarx framework (Anjorin et al., 2017), which is intended to compare BX approaches. Please note that our solution was provided as part of the reference solutions and therefore it was not published yet.

5.1 Problem

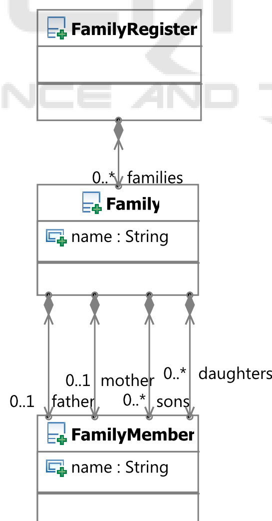

Figure 2: Families meta model.

Figures 2 and 3 depict the two meta models which are involved in the transformation scenario. In the following we refer to the families model as "source model" and the persons model as the "target model". The root element of the source model is a FamilyRegister which may contain an arbitrary number of families (Family). Each family has members who are distinguished by their roles. The meta model allows for at most one father and mother and an arbitrary number of sons and daughters. The PersonRegister on the other hand maintains a flat collection of persons (Person) who are either male (Male) or female (Female). It is important to note that we can not assume any key properties in neither model. Consequently, there may be multiple families with the same name and name clashes may also occur within the same family. In the person register there may be multiple persons with the same name and even the same birth date. No specific ordering of the collection is assumed.

[3] http://transformation-tool-contest.eu/

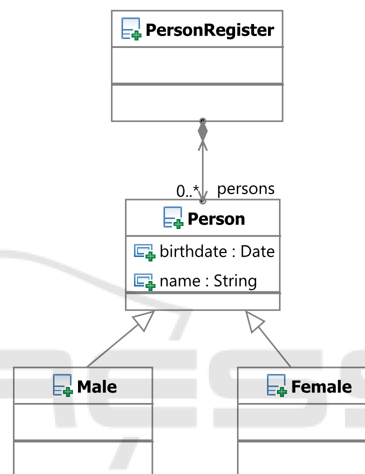

Figure 3: Persons meta model.

Consistency between a families and a persons model is established, if a bijective mapping exists which pairs mothers and daughters (fathers and sons) with females (males) and the name of every person p is "*f.name, m.name*", where m is the member in the family f paired with p.

Please note that in this example, the forward transformation is straightforward by mapping each family member to a person of the same gender and composing the person's name from the surname and first name while leaving the birthday unset. The backward transformation however is non-deterministic, as a person may be mapped either to a parent or a child, and persons may be grouped into families in different ways. As a consequence, two boolean parameters are introduced which control the behavior of the backward transformation and allow for user-defined configurations. One parameter controls whether a new person is added to an existing family (if available), or a new family is created with the person as a single family member. The other parameter controls whether a person is mapped to the parent or child role

within the family. In case both parameters are set to true, the first one takes precedence, which means in case there is a family with a matching surname but the parent role is already occupied, the person is added as a child.

This transformation scenario comes with information loss in both directions, as in the forward direction the distinction between parents and children as well as the aggregation into families is lost. Birthdays are lost in the backward direction. As a result, this case is an example for a round-trip scenario, where both models may be edited and changes have to be propagated back and forth.

5.2 Solution

We solved this transformation scenario with our BXtend framework. Our solution comprises three classes, implementing rules for transforming:

- a FamilyRegister to a PersonRegister and vice versa
- fathers and sons (identified by the respective containment relation ships between Family and FamilyMember) to Male persons and vice versa
- mothers and daughters to Female persons and vice versa

We did not need a separate rule dealing with Familys as there is no counterpart for a family in the person model.

Listing 3: Forward transformation of mothers and daughters to female persons.

```
1  override sourceToTarget() {
2    sourceModel.allContents.filter(typeof(Family))
3      .forEach[
4        family |
5        val corrRegister = family.eContainer.
             getCorrModelElement
6        val females = newArrayList
7        females.addAll(family.daughters)
8        if (family.mother != null)
9          females.add(family.mother)
10       females.forEach[
11         member |
12         val corrFemale = member.
               getOrCreateCorrModelElement()
13         val female = corrFemale.
               getOrCreatePersonElement(family.name + ",
                " + member.name,
14           PersonsPackage.eINSTANCE.female)
15       ]
16     ]
17 }
```

As stated earlier, our BXtend framework works incrementally. To this end, we need to check the correspondence model prior to each modification of source and target models. This check is done in line 8 of Listing 4.

As stated above, the backward transformation is non-deterministic for the following reasons:

- it is not clear to which family a person belongs, especially if there are duplicate names
- a person may be added as a parent or a child

To this end, we developed a configurable backward transformation which can be executed in three different modes:

- using default values for the decision parameters
- a runtime-configurable version, where the parameters may be supplied via the API
- by providing a GUI where the user may edit correspondences between family members and persons

Listing 4: Backward transformation of male persons.

```
1  override targetToSource() {
2    targetModel.allContents.filter(typeof(Male)).forEach
         [ p |
3      val corr = p.getOrCreateCorrModelElement
4      val source = corr.getOrCreateSourceElem(
           FamiliesPackage.eINSTANCE.familyMember) as
           FamilyMember
5      val firstName = p.name.split(", ").get(1)
6      val familyName = p.name.split(", ").get(0)
7      val sourceFamily = source.eContainer() as Family
8      val famRegister = p.eContainer().getCorrModelElem
           ().source as FamilyRegister
9
10     source.name = firstname
11
12     if (sourceFamily == null || sourceFamily.name !=
           familyName) {
13       val family = sourceFamily.getOrCreateFamily(p,
             famRegister)
14       p.addToFamily(family, source, [family.father], [
             family.father = it], [family.sons += it])
15
16       if (sourceFamily != null && sourceFamily.father
             == null && sourceFamily.mother == null &&
17         sourceFamily.sons.empty && sourceFamily.
               daughters.empty && decision.
               deleteEmptyFamily(sourceFamily, source))
18         EcoreUtil.delete(sourceFamily, true)
19     }
20   ]
21 }
```

Listing 5: Helper method for backward transformation.

```
1  def protected getOrCreateFamily(Family sourceFamily,
       Person p, FamilyRegister fregister) {
2    val familyname = p.name.split(", ").get(0)
3    val families = fregister.getFamilies(familyname)
4    var family = decision.getFamily(families, p,
         sourceFamily)
5    if (family == null) {
6      family = createFamilyElement(FamiliesPackage.
           eINSTANCE.family) as Family => [name =
           familyname]
7      fregister.families += family
8      decision.linkPersonToFamily(p, family)
9    }
10   family
11 }
```

Listing 4 depicts the backward transformation of male persons. The transformation is invoked for any male person in the target model. For each matching model element, the correspondence model element is retrieved (or created if there is no such element) as shown in line 3 of Listing 4. Afterwards, the FamilyMember element of the source model is retrieved or

created in line 4. As explained earlier, in the person model only one attribute is used to store both first and last names, separated by a comma. The value of this attribute needs to be split in the backward direction in order to obtain the corresponding attribute values for Family and FamilyMember. FamilyMembers are connected to their respective Family by a containment relationship. In order to get the family for a family member, the generic eContainer method of the EMF framework is used. In case no such family exists (the family member has been newly created), or the family name does not match the expected value (person name has been changed), a helper method getOrCreateFamily (c.f., Listing 5) is called. The helper method getOrCreateFamily retrieves all matching families from the family register. Depending on the value of the boolean parameters controlling the association of family members to families a family or null is returned. If no family was returned, it is created in lines 6-8 of Listing 5. Afterwards the person is added to the retrieved family in line 14 of Listing 4 by calling the helper method addToFamily, where the family member is added to the family using the proper containment relation and taking into account the configuration parameters. E.g., a male person may be added as a father (in case the boolean variable is set accordingly), or as a son otherwise. Please note, that in case the parent role of the family is taken already, the family member will be added as a son, regardless the value of the configuration parameter.

6 EVALUATION

In order to demonstrate the feasibility of our framework, we tested it in several bidirectional transformation scenarios. For the transformation tool contest, we implemented the Families 2 Persons example as discussed in Section 5. Our solution was able to pass all provided test cases, while the TGG and QVT-R based solutions implemented with *eMoflon*(Anjorin et al., 2012) and *medini QVT*[4] failed in some of them. Surprisingly, when applying LOC metrics on the respective transformation descriptions it turned out that our solution did not require significantly more implementation effort. Even if the other formalisms are designed explicitly for bidirectional transformations and they provide a highly-declarative syntax,

Table 1 summarizes the result of a comparison of our solution to the Families 2 Persons test case of the transformation tool contest 2017 with other tools. The provided transformation case contains 34 different JUnit test cases, classified into forward and backward tests with both batch and incremental behavior. Our solution written with the BXtend framework presented in this paper was able to pass all supplied test cases. We compared the BXtend solution with the other provided reference solutions for this transformation case, which are all based on bidirectional languages. Please note that BX languages have a predefined semantics for consistency analysis and repair, which can be adopted to the specific use case only to a very limited extent and thus the provided transformation tool does not always fit the specific requirements of the transformation case.

We also evaluated our BXtend solution in a real-world scenario: Round-trip engineering between UML class diagrams and Java source code. The transformation problem was solved using three different formalisms: TGGs and the tool TGG Interpreter (Buchmann and Westfechtel, 2016), QVT-R (Greiner et al., 2016) and a hand-crafted triple graph transformation system (Buchmann and Greiner, 2016). The following lessons have been learned and motivated us to provide the BXtend framework:

Implementation Effort. The most surprising result was that the BXtend solution outperforms the QVT-R solution in terms of LOC metrics. Even if TGGs and QVT-R provide support for bidirectional and incremental model-to-model transformations, both approaches required significant effort to specify bidirectional rules for the round-trip engineering use case. **1032** lines of BXtend code were required compared to **1905** lines of QVT-R code (in the round-trip engineering use case).

Redundancy of the Rule Set. The TGG approach suffers from the *redundancy of the rule set*, caused by inherent properties of triple rules: As each rule describes a synchronous extension of source, correspondence and target graph respectively, the subgraph to be added is fixed in the rule. Consequently, it has to be defined *statically*, which nodes and edges of which types have to be created and how the attribute values have to be calculated. As a result, variability in right hand sides of rules leads to several similar rules. We observed a similar effect in the QVT-R specification, as it does not support variability either and thus, slight differences have to be addressed in different rules where large parts of the rule body needs to be copied & pasted. The BXtend solution does not suffer from this problem, since language features like inheritance, genericity or control structures are available and extensively exploited in our solution.

[4]http://projects.ikv.de/qvt

Table 1: Results of running the test cases for the transformation tool contest.

	Total test cases	**BXtend**	**eMoflon**	**BiGUL**	**QVT-R**
Batch forward	7	7	7	7	7
Batch backward	11	11	11	11	11
Incremental forward	8	8	5	3	4
Incremental backward	8	8	5	4	0
Total	34	34	28	25	22

Cognitive Complexity. From the three solutions, QVT-R imposes the highest cognitive complexity to the transformation developer for several reasons: The rules are quite complex, even if there is a declarative way of specifying them using a textual syntax. Often, it is not clear in which order statements in *when* and *where* clauses are actually executed. Equations play a dual role, as depending on the binding state of its left and right-hand side, an equation may serve as an additional constraint or as a simple assignment. Nevertheless, the transformation developer needs to specify rules which can be executed in both directions, which means that operational semantics of QVT-R needs to be taken into account carefully. In the TGG approach, the transformation engineer has to specify the rules in a way which allows for a correct interplay, without explicitly specifying dependencies between rules. Again, the execution in both transformation directions has to be considered. In the BXtend approach, the cognitive complexity for the developer is reduced, as there are separate rules for each transformation direction. On the other hand, the developer needs to make sure that both transformation directions actually match.

Level of Abstraction. The TGG specification is highly declarative, allowing for transformations in both directions from only a single rule set. Furthermore, the TGG engine deals with changes in the participating models. Triple rules may be specified in an intuitive graphical syntax and a large number of rules in the considered transformation scenario have modest size and complexity. QVT-R rules are also highly declarative and they also only require one single rule set for executing transformations in both directions. Furthermore, QVT-R employs OCL statements in both queries and *when/where* clauses allowing for a compact and concise specification of dependencies between relations. The BXtend solution resides on the lowest level of abstraction, as Xtend (the programming language on which our solution is based) primarily is a procedural object-oriented language augmented with some declarative language constructs like lambda expressions. However, in the use cases considered so far this fact did not cause negative effects on the required implementation effort or on the complexity of the resulting forward and backward rules.

7 CONCLUSION AND FUTURE WORK

In this paper, we presented a light-weight frame work for specifying bidirectional and incremental model-to-model transformations based on the Xtend programming language. We showed the feasibility of our approach by comparing it to standard M2M approaches in the popular Families 2 Persons transformation scenario. Furthermore, we also applied our approach in a real-world case study dealing with round-trip engineering between UML class diagrams and Java source code, which was also solved using TGGs and QVT-R. This case study unveiled several drawbacks of the TGG and QVT-R approach which are not present in our BXtend solution.

Current work addresses the implementation of further model-to-model transformation scenarios, as described in (Westfechtel, 2016). Once the solutions are implemented, we plan to add respective test cases to the Benchmarx framework (Anjorin et al., 2017) to obtain further results comparing our BXtend framework with other solutions based on TGGs or QVT-R.

REFERENCES

Anjorin, A., Diskin, Z., Jouault, F., Ko, H.-S., Leblebici, E., and Westfechtel, B. (2017). Benchmarx reloaded: A practical framework for bidirectional transformations. In Eramo, R. and Johnson, M., editors, *Sixth International Workshop on Bidirectional Transformations (BX 2017)*, CEUR Workshop Proceedings.

Anjorin, A., Lauder, M., and Schürr, A. (2012). eMoflon: A Metamodelling and Model Transformation Tool. In Störrle, H., Botterweck, G., Bourdellès, M., Kolovos, D., Paige, R., Roubtsova, E., Rubin, J., and Tolvanen, J., editors, *Joint Proceedings of the Co-located Events at the 8th European Conference on Modelling Foun-*

dations and Applications (ECMFA 2012), page 348, Copenhagen, Denmark. Technical University of Denmark (DTU). ISBN: 978-87-643-1014-6.

Arendt, T., Biermann, E., Jurack, S., Krause, C., and Taentzer, G. (2010). Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In Petriu, D. C., Rouquette, N., and Haugen, Ø., editors, *Proceedings 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010), Part I*, volume 6394 of *Lecture Notes in Computer Science*, pages 121–135, Oslo, Norway. Springer-Verlag.

Buchmann, T., Dotor, A., and Westfechtel, B. (2009). Triple Graph Grammars or Triple Graph Transformation Systems? In Chaudron, M. R., editor, *Models in Software Engineering, Workshops and Symposia at MODELS 2008*, volume 5421 of *Lecture Notes in Computer Science*, pages 138–150. Springer Verlag. best paper of Workshop MCCM 2008, Toulouse, France, September/October 2008.

Buchmann, T. and Greiner, S. (2016). Handcrafting a Triple Graph Transformation System to Realize Round-trip Engineering Between UML Class Models and Java Source Code. In Maciaszek, L. A., Cardoso, J. S., Ludwig, A., van Sinderen, M., and Cabello, E., editors, *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 2: ICSOFT-PT, Lisbon, Portugal, July 24 - 26, 2016.*, pages 27–38. SciTePress.

Buchmann, T. and Westfechtel, B. (2016). Using Triple Graph Grammars to Realize Incremental Round-Trip Engineering. *IET Software*. Online first, http://digital-library.theiet.org/content/journals/10.1049/iet-sen.2015.0125.

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646.

Ehrig, K., Guerra, E., Lara, J. D., Lengyel, L., Levendovszky, T., Prange, U., Taentzer, G., Varro, D., and Varro-Gyapay, S. (2005). Model transformation by graph transformation: A comparative study. In *Proceedings of the International workshop on model transformations in practice (MTiP 2005), Satellite Event of MoDELS 2005*, volume 3844 of *Lecture Notes in Computer Science*, pages 71–80, Montego Bay, Jamaica. Springer-Verlag.

Giese, H., Hildebrandt, S., and Lambers, L. (2014). Bridging the gap between formal semantics and implementation of triple graph grammars - Ensuring conformance of relational model transformation specifications and implementations. *Software and System Modeling*, 13(1):273–299.

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 *MODELSWARD 2016 - Proceedings of the 4rd International Conference on Model-Driven Engineering and Software Development, Rome, Italy, 19-21 February, 2016.*, pages 15–27.

Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39.

Kindler, E. and Wagner, R. (2007). Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical Report tr-ri-07-284, Software Engineering Group, Department of Computer Science, University of Paderborn.

Klassen, L. and Wagner, R. (2012). EMorF - a tool for model transformations. In Krause, C. and Westfechtel, B., editors, *Proceedings of the 7th International Workshop on Graph Based Tools (GraBaTs 2012)*, ECEASST 54, page 6 p., Bremen, Germany.

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). *Meta Object Facility (MOF) Version 2.5*. OMG, Needham, MA, formal/2015-06-05 edition.

Popoola, S., Kolovos, D. S., and Rodriguez, H. H. (2016). EMG: A domain-specific transformation language for synthetic model generation. In *Theory and Practice of Model Transformations - 9th International Conference, ICMT 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-5, 2016, Proceedings*, pages 36–51.

Rose, L. M., Kolovos, D. S., Paige, R. F., Polack, F. A. C., and Poulding, S. M. (2014). Epsilon flock: a model migration language. *Software and System Modeling*, 13(2):735–755.

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

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

Stevens, P. (2007). Bidirectional model transformations in QVT: semantic issues and open questions. In *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*, pages 1–15.

Völter, M., Stahl, T., Bettin, J., Haase, A., and Helsen, S. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.

Westfechtel, B. (Online first, 2016). Case-Based Exploration of Bidirectional Transformations in QVT Relations. *Software and Systems Modeling*. doi:10.1007/s10270-016-0527-z.