The Journal of Systems & Software 189 (2022) 111288

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

BXtendDSL: A layered framework for bidirectional model transformations combining a declarative and an imperative language☆,☆☆

Thomas Buchmann *, Matthias Bank, Bernhard Westfechtel

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

ARTICLE INFO

Article history:
Received 1 April 2021
Received in revised form 20 January 2022
Accepted 23 February 2022
Available online 17 March 2022

Keywords:
Model transformations
Bidirectional
Declarative
Imperative

ABSTRACT

Model-driven software development (MDSD) heavily relies on model transformations. While in a strict forward engineering process unidirectional transformations are used, bidirectional transformations are crucial as soon as roundtrip engineering comes into play. In this paper, we present a hybrid language specifically designed to describe bidirectional model transformations. From a declarative transformation specification code is generated which uses our framework for bidirectional and incremental model transformations. A sophisticated code generation mechanism allows for hooking into the generated transformation code at the imperative level to supply behavior that cannot be expressed declaratively. A thorough evaluation demonstrates conciseness, expressiveness, and scalability of our approach.

1. Introduction

Models and *model transformations* are the driving forces behind model-driven software development (MDSD) (Völter et al., 2006). In order to be successful, key enabling technologies are required for defining modeling languages as well as for defining and executing model transformations.

Metamodels are used to define the core concepts (abstract syntax) of modeling languages. For object-oriented modeling, the Object Management Group (OMG) provides the *Meta Object Facility (MOF)* standard (OMG, 2015) that has been adopted widely for defining metamodels. Frequently, metamodels are defined in *Ecore*, a subset of MOF that has been implemented in the *Eclipse Modeling Framework (EMF)* (Steinberg et al., 2009).

While there seems to be an emerging consensus on the definition of modeling languages and representation of models as instances of metamodels, the landscape of *model transformation languages* is characterized by a wide spectrum of approaches (Czarnecki and Helsen, 2006), even though there are OMG standards for model transformation languages, as well (Object Management Group, 2016). Languages for model transformations rely on different computational paradigms, support batch or incremental, in-place or out-place transformations, etc.

Model transformations may also be classified with respect to their direction. A *unidirectional transformation* receives a source model as input and returns a target model. In contrast, a *bidirectional transformation* (Abou-Saleh et al., 2016) may be executed in both directions.

In software engineering, a strict forward engineering process requires a chain of unidirectional transformations, which transform requirements eventually into code. In contrast, *roundtrip engineering* calls for bidirectional transformations that propagate changes back and forth. Bidirectional transformations have been studied also in a wide range of other application domains, including e.g. view-update problems in databases or bidirectional data exchange (Czarnecki et al., 2009).

Developing bidirectional transformations in a conventional programming language is awkward and error-prone: Both transformation directions have to be programmed explicitly at a low level of abstraction, taking algorithmic details of change propagation into account. Therefore, *domain-specific languages (DSLs)* and *theoretical approaches* tailored towards bidirectional transformations have been developed (Hidaka et al., 2016).

DSLs for bidirectional transformations should satisfy a number of challenging requirements: They should allow for concise transformation definitions, transformations should be specified at a high level of abstraction, forward and backward transformations should behave in a mutually consistent way, and a large class of transformation problems should be solvable.

Research on bidirectional transformations has often focused on the development of declarative approaches that guarantee consistent behavior of forward and backward transformations. Consistency has been formalized by a number of *bidirectional transformation laws* of different kinds (Schürr, 1994; Foster et al., 2007; Stevens, 2010).

☆ This article is an extended version of Bank et al. (2021).
☆☆ Editor: Doo-Hwan Bae.
* Corresponding author.
E-mail address: thomas.buchmann@uni-bayreuth.de (T. Buchmann).

https://doi.org/10.1016/j.jss.2022.111288
0164-1212/

However, these approaches have a major drawback which is shared by all of them: While they guarantee certain bidirectional transformation laws, they lack expressiveness: Certain bidirectional transformation problems cannot be solved at all, or they can be solved only partially, or they can be solved but with a considerably higher specification effort than expected. In previous work, this claim was substantiated by a number of benchmarks and case studies (Anjorin et al., 2020, 2017a; Buchmann and Westfechtel, 2016; Greiner et al., 2016; Buchmann and Greiner, 2016).

Our research follows a more pragmatic approach. Our primary focus lies on *expressiveness*, *conciseness*, and *scalability*. To this end, we combine a *declarative DSL* with an *imperative DSL*, the latter of which allows to specify all parts of the transformation that may not be addressed by the declarative DSL.

BXtendDSL is a framework for the definition and execution of bidirectional incremental model transformations. The framework is based on EMF and has been implemented in Xtend,[1] an expressive dialect of the programming language Java. The models to be transformed are assumed to be instances of metamodels, which in turn are defined with the help of Ecore, the metametamodel provided by EMF. A transformation is specified in a small and light-weight external DSL called *BXtendDSL Declarative*, which essentially serves to define correspondences between model elements with the help of transformation rules. From a declarative specification, code is generated against the libraries of the BXtendDSL framework. The application programming interfaces of these libraries constitute an internal, imperative DSL called *BXtendDSL Imperative*. The generated code is extended with handwritten code that is written in the internal DSL. In this way, the transformation developer may take care of operational details that go beyond the capabilities of the declarative language. Altogether, BXtendDSL allows to write concise transformation definitions and at the same time provides the expressiveness required for solving a wide range of bidirectional transformation problems. Furthermore, execution of BXtendDSL transformations proves scalable to large model sizes.

This paper is structured as follows: In Section 2, we introduce the research domain to which our work contributes. Section 3 provides an overview of our approach. Section 4 explains the declarative DSL and its connection to the imperative DSL. Section 5 presents transformation cases, which are evaluated in Section 6. Section 7 discussed related work, and Section 8 concludes the paper.

This paper is a considerably extended version of Bank et al. (2021). It provides a comprehensive introduction to the DSLs provided for specifying bidirectional transformations and illustrates their use through a number of small, yet challenging bidirectional transformation problems. Furthermore, it includes a thorough evaluation by a number of benchmarks that prove conciseness, expressiveness, and scalability.

2. Background

This section introduces models, model transformations, and bidirectional transformations.

2.1. Models and metamodels

Model-driven software development (MDSD) (Völter et al., 2006) emphasizes the use of *models* in the development of software systems. Generally speaking, a model is an abstraction of a system under study. Models may be employed for different purposes such as understanding, communication, analysis, simulation, and execution.

Models are expressed in *modeling languages*. A language definition comprises both *syntax* and *semantics*. *Concrete syntax* defines the notation (e.g., textual or graphical) employed by the user. *Abstract syntax* ignores irrelevant details of the notation and defines the semantically relevant elements of the language without considering their external representation.

In MDSD, abstract syntax is usually defined in terms of a *metamodel* that is composed of classes, attributes, and associations. Then, a model is represented as an *instance of a metamodel* being composed of objects, attribute values, and links.

In the *Eclipse Modeling Framework* (Steinberg et al., 2009), a metamodel is defined as instance of the metametamodel *Ecore*. An Ecore-based metamodel is composed of *classes* that may be organized into an *inheritance hierarchy*, as well as *features* of classes. A *structural feature* is an *attribute*, typed by an elementary type (e.g., boolean or string), or a *reference* to a class. References are *unidirectional*; corresponding opposite references may be grouped into pairs (*bidirectional references*). *Containment references* define exclusive containers. Finally, *behavioral features* are operations that may have typed parameters and a return type. Only the signatures of operations are defined; their bodies are not specified.

2.2. Model transformations

A *(model) transformation* (Czarnecki and Helsen, 2006) is a procedure that reads, creates, or updates a set of models. This paper deals with transformations operating on exactly two models, called *source* and *target model*, respectively.

A transformation is *endogenous* if the source and the target model are instantiated from the same metamodel; otherwise, it is called *exogenous*.

A *transformation definition* is a "program" that specifies a transformation. Transformations may be defined *declaratively* — e.g., in terms of *transformation rules* — or *imperatively*.

A *trace* records the execution of a transformation by capturing the applied rules as well as the matched source and the created target elements. A *transient* trace is maintained only during the transformation, a *persistent* trace is stored permanently. A trace may be represented as a model, too (called *correspondence model*).

A transformation may be defined either in a general-purpose language or in a *domain-specific language (DSL)* (Fowler, 2011). An *external DSL* is a separate language with dedicated syntax and semantics. An *internal DSL* is embedded into a host language and may be provided e.g. as an application programming interface (API).

A *batch transformation* reads the source model and creates the target model from scratch. An *incremental transformation* receives a pair of models as input and modifies the already existing target model. If the correspondence model is persisted, it may be used as a third input to facilitate change detection and propagation.

An incremental transformation should satisfy two crucial requirements. First, like a batch transformation, it should establish *consistency* of the target model with the source model. Consistency may be defined by a (usually complex) predicate, being composed of constraints on the contents of interrelated models. Second, an incremental transformation should minimize the changes required to reestablish consistency. Several notions of *least change* have been proposed in the literature (Cheney et al., 2017), but there is no universally accepted definition.

An incremental transformation tool is called *change-based* if it is supplied with the changes on the source model. A *state-based* tool receives only model states as inputs and infers the changes from the states.

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

An incremental transformation is used to synchronize changes on interdependent models. Synchronization may be performed *live* — each change on one model is propagated immediately to the opposite model — or *on demand* — a command has to be issued explicitly to launch synchronization.

2.3. Bidirectional transformations

A transformation is *unidirectional* if it is executed only in the direction of the target model. A *bidirectional transformation* (Hidaka et al., 2016) is executed in both directions. A *forward transformation* modifies the target model, and a *backward transformation* updates the source model.

Implementing bidirectional transformations in general-purpose programming languages is awkward and error-prone: A transformation developer has to encode each transformation direction explicitly and has to ensure that forward and backward transformations are mutually consistent. This observation motivated the development of *DSLs for bidirectional transformations* (Hidaka et al., 2016) that should satisfy the following requirements:

1. The DSL should facilitate *concise* transformation definitions, i.e., short solutions of transformation problems.
2. The DSL should be as *declarative* as possible. In particular, it should relieve the transformation developer from specifying the operational details of restoring consistency.
3. The DSL should provide guarantees with respect to the consistent behavior of forward and backward transformations. These guarantees have been formalized in terms of *bidirectional transformation laws* of different kinds (Schürr, 1994; Foster et al., 2007; Stevens, 2010).
4. The DSL should be *expressive*, i.e., it should be possible to solve a wide range of bidirectional transformation problems in that language (Westfechtel, 2018a).
5. Execution of transformation definitions should be *scalable* to large model sizes.

3. Approach

BXtendDSL is a state-based framework for defining and executing bidirectional incremental model transformations on demand that is based on EMF and the programming language Xtend. The following subsections introduce BXtend, a framework that we developed in previous work, and BXtendDSL, which builds upon and extends BXtend.

3.1. Bxtend

BXtend (Buchmann, 2018) is a pragmatic approach to programming bidirectional transformations, with a special emphasis on problems encountered in the practical application of existing bidirectional transformation languages and tools. Built upon the programming language *Xtend* (Bettini, 2016), BXtend provides the full-fledged potential of imperative programming, combined with declarative parts used for describing transformation patterns. Xtend is based on Java, but extends it with advanced concepts such as extension methods, operator overloading, switch expressions, and multiple dispatch. In addition, BXtend is based on *EMF*, which provides a platform for research on MDSD that is used widely in the academic world.

In BXtend, the transformation developer may take advantage of a framework for bidirectional incremental model transformations. A correspondence model is placed in between the source model and the target model, following the well-known *Triple Graph Grammar (TGG)* approach (Schürr, 1994). However, in contrast to TGGs, where correspondences are defined declaratively by graph grammar rules, bidirectional transformations are programmed imperatively, resulting in an (imperative) *Triple Graph Transformation System (TGTS)* (Buchmann et al., 2009). To this end, the BXtend framework provides generic, reusable algorithms that are extended and adapted to solve the transformation problem at hand.

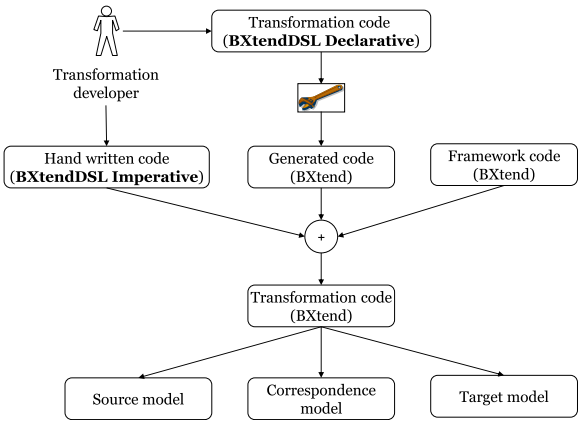

Fig. 1. Layered approach to bidirectional transformations.

When working with the stand-alone BXtend framework, the transformation developer needs to specify both transformation directions separately, resulting in BXtend transformation rules with a significant portion of repetitive code. Furthermore, the standard generic correspondence model supports only $1 : 1$ correspondences; for $1 : n$ or $m : n$ correspondences, manual adaptations are required.

3.2. BXtendDSL

BXtendDSL (Bank et al., 2020; Bank, 2019) reduces the effort for the transformation developer significantly by a *layered approach* (Fig. 1): First, the external DSL (*BXtendDSL Declarative*) is used to specify correspondences declaratively. Second, the internal DSL (*BXtendDSL Imperative*) is employed to take care of all algorithmic details required to solve the respective bidirectional transformation problem completely. The handwritten code and the generated code are combined with framework code to provide for an executable transformation.

At the declarative level, BXtendDSL provides a small, lightweight language that follows a *relational approach*: A transformation definition written in the declarative language declares correspondences between source and target model elements by means of transformation rules. Transformation rules may depend on each other, but the declarative language completely abstracts from the correspondence model (trace) that is maintained automatically at the imperative level.

By using the code generator, the transformation developer is relieved from writing repetitive routine parts of a transformation by hand. For simple parts of the transformation, the generated code ensures roundtrip properties (Section 4.4). Since the declarative DSL usually is not expressive enough to solve the transformation problem at hand completely, the generated code needs to be combined with handwritten imperative code. The interface between the declarative parts and the imperative parts is defined by certain language constructs of the declarative DSL. From these constructs, (stubs of) *hook methods* are generated that have to be implemented manually, e.g., for implementing filters or actions to be executed in response to the deletion or creation of objects.

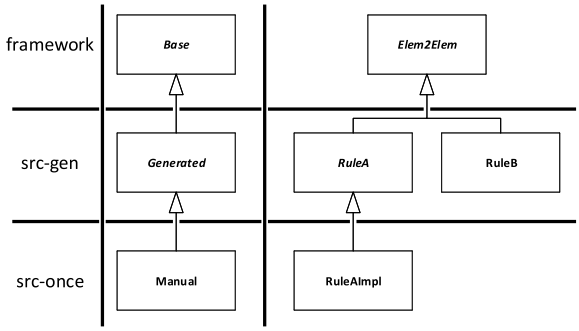

Fig. 2. Architecture, based on the generation gap pattern.

Fig. 2 depicts the *layered architecture* of the BXtendDSL framework, 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 cannot be expressed 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.

Incremental change propagation relies on a persistently stored *correspondence model* which has been generalized significantly compared to its predecessor version (Buchmann, 2018). Fig. 3 depicts the underlying EMF-based *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 (Section 5).

To program at the imperative level, we provide a powerful *internal DSL* for accessing and manipulating the correspondence model. It allows to easily retrieve the correspondence model element associated with a given element from source or target models respectively. Furthermore, methods for *wrapping* and *unwrapping* elements or even sets of elements to/from the respective wrapper classes SingleElem and MultiElem are provided.

Please note that the transformation developer does not have to deal with managing correspondences at the declarative level: Creation and deletion of correspondences need not be specified in the declarative DSL. Rather, all of the algorithmic details of managing the correspondence model are handled by our framework and are hidden from the user.

4. Language

This section describes design decisions, syntax, semantics, and roundtrip properties of *BXtendDSL Declarative* with the help of a running example.

4.1. Design rationale

The design of BXtendDSL Declarative was driven by the following design decisions:

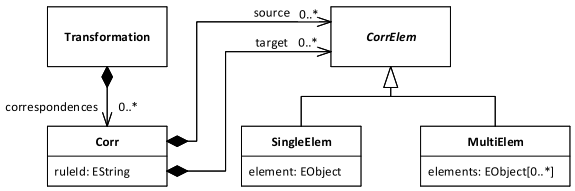

Fig. 3. Correspondence metamodel (Ecore).

Decision 1 (*Declarative Language*)**.** *BXtendDSL Declarative should be purely declarative, without any imperative language constructs.*

BXtendDSL Declarative is a *relational language* that specifies relations between source and target model elements in a declarative way. The language abstracts from all algorithmic details of consistency restoration.

Decision 2 (*External DSL*)**.** *BXtendDSL Declarative should be provided as an external domain-specific language.*

In BXtendDSL, we opted for a declarative external DSL because it allows the transformation developer to conveniently express rules concisely in a human-readable notation in terms of domain-specific concepts.

Decision 3 (*Rule-based Language*)**.** *The consistency relation underlying a bidirectional transformation should be specified in terms of rules.*

In that respect, the declarative language follows a similar approach as e.g. the language QVT-R, which was defined in an OMG standard (Object Management Group, 2016): The consistency relation is defined by of a set of rules, each of which relates a set of source elements to a set of target elements.

Decision 4 (*Declarative Rule Dependencies*)**.** *Dependencies between rules should be defined declaratively.*

In particular, the transformation developer should not be concerned with the management of the correspondence model (creation/deletion of correspondences as well as their relations to source and target elements).

Decision 5 (*Support of $m : n$ Mappings*)**.** *BXtendDSL Declarative should allow for the definition of complex mappings between m source and n target elements.*

In this respect, BXtendDSL Declarative considerably goes beyond our previous work on BXtend, which focused primarily on $1 : 1$ mappings.

Decision 6 (*Roundtrip Properties*)**.** *BXtendDSL Declarative should support the establishment of roundtrip properties.*

In BXtend, the transformation developer took full responsibility for the establishment of roundtrip properties. Using BXtendDSL Declarative, this burden may be reduced by partial code generation (see below).

Decision 7 (*Computationally Incomplete Language*)**.** *BXtendDSL Declarative should support the partial specification of bidirectional transformations.*

The language is designed to complement the imperative layer rather than to replace it. What cannot be expressed at the declarative layer, may be handled at the imperative layer.

```
1  sourcemodel "http://pdb1.ecore"
2  targetmodel "http://pdb2.ecore"
3
4  options
5    PREFER_USING_FIRST_SPACE_TO_LAST
6
7  rule Person2Person
8    src Person s;
9    trg Person t;
10   s.birthday <--> t.birthday;
11   s.placeOfBirth <--> t.placeOfBirth;
12   s.id <--> t.id;
13   s.firstName s.lastName <--> s t.name;
14
15 rule Database2Database
16   src Database s;
17   trg Database t;
18   s.name <--> t.name;
19   {s.persons: Person2Person} <--> {t.persons:
         Person2Person};
```

Listing 1: Bidirectional transformation between person databases

Decision 8 (*Partial Code Generation*)**.** *From BXtendDSL Declarative, code should be generated into the BXtendDSL framework which may be extended at the imperative layer.*

This design decision follows from the previous decision to go for an incomplete language. Thus, the overall bidirectional transformation problem is solved by a combination of declarative and imperative code.

Decision 9 (*Hooks*)**.** *BXtendDSL Declarative should provide for hooks into the imperative layer.*

A *hook* defines an extension point. In BXtendDSL Declarative, hooks are only defined but not implemented. Hooks are compiled into hook methods, the bodies of which have to be filled in by the transformation developer. In this way, the declarative code and the imperative code are glued together.

4.2. Sample transformation

As a running example for the current section, we use a simple bidirectional transformation between person databases (Westfechtel, 2018a): In the *Persons to Persons* case, two databases have to be synchronized, each of which contains a flat set of persons. For each person, an identifier, the birthday, and the place of birth are stored. Names are handled differently (first and last name are separate attributes in the source database, while the target database maintains only a single name attribute). We assume that the source and target metamodels both define a class Database, a class Person, and a containment reference persons connecting them. Furthermore, each model instance is composed of a single Database instance, to which a collection of Person instances is attached by instances of the reference persons.

This transformation problem is solved almost completely at the declarative layer (Listing 1). Explanations will follow in the course of Section 4.3. Only the mapping of name attributes has to be handled at the imperative layer (Section 4.5).

4.3. Syntax

This section describes the syntax of BXtendDSL Declarative in terms of EBNF rules. For the purpose of illustration, we will refer to the Persons to Persons case presented above. It should be noted that this sample transformation uses only a part of the language constructs offered by BXtendDSL Declarative. Examples of the remaining language constructs will be given in Section 5.

```
1  BxtendDSL :
2     Metamodels
3     (Config)?
4     (TransformationRule)*
5
6  Metamodels :
7     'sourcemodel' MetamodelURI
8     'targetmodel' MetamodelURI
9
10 Config :
11    'options' (OptionId)+
12
13 TransformationRule :
14    'rule' RuleId
15       'src' (ObjectMatcher)+
16       'trg' (ObjectMatcher)+
17       (FeatureMapping)*
18
19 ObjectMatcher :
20    ClassId ObjectId ('|' Modifier (',' Modifier)*)?
21
22 Modifier :
23    'filter' | 'sort'| 'deletion' |
24    'creation' | 'group'
25
26 FeatureMapping :
27    (MappingFeature)+ Direction (MappingFeature)+
28
29 Direction : '-->' | '<--' | '<-->'
30
31 MappingFeature :
32    FeatureAccess | ObjectId | FeatureCorrespondence
33
34 FeatureAccess :
35    ObjectId '.' FeatureId
36
37 FeatureCorrespondence :
38    '{' FeatureAccess ':' RuleCorrespondence (','
         RuleCorrespondence)* '}'
39
40 RuleCorrespondence :
41    RuleId ("[" ObjectId ("," ObjectId)* "]")?
```

Listing 2: EBNF for BXtendDSL Declarative

The *EBNF* shown in Listing 2 defines the syntax of BXtendDSL Declarative. The small number of rules supports the claim that BXtendDSL Declarative indeed is a lightweight language. The notation is straightforward: Identifiers and strings denote nonterminal and terminal symbols, respectively. A colon separates the left-hand side from the right-hand side of a grammar rule. On the right-hand side, (.)? denotes an optional part, vertical bars separate alternatives, and (.)* and (.)+ designate potentially empty and non-empty sequences, respectively. Lexical rules are not shown. Names for identifiers indicate their logical roles (e.g., RuleId); the lexical syntax for all of them is the same.

To provide a concise notation, minimizing the required number of lexical units, we decided to design a *whitespace-aware grammar*, where whitespaces and indentation characters carry semantics, similar to Python. As a benefit of this design decision, no curly braces are needed to wrap code blocks, resulting in an easier to read language with less syntactic noise.

A BXtendDSL transformation definition consists of three parts: a reference to the metamodels of the models to be synchronized, an optional configuration part defining parameters for controlling the transformation, and a sequence of transformation rules, which are executed in their textual order.

BXtendDSL Declarative is designed for the bidirectional transformation between two models, called *source* and *target model*, respectively. The rule for Metamodels serves to provide URI references to the source and target *metamodels*, which must be based on Ecore (lines 1–2 in Listing 1).

The grammar rule Config is used to define options, i.e., configuration parameters. Each *option* will be bound to a Boolean

value at runtime when the transformation is executed. Options are queried in the imperative code to control the behavior of the transformation. The option defined in line 5 of Listing 1 is used to control where a person name is split in the backward transformation.

A TransformationRule has a unique identifier and is composed of three parts: a *source*, a *target*, and a potentially empty sequence of feature mappings. The keywords src and trg are followed by non-empty sequences of object matchers. In this way, it is possible to define $m : n$ correspondences between m objects in the source model and n objects in the target model, respectively. An ObjectMatcher specifies an object which is matched by its class. Optionally, a sequence of modifiers may be attached to an object matcher.

Listing 1 contains two rules: Person2Person (lines 7–13) and Database2Database (lines 15–19) map objects of the classes Person and Database onto each other.

A Modifier is used to modify the behavior of a transformation rule. Modifiers are hooks to the imperative level, where their concrete behavior is implemented. They are used when the declarative language is not expressive enough to completely solve the transformation problem at hand. In this way, the declarative DSL is kept small and lightweight, and all language constructs available in the Xtend programming language may be exploited to realize the imperative parts of the transformation.

Modifiers are used for different purposes: filter serves to constrain the match of the respective object by an application condition. sort is used to sort the matches for applying the enclosing transformation rule to ensure that the matches are processed in a specific order. deletion provides for a method hook that performs clean-up actions required in the course of the deletion of an object. creation allows to execute code in response to the creation of an object. Finally, group indicates a *multi-object*, i.e., an element of a transformation rule that is matched to a set of objects in the respective model.

Modifiers are not employed in Listing 1; examples will be given later.

As described so far, a transformation rule defines correspondences at the level of objects. These object-level correspondences may be refined by correspondences at the level of features.

A FeatureMapping defines a correspondence in which features are involved. In both models, multiple elements may participate in such a correspondence. The source model elements are specified first, followed by a direction specification, and a sequence of target model elements.

The Direction specifies in which direction(s) the respective feature mapping is relevant. $\langle --\rangle$, $--\ >$, and $<\ --$ indicate bidirectional, forward, and backward mappings, respectively.

A MappingFeature declares an element which is involved in a mapping. There are three types of mapping features: feature accesses, objects, and feature correspondences.

A FeatureAccess is composed of an object identifier, followed by a feature identifier. A feature mapping that consists of feature accesses only specifies correspondences between attributes.

Listing 1 includes multiple $1 : 1$ feature mappings between the attributes birthday, placeOfBirth, and id of Person objects (lines 10–12), as well as name attributes of Database objects (line 18). Only the feature mapping in line 13 lists two feature accesses on the source model and one feature access on the target model.

Objects, referenced by an ObjectId, may also be involved in feature mappings. This alternative allows to define correspondences between objects in one model and references in the opposite model. The use of objects in feature mappings will be demonstrated later.

Finally, a FeatureCorrespondence is used to ensure that corresponding objects are referenced in the source model and the target model, respectively. Which objects correspond to each other, is defined by other transformation rules. In addition to a feature access, a comma-separated sequence of rule correspondences is specified.

A RuleCorrespondence references a transformation rule. Without further qualification, this reference refers to all objects in the opposite model listed in the respective transformation rule. This set may be constrained by listing the relevant objects (using their identifiers in the transformation rule), enclosed in parentheses. Since rules are executed in their textual order, a rule correspondence must reference a preceding rule.

A feature correspondence works as follows: If the transformation is executed from the end in which the feature correspondence occurs, references in the opposite model are populated by navigating along the links in this model and along the correspondences for the respective transformation rules to objects in the opposite model. If the transformation is executed in the opposite direction, the feature access of the feature correspondence defines which reference is to be populated, and the rule correspondence is ignored.

In Listing 1, the feature mapping in line 19 is used to align the instances of persons references in the source and the target model, respectively. Both Database objects reference corresponding Person objects. In forward direction, t.persons is obtained from s.persons by navigating for each person in this collection along the Person2Person correspondence to a person in the target model; likewise for the backward direction.

The feature mapping described above serves as an example for the *alignment* of *unordered collections*. Alignment is performed with the help of previously established correspondences between elements participating in the unordered collections. Insertions and deletions of elements are propagated to the opposite model.

4.4. Roundtrip properties

Bidirectional transformations should satisfy *roundtrip properties* that ensure their mutually consistent behavior. Below, we state the roundtrip properties that are relevant for our work, define conditions on transformation definitions, and sketch a proof that roundtrip properties are satisfied under these conditions.

4.4.1. Basic definitions

Roundtrip properties differ according to the underlying bidirectional transformation approach (Abou-Saleh et al., 2016). We roughly follow the relational approach pioneered by Stevens (2010). Our approach differs inasmuch as the BXtendDSL framework utilizes a persistently stored correspondence model.

Let S, T, and C denote sets of source, target, and correspondence models, respectively. Furthermore, let $R \subseteq S \times C \times T$ denote a *consistency relation*, which defines the set of consistent triples $(s, c, t) \in R$.

Let (s, c, t) be consistent. A *forward transformation* is a function $\overrightarrow{R}$ that updates c and t in response to updates on s:

$$\overrightarrow{R} : S \times C \times T \to C \times T \tag{1}$$

Starting from $(s, c, t) \in R$, let us assume that s has been updated to s'. $\overrightarrow{R}$ is called *correct* with respect to R if it reestablishes consistency:

$$\overrightarrow{R}(s', c, t) = (c', t') \Rightarrow (s', c', t') \in R \tag{2}$$

$\overrightarrow{R}$ is *hippocratic* if its call on a consistent triple (s, c, t) does not perform any changes:

$$(s, c, t) \in R \wedge \overrightarrow{R}(s, c, t) = (c', t') \Rightarrow (c', t') = (c, t) \tag{3}$$

The *backward transformation* $\overleftarrow{R}$ as well as its correctness and hippocraticness are defined analogously.

A *roundtrip* is a chain of opposite transformations. If correctness and hippocraticness hold for both forward and backward transformations, the first transformation in the chain will establish consistency without modifying the previously updated model (s' if the chain starts with a forward transformation). Due to hippocraticness, the opposite transformation will not perform updates any more. Thus, s' remains unaffected by a roundtrip.

4.4.2. Well-behavedness conditions

A bidirectional transformation is *well-behaved* if the forward and the backward transformation are correct and hippocratic. The following conditions on BXtendDSL transformation definitions are *sufficient* for that:

Condition 1. *The class of any object matcher must not be abstract.*

If the class of an object matcher is abstract, the respective containing rule cannot be executed in its direction (the object cannot be instantiated).

Condition 2. *The class of any object matcher must be a leaf class.*

Condition 3. *Each class may occur in at most one object matcher.*

Conditions 2 and 3 ensure that any object is transformed at most once. Otherwise, change propagation in the opposite direction may result in inconsistencies due to a non-injective mapping.

Condition 4. *The transformation definition does not contain any modifiers.*

For modifiers, imperative code is required that is not guaranteed to preserve well-behavedness.

Condition 5. *Each feature mapping must be bidirectional.*

After executing a transformation in the opposite direction of a unidirectional mapping, a subsequent inverse transformation may update the original model, which violates hippocraticness.

Condition 6. *Exactly one mapping feature occurs on either side of a feature mapping.*

If more than one mapping feature occurs on one side, it is not clear how to split or merge the respective feature values. Thus, imperative code is required in this case.

Condition 7. *In a feature mapping, either all mapping features are feature accesses, or all mapping features are feature correspondences.*

Feature accesses and feature correspondences are used to map attributes and references, respectively. Objects in a feature mapping are excluded because they always require imperative code to be written.

Condition 8. *Either all features referenced in a feature mapping are single-valued, or all referenced features are multi-valued.*

Multi-valued features cannot be mapped automatically to single-valued features.

Condition 9. *Each feature referenced in a feature mapping may occur only once in the containing rule.*

A feature occurring multiple times in the feature mappings of a given rule may result in conflicting definitions of its value.

Condition 10. *If one end of a bidirectional reference occurs in a feature mapping, its opposite end must not occur in any rule.*

In this case, executing the respective rule automatically determines the opposite end, as well. Therefore, a feature mapping for the opposite end is not needed and may even result in inconsistencies.

Condition 11. *The types of feature accesses in a feature mapping must be the same.*

Only if the types of the attributes occurring in feature accesses coincide is it possible to copy attribute values back and forth.

Condition 12. *Each feature mapping may reference at most one transformation rule.*

This condition guarantees that the same rule is used on either side of a feature mapping (if any). If different rules were allowed, transformations may not be hippocratic, for a similar reason as given for Condition 5.

All conditions for well-behavedness may be checked by static analysis of the transformation definition. They are formalized as OCL (OMG, 2014) constraints; see Listing 3 (from which a few auxiliary declarations were omitted.). The constraints refer to an Ecore-based metamodel for the abstract syntax of BXtendDSL Declarative that is derived from the grammar of Listing 2.

For the Persons to Persons case, the conditions for well-behavedness hold with only one exception: the feature mapping for name attributes in Listing 1, line 13, which violates Condition 6. When we eliminate this feature mapping, we indeed obtain a well-behaved bidirectional transformation.

Some of the conditions defined above *hard*; in case of violations, the transformation definition is considered erroneous. This applies to Condition 1 because an abstract class prevents the instantiation of an object matcher. Furthermore, violations of Conditions 9 and 10 indicate conflicting definitions of feature values. Finally, violation of Condition 2 results in an asymmetric behavior of the respective rule: In one direction, objects of subclasses may be matched; in the opposite direction, only an object of the superclass may be instantiated.

The remaining conditions are *soft*. Violations are not treated as errors; however, imperative code is required to provide for a transformation definition satisfying roundtrip properties. For example, this applies to the mapping of names in the Persons to Persons case; see Listing 7 for the respective imperative code.

Section 5 will provide further examples of transformation definitions that partially violate soft conditions, but nevertheless ensure roundtrip properties. Furthermore, the electronic resources referenced at the end of this paper include further sample transformations. In particular, the *Gantt to CPM* case described in Westfechtel (2018a) involves a non-injective mapping at the type level which requires two rules with object matchers of the same class: Both activities and dependencies in Gantt diagrams are mapped to activities in CMP networks, violating the soft Condition 3. Disjoint filters on the object matchers ensure that each CPM activity is transformed only once in backward direction.

4.4.3. Abstract semantic model

The abstract computational model for the functional behavior of BXtendDSL transformation definitions which is presented below is based on *triple graph grammars (TGGs)* (Schürr, 1994). We assume the conditions for well-behavedness introduced above and provide a proof sketch for correctness and hippocraticness. The Persons to Persons case (Listing 1) serves as a running example (ignoring the mapping between name attributes, which violates Condition 6).

```
import 'BXtendDSLNew.ecore'
import 'http://www.eclipse.org/emf/2002/Ecore'

package bXtendDSL

context ObjectMatcher inv Condition1:
  not self.clazz.abstract

context ObjectMatcher inv Condition2:
  self.clazz.isLeaf

context ObjectMatcher inv Condition3:
  self.parent.parent.rules.matchers->select(m | m.clazz = self.clazz)->size() = 1

context ObjectMatcher inv Condition4:
  self.modifiers->isEmpty()

context FeatureMapping inv Condition5:
  self.direction = Direction::Bidirectional

context FeatureMapping inv Condition6:
  self.srcMappingFeatures->size() = 1 and self.trgMappingFeatures->size() = 1

context FeatureMapping inv Condition7:
  self.mappingFeatures->forAll(mf | mf.oclIsTypeOf(FeatureAccess)) or
  self.mappingFeatures->forAll(mf | mf.oclIsTypeOf(FeatureCorrespondence))

context FeatureMapping inv Condition8:
  self.mappingFeatures->selectByKind(FeatureAccessOrCorrespondence).feature->forAll(f | f.many) or
  self.mappingFeatures->selectByKind(FeatureAccessOrCorrespondence).feature->forAll(f | not f.many)

context FeatureAccessOrCorrespondence inv Condition9:
  self.parent.parent.featureMappings.mappingFeatures->
    selectByKind(FeatureAccessOrCorrespondence)->
      select(fac | fac.feature = self.feature)->size() = 1

context FeatureCorrespondence inv Condition10:
  self.parent.parent.parent.rules.featureMappings.mappingFeatures->
    selectByKind(FeatureCorrespondence).feature->selectByKind(ecore::EReference)->
      select(ref | ref.eOpposite = self.feature)->size() = 0

context FeatureMapping inv Condition11:
  self.mappingFeatures->selectByKind(FeatureAccess).feature.eType->asSet()->size() <= 1

context FeatureMapping inv Condition12:
  self.mappingFeatures->selectByKind(FeatureCorrespondence).correspondences.rule->asSet()->size() <= 1

endpackage
```

Listing 3: OCL constraints for roundtrip properties

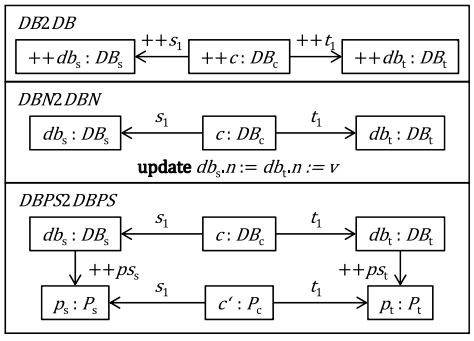

Fig. 4. Triple graph rules.

Each transformation rule is mapped to a set of synchronous *triple rules*: one rule for object matchers and one rule for each feature mapping. Three types of rules are derived: for object matchers, feature accesses, and feature correspondences.

Fig. 4 gives one example for each rule type. To save space, node and edge types are abbreviated. Subscripts s, c, and t distinguish between source, correspondence, and target graph. $++$ indicates a node or edge to be created.

An *object rule* for *object matchers* (e.g., *DB2DB*) is constructed as follows: Introduce a node for each object matcher and one correspondence node to which all source and target nodes are connected. All nodes and edges are marked by $++$. Since Condition 4 excludes modifiers, each node stands for a single object (rather than a group). Condition 1 ensures that all types may be instantiated.

For a feature mapping involving *feature accesses* (e.g., *DBN2DBN*), an *attribute rule* is created that includes nodes for source and target objects, as well as the respective correspondence node and connecting edges. Furthermore, the update part of the rule assigns the same value (denoted by the free variable v) to the corresponding attributes of source and target nodes. Please note that the construction of an attribute rule assumes a bidirectional feature mapping (Condition 5). Furthermore, Conditions 6 and 7 ensure that exactly one attribute occurs on each side of the feature mapping.

A *reference rule* for *feature correspondences* (e.g., *DBPS2DBPS*) includes graph triples ensuring that end nodes correspond to each other, and edges to be created on either side. Similarly to attribute rules, Condition 5 ensures that edges may be created synchronously, while Conditions 6 and 7 imply a $1:1$ correspondence between references. Finally, due to Condition 12, the same rule is involved in both directions, resulting in a single correspondence node connecting the targets of the edges to be created.

Directed rules for both directions are derived from triple rules, as demonstrated in Fig. 5 for forward rules; backward rules are

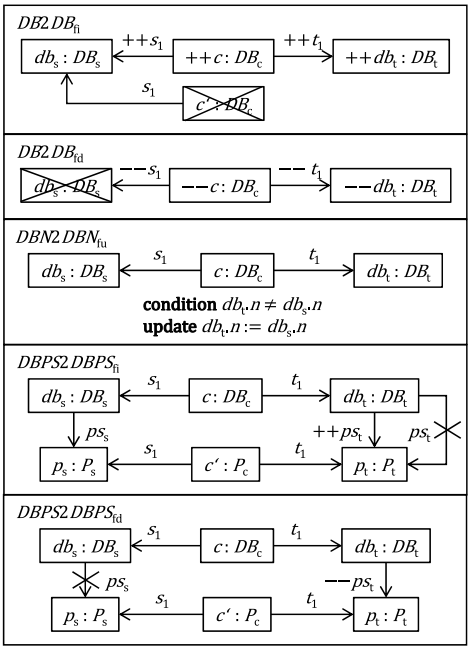

Fig. 5. Forward rules.

constructed analogously. −− denotes deletion, and crosses indicate negative application conditions. In a directed transformation, all rules are applied until all matches are exhausted.

For an object rule, an *insertion rule* ($DB2DB_{fi}$) propagates insertions of nodes that have not been transformed so far. A *deletion rule* ($DB2DB_{fd}$) deletes the correspondence node and the target nodes after one of the source nodes has been deleted. In the case of multiple source nodes, one deletion rule is required for each of them.

For an attribute rule, an *update rule* ($DBN2DBN_{fu}$) restores consistency of attribute values.

For a reference rule, two rules ($DBPS2DBPS_{fi}$ and $DBPS2DBPS_{fd}$) propagate insertions and deletion of edges, respectively.

For the proof sketch of roundtrip properties, let us first examine the *soundness* of the rule set. To this end, it has to be shown that each feature value is defined uniquely and consistently. Below, this is demonstrated for attributes; references are handled analogously.

Due to Conditions 2 and 3, an object class may participate in at most one transformation rule. Furthermore, an attribute may occur only once in at most one feature mapping per rule (Condition 9). Altogether, at most one attribute rule is derived. Due to Conditions 6 and 7, only a single attribute occurs on the opposite side. Furthermore, both attributes have the same base type (Condition 11) and the same multiplicity (Condition 8). Therefore, both attributes may be assigned the same value.

Based on a sound rule set, the *consistency relation R* is defined by the set of all graph triples that may be derived by applying the triple rules.

To prove *correctness*, we have to show that a directed transformation results in a consistent graph triple. For insertion and update rules, we observe that they establish the same postconditions as the respective triple rules. In addition, Conditions 2 and 3 ensure that only one insertion rule may be applied to each object. Thus, each object participates in at most one rule instance, as it is the case with synchronous object rules. Finally, deletion rules invert insertion rules; they are required for propagating updates in general (rather than insertions only), but do not result in graph triples which could not be generated by the set of triple rules. Altogether, we may conclude that we obtain the same result by running a directed transformation after a change to the source or target graph as if we had created the resulting graph triple by applying triple rules synchronously.

Since a directed transformation results in a consistent graph triple, none of the directed rules is applicable since each of them removes an inconsistency and thus invalidates itself. For example, an update rule cannot be executed when the values of corresponding attributes are equal, and an insertion rule requires an object that has not been transformed yet. This implies *hippocraticness*: In the course of a directed transformation, all directed rules are applied until all matches are exhausted.

4.5. Semantics

The abstract semantic model presented above assumes conditions for well-behavedness. Below, we describe the actual *operational semantics* of BXtendDSL Declarative, which is defined by generating code according to the generation gap architecture shown in Fig. 2. If well-behavedness conditions are satisfied, the operational semantics conforms to the semantic model presented above, and the generated code is fully executable.

Altogether, the transformation developer may benefit from having correct code generated for the well-behaved parts, relieving her from the painstaking duty to manually program simple and redundant code. This allows her to focus on the challenging parts of the transformation, whose well-behavedness has to be established by testing.

In the following, we explain the layers of the generation gap architecture in turn, referring to the Persons to Persons case as running example throughout. In particular, we describe the mapping of language constructs requiring imperative code to hook methods, and their integration into the framework code by method calls.

4.5.1. Framework layer

The central component of the generic framework layer is a class that is responsible for executing the overall transformation (Listing 4). Following the template-method design pattern, the class is abstract since it relies on calling transformation-specific methods to be supplied on the src-gen layer. For example, the abstract method createRules has to create the rules to be applied.

Two methods sourceToTarget and targetToSource execute transformations in forward and backward direction, respectively. Since they operate symmetrically, we explain only the forward transformation.

The transformation is organized into five phases. First, rules are executed in their textual order (lines 8–13). Each rule is applied to all matches, taking modifiers for filtering, sorting, and grouping into account.

In the second phase, by iterating over all correspondences, new target elements are added to the target model resource (lines 14–18). This is a technical step that is required in EMF.

Subsequently, for all rules, creation hooks attached to new target elements are executed (lines 20–24). Note that elements scheduled for deletion in phase 1 are still present and may be accessed in creation hooks.

In the fourth phase, for elements that have been scheduled for deletion, deletion hooks are executed before the elements are eventually deleted (lines 25–30).

In a final clean-up phase (line 31), target elements which are referenced by an obsolete correspondence model element are actually deleted.

```
1  abstract class AbstractPdb12Pdb2 implements BXtendTransformation {
2  ...
3    override void sourceToTarget() {
4      val createdElems = new HashMap<Elem2Elem, List<EObject>>()
5      val spareElems = new HashMap<Elem2Elem, List<EObject>>()
6      var Set<EObject> detachedCorrElems = new HashSet<EObject>()
7
8      for (rule : rules) {
9        val delta = rule.sourceToTarget(detachedCorrElems)
10       createdElems.put(rule, delta.createdElems)
11       spareElems.put(rule, delta.spareElems)
12       detachedCorrElems = delta.detachedCorrElems
13     }
14     for (Corr corr : (corrModel.contents.get(0) as Transformation).correspondences) {
15       for (EObject trg : corr.flatTrg.filter[eContainer === null]) {
16         targetModel.contents += trg
17       }
18     }
19
20     for (rule : rules) {
21       for (createdElem : createdElems.get(rule)) {
22         rule.onTrgElemCreation(createdElem)
23       }
24     }
25     for (rule : rules) {
26       for (spareElem : spareElems.get(rule)) {
27         rule.onTrgElemDeletion(spareElem)
28         EcoreUtil.delete(spareElem, false)
29       }
30     }
31     deleteUnreferencedTargetElements()
32   }
33
34   def protected abstract List<Elem2Elem> createRules();
35 ...
36 }
```

Listing 4: Abstract class for the transformation between person databases (generated on the framework layer)

```
1  class Pdb12Pdb2 extends AbstractPdb12Pdb2 {
2  ...
3    override protected List<Elem2Elem> createRules() {
4      var result = new ArrayList<Elem2Elem>()
5      result += new Person2PersonImpl(this)
6      result += new Database2Database(this)
7      return result
8    }
9  ...
10 }
```

Listing 5: Concrete class for the transformation between person databases (generated on the src-gen layer)

4.5.2. src-gen layer

The src-gen layer contains specific, generated code to be adapted only at the src-once layer, including a concrete class for the overall transformation as well as classes for all transformation rules.

Listing 5 shows an excerpt from the concrete transformation class Pdb12Pdb2, which extends the respective abstract framework class (Listing 4). The concrete class provides implementations of abstract methods such as createRules (lines 3–8), which inserts the rules into the list in the same order as they appear in the declarative transformation definition (Listing 1).

For each rule, a corresponding class is generated, which is concrete if it is does not rely on methods to be provided at the src-once layer. In Persons to Persons, the rule classes Database2Database and Person2Person are concrete and abstract, respectively.

Listing 6 displays an excerpt from the rule class Person2Person. In addition to the method sourceToTarget to be explained below, it contains abstract hook methods for converting between first names, last names, and names (lines 32 and 34) that were generated from the feature mapping in Listing 1, line 13.

sourceToTarget is a template method that calls hook methods for modifiers (not present in this example) and feature mappings. It is structured into two phases First, all matches of the source pattern are collected (lines 9–11). In general, matching is performed in nested loops over object matchers, taking group, filter, and sort modifiers into account. Here, a single loop matching objects by their class suffices.

Second, all matches are processed in turn (lines 13–27). For each match (line 14), correspondences and target elements are searched and created if they do not exist yet (lines 16–19). Next, the feature mappings are applied to the retrieved (or newly created) target model elements (lines 21–23). Lines 25 and 26 show the use of the hook method nameFrom, which composes the name in the target model from the first name and the last name in the source model.

4.5.3. src-once Layer

At the src-once layer, classes are generated with implementation stubs for all abstract hook methods introduced at the src-gen layer. In the Persons to Persons case, the class Person2PersonImpl is generated which has to be extended by the transformation developer.

Listing 7 shows the source code of this class after the transformation developer has filled in the bodies of the hook methods. nameFrom is used to concatenate first and last name to a single name attribute, which is used in the forward direction, while firstName_lastNameFrom describes the consistent behavior for the backward direction. In backward direction, a configuration option having been introduced in the declarative transformation definition (line 5 of Listing 1) is used to control where the name is split if it contains multiple spaces.

```
abstract class Person2Person extends Elem2Elem {
...
  override CorrModelDelta sourceToTarget(Set<EObject> _detachedCorrElems) {
    this.createdElems = new ArrayList<EObject>()
    this.spareElems = new ArrayList<EObject>()
    this.detachedCorrElems = _detachedCorrElems

    val _matches = new ArrayList<Source>()
    for (s : sourceModel.allContents.filter(typeof(pdb1.Person)).toIterable()) {
      _matches += new Source(s)
    }

    for (_match : _matches) {
      val s = _match.s

      val _corr = wrap(s).updateOrCreateCorrSrc()
      val _tType = new CorrElemType("Person", false)
      val _trg = _corr.getOrCreateTrg(_tType)
      val t = unwrap(_trg.get(0) as SingleElem) as pdb2.Person

      t.setBirthday(s.getBirthday())
      t.setPlaceOfBirth(s.getPlaceOfBirth())
      t.setId(s.getId())

      val _name = nameFrom(s.getFirstName(), s.getLastName())
      t.setName(_name.name)
    }

    return new CorrModelDelta(this.createdElems, this.spareElems, this.detachedCorrElems)
  }

  def protected abstract nameFrom(String firstName, String lastName);

  def protected abstract firstName_lastNameFrom(Person s, String name);
...
}
```

Listing 6: Generated class for the rule Person2Person (generated on the src-gen layer)

```
class Person2PersonImpl extends Person2Person {

  override protected nameFrom(String firstName, String lastName) {
    new Type4name(firstName + " " + lastName)
  }

  override protected firstName_lastNameFrom(Person s, String name) {
    if (name == s.firstName + " " + s.lastName) {
      new Type4firstName_lastName(s.firstName, s.lastName)

    } else {
      val preferFirstSpace = trafo.getOption(Pdb12Pdb2.OPT_PREFER_USING_FIRST_SPACE_TO_LAST)
      val spacePosition = if (preferFirstSpace == true) name.indexOf(" ") else name.lastIndexOf(" ")
      new Type4firstName_lastName(name.substring(0, spacePosition), name.substring(spacePosition + 1))
    }
  }
}
```

Listing 7: Class with implemented hook methods

5. Transformation cases

In Section 4, we employed the Persons to Persons case (P2P) as a running example. Below, we present more challenging cases that cover a wide spectrum of transformation problems and also employ nearly all constructs of BXtendDSL Declarative. The declarative code is shown completely, only excerpts of imperative code (one example for each modifier) are included.

5.1. Families to persons

The *Families to Persons* case (F2P), originally proposed as part of the ATL (Jouault et al., 2008) transformation zoo,[2] probably is the best explored bidirectional transformation problem so far. Below, we present our solution to the benchmark as presented in Anjorin et al. (2020).

In the F2P case, the families model contains a set of families with father, mother, sons, and daughters. In the persons model, a flat collection of male and female persons is stored (with birthdays, which are not present in the families model). The models are mutually consistent if there is a name- and gender-preserving bijection between family members and persons. Updates may be performed on both models, and have to be propagated in both directions. While the F2P case is rather small and thus implementable with acceptable effort, it poses a number of challenges such as heterogeneous metamodels, loss of information, the absence of keys (uniquely identifying properties of model elements), configurability (of the backward transformation), renamings and moves, order dependent update behavior, and application-specific requirements to change operations.

[2] https://www.eclipse.org/atl/atlTransformations/#Families2Persons.

```
sourcemodel "families.ecore"
targetmodel "persons.ecore"

options
  PREFER_CREATING_PARENT_TO_CHILD
  PREFER_EXISTING_FAMILY_TO_NEW

rule Register2Register
  src FamilyRegister s;
  trg PersonRegister t;

rule Member2Female
  src FamilyMember member | filter;
  trg Female female | creation;
  member.name member.motherInverse member.daughtersInverse --> female.name;
  member.daughtersInverse member.motherInverse --> female.personsInverse;
  member.name <-- female.name member;

rule Member2Male
  src FamilyMember member | filter;
  trg Male male | creation;
  member.name member.fatherInverse member.sonsInverse --> male.name;
  member.sonsInverse member.fatherInverse --> male.personsInverse;
  member.name <-- male.name member;
```

Listing 8: Transformation of Families to Persons (BXtendDSL Declarative)

```
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 9: Code for implementing modifiers (BXtendDSL Imperative)

```
sourcemodel "http://bags1.ecore"
targetmodel "http://bags2.ecore"

rule Element2Element
  src Element s | group, filter;
  trg Element t;
  s <--> t.value t.multiplicity;

rule Bag2Bag
  src MyBag s;
  trg MyBag t;
  {s.elements: Element2Element} <--> {t.elements: Element2Element};
```

Listing 10: Bidirectional transformation between bags

```
override protected groupSElem(Element sElem) {
    sElem.value
}
override protected filterS(List<Element> s) {
    !s.empty
}

override protected tValue_multiplicityFrom(List<Element> s) {
    new Type4tValue_multiplicity(s.get(0).value, s.size())
}

override protected sFrom(SrcMultiElemUpdater<Element> sUpdater, String tValue, int multiplicity) {
    for (var i = 0; i < multiplicity; i++) {
        sUpdater.update[true].value = tValue
    }
    new Type4s(sUpdater.finish())
}
```

Listing 11: Code for implementing modifiers and mappings (BXtendDSL Imperative)

```
sourcemodel "http://ast.ecore"
targetmodel "http://dag.ecore"

rule Operator2Operator
  src Operator s | group, filter, sort;
  trg Operator t | sort;
  s <--> t.op {t.leftInverse: Operator2Operator} {t.rightInverse: Operator2Operator};

rule Variable2Variable
  src Variable s | group, filter;
  trg Variable t;
  s  <--> t.name {t.leftInverse: Operator2Operator} {t.rightInverse: Operator2Operator};

rule Number2Number
  src Number s | group, filter;
  trg Number t;
  s <--> t.value {t.leftInverse: Operator2Operator} {t.rightInverse: Operator2Operator};

rule Model2Model
  src Model s;
  trg Model t;
  {s.expr: Operator2Operator, Variable2Variable, Number2Number} <--> {t.exprs: Operator2Operator, Variable2Variable,
    Number2Number};
```

Listing 12: Bidirectional transformation between abstract syntax trees and directed acyclic graphs

```
override protected compareSource(Source lhs, Source rhs) {
    if (computeKey(lhs.s.get(0)).contains(computeKey(rhs.s.get(0)))) {
      return -1
    } else if (computeKey(rhs.s.get(0)).contains(computeKey(lhs.s.get(0)))) {
      return 1
    } else {
      return 0
    }
  }

override protected compareTarget(Target lhs, Target rhs) {
  if (lhs.t.isParent(rhs.t)) {
    return -1
  } else if (rhs.t.isParent(lhs.t)) {
    return 1
  } else {
    return 0
  }
}
```

Listing 13: Code for sorting operators (BXtendDSL Imperative)

```
sourcemodel "http://pn.ecore"
targetmodel "http://pnw.ecore"

rule Place2Place
  src Place s;
  trg Place t;
  s.name <--> t.name;
  s.noOfTokens <--> t.noOfTokens;

rule Transition2Transition
  src Transition s;
  trg Transition t;
    TPEdge tpEdges | group, filter;
    PTEdge ptEdges | group, filter;
  s.name <--> t.name;
  {s.trgT2P: Place2Place} t <--> tpEdges;
  {s.srcP2T: Place2Place} t <--> ptEdges;

rule Net2Net
  src Net s;
  trg Net t;
  s.name <--> t.name;
  {s.elements: Place2Place, Transition2Transition} <--> {t.elements: Place2Place, Transition2Transition};
```

Listing 14: Bidirectional transformation between Petri nets and weighted Petri nets

An example of mutually consistent models is given in Fig. 6. The figure displays a slightly simplified object diagram (birthday attributes and labels of correspondence links are omitted). The source model (left) and the target model (right) are connected by a correspondence model (middle) conforming to the metamodel of Fig. 3.

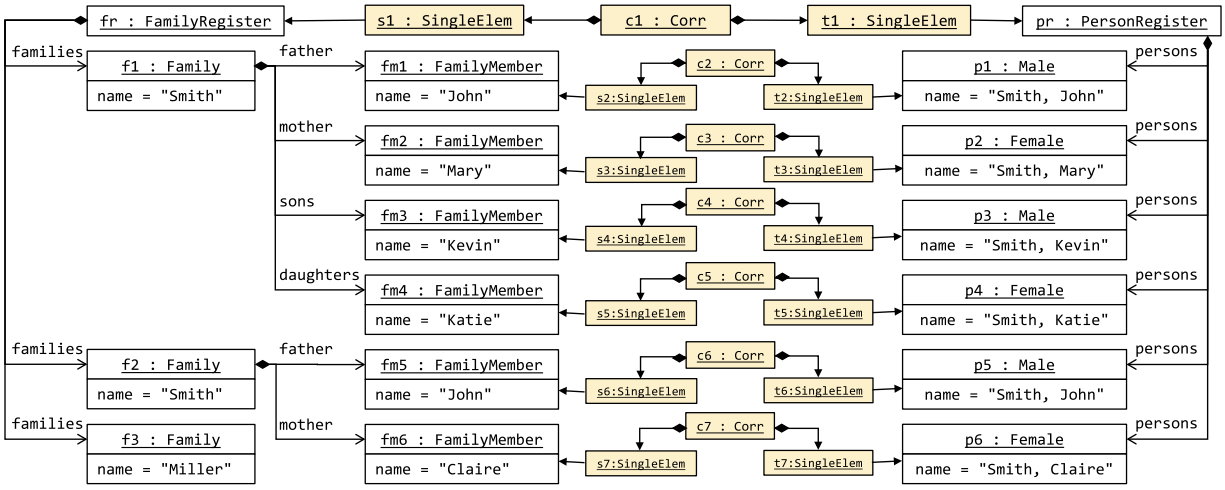

Fig. 6. Example models and correspondence model.

Listing 8 shows the BXtendDSL code for the F2P transformation. On this declarative level, the overall transformation is defined only partially. The transformation definition employs several language constructs which were not needed in the P2P case (unidirectional feature mappings, creation and filter modifiers).

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 are used only at 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 containment 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: filter (line 13) ensures that only mothers and daughters are considered by this rule. creation (line 14) is required for a special case: After a move in the families model that results in a change of gender, the corresponding object in the persons model will be deleted and recreated; on recreation, the birthday has to be copied from the person to be deleted. Furthermore, the rule contains three unidirectional feature mappings for calculating the name of a person and the link to its container, as well as the name of a member. The rule Member2Male works analogously.

From modifiers and unidirectional mappings, hook methods are generated that have to be implemented at the src-once layer. Listing 9 displays the code for modifiers in Member2Female. filterMember checks the gender (line 6) and builds a birthday map (lines 3–5) that is required in onFemaleCreation to copy the birthday in the case of a gender switch. Regarding the code for feature mappings, the reader is referred to Bank et al. (2021).

5.2. Bags to bags

The *Bags to Bags* case (B2B), proposed in Westfechtel (2018a), involves a $1 : n$ mapping, where a single object is mapped to a collection of objects of statically unknown cardinality. B2B involves a *folding* in forward direction and an *unfolding* in backward direction. The case is solved with the help of a group modifier.

Two bags have to be synchronized. Each element of a bag is represented by an object with a value attribute in both metamodels. In the first representation, each occurrence of some value is represented by one object, while in the second representation there is only one object for each value, carrying a multiplicity attribute.

Fig. 7 shows an object diagram for a simple example. Both models have a single root object which holds containment links to the elements of the respective bag. The correspondence model makes use of *multi-elements* that reference elements in the source model. Multi-elements result from the application of a group modifier.

Listing 10 depicts the declarative part of our BXtendDSL solution for this transformation case. The rule Bag2Bag is similar to the rule Database2Database of Listing 1. The rule Element2Element maps a group s to a single object t. The filter modifier is required to exclude empty groups (which are allowed in general: a pattern may match even with an empty group). The feature mapping in line 7 relates the group s to the attributes value and multiplicity of t.

The required imperative code is shown in Listing 11. Elements are grouped by their values (lines 1–3). Empty groups are filtered (3–6). value and multiplicity are calculated from the value of any element and the size of the collection, respectively (8–10). Finally, the method sFrom (12–17) incrementally performs the unfolding transformation with the help of an updater supplied by the framework code (see Bank (2019) for details).

5.3. Expression trees to directed acyclic graphs

This case (Westfechtel, 2018a), called Ast2Dag, involves a transformation between an *abstract syntax tree* and a *directed acyclic graph*, both representing an expression being composed of binary arithmetic operators and variables and numbers as operands. Folding and unfolding are much more complex than for B2B since they require restructurings. Furthermore, matches have to be sorted, using the sort modifier.

Fig. 8 shows an object diagram for a sample expression, including a tree, a dag, and a correspondence model in between. Please note the $n : 1$ mapping of variable a. Change propagation is complex in both directions. For example, if variable v2 is renamed to b in Fig. 8, the dag has to be restructured into a tree. Conversely, if v3 is renamed to b (again starting from the state displayed in Fig. 8), renaming has to be applied to all copies in the tree.

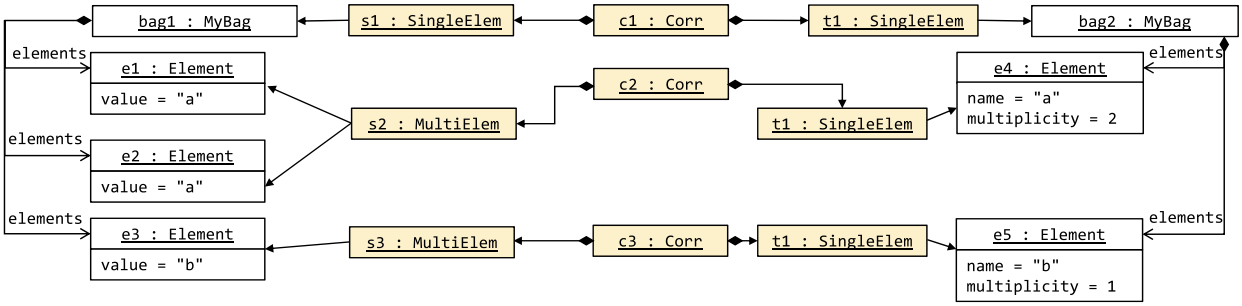

Fig. 7. Object diagram for a sample bag ({"*a*", "*a*", "*b*"}).

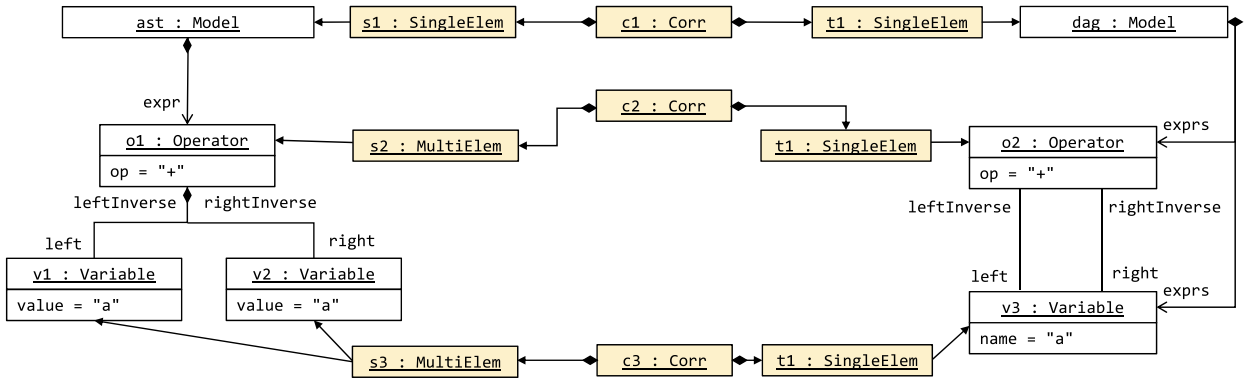

Fig. 8. Object diagram for a sample expression (a + a).

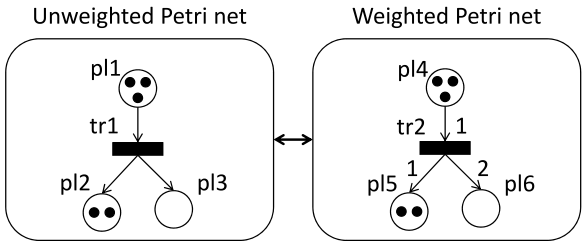

Fig. 9. Unweighted and weighted Petri nets (concrete syntax).

Listing 12 shows the BXtendDSL Declarative code for this transformation. Each of the rules transforming operators, variables, and numbers requires a group modifier for grouping equal subexpressions, and a filter modifier to exclude empty groups. Furthermore, operators have to be processed top-down, which is achieved with the help of sort modifiers on both sides of the rule Operator2Operator. In this way, it is guaranteed that each link to a parent in a feature mapping is accessed only after the parent has been processed. Finally, imperative code is required for all feature mappings in all rules.

Altogether, quite a number of hook methods have to be implemented at the imperative layer (Bank, 2019). Due to the lack of space, we present only the methods for sorting operators (Listing 13). The sorting algorithm itself is part of the framework code and need not be implemented by the transformation developer, who has to supply the required sorting predicates (Listing 13) only.

The methods compareSource and compareTarget are used to sort elements of source and target models, respectively. In the tree, for each node the textual representation of the respective subexpression is computed as a key. If the key of some node includes the key of another node, the former is a parent of the latter and precedes it in the sorting order. This ensures that the dag is constructed top-down. In the dag, we may simply compare the parent links to sort the matches.

5.4. Petri nets to weighted Petri nets

The *Petri Nets to Weighted Petri nets* case (Pn2Pnw) (Westfechtel, 2018a) serves as an example of a *link-to-object mapping*. Furthermore, it includes a rule with *multiple object matchers*.

When a transition is fired in an unweighted Petri net, one token from each input place is removed and one token is added to each output place. Contrastingly, in a weighted Petri net, edge weights determine the number of tokens which are produced and consumed. The forward transformation adds the default weight 1 to each transition, while the backward transformation simply discards the weights (Fig. 9).

Fig. 10 displays the same Petri nets in abstract syntax on the left-hand side and on the right-hand side, respectively (name attributes are omitted from the figure). In an unweighted Petri net, each edge is modeled by *link*. In a weighted Petri net, an *object* (with adjacent links) is required for storing the attribute weight, whose default value is 1.

Listing 14 shows the solution to this case in BXtendDSL Declarative; we refrain from explaining the complementing imperative code (which is very concise, too). The rules Place2Place and Net2Net define 1 : 1 mappings and are compiled to fully executable imperative code. In contrast, Transition2Transition defines a 1 : *n mapping*: A transition in an unweighted Petri Net is mapped to a transition in a weighted Petri Net as well as sets of edge objects for incoming and outgoing edges.

Imperative code is required for the modifiers in lines 13–14 and for the feature mappings in lines 16–17, which realize *link-to-object mappings*. The group modifiers group edge objects by their adjacent transition. The filter modifiers check whether grouped edges are adjacent to the current transition. The hook methods for link-to-object mappings create an object for each link and vice versa.

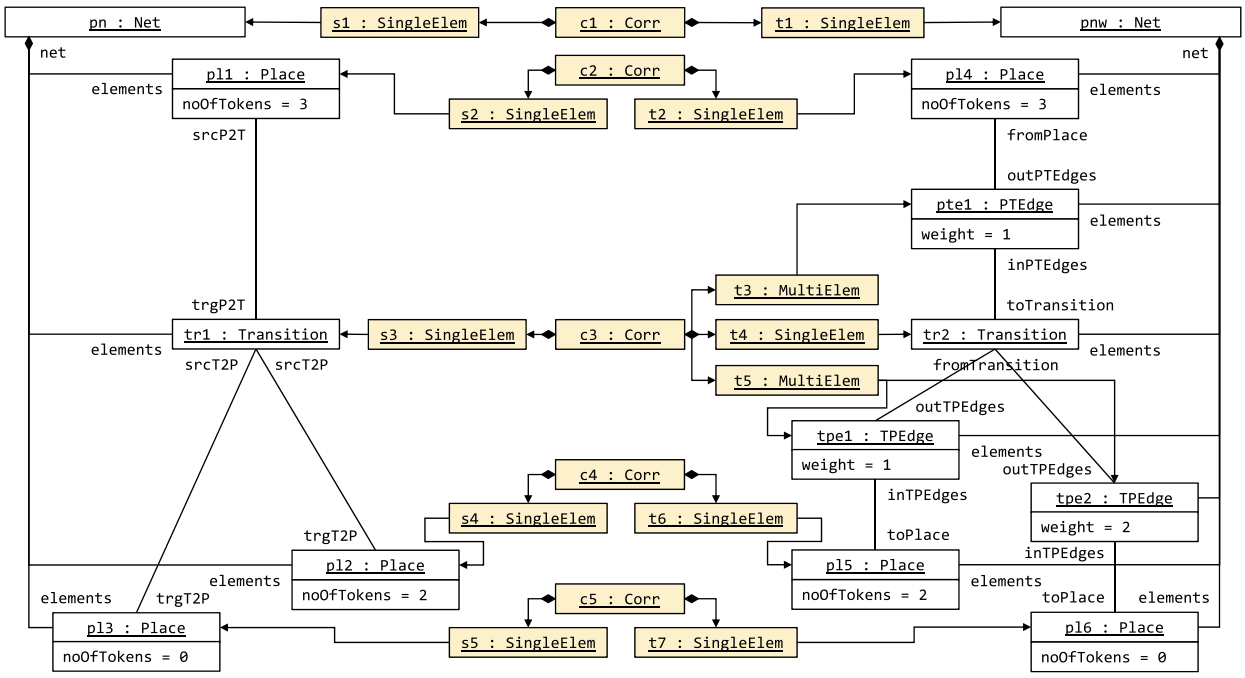

Fig. 10. Example models and correspondence models (Petri nets).

6. Evaluation

We conducted a quantitative evaluation referring to all cases presented in this paper. Below, we describe the investigated research questions, the conducted experiment, the obtained results, answers to the research questions, and threats to validity.

6.1. Research questions

Our evaluation addresses three research questions that are introduced below.

Question 1 (*Conciseness*)**.** *How concise are transformation definitions written in BXtendDSL?*

Thus, we are interested in the size of transformation definitions required to solve bidirectional transformation problems. In this way, we intend to measure the effort to be invested by transformation developers.

Question 2 (*Expressiveness*)**.** *To what extent is it possible to solve bidirectional transformation problems in BXtendDSL?*

This question refers to the correctness of BXtendDSL solutions with regard to the requirements of the respective transformation cases. Expressiveness is measured in terms of passed test cases.

Question 3 (*Scalability*)**.** *Are BXtendDSL transformations scalable to large model sizes?*

By means of this research question, we would like to investigate whether the performance of BXtendDSL suffices to apply transformations to large models of practical relevance. To answer this question, we perform measurements of the execution time.

6.2. Experiment

Our evaluation is based on the Benchmarx infrastructure, which was presented first in Anjorin et al. (2017b). Benchmarx has been designed as a general framework for implementing bidirectional transformation cases in heterogeneous tools living in different technological spaces. The infrastructure acknowledges the existence of a spectrum of tool architectures (batch, incremental, state-, correspondence-, and change-based).

On top of the Benchmarx infrastructure, a benchmark for the Families to Persons (F2P) case was designed and implemented. This benchmark was accepted for the Transformation Tool Contest (TTC) 2017 (Anjorin et al., 2017a), where it was implemented in a number of transformation tools. Furthermore, the F2P case was also presented in depth in a journal paper (Anjorin et al., 2020). In this paper, we reuse the F2P benchmark, which to date is the only benchmark that has been implemented in a fairly large number of bidirectional transformation tools.

We compare the BXtendDSL solution of Section 5.1 against the BXtend solution published in Anjorin et al. (2020), and a solution (Westfechtel and Buchmann, 2019) in *medini QVT* (ikv++ technologies, 2017), a tool that is based on *QVT-R* (Object Management Group, 2016), but maintains a persistent correspondence model (see Section 7.2 for a brief description). Thus, our evaluation compares tools of different categories: A *declarative* tool (medini QVT), an *imperative* tool (BXtend), and a *hybrid tool* (BXtendDSL), combining a declarative with an imperative DSL. We also present some aggregate data drawn from Anjorin et al. (2020) to allow a comparison against the other F2P solutions.

To reduce the potential bias of a single transformation case, we extend our evaluation to the other cases presented in this paper, employing the same set of tools as for the F2P case. All of these cases and their QVT-R solutions were presented in Westfechtel (2018a), which considered only batch transformations and did not explore the incremental behavior of the presented solutions.

The following sections on conciseness, expressiveness, and scalability refer to the F2P case only. Section 6.6 gives a brief summary of the remaining cases.

6.3. Conciseness

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

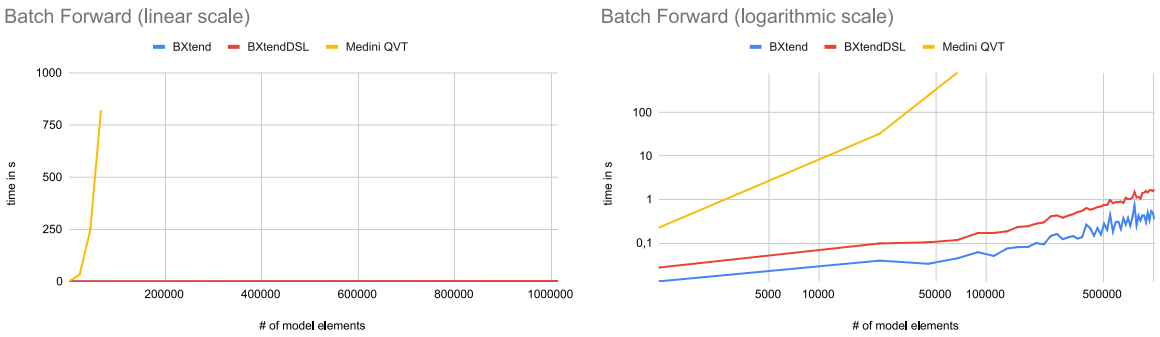

Fig. 11. Forward batch transformation: Linear/linear scale and logarithmic/logarithmic scale.

Table 1
Sizes of transformation definitions.

	BXtendDSL	BXtend	Medini QVT
LOC	89	211	320
# words	276	565	956
# characters	3030	7571	9740

specification effort. Since all compared solutions are based on textual languages, we measure the size of transformation definitions by counting the *numbers* of *lines of code*, *words* (character strings separated by whitespace), and *characters* (excluding empty lines, comments, and generated code lines). Table 1 depicts the values obtained for those metrics for BXtendDSL, BXtend, and medini QVT. In the case of BXtendDSL, both the code in the declarative DSL as well as the supplementary, handwritten imperative code was counted.

In the case of BXtendDSL and BXtend, 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 the size metrics.

Although in medini QVT the transformation definition consists of a single set of bidirectional rules, the solution is much larger than even the BXtend solution, in which both transformation directions have to be specified explicitly and separately. Although the medini QVT solution is bidirectional, it does contain a large number of conditional expressions that depend on the transformation direction and blow up the specification considerably.

With respect to conciseness, BXtendDSL also beats all solutions compared in Anjorin et al. (2020). Table 2 briefly summarizes the respective results (showing only the lines of code metric). For a more detailed description the reader is referred to Anjorin et al. (2020).

6.4. Expressiveness

The test suite supplied with the benchmark contains different test cases for each transformation direction, as well as test cases carried out in batch and incremental mode of operation. In total, the F2P benchmark comprises 34 test cases (Table 4).

The BXtend as well as the BXtendDSL solution successfully pass all test cases — in forward and backward direction as well as in batch and in incremental mode. In the case of BXtend, the transformation is defined completely in Xtend; thus, the full flexibility of an imperative programming language may be exploited. Notably, the blend of declarative and imperative code in BXtendDSL does not affect expressiveness negatively. In both tools, the solutions may be programmed precisely according to the requirements of the benchmark.

The medini QVT solution successfully passes all test cases in batch mode, but fails in most of the incremental test cases. In contrast to BXtend and BXtendDSL, the transformation developer has to "buy" the built-in incremental behavior and cannot adjust it in such a way that the requirements are satisfied (Westfechtel, 2018b; Westfechtel and Buchmann, 2019).

When comparing the results with other tools performing the same benchmark (Anjorin et al., 2020), it becomes evident that BXtend and also BXtendDSL are the only tools which do not fail for any of the provided test cases. A summary of the numbers given in Anjorin et al. (2020) is shown in Table 3.

6.5. Scalability

In order to evaluate efficiency and scalability with respect to increasing model size, two experiments were conducted: (1) batch transformations in forward and backward direction, and (2) incremental transformations in forward and backward direction. 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 is measured after large models have been constructed (using the same range of model sizes as for batch transformations). In each direction, a single member/person is inserted, resulting in a corresponding change in the opposite model.

The tests were performed on the same machine and in isolation for each solution. 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 Figs. 11–14. All figures are 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 of 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.

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 bidirectional transformation 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

Table 2
Sizes of transformation definitions for all solutions participating in the F2P benchmark (Anjorin et al., 2020) (LOC metric only).

	BXtendDSL	BXtend	Medini QVT	BiGUL	eMoflon	EVL + Strace	JTL	NMF	SDMLib
Lines of code	89	211	320	176	217	1299	227	279	236

Table 3
Aggregate test results for all solutions participating in the F2P benchmark (Anjorin et al., 2020).

Result	BXtendDSL	BXtend	Medini QVT	BiGUL	eMoflon	EVL + Strace	JTL	NMF	SDMLib
Pass	34	34	22	25	28	26	24	31	31
Fail	0	0	12	9	6	8	10	3	3

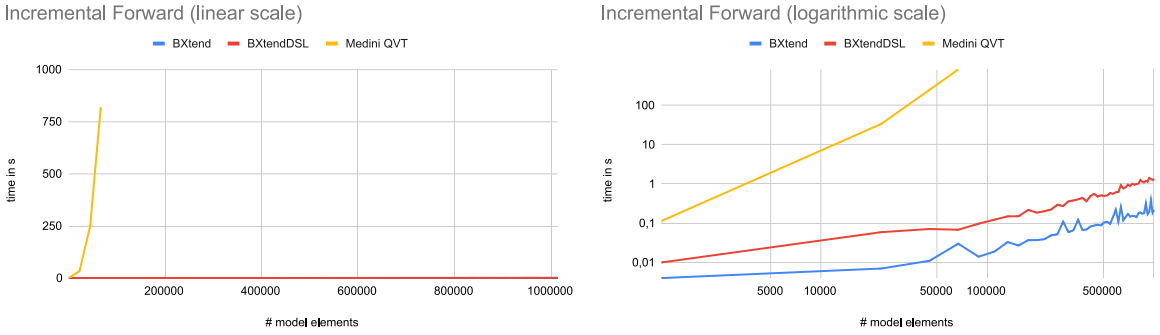

Fig. 12. Forward incremental transformation: Linear/linear scale and logarithmic/logarithmic scale.

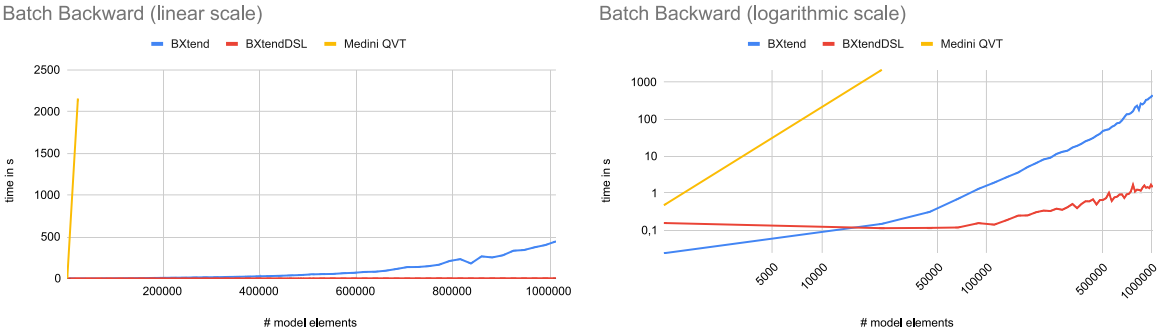

Fig. 13. Backward batch transformation: Linear/linear scale and logarithmic/logarithmic scale.

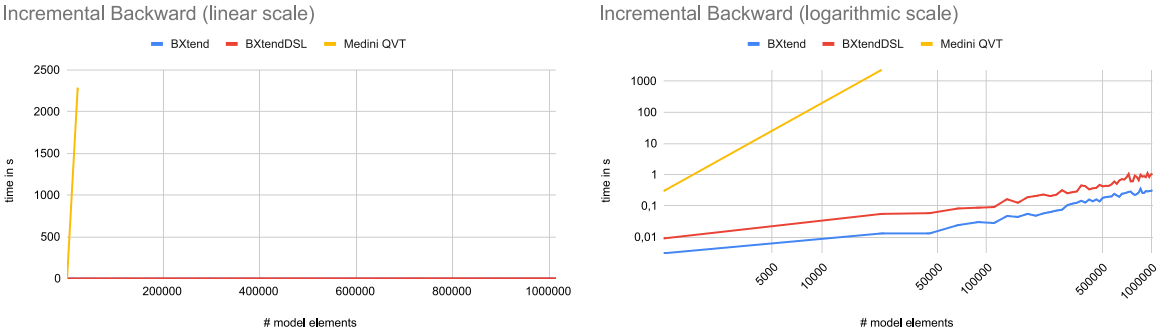

Fig. 14. Backward incremental transformation: Linear/linear scale and logarithmic/logarithmic scale.

matching families is expensive and results in a sharp increase of runtime. In contrast to the BXtend solution, only a single iteration over the families collection has to be performed in the BXtendDSL solution, resulting in an almost linear runtime. For the sake of comparability and traceability, we did not modify the original BXtend solution, since it was one of the reference solutions submitted to the TTC 2017 (Anjorin et al., 2017a) and also the one that was discussed in Anjorin et al. (2020).

For the medini QVT solution, the tests had to be stopped at even small model sizes due to exponential growth of computation times. The plots clearly indicate that the medini QVT solution is practically unusable for even modest model sizes of approximately 50.000 elements.

We limited ourselves to provide plots only for BXtendDSL, BXtend and Medini QVT in this paper, as a detailed comparison of BXtend with seven other tools for the F2P case may also be

Table 4
Aggregate test results, grouped into categories and classified as passes/fails.

Category	Result	BXtendDSL	BXtend	Medini
Batch	Pass	7	7	7
FWD	Fail	0	0	0
Batch	Pass	11	11	11
BWD	Fail	0	0	0
Incr.	Pass	8	8	4
FWD	Fail	0	0	4
Incr.	Pass	8	8	0
BWD	Fail	0	0	8
Total	Pass	34	34	22
	Fail	0	0	12

Table 5
Lines of code of the transformation definitions for other transformation cases.

Case	BXtendDSL	BXtend	Medini QVT
P2P	34	59	56
B2B	31	90	84
Ast2Dag	120	282	353
Pn2Pnw	34	102	78

Table 6
Passed test cases/all test cases for other transformation cases.

Case	BXtendDSL	BXtend	Medini QVT
P2P	13/13	13/13	13/13
B2B	14/14	14/14	7/14
Ast2Dag	18/18	18/18	16/18
Pn2Pnw	17/17	17/17	15/17

found in Anjorin et al. (2020). Some tools were omitted from several plots presented in Anjorin et al. (2020) due to excessive execution times. All other tools proved to have a reasonable scalability in terms of runtime behavior for increasing model sizes. With respect to incremental changes, only a change-based tool (NMF) performed significantly more efficiently than BXtend and BXtendDSL, both of which are state-based and thus have to reconstruct changes before applying them.

6.6. Other transformation cases

We performed analogous experiments for the other transformation cases presented in this paper, again using the Benchmarx framework. These supplementary transformation cases have been implemented only in BXtendDSL, BXtend, and medini QVT so far (i.e., the same tools that we compared for the F2P benchmark). The data on the size of transformation definitions and the passed test cases are summarized in Tables 5 and 6, respectively. These data confirm the results observed for the F2P benchmark. This applies to performance measurements, as well, which are not reported here to save space.

Table 5 shows a significant reduction in size for the BXtendDSL solution compared to the other solutions in all cases. Furthermore, the sizes of the solutions in BXtend and medini QVT are roughly comparable. Remarkably, each of the medini QVT solutions for B2B and Ast2Dag required two unidirectional transformations; we did not manage to synthesize a single bidirectional transformation (Westfechtel, 2018a).

Table 6 shows 100% success rates for both BXtendDSL and BXtend. For B2B, the backward transformation in medini QVT is not executable at all, presumably due to a bug in medini QVT. For Ast2Dag and Pn2Pnw, a few incremental test cases fail, confirming the observation from F2P that incremental behavior cannot be adapted as required.

6.7. Answers to research questions

In the following, we summarize our answers to the research questions stated in Section 6.1, based on the evaluation of the benchmarks presented above.

Answer 1 (*Conciseness*)**.** *The use of BXtendDSL results in significant savings with respect to the size of transformation definitions.*

This observation applies to all cases studied in this paper. Compared to BXtend, the declarative layer of BXtendDSL increases conciseness significantly. Even a fully declarative language, as implemented in medini QVT, performs worse than BXtendDSL with respect to conciseness.

Answer 2 (*Expressiveness*)**.** *Both BXtendDSL and BXtend achieve full functional correctness in all benchmarks.*

Functional correctness is not affected negatively by the layered architecture of BXtendDSL. Both BXtendDSL and BXtend benefit from the flexibility obtained by relying (partially in the case of BXtendDSL) on an imperative language. The solutions in both tools are the only ones passing all tests for all cases.

Answer 3 (*Scalability*)**.** *BXtendDSL transformations are scalable with acceptable performance.*

Our measurements have shown that both BXtend and BXtendDSL are scalable to large model sizes. In contrast, medini QVT and some of the tools compared in the F2P benchmark (Anjorin et al., 2020) do not scale at all. Compared to BXtend, performance of BXtendDSL is slightly degraded, but to an acceptable degree.

6.8. Threats to validity

Concerning conciseness, we are well aware of the fact that lines of code is an old but controversial metric. For measuring the size of the solutions, we used a simple tool which counts the numbers of handwritten lines of code, words, and characters for each solution. These numbers have to be taken with a grain of salt, as they refer to transformation definitions written in different languages and they are sensitive to layout conventions as well as programming practices. However, the results are consistent over all test cases studied in this paper.

Concerning expressiveness, we followed our experiences gained when developing a test suite for the F2P benchmark (Anjorin et al., 2017a, 2020) and provided similar test suites for the other transformation examples presented in this paper. Since these test suites provide for feature coverage with respect to the feature model for classifying test cases and model coverage with respect to the types of model elements defined in the respective metamodels, they suffice for benchmarking purposes, but not for detecting each and every error.

Concerning scalability, we have interpreted runtime results with care. However, it is obvious that the majority of declarative solutions presented in Anjorin et al. (2020) and the medini QVT solution do not scale at all, while BXtend and BXtendDSL roughly exhibit linear performance.

7. Related work

To discuss related work, we introduce a taxonomy which we use to classify and describe other approaches to bidirectional transformations. Subsequently, we compare BXtendDSL against these approaches and conclude with a summary of distinctive features.

Table 7
Classification of bidirectional transformation approaches.

	BiGUL	Medini QVT	JTL	eMoflon	NMF	EVL + Strace	BXtend	BXtendDSL
1	Asymmetric	Symmetric	Symmetric	Symmetric	Symmetric	Symmetric	Symmetric	Symmetric
2	State-based	State-based	State-based	Change-based	Change-based	State-based	State-based	State-based
3	Declarative	Declarative	Declarative	Declarative	Imperative	Both	Imperative	Both
4	Put-get get-put	None	Correctness hippocraticness	Correctness (completeness)	(Correctness) (hippocraticness)	None	None	(Correctness) (hippocraticness)

7.1. Classification

A wide variety of approaches to bidirectional transformations have been developed (Czarnecki et al., 2009; Abou-Saleh et al., 2016; Stevens, 2007). Different taxonomies for classifying these approaches have been proposed (Hidaka et al., 2016; Diskin et al., 2016). In the context of this paper, the following dimensions are relevant:

1. In *asymmetric* approaches (Diskin et al., 2011a), one model plays the role of the *source*, and the other model constitutes a *view* that is derived from the source. In *symmetric* approaches (Diskin et al., 2011b), both models are considered as *peers* that have to be maintained mutually consistent.
2. *State-based* approaches (Stevens, 2010) operate on model states, while *change-based* approaches (Diskin et al., 2010) operate on deltas, i.e., sequences of change operations.
3. *Declarative* approaches (Object Management Group, 2016) are based on declarative specifications of a bidirectional transformation, while *imperative* approaches (Buchmann, 2018) explicitly specify the steps to be executed in an imperative language.
4. Finally, approaches may be classified with respect to the *bidirectional transformation laws* (Abou-Saleh et al., 2016) that they guarantee to hold.

7.2. Other approaches

While quite a number of theoretical approaches to bidirectional transformations have been developed, the landscape of tools is populated sparsely. Since our contribution primarily focuses on tool support, we confine the comparison to bidirectional transformation tools, disregarding pure theoretical work. To this end, we selected the tools used in the evaluation of the Families to Persons benchmark (Section 6, Tables 2 and 3). Since this selection includes a wide spectrum of approaches, it is well suited for a comprehensive comparison.

Table 7 classifies the selected tools with respect to the taxonomy presented above. In row 4 of the table, parentheses indicate that the respective law holds only under certain conditions.

Among the compared tools, *BiGUL* (Hu and Ko, 2016; Ko et al., 2016) is the only tool that is based on an asymmetric approach. Asymmetric approaches have been developed to solve the well-known *view-update problem*, as it occurs e.g. in databases. These approaches address a number of laws, the most important ones being the *get-put* and *put-get* laws: Assuming a set S of source models, a set V of view models, a function $get : S \rightarrow V$ that constructs a view, and a function $put : S \times V \rightarrow S$ that updates the source according to updates of the view, $put(s, get(s)) = s$ and $get(put(s, v)) = v$ should hold. In BiGUL, these laws are guaranteed to hold: Each BiGUL program is composed of *combinators* (Foster et al., 2007; Terwilliger, 2011), each of which corresponds to a transformation step and is invertible. BiGUL supports *putback-based* programming, i.e., the programmer provides the put transformation, from which the get transformation is derived automatically. To this end, BiGUL provides a rich set of primitives and combinators on tree-based data structures.

In *medini QVT* (ikv++ technologies, 2017), the transformation developer may specify bidirectional transformations using the language QVT-R (Object Management Group, 2016). QVT-R follows a *constraint-based approach*: A transformation is defined in terms of a *consistency relation*, from which the update behavior for consistency restoration is derived automatically. In QVT-R, consistency is defined in terms of a set of relations between source and target patterns. OCL (OMG, 2014) is used as expression language. Pre- and postconditions may be defined by when and where clauses, which may include calls to other relations. While medini QVT implements the official QVT-R standard at the syntactic level, there are some semantic deviations. E.g., medini QVT uses a trace model for a state-based propagation of changes, while the QVT-R standard defines a trace-based semantics. No guarantees can be given for correctness and hippocraticness if the language is used in an unconstrained way (Westfechtel, 2018a).

JTL (Cicchetti et al., 2010) is a model transformation language with dedicated support for bidirectional transformations and non-determinism. When using JTL, the transformation developer provides a set of constraints specifying a consistency relation in a declarative way. These constraints and the involved metamodels are transformed into an Answer Set Programming (ASP) (Gelfond and Lifschitz, 1988) problem, which an ASP solver can use to enable consistency restoration. Syntactically, JTL is based on (a subset of) QVT-R, but its semantics deviates significantly. Like medini QVT, JTL operates on a persistent trace which does not require the exact delta which was applied to one of the involved models. By generating all possible models which may be obtained by selecting one of multiple rules being applicable to a given match, JTL takes non-determinism into account. The desired result has to be selected by the user from all those candidates. JTL ensures correctness with respect to the specified set of constraints, and hippocraticness as the current state of all models can always be taken as a solution if it already fulfills all constraints.

eMoflon (Anjorin et al., 2012; Weidmann et al., 2019) is based on the concept of triple graph grammars (Schürr, 1994). Source and target models are considered as graphs with a correspondence graph between both models keeping track of relationships between source and target elements. By providing a triple graph grammar (TGG), consistency is specified in a highly declarative way. Triple rules describe a synchronous extension of the whole triple graph. From synchronous triple rules, directed rules are automatically derived. eMoflon operates in a propagation-based style, which means the tool expects old versions of the graphs, as well as a *structural delta* between the old and new versions of the graph that needs to be propagated. After a directed synchronization, a triple graph is obtained that is guaranteed to be a member of the language generated by the TGG. As a consequence, eMoflon guarantees correctness. Under certain conditions, completeness is guaranteed, as well. For each structural delta which results in a source graph for which a consistent triple exists, changes can be propagated successfully to the correspondence graph and the target graph.

NMF (Hinkel and Burger, 2019) uses C# as a host language to realize a bidirectional transformation language as an internal DSL. When programming a bidirectional transformation with NMF, the developer specifies a set of consistency relations between

elements of source and target models. Relations are coupled in so called *synchronization blocks*, which can be viewed as pairs of updates (and queries). NMF uses operational deltas as an input for the transformation. In Hinkel and Burger (2019) a proof for correctness and hippocraticness under the restriction, that the transformation developer ensures that get-put and put-get laws are actually satisfied for all intra-model lenses, is given. While in simple cases a put operation may automatically be derived from the get operation, an explicit specification of put satisfying the round-trip laws must be provided by the transformation developer in more complex cases.

The tool *EVL+Strace* (Samimi-Dehkordi et al., 2018) is based on the Epsilon framework (Kolovos et al., 2018), which provides tool support for a variety of DSLs for model transformation. For a bidirectional transformation, a trace metamodel has to be specified first. A trace model conforming to this metamodel contains copies of all relevant elements of source and target models, as well as links connecting these elements. Subsequently, a pair of synchronizers has to be specified, one for each direction. To this end, EVL+Strace employs two languages: The declarative Epsilon Validation Language (EVL) is used to detect inconsistencies between the changed model and the trace model. For each constraint violation, repair actions are programmed in the imperative Epsilon Object Language (EOL). In general, no adherence to any formal property can be guaranteed, as the transformation developer is free to implement consistency checks and repairs for both directions.

BXtend (Buchmann, 2018), the precursor of BXtendDSL, has already been described in Section 3.1. BXtend is a state-based tool; each transformation direction has to be programmed explicitly. No guarantees can be given with respect to roundtrip properties.

7.3. Comparison to other approaches

BXtendDSL supports a symmetric approach, considering source and target models as peers that have to be kept consistent. Based on model states, transformations are performed incrementally to restore consistency. A declarative external DSL is combined with an imperative internal DSL. Bidirectional transformations are correct and hippocratic under the constraints specified in Section 4.4.2.

In contrast to BXtendDSL, *BiGUL* has been designed for the asymmetric case (view-update problems). While BXtendDSL deals with graph-structured models, transformations in BiGUL operate on trees. In BiGUL, all transformation definitions need to be composed from primitives and combinators that satisfy roundtrip laws. BXtendDSL combines declarative with imperative code. Handwritten code provides the flexibility to solve a large variety of bidirectional transformation problems, but requires testing of roundtrip properties.

Both *medini QVT* and *JTL* are based syntactically on QVT-R. In both tools, transformations are specified declaratively by consistency relations. BXtendDSL provides a declarative language, as well, which, however, is combined with an imperative language. It is the imperative code which provides the expressive power required to overcome the limitations of the declarative language. Furthermore, while both medini QVT and JTL are based on constraint solvers, BXtendDSL provides a scalable execution engine implemented in Xtend.

In *eMoflon*, bidirectional transformations are specified declaratively by synchronous triple graph rules. Incremental change propagation is obtained for free. In contrast, BXtendDSL relies on *triple graph transformation systems* rather than triple graph grammars. While BXtendDSL may ensure roundtrip properties only under restrictive constraints, the combination of imperative with declarative code improves expressiveness. In contrast, eMoflon guarantees correctness, but assumes that (1) source and target models may be specified by its graph grammar language, and (2) the respective rules may be combined 1 : 1 into triple rules. If these assumptions do not hold, a solution cannot be provided.

While BXtendDSL transformations are state-based and are performed on demand only, *NMF* is change-based and has been built for live synchronization: Each change is propagated immediately to the opposite model. While BXtendDSL combines a declarative external DSL with an imperative internal DSL, NMF relies on a single internal DSL, employing C# as a host language. Like BXtendDSL, NMF gains its expressiveness through imperative code, the use of which requires testing of roundtrip properties.

Like BXtendDSL, *EVL+Strace* combines a declarative with an imperative language. However, in EVL+Strace these languages are not linked through code generation. Rather, the declarative language is used to specify constraint violations, which are repaired in the imperative language. In contrast to BXtendDSL, both transformation directions are specified completely separately, resulting in a significant amount of redundant code (Table 2).

Compared to *BXtend*, BXtendDSL adds a whole new layer comprising a newly developed declarative language on top of the imperative framework. Now, 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 seamless integration of BXtend code generated from the BXtendDSL specification and handwritten imperative extensions to certain parts of the transformation rules. By employing the code generation approach, roundtrip properties are established for those parts of the declarative transformation specification satisfying well-behavedness conditions. Furthermore, the underlying BXtend framework has been revised significantly, providing a correspondence model which now allows also for $m : n$ correspondences, and a whole new internal DSL for manipulating the correspondence model.

7.4. Distinctive features

BXtendDSL is optimized towards *expressiveness*: Exploiting the flexibility of the imperative internal DSL, the transformation definition may be tailored exactly to the requirements imposed by the respective transformation case. This claim is supported by 100% success rates for all benchmarks reported in this paper.

In contrast, purely declarative tools such as BiGUL, eMoflon, and JTL derive the operational behavior of a bidirectional transformation from a high level specification and ensure that transformation definitions satisfy certain bidirectional transformation laws by construction. But either the transformation developer succeeds in solving the transformation case at hand, or (s)he has to look for another approach. We call approaches of this kind *closed* because they do not offer any extension mechanism to surmount the restrictions of the respective transformation definition language.

Furthermore, BXtendDSL is also optimized towards *conciseness*. In all benchmarks reported in this paper, we managed to provide the far shortest solution in BXtendDSL. Compared to the next best solution for the Families to Persons benchmark (in terms of conciseness) reported in this paper and in Anjorin et al. (2020), BXtendDSL needs about the half number of lines of code and thus offers a reduction by 50% (while at the same time the solutions in BXtendDSL and BXtend are the only ones offering a 100% success rate).

To the best of our knowledge, BXtendDSL is unique among the bidirectional transformation approaches with respect to the combination of an *external declarative DSL* and an *internal imperative DSL*, which are linked by code generation: The declarative DSL is compiled into the imperative DSL; the generated code is supplemented with handwritten bodies for hook methods.

8. Conclusion

In this paper, we presented *BXtendDSL*, a layered framework for bidirectional model transformations combining a declarative and an imperative language linked by code generation. BXtendDSL satisfies the requirements stated in Section 2.3 as follows:

1. The evaluation has shown its *conciseness*, resulting in the smallest sizes of transformation definitions compared with other languages and tools (Section 6.3).
2. BXtendDSL follows a hybrid approach to the definition of bidirectional incremental transformations: The *declarative DSL* provides for a relational specification and is complemented with an *imperative DSL* (Section 4).
3. *Roundtrip properties* are ensured for those parts of a transformation definition that satisfy well-behavedness constraints (Section 4.4).
4. BXtendDSL is optimized towards *expressiveness*, which is achieved with the help of the imperative DSL: By combining the declarative with an imperative DSL, a wide spectrum of transformation problems may be solved, resulting in 100% success rates for all benchmarks reported in this paper (Section 6.4).
5. Finally, executions of BXtendDSL transformation definitions are *scalable*, i.e., transformation definitions may be executed efficiently on large models (Section 6.5).

Future work will be performed in various directions. First, the transformation cases we have studied so far already cover a number of challenges such as $m : n$ correspondences, mapping of objects to links, foldings and unfoldings, heterogeneous metamodels, loss of information, configurability of transformations, renamings and moves, or application-specific requirements to change operations. Nevertheless, we are continuing our application-oriented work by studying other transformation cases that may reveal additional challenges. Furthermore, we are planning to address industrially relevant transformation cases involving large metamodels and large model instances to prove the feasibility of using BXtendDSL for practically relevant problems such as e.g. roundtrip engineering UML class models and Java source code, which we have already implemented in other bidirectional transformation tools (Buchmann and Westfechtel, 2016; Buchmann and Greiner, 2016; Greiner et al., 2016).

So far, we have evaluated BXtendDSL with respect to conciseness, measured by lines of code, expressiveness, measured by the number of passed test cases, and scalability, measured in execution time. While these are important aspects of usability, ease of use and understandability, as perceived by developers, are highly relevant, as well. While these aspects are more difficult to evaluate quantitatively, valuable feedback may be gained from qualitative studies carried out with practitioners, ideally on real or pilot projects in industry.

Finally, BXtendDSL Declarative has been designed as a small and light-weight declarative language which may be complemented by writing imperative code. We have designed this language incrementally, adding language constructs on demand to solve the transformation problems at hand. The work on this language is still ongoing, i.e., we are still extending the declarative language incrementally to shift the borderline between the declarative and the imperative layer of bidirectional transformation specifications. Our goal is to make the declarative language more expressive such that larger parts of the overall bidirectional transformation may be covered. At the same time, however, the language should stay small, lightweight and declarative, saving the effort of developing a full-fledged programming language capable of addressing all computational details (that may be expressed conveniently in Xtend anyway).

CRediT authorship contribution statement

Thomas Buchmann: Writing, Concept, Evaluation, Project administration. **Matthias Bank:** Implementation, Evaluation. **Bernhard Westfechtel:** Writing, Discussion, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The constructive comments of the unknown reviewers are gratefully acknowledged.

Resources

The Eclipse Update Site for BXtendDSL may be found at: http://btn1x4.inf.uni-bayreuth.de/bxtenddsl/update/.

A sample workspace comprising Benchmarx and the examples discussed in this paper is provided as a zip archive via http://btn1x4.inf.uni-bayreuth.de/bxtenddsl/examples/workspace.zip.

Please make sure you have installed BXtendDSL in your Eclipse and also the Benchmarx connector for BXtendDSL available at: http://btn1x4.inf.uni-bayreuth.de/bxtenddsl/connector/.

References

Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., Stevens, P., 2016. Introduction to bidirectional transformations. In: Gibbons, J., Stevens, P. (Eds.), Bidirectional Transformations - International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures. In: Lecture Notes in Computer Science, vol. 9715, Springer, pp. 1–28. http://dx.doi.org/10.1007/978-3-319-79108-1_1.

Anjorin, A., Buchmann, T., Westfechtel, B., 2017a. The families to persons case. In: García-Domínguez, A., Hinkel, G., Krikava, F. (Eds.), 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. In: CEUR Workshop Proceedings, vol. 2026, CEUR-WS.org, pp. 27–34.

Anjorin, A., Buchmann, T., Westfechtel, B., Diskin, Z., Ko, H., Eramo, R., Hinkel, G., Samimi-Dehkordi, L., Zündorf, A., 2020. Benchmarking bidirectional transformations: Theory, implementation, application, and assessment. Softw. Syst. Model. 19 (3), 647–691. http://dx.doi.org/10.1007/s10270-019-00752-x.

Anjorin, A., Diskin, Z., Jouault, F., Ko, H., Leblebici, E., Westfechtel, B., 2017b. BenchmarX reloaded: A practical benchmark framework for bidirectional transformations. In: Eramo, R., Johnson, M. (Eds.), Proceedings of the 6th International Workshop on Bidirectional Transformations Co-Located with the European Joint Conferences on Theory and Practice of Software, BX@ETAPS 2017, Uppsala, Sweden, April 29, 2017. In: CEUR Workshop Proceedings, vol. 1827, CEUR-WS.org, pp. 15–30, URL http://ceur-ws.org/Vol-1827/paper6.pdf.

Anjorin, A., Lauder, M., 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., Tolvanen, J. (Eds.), Joint Proceedings of the Co-Located Events at the 8th European Conference on Modelling Foundations and Applications (ECMFA 2012). Technical University of Denmark (DTU), Copenhagen, Denmark, ISBN: 978-87-643-1014-6, p. 348.

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

Bank, M., Buchmann, T., Westfechtel, B., 2021. Combining a declarative language and an imperative language for bidirectional incremental model transformations. In: Hammoudi, S., Pires, L.F., Seidewitz, E., Soley, R. (Eds.), Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2021, Online Streaming, February 8-10, 2021. SciTePress, pp. 15–27. http://dx.doi.org/10.5220/0010188200150027.

Bank, M., Kaske, S., Buchmann, T., Westfechtel, B., 2020. Incremental bidirectional transformations: Evaluating declarative and imperative approaches using the AST2Dag benchmark. In: Ali, R., Kaindl, H., Maciaszek, L.A. (Eds.), Proceedings of the 15th International Conference on Evaluation of Novel Approaches To Software Engineering, ENASE 2020, Prague, Czech Republic, May 5-6, 2020. SciTePress, pp. 249–260. http://dx.doi.org/10.5220/0009206602490260.

Bettini, L., 2016. Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing, Birmingham, UK.

Buchmann, T., 2018. BXtend - A framework for (bidirectional) incremental model transformations. In: Hammoudi, S., Pires, L.F., Selic, B. (Eds.), Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018.. SciTePress, pp. 336–345. http://dx.doi.org/10.5220/0006563503360345.

Buchmann, T., Dotor, A., Westfechtel, B., 2009. Triple graph grammars or triple graph transformation systems? In: Chaudron, M.R. (Ed.), Models in Software Engineering, Workshops and Symposia at MODELS 2008. In: Lecture Notes in Computer Science, vol. 5421, Springer, pp. 138–150.

Buchmann, T., 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., Cabello, E. (Eds.), Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 2: ICSOFT-PT, Lisbon, Portugal, July 24 - 26, 2016.. SciTePress, pp. 27–38. http://dx.doi.org/10.5220/0005957100270038.

Buchmann, T., Westfechtel, B., 2016. Using triple graph grammars to realize incremental round-trip engineering. IET Softw. 10 (6), 173–181, URL http://digital-library.theiet.org/content/journals/10.1049/iet-sen.2015.0125.

Cheney, J., Gibbons, J., McKinna, J., Stevens, P., 2017. On principles of least change and least surprise for bidirectional transformations. J. Object Technol. 16 (1), 3:1–31. http://dx.doi.org/10.5381/jot.2017.16.1.a3.

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

Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F., 2009. Bidirectional transformations: A cross-discipline perspective. In: Paige, R.F. (Ed.), Proceedings of the Second International Conference on Theory and Practice of Model Transformations (ICMT 2009). In: Lecture Notes in Computer Science, vol. 5563, Springer, Zurich, Switzerland, pp. 260–283.

Czarnecki, K., Helsen, S., 2006. Feature-based survey of model transformation approaches. IBM Syst. J. 45 (3), 621–646. http://dx.doi.org/10.1147/sj.453.0621.

Diskin, Z., Gholizadeh, H., Wider, A., Czarnecki, K., 2016. A three-dimensional taxonomy for bidirectional model synchronization. J. Syst. Softw. 111, 298–322. http://dx.doi.org/10.1016/j.jss.2015.06.003.

Diskin, Z., Xiong, Y., Czarnecki, K., 2010. From state- to delta-based bidirectional model transformations. In: Tratt, L., Gogolla, M. (Eds.), Theory and Practice of Model Transformations - 3rd International Conference, ICMT@TOOLS 2010, MáLaga, Spain, June 28-July 2, 2010. Proceedings. In: Lecture Notes in Computer Science, vol. 6142, Springer, pp. 61–76. http://dx.doi.org/10.1007/978-3-642-13688-7_5.

Diskin, Z., Xiong, Y., Czarnecki, K., 2011a. From state- to delta-based bidirectional model transformations: The asymmetric case. J. Object Technol. 10, 6: 1–25. http://dx.doi.org/10.5381/jot.2011.10.1.a6.

Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F., 2011b. From state- to delta-based bidirectional model transformations: The symmetric case. In: Whittle, J., Clark, T., Kühne, T. (Eds.), Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings. In: Lecture Notes in Computer Science, vol. 6981, Springer, pp. 304–318. http://dx.doi.org/10.1007/978-3-642-24485-8_22.

Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A., 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. 29 (3), 17:1–17:65.

Fowler, M., 2011. Domain-Specific Languages. In: The Addison-Wesley signature series, Addison-Wesley, Boston, MA, URL http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0321712943,00.html.

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

Greiner, S., Buchmann, T., 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., Desfray, P. (Eds.), MODELSWARD 2016 - Proceedings of the 4rd International Conference on Model-Driven Engineering and Software Development, Rome, Italy, 19-21 February, 2016. SciTePress, pp. 15–27. http://dx.doi.org/10.5220/0005644700150027.

Hidaka, S., Tisi, M., Cabot, J., Hu, Z., 2016. Feature-based classification of bidirectional transformation approaches. Softw. Syst. Model. 15 (3), 907–928.

Hinkel, G., Burger, E., 2019. Change propagation and bidirectionality in internal transformation DSLs. Softw. Syst. Model. 18 (1), 249–278.

Hu, Z., Ko, H., 2016. Principles and practice of bidirectional programming in BiGUL. In: Gibbons, J., Stevens, P. (Eds.), Bidirectional Transformations - International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures. In: Lecture Notes in Computer Science, vol. 9715, Springer, pp. 100–150. http://dx.doi.org/10.1007/978-3-319-79108-1_4.

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., 2008. ATL: A model transformation tool. Sci. Comput. Program. 72, 31–39, Special Issue on Experimental Software and Toolkits (EST).

Ko, H., Zan, T., 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. pp. 61–72. http://dx.doi.org/10.1145/2847538.2847544.

Kolovos, D., Rose, L., Paige, R., Garcia-Dominguez, A., 2018. The Epsilon Book. http://www.eclipse.org/epsilon.

Object Management Group, 2016. Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification Version 1.3, formal/2016-06-03 ed. Needham, MA.

OMG, 2014. Object Constraint Language, formal/2014-02-03 ed. OMG, Needham, MA.

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

Samimi-Dehkordi, L., Zamani, B., Rahimi, S.K., 2018. EVL+Strace: A novel bidirectional model transformation approach. Inf. Softw. Technol. 100, 47–72. http://dx.doi.org/10.1016/j.infsof.2018.03.011.

Schürr, A., 1994. Specification of graph translators with triple graph grammars. In: Tinhofer, G. (Ed.), Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 1994). In: Lecture Notes in Computer Science, vol. 903, Springer, Herrsching, Germany, pp. 151–163.

Steinberg, D., Budinsky, F., Paternostro, M., Merks, E., 2009. EMF Eclipse Modeling Framework, second ed. In: The Eclipse Series, Addison-Wesley, Boston, MA.

Stevens, P., 2007. A landscape of bidirectional model transformations. In: Lämmel, R., Visser, J., Saraiva, J. (Eds.), Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers. In: Lecture Notes in Computer Science, vol. 5235, Springer, pp. 408–424. http://dx.doi.org/10.1007/978-3-540-88643-3_10.

Stevens, P., 2010. Bidirectional model transformations in QVT: Semantic issues and open questions. Softw. Syst. Model. 9 (1), 7–20.

ikv++ technologies, 2017. Medini QVT. http://projects.ikv.de/qvt.

Terwilliger, J.F., 2011. Bidirectional by necessity: Data persistence and adaptability for evolving application development. In: Lämmel, R., Saraiva, J., Visser, J. (Eds.), Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers. In: Lecture Notes in Computer Science, vol. 7680, Springer, pp. 219–270. http://dx.doi.org/10.1007/978-3-642-35992-7_6.

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

Weidmann, N., Anjorin, A., Fritsche, L., Varró, G., Schürr, A., Leblebici, E., 2019. Incremental bidirectional model transformation with eMoflon::IBeX. In: Cheney, J., Ko, H. (Eds.), Proceedings of the 8th International Workshop on Bidirectional Transformations Co-Located with the Philadelphia Logic Week, Bx@PLW 2019, Philadelphia, PA, USA, June 4, 2019. In: CEUR Workshop Proceedings, vol. 2355, CEUR-WS.org, pp. 45–55, URL http://ceur-ws.org/Vol-2355/paper4.pdf.

Westfechtel, B., 2018a. Case-based exploration of bidirectional transformations in QVT relations. Softw. Syst. Model. 17 (3), 989–1029. http://dx.doi.org/10.1007/s10270-016-0527-z.

Westfechtel, B., 2018b. Incremental bidirectional transformations: Applying QVT relations to the families to persons benchmark. In: Damiani, E., Spanoudakis, G., Maciaszek, L.A. (Eds.), Proceedings of the 13th International Conference on Evaluation of Novel Approaches To Software Engineering, ENASE 2018, Funchal, Madeira, Portugal, March 23-24, 2018. SciTePress, pp. 39–53. http://dx.doi.org/10.5220/0006679700390053.

Westfechtel, B., Buchmann, T., 2019. Incremental bidirectional transformations: Comparing declarative and procedural approaches using the families to persons benchmark. In: Damiani, E., Spanoudakis, G., Maciaszek, L.A. (Eds.), Evaluation of Novel Approaches To Software Engineering. ENASE 2018. In: Communications in Computer and Information Science, Springer, pp. 98–118. http://dx.doi.org/10.1007/978-3-030-22559-9.

Thomas Buchmann received his diploma degree in mathematics from the University of Bayreuth in 2001. For the following three years he was employed as the manager of the software engineering department of a medium-sized local company. He obtained his doctoral as well as his habilitation degree

(all in computer science) from the University of Bayreuth in 2010 and 2017, respectively. His research interests include model transformations, model-driven engineering, software product line engineering, domain-specific languages, and software architecture.

Matthias Bank received his Master degree in Computer Science from the University of Bayreuth in 2019. He is now working as a software engineer.

Bernhard Westfechtel received his diploma degree from University of Erlangen–Nuremberg in 1983 and his doctoral as well as his habilitation degree (all in computer science) from RWTH Aachen University in 1991 and 1999, respectively. Since 2004, he has been a full professor of computer science (in software engineering) at University of Bayreuth. His research interests include graph transformations, model-driven engineering, software product line engineering, software configuration management, software process modeling, software architecture, and re-engineering.