Combining a Declarative Language and an Imperative Language for Bidirectional Incremental Model Transformations

Matthias Bank, Thomas Buchmann and Bernhard Westfechtel
Chair of Applied Computer Science I, University of Bayreuth, Universitätsstrasse 30, 95440 Bayreuth, Germany

Keywords: Model-Driven Software Development, Model Transformation, Bidirectional Transformation, Incremental Transformation.

Abstract: Bidirectional incremental model transformations are crucial for supporting round-trip engineering in model-driven software development. A variety of domain-specific languages (DSLs) have been proposed for the declarative specification of bidirectional transformations. Unfortunately, previous proposals fail to provide the expressiveness required for solving practically relevant bidirectional transformation problems. To address this shortcoming, we propose a layered approach: On the declarative level, a bidirectional transformation is specified concisely in a small and light-weight external DSL. From this specification, code is generarated into an object-oriented framework, on top of which the behavior of the transformation may be complemented and adapted in an imperative internal DSL. An evaluation with the help of a well-known transformation case demonstrates that this layered hybrid approach is both concise and expressive, and also scalable.

1 INTRODUCTION

A wide range of application domains demands for *bidirectional transformations* (bx), which are mechanisms for specifying and maintaining consistency between two or more models. Many areas, including model-driven software development (MDSD) (Völter et al., 2006) have been subject for studies of bx. Roundtrip engineering in model-driven architecture (MDA)(Mellor et al., 2002) processes e.g. is a use case for bx.

Unidirectional languages and tools may be used to solve bx problems, resulting in increased effort of maintaining separate implementations with mutually consistent behavior. To cope with this fact, a wide range of dedicated *domain-specific languages* (DSLs) and accompanying tools for bx have been developed in the past, promising to assist transformation developers in solving bx problems more efficiently and reliably. A detailed and feature-based classification of numerous bx approaches and tools may be found in (Czarnecki and Helsen, 2006) and (Hidaka et al., 2016).

Typically, bx approaches reside on a high level of abstraction. They provide *declarative languages* that relieve the transformation developer from explicitly specifying both transformation directions as well as incremental behavior. This is achieved in different ways: In grammar-based approaches, a grammar is defined to generate consistent pairs of models (Schürr, 1994). In relational approaches (Cicchetti et al., 2010), consistency relations between source and target models are defined. In functional approaches (Foster et al., 2007), one transformation direction is specified explicitly, and the opposite transformation is derived automatically.

Unfortunately, the declarative bx approaches that have been proposed so far suffer from a problem which is shared by all of them: While they guarantee certain bx laws (regarding consistent behavior of forward and backward transformations), they lack *expressiveness*: Certain bx problems cannot be solved at all, or they can be solved only partially, or they can be solved, but the specification effort is much higher than expected. In previous work, this claim was substantiated by a number of *benchmarks* and *case studies*, covering both artificial and real-world transformation scenarios (Anjorin et al., 2020; Westfechtel, 2019; Westfechtel and Buchmann, 2019; Buchmann and Westfechtel, 2013; Greiner et al., 2016; Buchmann and Westfechtel, 2016).

These observations motivated us to setup *BXtend* (Buchmann, 2018) – a framework for bidirectional incremental model transformations based on the pro-

Bank, M., Buchmann, T. and Westfechtel, B.
Combining a Declarative Language and an Imperative Language for Bidirectional Incremental Model Transformations.
DOI: 10.5220/0010188200150027
In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2021)*, pages 15-27
ISBN: 978-989-758-487-9

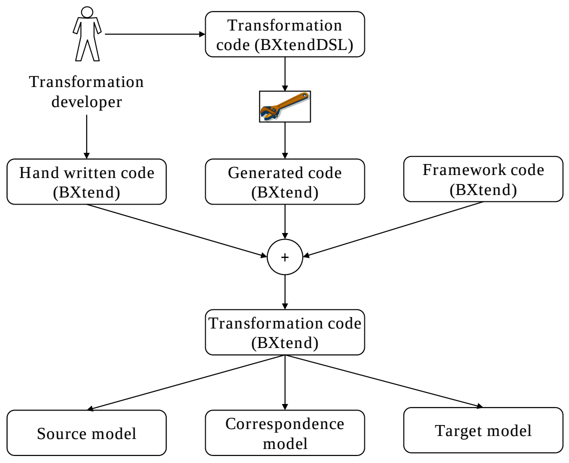

Figure 1: Layered approach to bidirectional transformations.

gramming language Xtend[1]. The framework provides a set of reusable classes that are extended to program bidirectional transformations. Thus, a bidirectional transformation is specified in an *imperative language* which provides for the required expressiveness to solve non-trivial bx problems. The BXtend approach was evaluated successfully in several transformation scenarios (Anjorin et al., 2020; Westfechtel and Buchmann, 2019; Bank et al., 2020; Greiner et al., 2016). Although in BXtend both transformation directions have to be programmed explicitly, the resulting transformation definitions are still even more concise than in some declarative approaches. This is due to lacking flexibility of the respective languages, resulting in redundant rule sets.

This paper complements previous work on BXtend by providing a declarative language called *BXtendDSL*. While the BXtend framework provides an *imperative internal DSL* (based on Xtend), BXtendDSL is a small and light-weight *declarative external DSL*. In BXtendDSL, the transformation developer essentially declares correspondences between elements of source and target languages. Intentionally, BXtendDSL is incomplete, i.e., usually it is not possible to solve the respective transformation case completely in this external DSL (this would have required a considerably more expressive and comprehensive language). Rather, from a transformation definition written in BXtendDSL code may be generated on top of the BXtend framework. Subsequently, the generated code is extended with manually written imperative code. Altogether, this results in a *layered approach* (Figure 1): First, the external DSL is used to specify correspondences declaratively. In a second step, the internal DSL is employed to take care of all the algorithmic details required to solve the respective bx problem completely.

In this paper, we start by outlining our overall approach to specifying bidirectional incremental transformations (Section 2). Subsequently, we present BXtendDSL (Section 3). We illustrate our approach by a well-known bx problem: the Families-to-Persons case (Section 4). This example is used in Section 5 to perform a thorough evaluation comparing our new layered approach to the plain BXtend solution, regarding the following research questions:

1. Is the size of the resulting transformation definition smaller, indicating a reduced implementation effort for the transformation developer?
2. Does the resulting transformation fulfill the quality criteria? I.e., it must pass all tests that the BXtend solution also passes.
3. Is the overall performance affected by introducing the additional layer of abstraction?

Finally, related work is discussed in Section 6, while Section 7 concludes the paper.

2 APPROACH

As stated above, BXtend (Buchmann, 2018) is a pragmatic approach to programming bx, with a special emphasis to address problems encountered in the practical application of existing bx languages and tools. Built upon the programming language Xtend, BXtend provides the full-fledged potential of imperative programming, combined with powerful declarative parts used for describing transformation patterns.

Technically, the transformation developer programs a *Triple Graph Transformation System* (TGTS) (Buchmann et al., 2009) using the BXtend framework, employing a correspondence model which may be manipulated using an internal DSL. The project presented in this paper – *BXtendDSL* – extends the framework and furthermore provides an external DSL, which allows for a concise specification of (bidirectional) transformation rules in a declarative textual syntax.

When working with the stand-alone BXtend framework, certain classes have to be adapted for specific transformations in several places, e.g. when model elements are created or deleted, when user supplied rules are called and for rule orchestration. Furthermore, the standard generic correspondence model only supports 1:1 correspondences. If 1:n or m:n correspondences are required, manual adaptions are possible – resulting in additional adaptations of the internal DSL for accessing and manipulating the cor-

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

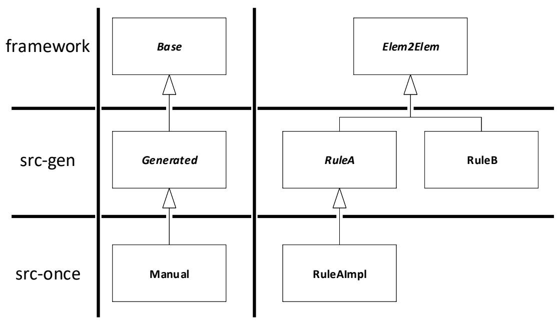

Figure 2: Architecture, based on the generation gap pattern.

respondence model. The transformation developer needs to specify both transformation directions separately, resulting in BXtend transformation rules with a significant portion of repetitive standard code.

BXtendDSL aims at minimizing the effort for the transformation developer significantly, by automatically generating the transformation-specific code from the DSL containing the transformation definition into the corresponding BXtend framework classes by keeping the flexibility of BXtend at the same time, since all generated classes may still be adapted freely. Furthermore, BXtendDSL supports m:n correspondences and provides a new internal DSL, which allows for an even more efficient access to the correspondence model. The concise and declarative notation of transformation rules allows to specify patterns between attributes and references of classes of both source and target model which are subject to transformation. The rule body provides a less-verbose syntax for model access and allows for a consistent specification of both transformation directions. BXtend transformation classes are automatically derived from the DSL avoiding the need of manually supplying repetitive standard code. All aspects of the transformation which can not be expressed adequately in a declarative way may be supplied imperatively using predefined hooks in the generated code. Figure 2 depicts the architecture with the DSL and all derived artifacts on the different levels, designed as a generation gap pattern (Fowler, 2011).

The framework comprises generic code for accessing the correspondence model and for transformation rules (depicted in the row framework). Based on code specified in the DSL file, generated classes inherit from those base classes adding specific behavior (src-gen). If parts of the transformation can not be expressed adequately in a declarative way in the DSL, extension points are generated (src-once), where imperative code may be added by the transformation developer on the level of the Xtend programming language.

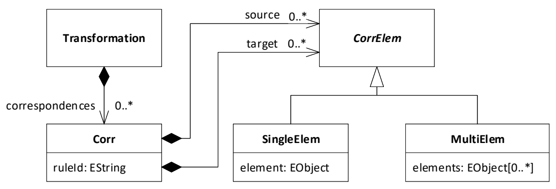

Figure 3: Correspondence metamodel (Ecore).

Incremental change propagation relies on a persistently stored *correspondence model* which has been generalized significantly compared to its predecessor version (Buchmann, 2018). Figure 3 depicts the underlying *metamodel*. A Transformation maintains a set of correspondences of class Corr, each of which stores the respective applied rule (attribute ruleId) and references to source and target elements of class CorrElem. In this way, m:n correspondences may be represented. Correspondence elements refer to the actual objects of the source and the target model by attributes of type EObject. A correspondence element may in turn represent a single object (SingleElem) or a set of objects (MultiElem). Sets of objects may be used for example when dealing with transformation problems which require folding/unfolding operations of source and target elements respectively and go beyond the scope of this paper. However, the interested reader is referred to (Bank, 2019) for further details.

3 LANGUAGE

In this section, we describe the syntax and semantics of BXtendDSL in general terms; an application follows in the next section.

3.1 Syntax

BXtendDSL is a small and light-weight textual language for the declarative specification of bidirectional incremental model transformations. Transformations defined in BXtendDSL define relations between language elements and are located at a high level of abstraction. However, the expressiveness of BXtendDSL is limited. Several language constructs provide hooks for the implementation of imperative aspects of the transformation at the BXtend layer. Below, we will explain the language constructs of BXtendDSL, referring to Listing 1, which will be explained in detail in Section 4.

BXtendDSL is designed for the bidirectional transformation between two models, called *source model* and *target model*, respectively. The keywords sourcemodel and targetmodel are followed by references (URLs) to the respective metamodels. Actual instances of these metamodels are supplied when the transformation is executed.

A transformation may be configured with the help of *options*. Each option will be bound to a Boolean value at runtime when the transformation is executed. Options are only defined, but not used at the BXtendDSL layer; they will be queried in BXtend code to be written by the transformation developer. In this way, the behavior of the transformation may be controlled.

BXtendDSL is a rule-based language. A BXtendDSL program consists of a set of *rules* that specify correspondences between elements of the source and the target model. The keywords src and trg are followed by lists of typed elements. In this way, it is possible to define m:n correspondences.

Each element may be decorated by a list of *modifiers*, each of which is indicated by a respective keyword. Each modifier requires an implementation in BXtend; the code generator provides for the respective method hook. Modifiers are used for different purposes. A filter indicates an application condition. A creation modifier stands for code that needs to be executed when the respective element is created; likewise, a *deletion* modifier is available for clean-up actions required in the course of the deletion of an element. BXtendDSL provides a few more modifiers for other purposes which are not relevant for our running example; see (Bank, 2019).

Each rule may define a list of *mappings*, which define correspondences between structural features (attributes or references) of source and target elements. At the BXtendDSL layer, mappings are only defined; calculation rules for the values of structural features may have to be provided at the BXtend layer.

A mapping is defined by a list of source features, a mapping operator, and a list of target features. The mapping operator $<-->$ indicates a *bidirectional mapping* that is executed in both transformation directions. Likewise, the operators $-->$ and $<--$ represent *unidirectional mappings* (in forward and backward direction, respectively).

In simple cases, a mapping may be translated into executable code. For example, in the case of a birectional 1:1 mapping between attributes of the same type, code is generated to copy attribute values back and forth. Similarly, in the case of a bidirectional 1:1 mapping between references to corresponding objects, code is generated for "copying" links: When an object *s* is referenced in the source model, the corresponding object *t* is referenced in the target model. In this case, the mapping defines correspondence by referring to another rule that connects the corresponding objects. If a mapping is not 1:1, method hooks will be generated, where the mapping defines the signature of the respective generated method.

3.2 Semantics

There is no declarative specification of the semantics of BXtendDSL. From a program written in BXtendDSL, code is generated on top of the BXtend framework code according to the generation gap architecture of Figure 2. In the src-gen layer, either abstract or concrete classes are generated, depending on whether the BXtendDSL code defines the semantics partially or completely. In the src-once layer, the transformation developer has to provide all missing method implementations, but (s)he is also free to override default behavior as required. Altogether, the semantics of BXtendDSL programs may be adapted in a flexible way.

The framework contains an abstract class Transformation which defines the default behavior of transformations. This class provides two methods sourceToTarget and targetToSource for executing transformations in forward and backward direction, respectively. Execution is structured into the following phases (assuming forward direction in the following):

1. Rules are executed in their textual order in the BXtendDSL program. For each rule, its sourceToTarget method is called. All matches for the current rule are retrieved (taking filters into account), and consistency of the target elements with the source elements is established (including the execution of mappings). Furthermore, the method sourceToTarget collects all created elements and all elements to be deleted (in the target model) for further processing.
2. For all rules, their creation hooks are executed in turn. In this phase, the target elements to be deleted are still available.
3. For all rules, their deletion hooks are executed in turn. This allows for the execution of additional operations on the model which require the elements to be deleted.
4. In a final clean-up phase, target elements scheduled for deletion are actually deleted.

4 EXAMPLE

In this section we present the transformation problem which is used as a running example in this paper, before we discuss details of our solution.

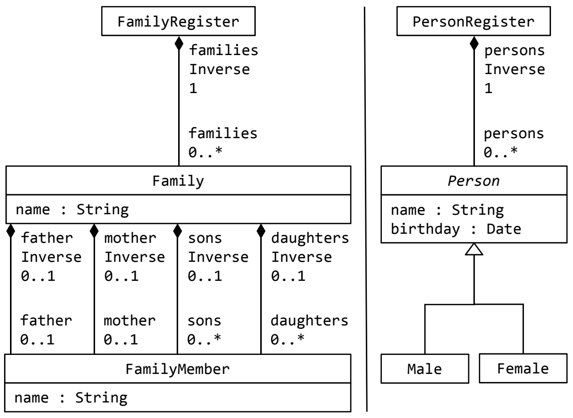

Figure 4: Metamodels.

4.1 Transformation Problem

As a running example in this paper, we chose to use the well-known Families-to-Persons benchmark (Anjorin et al., 2017), since it allows for a quantitative and qualitative evaluation of our DSL compared to other bx approaches (Anjorin et al., 2020). In the transformation case, two related, but differently structured models have to be kept in synch: A *families model*, with family members taking different roles within a family, and a flat set of males and females in a *persons model* (c.f., Figure 4). A unique root in each model is assumed. A family register stores an unordered collection of families, where each family has members who are distinguished by their roles. The metamodel permits at most one mother and at most one father as well as an arbitrary number of daughters and sons. In contrast, a person register maintains a flat unordered collection of persons who have a birthday and are either male or female. Please note that there may be multiple families with the same name, family members with the same name even within a single family, and multiple persons with the same name and even the same birthday.

A families model is *consistent* with a persons model if a bijective mapping between family members and persons can be established such that:

1. Mothers and daughters (fathers and sons) are paired with females (males).
2. The name of every person p is "$f.name$, $m.name$", where m is the member (in family f) paired with p.

Any transformation should result in a pair of mutually consistent models. However, this requirement does not determine the behavior of the transformation in a unique way. For example, when a person is inserted into the persons model, it is not clear whether the corresponding new member should be inserted into an existing or a new family, and whether the member should occupy the role of a parent or a child. Therefore, the definition of the Families-to-Persons benchmark includes behavioral descriptions for all types of model changes both for forward and backward transformations; see (Anjorin et al., 2020) for further details.

4.2 Solution

The solution for the Families-to-Persons benchmark is structured into a declarative and an imperative layer, which are explained in turn below.

4.2.1 Declarative Layer (BXtendDSL)

Listing 1 shows the BXtendDSL code for the Families-to-Persons transformation. On this declarative level, the overall transformation is defined only partially. The BXtendDSL code is supplemented with Xtend code which will be explained in the next subsection.

In lines 1–2, the source and target metamodels are defined on which the transformation is based. Lines 4–6 introduce Boolean options which are used to control the backward transformation. These options, which will be used only on the imperative layer, define whether a member should be inserted as a parent or a child and whether (s)he should be inserted into a new or into an existing family.

The rule Register2Register maps the root containers of both models (lines 8–10). The corresponding references families and persons cannot be mapped onto each other since there is no corresponding class for a Family in the persons model. These references are managed via the rules Member2Female and Member2Male, respectively.

Member2Female is used to specify the transformation of a FamilyMember to a Female person and vice versa (lines 12–17). The rule defines two modifiers, resulting in the creation of methods stubs at the imperative layer. The transformation developer needs to implement the respective bodies in order to realizes the desired behavior. Both modifiers are relevant only in forward direction. The filter modifier on the source element member (line 13) has to ensure that only mothers and daughters are considered by the rule. The creation modifier (line 14) has to handle a special case that may occur due to a move operation in the families model: A father or son may be reassigned to one of the references daugthers or mother within a family. As a consequence, a male person has to be deleted, and a female person has to be created. To avoid loss of information, the birthday has to be

```
 1 sourcemodel "platform:/plugin/Families/model/Families.ecore"
 2 targetmodel "platform:/plugin/Persons/model/Persons.ecore"
 3
 4 options
 5   PREFER_CREATING_PARENT_TO_CHILD
 6   PREFER_EXISTING_FAMILY_TO_NEW
 7
 8 rule Register2Register
 9   src FamilyRegister s;
10   trg PersonRegister t;
11
12 rule Member2Female
13   src FamilyMember member | filter;
14   trg Female female | creation;
15   member.name member.motherInverse member.daughtersInverse --> female.name;
16   member.daughtersInverse member.motherInverse --> female.personsInverse;
17   member.name <-- female.name member;
18
19 rule Member2Male
20   src FamilyMember member | filter;
21   trg Male male | creation;
22   member.name member.fatherInverse member.sonsInverse --> male.name;
23   member.sonsInverse member.fatherInverse --> male.personsInverse;
24   member.name <-- male.name member;
```

Listing 1: Transformation of Families to Persons.

copied from the male person to be deleted to the female person to be created (the gender has changed, but the person still is the same).

Furthermore, the rule Member2Female includes three mappings (lines 15–17). All of these mappings are directed, either in forward direction ($--->$) or in backward direction ($<---$). Mappings are handled in a similar way as filters: A mapping defines the inputs and outputs of some calculation; the actual calculation is programmed at the imperative layer.

The mapping in line 15 is required in forward direction for the calculation of the name of the female person, based on the name of the member and the name of the family to which the member belongs. The family name is not supplied directly as a second parameter (this would require a more general expression syntax in BXtendDSL). Rather, the family is determined by navigating along an inverse reference (motherInverse or daughtersInverse). In a similar way, the mapping in line 16 is used to connect the female person to the person register.

Finally, the mapping in line 17 is required in backward direction to calculate the member's name. In addition, this mapping is used for the management of families. The implementation of the mapping at the imperative layer has to provide a policy for inserting the member into a new or existing family as a parent or a child. For this purpose, member is required as a second argument.

The rule Member2Male works analogously to Member2Female.

4.2.2 Imperative Layer(BXtend)

At the imperative layer, we have to provide implementations of filters and modifiers defined at the declarative layer. These implementations are located at the bottom layer of the generation gap architecture displayed in Figure 2. The transformation developer implements bodies of hook methods which are called by the framework code. At the imperative layer, we exploit the expressiveness of imperative languages to exactly implement the required behavior of the bidirectional transformations.

Listing 2 displays the code for implementing the modifiers defined in lines 13–14 of Listing 1. The filter modifier is implemented in lines 2–7. The rule Member2Female may be applied only if the member is a daughter or a mother (line 6). In addition, the method filterMember contains management code which is required for handling the special case of switching the member's gender from male to female in response to a move operation. In this case, the rule Member2Female has to be applied to generate a new Female object; afterwards, the old Male object will be deleted. To save the male's birthday, a map from family members to dates is maintained which is declared in line 1. In lines 3–5, the map is extended with the birthday of the still existing male person.

The creation modifier (line 14 of Listing 1) is implemented by the method onFemaleCreation (lines 8–12 of Listing 2). If the member corresponding to the new female object has an entry in the birthdays map,

```
Map<FamilyMember, Date> birthdays = newHashMap()
override protected filterMember(FamilyMember member) {
  if (member.hasCorr) {
     birthdays.put(member, (unwrap(member.corr.target.get(0)) as Person).birthday)
  }
  return member.daughtersInverse !== null || member.motherInverse !== null
}
override protected onFemaleCreation(Female female) {
  if (birthdays.containsKey(female.corr.source().member)) {
    female.birthday = birthdays.get(female.corr.source().member)
  }
}
```

Listing 2: Code for implementing modifiers.

its value is retrieved from the map and used to assign the birthday attribute of the object female.

Mappings are translated into methods which calculate the requested outputs. Listing 3 shows the methods implementing the mappings defined in lines 16–17 of Listing 1. The method femNameFrom composes the name of the female person from the family's name and the member's name, as described at the end of Section 4.1. The respective string is not returned directly; rather, it is wrapped into an instance of some generated type which in general may aggregate multiple outputs (which are allowed in the definition of mappings).

Similarly, the method personsInverseFrom (lines 5–8) returns the target of the reference personsInverse connecting the Female object to the PersonRegister object. The target is retrieved by navigating from the family to the family register and then using the correspondence for the rule Register2Register to navigate from the source to the target.

Finally, the mapping defined in line 19 of Listing 1 is implemented by the method shown in Listing 4. The name of the female person is split into the family name and the member name (lines 2–3), the latter of which is returned as the result of the method (line 25). The remaining code manages families, according to the policy determined by the options introduced in lines 5–6 of Listing 1. Any further action is required only if the member has not been inserted yet into a family or the family name of the person differs from the current family name in the families model (line 5). If an existing family is preferred, it is attempted to retrieve a family with the given family name, giving priority to families without a mother (lines 6–11). If the attempt is successful, the retrieved family is used throughout the rest of the method; otherwise, a new family is created and inserted into the family register (lines 12–15). If a child is preferred or the selected family has a mother who is different from the member to be inserted, the member is inserted as a daughter; otherwise, the member is inserted as a mother (lines 17–22).

An analogous implementation is supplied for the corresponding filters and mappings for male persons, which are not shown here due to space restrictions.

5 EMPIRICAL RESULTS

The declarative language presented in this paper was evaluated with several transformation scenarios, e.g. the transformation problems discussed in (Westfechtel, 2019). Due to space restrictions we decided to give detailed insight into the *Families-to-Persons* transformation problem (Anjorin et al., 2017), which was evaluated with several bx approaches in (Anjorin et al., 2020). In (Anjorin et al., 2020), the BXtend solution for the Families to Persons case is discussed, allowing for an easy comparison with the BXtendDSL solution presented in this paper. However, the results of this case apply also to the other transformation scenarios studied in (Bank, 2019).

5.1 Quantitative Analysis

While BXtend already allows for concise transformation definitions (Anjorin et al., 2020; Bank et al., 2020), one of the goals of the work presented in this paper was to further minimize the specification effort. As our solution supports a textual concrete syntax, a quantitative impression of the size of the transformation definitions can be obtained by counting the number of *lines of code* (excluding empty lines and comments), the *number of words* (character strings separated by whitespace) in these lines, and the *number of characters* in these words. Empty lines, comments as well as generated code lines are omitted. Table 1 depicts the values obtained for those metrics for BXtend and BXtendDSL. For both solutions, only those lines were counted which had to be written manually. In the case of BXtendDSL, both the code in the DSL itself and the supplementary BXtend code was counted.

```
override protected femNameFrom(String memName, Family motherInverse, Family daughtersInverse) {
  new Type4femName((motherInverse ?: daughtersInverse).name + ", " memName)
}

override protected personsInverseFrom(Family daughtersInverse, Family motherInverse) {
  new Type4personsInverse
      (Register2Register.target((daughtersInverse ?: motherInverse).familiesInverse.corr).t)
}
```

Listing 3: Code for determining properties of females.

```
override protected memNameFrom(FamilyMember member, String femName) {
  val familyName = femName.split(", ").get(0)
  val memberName = femName.split(", ").get(1)

  if (member.eContainer === null || (member.eContainer as Family).name != familyName) {
    val preferExisting = trafo.getOption(Families2Persons.OPT_PREFER_EXISTING_FAMILY_TO_NEW)
    val families = srcRoot.families
    var family = if (preferExisting == true) {
      families.findFirst[name == familyName && mother === null] ?:
         families.findFirst[name == familyName]
    }
    family = family ?: FamiliesFactory.eINSTANCE.createFamily() => [
      name = familyName
      familiesInverse = srcRoot
    ]

    val preferParent = trafo.getOption(Families2Persons.OPT_PREFER_CREATING_PARENT_TO_CHILD)
      if (preferParent == false || (family.mother !== null && family.mother !== member)) {
        family.daughters += member
      } else {
        family.mother = member
      }
  }

  new Type4memName(memberName)
}
```

Listing 4: Code for calculating member names and managing families.

Table 1: Size of the transformation definitions of both solutions.

	BXtendDSL	BXtend
Lines of code	89	211
Number of words	276	565
Number of characters	3030	7571

The transformation definition was written by the same developer in both tools which are subject to this comparison. The same layout conventions and programming practices have been applied. Consequently, those numbers give a good indication that the goal of reducing the size of the transformation definition was reached. In fact, they yield a significant reduction in terms of this LOC metrics.

5.2 Qualitative Analysis

In order to perform a qualitative analysis, test cases for the different transformation directions have been specified and executed for both batch and incremental mode of operation. We assume a test case to be passed, if the resulting model matches a predefined expected model state. The BXtend solution is able to pass all tests specified in (Anjorin et al., 2020). The same holds for the BXtendDSL solution. Table 2 gives an overview of the tests and the obtained results following the criteria used in (Anjorin et al., 2020). Please note that two unexpected passes are due to cases that test for order-dependent update behavior (which state-based tools cannot provide).

The qualitative analysis shows that the correctness of the transformation is not affected by introducing the additional DSL layer. Moreover, with respect to functional requirements both BXtend and

Table 2: Aggregate test results, grouped into categories and classified as expected/unexpected passes/fails.

Category	Result	BXtendDSL	BXtend
Batch FWD	expected pass	7	7
	expected fail	0	0
	unexpected pass	0	0
	unexpected fail	0	0
Batch BWD	expected pass	11	11
	expected fail	0	0
	unexpected pass	0	0
	unexpected fail	0	0
Incr. FWD	expected pass	8	8
	expected fail	0	0
	unexpected pass	0	0
	unexpected fail	0	0
Incr. BWD	expected pass	7	7
	expected fail	0	0
	unexpected pass	1	1
	unexpected fail	0	0
Total	expected pass	33	33
	expected fail	0	0
	unexpected pass	1	1
	unexpected fail	0	0

BXtendDSL achieve a perfect result: Both solutions pass all test cases. None of the other solutions compared in (Anjorin et al., 2020) exhibit a pass rate of 100 %.

5.3 Performance Analysis

In order to evaluate the efficiency and scalability of the resulting transformation with respect to increasing model size, two experiments were conducted in both forward and backward directions for each of the transformation problems resulting in four sets of measurements: (1) batch transformations in forward and backward directions, and (2) incremental transformations in forward and backward directions. The batch transformations test how the solutions scale when creating corresponding opposite models of increasing size (model size up to 1.000.000 elements). For incremental transformations, the time required to locate and propagate corresponding changes to the dependent model is measured.

The tests were performed on the same machine and in isolation for each solution and each transformation problem. A desktop PC with an AMD Ryzen 7 3700x CPU was used, running at a standard clock of 3.60 GHz, with 32 GB of DDR4 RAM and with Microsoft Windows 10 64-bit as operating system. We used Java 13.0.2, Eclipse 4.11.0, and EMF version 2.17.0 to compile and execute the Java code for the scalability test suite. Each test was repeated 5 times and the median measured time was computed.

The four measurement results are depicted in Fig. 5, 6, 7, and 8. The figure for the batch backward measurement is composed of two plots – a plot with linear/linear scale to the left, and a plot with log/log scale to the right. While the linear plot provides a realistic impression for the actual complexity of each solution, the logarithmic one zooms into finer details for smaller models and zooms out for larger models, allowing to qualitatively present large differences in runtime. Since we observed a significant difference in runtime for the batch backward solution, we decided to supply those two plots.

In three out of four measurements, the BXtendDSL solution is slower than the stand-alone BXtend solution (as expected, due to the additional layer of abstraction). However, in all measurements the BXtendDSL solution roughly exhibits linear performance, proving its scalability. Thus, the gap between BXtend and BXtendDSL is still in a reasonable range, and even the BXtendDSL solution outperforms many of the bx approaches compared in (Anjorin et al., 2020).

The reason why the BXtendDSL solution is so much faster in the batch backward case is that in the BXtend solution, retrieving matching families is expensive and results in a sharp increase of runtime. In our original implementation we used an Xtend switch statement to switch over the family size, which results in Java code which iterates over the same collection several times using non-standard Java collection classes. Increasing model sizes lead to a significant decrease in performance in this case. In the BXtendDSL solution we did not use Xtend switch statements, and only a single iteration over the families collection was neccessary, resulting in an almost linear runtime. For the sake of comparability and traceability, we did not modify the original BXtend solution, since it was the solution submitted for the TTC 2017 (Anjorin et al., 2017) and also the one that was discussed in (Anjorin et al., 2020).

5.4 Summary

As expected, the use of BXtendDSL reduces the overall size of the transformation definition considerably. This is due to the fact that parts of the transformation may be specified in a declarative way, without being forced to write separate forward and backward transformations. Functional correctness is not affected negatively; the full expressiveness of plain BXtend is retained due to the combination of declarative and imperative code. Finally, the BXtendDSL solution proves scalable since it roughly exhibits linear performance. Compared to plain BXtend, it degrades performance to an acceptable degree, due to the additional layer of abstraction.

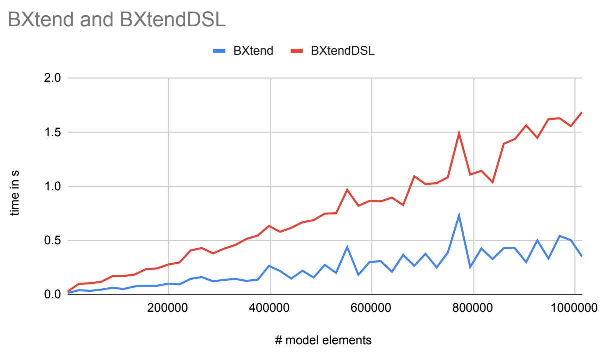

Figure 5: Forward batch transformation: Linear/linear scale.

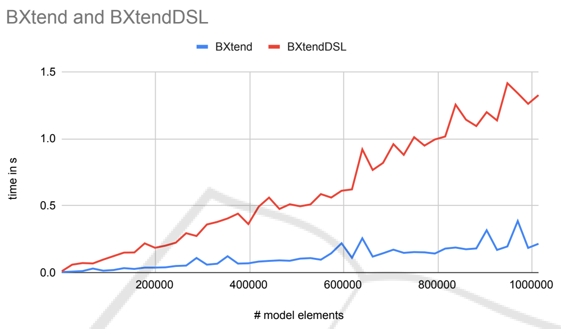

Figure 6: Forward incremental transformation: Linear/linear scale.

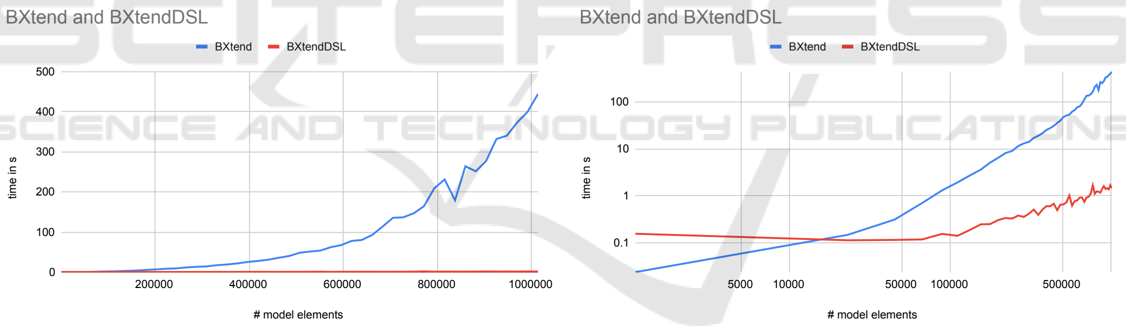

Figure 7: Backward batch transformation: Linear/linear scale (left) and log/log scale (right).

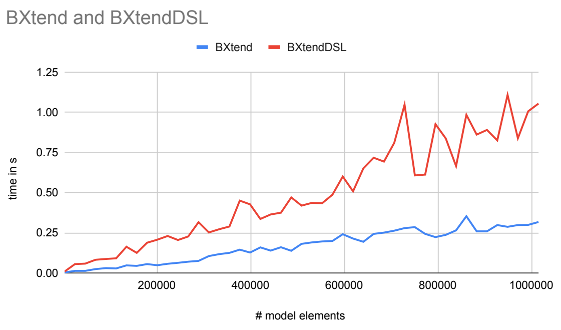

Figure 8: Backward incremental transformation: Linear/linear scale.

6 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, imperative approaches like BXtend are more powerful.

6.1 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) or TGG Interpreter (Kindler and Wagner, 2007), 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.

6.2 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, 2015) is a language following this principle. Unfortunately, there are only very few tools which are based on this standard. QVT-R allows for 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 (check-only and enforcement). Furthermore, QVT-R allows to propagate updates from the source model to the target model in subsequent transformation executions (incremental behavior). JTL (Cicchetti et al., 2010) is also a constraint-based approach, which provides guarantees of round-trip laws when executing a transformation specification. It is specifically tailored to support bidirectionality and non-determinism. When using JTL, the transformation developer supplies a set of constraints specifying a consistency relation. Those constraints are transformed together with the involved metamodels to an Answer Set Programming (ASP) (Gelfond and Lifschitz, 1988) problem, which an ASP solver can use to enable consistency restoration. However, since constraint solving is an NP-hard problem, JTL may only be applied to very small models and it does not scale with increasing model sizes.

6.3 FP-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). In this way, a pair of functions is provided which adheres to round tripping laws. These approaches origin from the BX community and are based on lenses (Foster et al., 2007), 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 (Ko et al., 2016).

6.4 Our Approach

The main differences of our approach compared to the approaches discussed above, is that BXtendDSL builds upon BXtend – a framework which is implemented in the imperative programming language *Xtend*. A switch of paradigms is possible whenever needed. The transformation developer may work on the declarative layer as long as possible and then switch to imperative constructs on demand. The generation gap pattern allows for a seamless integration of BXtend code generated from the BXtendDSL specification and hand-written imperative extensions to certain parts of the transformation rules. This approach allows for much greater flexibility in transformation definitions compared with traditional bx approaches. Furthermore, our BXtend environment is easy to use for Java developers, as the Xtend programming language directly builds upon Java and just makes it less verbose. Thus, the transformation developer does not need to learn a new programming language. Finally, 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.

7 CONCLUSION

In this paper we presented *BXtendDSL* – a programming language, specifically designed for bidirectional and incremental model transformation – which adds a declarative layer for specifying relations between model elements on top of the BXtend framework. We were able to significantly reduce the coding effort for the transformation developer, preserving all benefits from the BXtend framework at the same time by applying the generation gap pattern to our solution. A thorough evaluation proved the feasibility of our approach. Due to space restrictions, we limit ourselves to results from a popular benchmark – Families-to-Persons – in this paper. A detailed comparison with BXtend is performed in terms of quality (passed test cases), quantity (coding effort measured in LOC metrics) and performance (scalability tests). Following the Benchmarx approach, the results obtained are comparable with dedicated bx languages discussed in (Anjorin et al., 2020).

What makes our work unique is the layered approach to the specification of bidirectional incremental transformations. Quite a number of DSLs for bx have been developed. While they allow to specify bx on a high level of abstraction, they trade guarantees of bx laws for expressiveness. This is a severe problem in practice: If you cannot provide a solution in a DSL for bx which passes all test cases, you have to go for another language. In contrast, BXtendDSL is a small and lightweight DSL which allows to specify parts of the transformation declaratively. To obtain a complete solution, code is generated on top of the BXtend framework, and those problems not addressed at the declarative layer may be solved at the imperative layer. Thus, BXtendDSL in combination with BXtend provides for a pragmatic bx approach which is optimized towards conciseness, expressiveness, and scalability. Therefore, we consider this a practical contribution which facilitates the development of bidirectional incremental model transformations.

REFERENCES

Anjorin, A., Buchmann, T., and Westfechtel, B. (2017). The Families to Persons Case. In García-Domínguez, A., Hinkel, G., and Krikava, F., editors, *Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), Marburg, Germany, July 21, 2017*, volume 2026 of *CEUR Workshop Proceedings*, pages 27–34. CEUR-WS.org.

Anjorin, A., Buchmann, T., Westfechtel, B., Diskin, Z., Ko, H., Eramo, R., Hinkel, G., Samimi-Dehkordi, L., and Zündorf, A. (2020). Benchmarking bidirectional transformations: theory, implementation, application, and assessment. *Software and Systems Modeling*, 19(3):647–691.

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

Bank, M. (2019). Entwicklung einer deklarativen Sprache fr bidirektionale Modell-zu-Modell Transformationen. Master thesis (in German), University of Bayreuth, Germany.

Bank, M., Kaske, S., Buchmann, T., and Westfechtel, B. (2020). Incremental bidirectional transformations: Evaluating declarative and imperative approaches using the ast2dag benchmark. In Ali, R., Kaindl, H., and Maciaszek, L. A., editors, *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2020, Prague, Czech Republic, May 5-6, 2020*, pages 249–260. SCITEPRESS.

Buchmann, T. (2018). Bxtend - A framework for (bidirectional) incremental model transformations. In

Hammoudi, S., Pires, L. F., and Selic, B., editors, *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018.*, pages 336–345. SciTePress.

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.

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.

Buchmann, T. and Westfechtel, B. (2016). Using triple graph grammars to realize incremental round-trip engineering. *IET Software*, 10(6):173–181.

Cicchetti, A., Di Ruscio, D., Eramo, R., and Pierantonio, A. (2010). JTL: A bidirectional and change propagating transformation language. In Malloy, B., Staab, S., and van den Brand, M., editors, *Proceedings of the Third International Conference on Software Language Engineering (SLE 2010)*, volume 6563 of *Lecture Notes in Computer Science*, pages 183–202, Eindhoven, The Netherlands. Springer-Verlag.

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

Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A. (2007). Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17:1–17:65.

Fowler, M. (2011). *Domain-Specific Languages*. The Addison-Wesley signature series. Addison-Wesley.

Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In Kowalski, R. A. and Bowen, K. A., editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*, pages 1070–1080. MIT Press.

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 Hammoudi, S., Pires, L. F., Selic, B., and Desfray, P., editors, *MODELSWARD 2016 - Proceedings of the 4rd International Conference on Model-Driven Engineering and Software Development, Rome, Italy, 19-21 February, 2016*, pages 15–27. SciTePress.

Hidaka, S., Tisi, M., Cabot, J., and Hu, Z. (2016). Feature-based classification of bidirectional transformation approaches. *Software and Systems Modeling*, 15(3):907–928.

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.

Ko, H., Zan, T., and Hu, Z. (2016). BiGUL: a formally verified core language for putback-based bidirectional programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 61–72.

Mellor, S. J., Scott, K., Uhl, A., and Weise, D. (2002). Model-driven architecture. In Bruel, J. and Bellahsene, Z., editors, *Advances in Object-Oriented Information Systems, OOIS 2002 Workshops, Montpellier, France, September 2, 2002, Proceedings*, volume 2426 of *Lecture Notes in Computer Science*, pages 290–297. Springer.

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

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.

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

Westfechtel, B. (2019). Case-based exploration of bidirectional transformations in QVT relations. *Software and Systems Modeling*, 17(3):989–1019.

Westfechtel, B. and Buchmann, T. (2019). Incremental bidirectional transformations: Comparing declarative and procedural approaches using the families to persons benchmark. In Damiani, E., Spanoudakis, G., and Maciaszek, L. A., editors, *Evaluation of Novel Approaches to Software Engineering. ENASE 2018.*, Communications in Computer and Information Science, pages 98–118. Springer.