Putback-Based Bidirectional Model Transformations[*]

Xiao He
School of Computer and Communication Engineering,
University of Science and Technology Beijing
Beijing, China
hexiao@ustb.edu.cn

Zhenjiang Hu
National Informatics Institute/
The University of Tokyo
Tokyo, Japan
hu@nii.ac.jp

ABSTRACT

Bidirectional model transformation (BX) plays a vital role in Model-Driven Engineering. A major challenge in conventional *relational* and *bidirectionalization*-based BX approaches is the ambiguity issue, i.e., the backward transformation may not be uniquely determined by the consistency relation or the forward transformation. A promising solution to the ambiguity issue is to adopt *putback*-based bidirectional programming, which realizes a BX by specifying the backward transformation. However, existing *putback*-based approaches do not support multiple conversions of the same node (namely a shared node). Since a model is a graph, shared nodes are very common and inevitable. Consequently, existing *putback*-based approaches cannot be directly applied to bidirectional model transformation. This paper proposes a novel approach to BX. We define a new model-merging-based BX combinator, which can combine two BXs owning shared nodes into a well behaved composite BX. Afterwards, we propose a *putback*-based BX language XMU to address the ambiguity issue, which is built on the model-merging-based BX combinator. We present the formal semantics of XMU which can be proven well behaved. Finally, a tool support is also introduced to illustrate the usefulness of our approach.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; *Specification languages*;

KEYWORDS

bidirectional transformation, model, ambiguity, shared node

ACM Reference Format:
Xiao He and Zhenjiang Hu. 2018. Putback-Based Bidirectional Model Transformations. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3236024.3236070

[*]This work was partially supported by the National Key Research and Development Program of China (Nos. 2017YFB0202303, 2017YFB0202300), by the Japan Society for the Promotion of Science (JSPS) Grant-in-Aid for Scientific Research (S) No. 17H06099, and by the National Natural Science Foundation of China (Nos. 2015CB352200, 61620106007, 61472180, 61300009).

1 INTRODUCTION

In Model-Driven Engineering (MDE), developers employ various models to capture different views of the system under development. Those models are probably interrelated, and may evolve independently. When one is updated, the changes should be propagated to the related models. How to keep these models synchronized is known to be a crucial issue in a round-trip development.

It has been argued for a long time that bidirectional transformation (BX) [4, 31] could provide a software foundation for model synchronization. A BX program is a single program but can be viewed as a pair of forward and backward transformations, namely *get* and *put*, respectively, where *get* creates a view model from a source model, and *put* converts a source model into an updated source model according to a view model. We have seen good progress in BX and its application to model synchronization, particularly in the theoretical foundation [5, 7], the languages and/or the algorithms [3, 8, 12, 15, 21–23, 33], and the applications [13, 30, 34]. There are basically two approaches to writing BX. One is to ask users to write a declarative consistency relation (e.g., QVT relations [1]) between models, from which a suitable BX (forward and backward transformations) is derived; And the other is to ask users to write a forward transformation in a traditional unidirectional language (e.g., ATL [17]) or a domain specific language (e.g., lens [11]), from which a backward transformation is derived.

Despite the promising features of BX for model synchronization, there is a big issue that prevents it from being used in practice. As argued in [31], the existing bidirectional model transformation languages have inherited ambiguity in their semantics, and they never provide any effective way to remove the ambiguity to gain full control of the synchronization behavior. To be concrete, consider the classic bidirectional model transformation between UML class diagrams to Relational Database, namely `UML2RDBMS`[1]. We may ask the user to write a forward transformation in ATL from which a backward transformation is derived, or a more general flexible consistency relation in QVT from which both forward and backward transformations are derived. The ATL rule and the QVT relation are presented in Figure 1. In forward transformation, a table is created for each non-abstract class. But what about the backward transformation? There are many possibilities that cause ambiguity. For instance, if we delete a table and attempt to propagate the table deletion back to the class model, we may either delete the class that corresponds to the deleted table, or keep it by changing the class into an abstract one. The existing approaches automatically return one as the result. However, if it is not the one we wish to have for the backward transformation behavior, we have no way to specify our intention (in ATL or in QVT).

[1]The `UML2RDBMS` transformation has been implemented using QVT [1] and ATL (http://www.eclipse.org/atl/atlTransformations/#Class2Relational).

```
// ATL rule                    // QVT-R rule
rule ClassToTable {            relation ClassToTable {
 from c:uml!Class(              cn:String;
   c.isAbstract=false)          when { PackageToSchema(p,s) }
 to t:rdbms!Table{              domain uml p:Package{
   t.name <- c.name               classes=c:Class{name=cn,isAbstract=false}}
 }                              domain rdbms s:Schema{tables=t:Table{name=cn}}
 do {/* inner updates */}       where {/* inner updates */}
}                              }
```

Figure 1: ATL and QVT realization of ClassToTable

Fortunately, we have an important fact about BX that is little known: while there are generally many possible backward transformations for a forward transformation, there is at most one forward transformation for a backward transformation without ambiguity. In other words, the essence of bidirectional transformation is nothing but (*putback*) backward transformation [10, 19, 20]. This *putback*-based BX approach has been applied to build useful bidirectional transformations between tree-like data [25, 36].

Inspired by the success of the *putback*-based BX on tree-like data, we want to go further to see whether we can extend it to bidirectional model transformation, where models are basically graphs that may contain shared nodes (and cycles). As a matter of fact, the shared nodes introduce a new challenge. Consider the following two *putback*-based backward transformation rules:

- *R1* converts $\begin{tikzcd} B \arrow[r] & A \end{tikzcd}$ into $\begin{tikzcd} B \arrow[r, dotted] & D \end{tikzcd}$ given view node *X*, and
- *R2* converts $\begin{tikzcd} A & C \arrow[l] \end{tikzcd}$ into $\begin{tikzcd} D & C \arrow[l, dotted] \end{tikzcd}$ given view node *Y*.

When the original source model is $\begin{tikzcd} B \arrow[r] & A & C \arrow[l] \end{tikzcd}$ and the view model consists of *X* and *Y*, we expect to convert the original source and the view model into $\begin{tikzcd} B \arrow[r, dotted] & D & C \arrow[l, dotted] \end{tikzcd}$, where the shared node *A* in the source model is converted twice by *R1* and *R2*. It could not work if we would apply the two rules in a trivial sequential manner, say one after another. If *R1* is applied first, then the intermediate output is $\begin{tikzcd} B \arrow[r, dotted] & D & C \end{tikzcd}$, which cannot trigger *R2*. Similarly, if *R2* is applied first, then the intermediate output is $\begin{tikzcd} B & D & C \arrow[l, dotted] \end{tikzcd}$, which rejects *R1*. To resolve this problem, we carefully investigated the behavior of multiple rule applications on graphs, and found that if we could merge $\begin{tikzcd} B \arrow[r, dotted] & D & C \end{tikzcd}$ and $\begin{tikzcd} B & D & C \arrow[l, dotted] \end{tikzcd}$ well then we could get the expected result. Based on this observation, we propose to treat multiple conversions of a shared node as independent BXs (in each BX, this node is converted with an index/label), and then merge their outputs.

In this paper, we propose a novel bidirectional model transformation language XMU that can effectively handle both the ambiguity issue and the shared node issue. XMU is a *putback*-based BX language, enabling developers to define a BX over models in the form of a backward transformation, and automatically deriving the unique forward transformation from the backward transformation (and thus avoiding the ambiguity issue). In addition, XMU is established on a new model-merging-based BX combinator (rather than the existing parallel and sequential combinators [25]), supporting multiple conversions of the same node. The main contributions of this paper are twofolds: 1) a model-merging-based BX combinator to address the shared node issue; and 2) XMU, a *putback*-based BX language for ambiguity-free model synchronization. The preliminary results showed the practical feasibility of our approach.

The reminder of this paper is organized as follows: Section 2 introduces the background and the related work of this paper; Section 3 proposes the model-merging-based BX combinator; Section 4 proposes XMU, a bidirectional model transformation language; Section 5 presents the formal semantics of XMU; Section 6 introduces the tool support; The last section concludes the paper and briefly discusses the future work.

2 BACKGROUND AND RELATED WORK

2.1 Background

A bidirectional transformation (BX) can be viewed as a pair of a forward transformation *get* and a backward transformation *put* between two models.

The forward and backward transformations (*get* and *put*) can be defined symmetrically [7, 31] or asymmetrically [25]. *This paper mainly focuses on asymmetric BXs* because in theory, a symmetric BX can be composed of two asymmetric BXs. Without loss of generality, *get* and *put* can be defined as follows:

$$get \quad : \quad \mathbb{S} \rightarrow \mathbb{V} \tag{1}$$

$$put \quad : \quad \mathbb{S} \rightarrow \mathbb{V} \rightarrow \mathbb{S} \tag{2}$$

where *get* reads a source model of type (metamodel) $\mathbb{S}$, and creates a view model of type (metamodel) $\mathbb{V}$; and *put* yields an updated-source model by taking the original source and an updated view model as inputs.

A BX is well behaved if it satisfies the following round-trip properties:

$$get\ \mathrm{s} \neq \bot \implies put\ \mathrm{s}\ (get\ \mathrm{s}) = \mathrm{s} \tag{GetPut}$$

$$put\ \mathrm{s}\ \mathrm{v} \neq \bot \implies get\ (put\ \mathrm{s}\ \mathrm{v}) = \mathrm{v} \tag{PutGet}$$

$$put\ \mathrm{s}\ \mathrm{v} \neq \bot \implies put\ (put\ \mathrm{s}\ \mathrm{v})\ \mathrm{v} = put\ \mathrm{s}\ \mathrm{v} \tag{PutTwice}$$

where $\bot$ denotes a runtime error or an undefined value in this paper. The GetPut property says that if *get* succeeds, then no update on the view leads to no update on the source, the PutGet property says that if *put* succeeds, then update on the view should be fully put back to the source such that the updated view can be recovered from the updated source, and the last PutTwice property, which is derivable from the GetPut and PutGet properties, says that putting back twice has the same effect of putting back once.

One important fact that is used in this paper, as formally proven in [10], is that if *put* is well defined, then there is exactly one *get* that can be paired with the *put* to form a well-behaved BX. We say *put* is well defined, iff. (1) $\forall$s, v(put (put s v) v) = put s v, (2) *put* s is injective on view type, and (3) uncurry *put* is surjective on source type. This is the foundation of the putback-based bidirectional programming and the key to ambiguity issues.

2.2 Related Work

Diskin et al. [5, 7] proposed an algebraic framework of model synchronization. Their work focused on the theoretical aspect of model synchronization (e.g., the correctness properties).

Giese et al. [12] proposed an incremental model synchronization approach based on Triple Graph Grammar (TGG). Lauder et al. [22] also proposed TGG-based incremental BX approach based on the node precedence analysis. Ehrig and Hermann et al. [8, 15] discussed the correctness of TGG-based BX approaches. Lamo et al.

[21] proposed a graph-grammar-based BX approach, which shares many ideas with TGG-based approaches. These approaches are typical relational BX approaches that cannot deal with ambiguity issues properly.

Xiong el al. [33] proposed an ATL-based model synchronization framework. They derive a backward transformation automatically from a forward ATL transformation. Their approach imposed some restrictions on the transformation specification and view updating, e.g., element deletion is not allowed in the *view* model, to avoid the ambiguity issue.

Solvers are intensively used to realize BXs. Macedo et al. [23] proposed an Alloy-based approach that turns QVT relations and ATL rules into BXs over models. The basic idea is to encode QVT relations and ATL rules into Alloy [16] constraints that can be solved by the Alloy solver. Their approach uses a customizable graph edit distance (GED) or an operation-based distance (OBD) to guide the solver to find possible solutions. Semeráth et al. [29] also proposed an Alloy-based backward change propagation approach. They derive a backward change propagation operation from a model query. Cicchetti et al. [3] proposed a relational BX language named JTL, and mapped JTL onto Answer Set Programming. In solver-based approaches, developers may have to iterate the result sets returned by the solver to obtain the required result. Besides, solver-based approaches usually do not scale.

Some technologies, such as those in [6, 9, 14], support *uncertainty* and *variability* in BXs. Those approaches can be adapted for dealing with the ambiguity issue by asking the user to pick a required result from a number of candidates.

There are also some research efforts in bidirectional transformation on tree-like data formats, such as XML. BiXid is a relation-based bidirectional XML transformation language [18]. However, BiXid cannot handle ambiguity and shared node issues. BiFluX is a putback-based bidirectional XML transformation language [24, 25]. BiGUL [20] is a general purpose *putback*-based BX language, which is a revision of the core of BiFluX. BiFluX and BiGUL enable us to specify *put*, i.e., how to update a source according to a view. Afterwards, a unique *get* can be computed automatically from *put*. BiFluX and BiGUL are free from the ambiguity issue, but they do not support multiple conversion of the same node in a graph. So they cannot handle shared nodes well.

In summary, the state-of-the-art approaches to BX over models cannot avoid the ambiguity issue. Existing ambiguity-free BX approaches are mainly designed for tree structures and do not handle shared graph nodes properly.

3 MODEL-MERGING-BASED BX COMBINATION

This section address the shared node issue in bidirectional model (graph) transformation as discussed in the introduction. The basic idea of our solution is to treat the multiple conversions (transformations) of the same node as independent component BXs, each of which converts this node only once. These component BXs are then combined by merging their outputs. In this way, a BX over models suffering from the shared node issue can be viewed as a combination of component BXs, which do not suffer from the shared node issue. This section proposes a model-merging-based BX combinator.

3.1 Metamodel, Model and Merging

We start by explaining metamodels, models and model merging used in this paper. Generally, a metamodel (written as $\mathbb{A}, \mathbb{B}, etc.$) describes a type of graphs, and a model (written as $A, B, etc.$) is viewed as a typed graph that confirms to a metamodel. Concretely, a model consists of objects (e.g., o_1, o_2) and links (e.g., l_1, l_2). We assume that *typeOf* is a function that can return the type of an object or a link (i.e., a class or a relationship defined in a metamodel).

In this paper, a model M is regarded as a typed graph with indices, in which each object is associated with a set of indices. $index_{\mathrm{M}}(o)$ returns the indices of o. $object_{\mathrm{M}}(i)$ returns the object in M that owns the index i. We assume that each index appears once, i.e., $\forall o_1, o_2 (o_1 \neq o_2 \Rightarrow index_{\mathrm{M}}(o_1) \cap index_{\mathrm{M}}(o_2) = \emptyset)$. We do not assume that indices are persistent attributes. They can be transient information that is maintained at runtime by BXs.

Given two models A and B, we say $\mathrm{A} \sqsubseteq \mathrm{B}$ if B contains all objects (including attribute values) and links in A, and we say $\mathrm{A} = \mathrm{B}$ iff $\mathrm{A} \sqsubseteq \mathrm{B} \wedge \mathrm{B} \sqsubseteq \mathrm{A}$.

We identify two types of model merging, denoted by $\uplus$ and $\triangleright$. One is the three-way model merging [32], $\mathrm{M}_1 \uplus_{\mathrm{M}_0} \mathrm{M}_2$, denoting merging M_1 and M_2 based on a common model M_0. The basic idea is to compute the delta (i.e., changes) δ_1 and δ_2 from M_0 to M_1 and M_2, respectively. Then, we compute the result by applying both δ_1 and δ_2 to M_0. Intuitively, three-way model merging will preserve all the changes from M_0 to M_1 and M_2 (i.e., $\mathrm{M}_1 \sqsubseteq \mathrm{M}_1 \uplus_{\mathrm{M}_0} \mathrm{M}_2 \wedge \mathrm{M}_2 \sqsubseteq \mathrm{M}_1 \uplus_{\mathrm{M}_0} \mathrm{M}_2$) when there is no conflict. It is not difficult to verify that $\mathrm{M} \uplus_{\mathrm{M}} \mathrm{M} = \mathrm{M}$ and $\mathrm{M} \uplus_{\emptyset} \mathrm{M} = \mathrm{M}$. To determine which objects in M_1 and M_2 should be merged and to calculate the delta, we use object indices to align models and objects. If $index_{\mathrm{A}}(o) \cap index_{\mathrm{B}}(o') \neq \emptyset$, then o and o' are aligned. We may define other alignment strategies (e.g., using key properties as indices) to extend $\uplus$. It is our future work to investigate other possibilities.

The three-way merging may fail when there is a conflict. For example, assuming that a link l ($l \equiv \langle o, o' \rangle$) and an object o satisfy $l \in \mathrm{M}_1 \wedge l \notin \mathrm{M}_0$ and $o \in \mathrm{M}_0 \wedge o \notin \mathrm{M}_2$, $\mathrm{M}_1 \uplus_{\mathrm{M}_0} \mathrm{M}_2$ fails (namely $\mathrm{M}_1 \uplus_{\mathrm{M}_0} \mathrm{M}_2 = \bot$) because o is required to be preserved by M_1 but is deleted in M_2.

The other type of model merging is the additive merging, $\mathrm{M}_1 \triangleright \mathrm{M}_0$, denoting merging M_1 into M_0. For every object $o \in \mathrm{M}_1$, if there exists $o_0 \in \mathrm{M}_0$ and $index_{\mathrm{M}_1}(o) \cap index_{\mathrm{M}_0}(o_0) \neq \emptyset$, then replace o_0 with o (and preserve all compatible attributes and links); otherwise, add o to the result. For each link $l \in \mathrm{M}_1$, merge/copy l to the result. Obviously, $\mathrm{M}_1 \sqsubseteq \mathrm{M}_1 \triangleright \mathrm{M}_0$.

3.2 A Model-Merging-based Combinator

Now we show how to lift the model-merging operation from models to BXs, which is the key step towards our put-based bidirectional model transformation.

As required in [10] for a well-defined *put*, *put* s v must be injective for all v, i.e., all information in the view should appear in the new updated source. In this paper, we relax this condition because we intend to allow a *put* to use part of the view to convert the source model and leave the unused information to other BXs. For any $put : \mathbb{S} \rightarrow \mathbb{V} \rightarrow \mathbb{S}$, we assume that there exists a function $core : \mathbb{S} \rightarrow \mathbb{V} \rightarrow \mathbb{V}$ to capture the partial view from the current source and view:

$$core\ \mathrm{s}\ \mathrm{v} \sqsubseteq \mathrm{v} \tag{3}$$

which should satisfy the following properties:

$$put\ \mathrm{s}\ \mathrm{v} = put\ \mathrm{s}\ (core\ \mathrm{s}\ \mathrm{v}) \tag{4}$$

$$core\ \mathrm{s}\ \mathrm{v} = core\ (put\ \mathrm{s}\ \mathrm{v})\ \mathrm{v} \tag{5}$$

Intuitively, *core* extracts the necessary view information needed by *put*. Note that such function *core* always exists because we can know from *put* what view information is needed.

We refine the original PutGet into PutGet* as follows:

$$put\ \mathrm{s}\ \mathrm{v} \neq \bot \implies get\ (put\ \mathrm{s}\ \mathrm{v}) \simeq core\ \mathrm{s}\ \mathrm{v} \tag{PutGet*}$$

where $\simeq$ denotes graph isomorphism. If a BX satisfies PutGet* law, it is not difficult to verify that the following condition also holds: $\forall \mathrm{v}_1, \mathrm{v}_2 (core\ \mathrm{s}\ \mathrm{v}_1 \neq core\ \mathrm{s}\ \mathrm{v}_2 \implies put\ \mathrm{s}\ \mathrm{v}_1 \neq put\ \mathrm{s}\ \mathrm{v}_2)$. This condition is the refined condition for injective *put*. We can also prove that under the new definitions and conditions, the uniqueness of *get*[2] for a well-behaved *put* still holds. In brief, due to the uncurry *put* is surjective on source and PutTwice, for any s, there must exist a certain v that makes *put* s v = s. Afterwards, we can define *get* s ≡ *core* s v, where v satisfies *put* s v = s (i.e., the existence). Furthermore, for any s, we choose v such that *put* s v = s. If there is another *get*' that can be paired with this *put*, due to PutGet*, $get'\ \mathrm{s} = get'\ (put\ \mathrm{s}\ \mathrm{v}) \simeq core\ \mathrm{s}\ \mathrm{v} \equiv get\ \mathrm{s}$ (i.e., the uniqueness).

We are now ready to define our model-merging-based BX combinator $\uplus$ to combine two BXs $bx_1 \equiv (get_1, put_1)$ and $bx_2 \equiv (get_2, put_2)$ into a composite BX $bx = bx_1 \uplus bx_2 \equiv (get, put)$ as follows:

$$get\ \mathrm{s} \quad = \quad get_1\ \mathrm{s} \uplus_{\emptyset} get_2\ \mathrm{s} \tag{6}$$

$$put\ \mathrm{s}\ \mathrm{v} \quad = \quad put_1\ \mathrm{s}\ \mathrm{v} \uplus_{\mathrm{s}} put_2\ \mathrm{s}\ \mathrm{v} \tag{7}$$

The core function of *bx* is defined as $core_1\ \mathrm{s}\ \mathrm{v} \uplus_{\emptyset} core_2\ \mathrm{s}\ \mathrm{v}$, where $core_1$ and $core_2$ are core functions of bx_1 and bx_2. Note that bx_1 and bx_2 can be successfully combined, only when the following equations hold for any s and v $(i = 1, 2)$:

$$put_i\ \mathrm{s}\ (get_i\ \mathrm{s}) = put_i\ \mathrm{s}\ (get_1\ \mathrm{s} \uplus_{\emptyset} get_2\ \mathrm{s}) \tag{8}$$

$$core_i\ (put_i\ \mathrm{s}\ \mathrm{v})\ \mathrm{v} = core_i\ (put_1\ \mathrm{s}\ \mathrm{v} \uplus_{\mathrm{s}} put_2\ \mathrm{s}\ \mathrm{v})\ \mathrm{v} \tag{9}$$

$$put_i\ (put_1\ \mathrm{s}\ \mathrm{v} \uplus_{\mathrm{s}} put_2\ \mathrm{s}\ \mathrm{v})\ \mathrm{v} = put_1\ \mathrm{s}\ \mathrm{v} \uplus_{\mathrm{s}} put_2\ \mathrm{s}\ \mathrm{v} \tag{10}$$

$$core_i\ \mathrm{s}\ \mathrm{v} = core_i\ \mathrm{s}\ (core_1\ \mathrm{s}\ \mathrm{v} \uplus_{\emptyset} core_2\ \mathrm{s}\ \mathrm{v}) \tag{11}$$

$$get_1\ (put_1\ \mathrm{s}\ \mathrm{v}) \uplus_{\emptyset} get_2\ (put_2\ \mathrm{s}\ \mathrm{v}) \simeq core_1\ \mathrm{s}\ \mathrm{v} \uplus_{\emptyset} core_2\ \mathrm{s}\ \mathrm{v} \tag{12}$$

Now, we can prove that a composite BX that is constructed by using $\uplus$ is also well behaved, i.e., Theorem 3.1.

Theorem 3.1 (Correctness of $\uplus$). *Given two well behaved BXs, namely bx_1 and bx_2, and their core functions, namely $core_1$ and $core_2$, when equations (8)-(12) hold, $bx = bx_1 \uplus bx_2$ is a well behaved BX, and the core function of bx, i.e., $core_1\ s\ v \uplus_{\emptyset} core_2\ s\ v$, satisfies equations (3)-(5).*

In brief, equations (9) and (11) ensure that the composite core function works properly, and equations (8)-(10), and (12) ensure that the composite BX is well behaved. Due to space limitation, the details of the proof are omitted in this paper and are presented at our project website[3].

[2] *get* is unique in terms of graph isomorphism.
[3] https://bitbucket.org/ustbmde/morel/wiki/Home

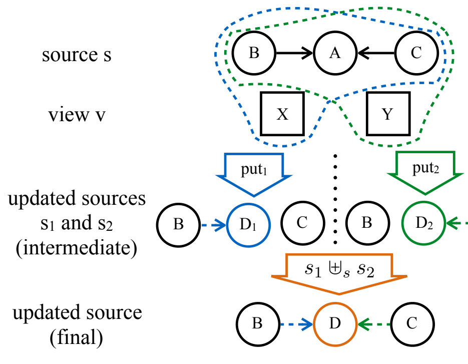

Figure 2: Example of merging-based combinator

Consider the shared node example in the Introduction, as shown in Figure 2. The original source must be converted by the two rules *R1* and *R2* (i.e., put_1 and put_2) simultaneously, as explained previously. After applying put_1 and put_2, we get two intermediate results, namely s_1 and s_2. s_1 and s_2 contain two *D*-elements (namely D_1 and D_2, respectively), which were created during put_1 and put_2. If we assign the same index to D_1 and D_2, then by using our model-merging-based BX combinator $\uplus$, we finally obtain the expected output, and the PutGet law holds.

4 XMU: A BIDIRECTIONAL MODEL TRANSFORMATION LANGUAGE

This section proposes a novel bidirectional model transformation language, namely XMU, which extends the putback BX approach with the unique model-merging-based BX combination.

XMU is a rule-based language. The main concrete syntax of XMU is listed in Figure 3. An XMU rule is defined as a sequence of formal parameters and a body statement. Each formal parameter is declared as a source/view/normal variable associated with a type (for a normal variable, its type can only be a primitive type). XMU statements, which are bidirectional, include `update-with-by`, `switch-case` and rule call statements. An `update-with-by` statement, which aligns the matches of source and view patterns first, has at most three clauses to indicate how to construct the updated source model according to the result of alignment. A `switch-case` statement consists of some branches. A branch condition can be a pattern or a boolean expression. Unidirectional statements (*uStmt*) of XMU include `enforce`, `delete` node/link and unidirectional `switch-case` statements. An `enforce` statement is responsible to construct the updated source model, while a `delete` statement is used to delete an object or a link from the updated source model. A unidirectional `switch-case` statement is similar to its bidirectional version, however its branch actions must be unidirectional statements. The definition of a model pattern (its syntax and meaning) in XMU is similar to that in QVT [1]. A pattern consists of a set of nodes and expressions. In the implementation, we extend the syntax presented in Figure 3 with some syntax sugar and variants to ease the BX specification. Figure 3 also omits some common language constructs that are shared by existing programming languages, such as basic arithmetic, relational and boolean expressions.

Take the classic conversion between classes and tables as an example. Figure 4 shows the XMU rule for this conversion, which corresponds to the ATL rule and QVT relation presented in Figure 1. This XMU rule can be read as follows: update each non-abstract class c within package p with a table t whose table name is identical to the class name within schema s; if c is paired with t, then perform the inner updates (i.e., the `match` clause); if c cannot be paired with any table, then turn c into an abstract class (i.e., the `unmatchs` clause); if t cannot be paired with any class, then create a new class c that is intended to be paired with t to continue the conversion (i.e., the `unmatchv` clause).

By comparing Figure 1 with Figure 4, we can easily discover the syntactic correspondence between XMU, ATL and QVT. Nevertheless, XMU is semantically different from ATL and QVT because an XMU rule specifies the *backward transformation*, rather than a forward transformation or a consistency relation between source and view. From the backward transformation, our approach is able to derive the unique forward transformation from source to view. For instance, in the forward direction, the behavior of the XMU rule *ClassToTable* is identical to that of the ATL and QVT rules in Figure 1: this XMU rule creates a table for each non-abstract class. In the forward semantics, the `update-with-by` statement can be roughly viewed as the `from-to` structure in ATL.

Elimination of ambiguity. A benefit of XMU is that it allows developers to explicitly specify the behavior of *put* to avoid the ambiguity issue. Compared with other BX technologies, XMU provides developers with better control over their BXs. For example, the XMU rule in Figure 4 can be substituted with either of the two rules in Figure 5. The first alternative rule will delete the unpaired class, rather than turning it into an abstract one. The second alternative rule will try to find and change an abstract class into a non-abstract one before creating a new class for an unpaired table. Note that forward transformations, which are derived from the initial XMU rule in Figure 4 and the two alternative rules in Figure 5, are identical to each other.

If we specify this rule as a mapping or a relation, we are unable to tell the transformation engine which backward semantics is required, and the ambiguity issue arises.

Addressing shared nodes. Consider a more complicated conversion between associations and foreign keys. We define an XMU

```
 ruleDef   ::=  rule ruleName ( fpars ){varDec stmt}
   fpars   ::=  fpars , fpars | source v:type | view v:type | [normal] v:type
  varDec   ::=  varDec varDec | v:type;
    stmt   ::=  update pat_S with pat_V by clause | switch(v){case}
            |   stmt where {index} | ruleName(apars) | stmt_1;stmt_2 | {stmt}
  clause   ::=  clause clause | match -> stmt | unmatchs -> uStmt
            |   unmatchv -> uStmt
    case   ::=  case case | case pat -> stmt | case boolexp -> stmt
   index   ::=  index,index | index(v_s, v_v)
   apars   ::=  apars,apars | expr
   uStmt   ::=  enforce pat_S | delete v | delete v.feature=expr
            |   switch(v){uCase} | uStmt_1;uStmt_2 | {uStmt}
   uCase   ::=  uCase uCase | case pat -> uStmt | case boolexp -> uStmt
     pat   ::=  patNode
 patNode   ::=  v:type{patExp}
  patExp   ::=  feature=patNode | feature=expr | patExp,patExp
    expr   ::=  constant | v
```

Figure 3: Concrete Syntax of XMU

```
// XMU rule
rule ClassToTable(source p:uml!Package, view s:rdbms!Schema) {
  cn:String;
  update p:uml!Package{classes=c:uml!Class{name=cn,isAbstract=false}}
    with s:rdbms!Schema{tables=t:rdbms!Table{name=cn}} by
      match -> {/* inner updates */}
      unmatchs -> enforce c:uml!Class{isAbstract=true}
      unmatchv -> enforce c:uml!Class{}
}
```

Figure 4: XMU Rule ClassToTable

rule *AssocToFKey* (as shown in Figure 6) to update each association between two classes with a foreign key between two tables. If an association cannot be paired with any foreign key, we must delete this association. If a foreign key cannot be paired with any association, we must create an association. Before creating a new association, we must check whether the related classes exist in the source model and create the absent class(es) if necessary to ensure the successful creation of this association.

The execution of an XMU program is based upon the model-merging-based BX combination. Informally, each XMU statement/rules (e.g., an `update-with` statement) is viewed as a primitive BX. A sequence of XMU statements/rules are combined by $\uplus$ defined in Section 3. However, when an object is converted multiple times, the shared issue may arise and the BX combination may fail.

Assume that we merge the XMU rules *ClassToTable* in Figure 4 and *AssocToFKey* in Figure 6 (i.e., $ClassToTable \uplus AssocToFKey$). Given the source model $m0$ and the view model as shown in Figure 7, we obtain two updated source models $m1$ and $m2$ by applying *ClassToTable* and *AssocToFKey*, respectively. Although a human can easily know that class $s3$ in $m1$ and class $s4$ in $m2$ are conceptually identical (they are both derived from table $t3$), the model merging operator $\uplus$ will treat them as two distinct objects because they have different indices. Consequently, $m1 \uplus_{m0} m2$ will contain three classes, and the combination of *ClassToTable* and *AssocToFKey* is invalid (PutGet* law will be violated).

```
// the first alternative rule
rule ClassToTable(source p:uml!Package, view s:rdbms!Schema) {
  cn:String;
  update p:uml!Package{classes=c:uml!Class{name=cn,isAbstract=false}}
    with s:rdbms!Schema{tables=t:rdbms!Table{name=cn}} by
      match -> {/* inner updates */}
     /* delete the unpaired class */
      unmatchs -> delete c
      unmatchv -> enforce c:uml!Class{}
}
// the second alternative rule
rule ClassToTable(source p:uml!Package, view s:rdbms!Schema) {
  cn:String;
  update p:uml!Package{classes=c:uml!Class{name=cn,isAbstract=false}}
    with s:rdbms!Schema{tables=t:rdbms!Table{name=cn}} by
      match -> {/* inner updates */}
      unmatchs -> enforce c:uml!Class{isAbstract=true}
      /* check whether there is an abstract class that can be turned
         into a non-abstract one before creating a new class */
      unmatchv -> switch(p) {
            case p:uml!Package{classes=c:uml!Class{name=cn, isAbstract=true}} ->
               enforce c:uml!Class{isAbstract=false}
            otherwise -> enforce c:uml!Class{}}
}
```

Figure 5: Alternative XMU rules to ClassToTable

```
rule AssocToFKey(source p:uml!Package, view s:rdbms!Schema) {
  sn, tn, an:String;
  update p:uml!Package{associations=assoc:uml!Association{name=an,
    source=sc:uml!Class{name=sn}, target=tc:uml!Class{name=tn}}}
   with s:rdbms!Schema{tables=st:rdbms!Table{name=sn,
    foreignKeys=f:ForeignKey{name=an,referTo=tt:rdbms!Table{name=tn}}}} by
    match -> {}
    unmatchs -> delete assoc
    unmatchv -> switch(p) {
                          case p:uml!Package{classes=sc:uml!Class{name=sn} -> {}
                          otherwise -> enforce sc:uml!Class{name=sn}
                        };
                        switch(p) {
                          case p:uml!Package{classes=tc:uml!Class{name=tn} -> {}
                          otherwise -> enforce tc:uml!Class{name=tn}
                        };
                        enforce assoc:uml!Association{name=an,
                          source=sc:uml!Class{}, target=tc:uml!Class{}}
}
```

Figure 6: Rule AssocToFKey

To ensure that ⊎ works properly, XMU provides an `index` function to compute object indices. For instance, during the backward transformation, `index(c,t)` computes an extra index *i* according to the indices of *c* and *t* and maps *i* onto *c* in the result. If the index of *c* is unused in the source model, its index is treated as `nil` during the computation. In forward transformation, `index(c,t)` computes and maps an index onto *t* according to the index of *c*.

Figure 8 shows the use of the `index` function. We add some `index` functions within `where` clauses appended to the `match` branches. When constructing the updated source models (from the same source and view in Figure 7), we compute an extra index for each class (the extra index computed by the `index` function is denoted as the concatenation of the function parameters). Class *s*3 in *m*1 and class *s*4 in *m*2 are mapped onto the same index, namely `nil-t3` (since neither *s*3 nor *s*4 exists in *m*0). In this way, $m1 \uplus_{m0} m2$ will contain only two classes, and $ClassToTable \uplus AssocToFKey$ is valid.

It is worthwhile to notice that rule *ClassToTable* and rule *AssocToFKey* cannot be applied sequentially. Otherwise, we will not get the expected result. As shown in Figure 9, assume that the original source model *m*0 contains two classes (namely *A* and *B*) that are connected by an association *R*; The view model contains table

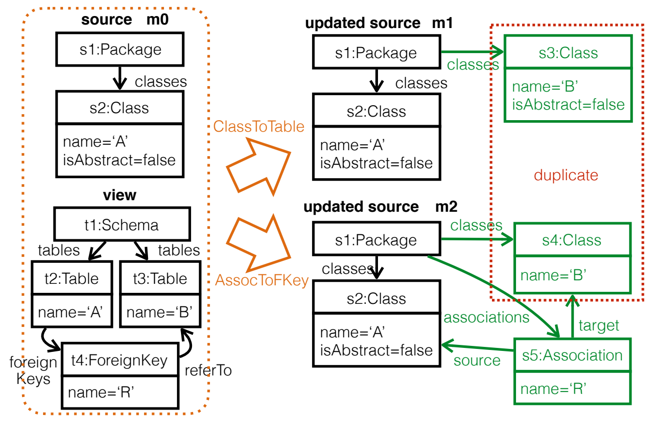

Figure 7: A shared node issue in UML2RDBMS

```
rule ClassToTable(source p:uml!Package,
    view s:rdbms!Schema) {
  cn:String;
  update p:uml!Package{
    classes=c:uml!Class{
      name=cn,isAbstract=false}}
   with s:rdbms!Schema{
    tables=t:rdbms!Table{name=cn}} by
     match -> {/* inner updates */}
      where {index(c,t)}
     ...
}
```

```
rule AssocToFKey(source p:uml!Package,
    view s:rdbms!Schema){
  sn, tn, an:String;
  update p:uml!Package{
   associations=assoc:uml!Association{
     name=an,source=sc:uml!Class{name=sn},
      target=tc:uml!Class{name=tn}}}
   with s:rdbms!Schema{
     tables=st:rdbms!Table{name=sn,
      foreignKeys=f:ForeignKey{name=an,
      referTo=tt:rdbms!Table{name=tn}}}} by
     match -> {}
      where {index(sc,st);index(tc,tt)}
      ...
}
```

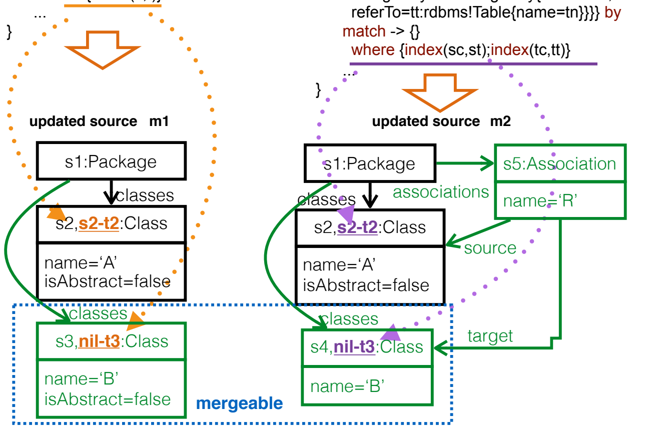

Figure 8: Index computation

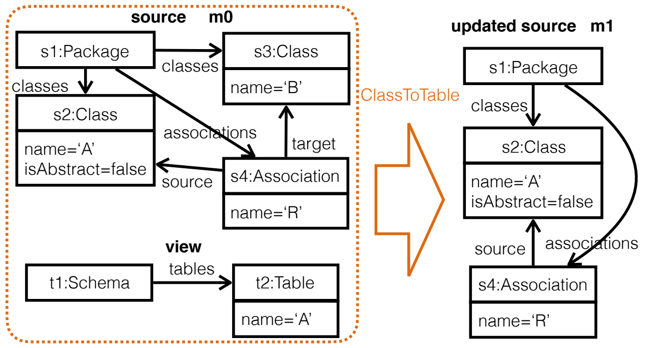

Figure 9: AssocToFKey will fail after ClassToTable

A only, and table *B* and the foreign key from table *A* to table *B* were removed (please refer to the view model as shown in Figure 7). If we apply *ClassToTable* against *m*0 in the backward direction first, then we get an (intermediate) updated source *m*1 in which class *B* was removed. However, in this case, we cannot continue to perform *AssocToFKey* backward against *m*1 because the source pattern of the `update` statement in *AssocToFKey* (as shown in Figure 6) cannot be matched. This source pattern matches an association that connects two classes, while in *m*1 in Figure 9, the association *R* connects class *A* only. The final result, which is equal to *m*1 and contains class *A* and an unwanted association *R*, is incorrect. The only way of obtaining the expected result is to combine *ClassToTable* and *AssocToFKey* by using our model-merging-based BX combinator, i.e., applying *ClassToTable* and *AssocToFKey* against *m*0 in parallel and then merging the two results.

$$
\begin{array}{lll|lll}
b & ::= & \texttt{updateWithBy}\ pat_S\ pat_V\ b\ u_c\ u_r \mid \texttt{ifThenElse}\ f\ b_1\ b_2 & u & ::= & \texttt{enforce}\ pat_S \\
 & \mid & \texttt{replace}\ pat_S\ pat_V \mid \texttt{matchS}\ pat_S\ b \mid \texttt{matchV}\ pat_V\ b & & \mid & \texttt{delete}\ v \mid \texttt{delete}\ v\ featur \\
 & \mid & \texttt{index}\ v_S\ v_V\ b \mid P(e_1, ..., e_n) \mid b_1 ; b_2 & & \mid & \texttt{condition}\ e\ u_1\ u_2 \mid u_1 ; u_2
\end{array}
$$

Figure 10: Core Language of XMU

5 SEMANTICS OF XMU

The concrete syntax of XMU in Figure 3 is designed for users, but it would be too complex to define the semantics directly on it. Rather, we propose a core language for XMU (namely XMU core), whose semantics is easier to define. XMU core is defined in Figure 10, where b denotes a bidirectional statement and u denotes a unidirectional statement.

The conversion between the concrete syntax of XMU and XMU core is straightforward, as follows: `update-with-by` is converted into `updateWithBy`; `switch-case` is converted into `ifThenElse` (a pattern used as a branch condition can also be viewed as a boolean condition [1]); `where-index` is converted into `index`; `replace`, `matchS` and `matchV` are used to realize bidirectional conversions implied by `update-with-by` and `switch-case`.

Take rule *AssocToFKey* as a concrete example. The concrete syntax of *AssocToFKey* has been presented in Figure 6 and Figure 8. This rule can be equivalently translated into an `updateWithBy` statement in XMU core, as shown in Figure 11. The bidirectional statement b of the `updateWithBy` statement is a `replace` statement wrapped by two `index` statements. The two unidirectional statements u_a and u_r of the `updateWithBy` statement are a `delete` statement and a chain of two `condition`s and an `enforce` statement.

```
// patS and patV are defined as follows
// patS = p:uml!Package{
//    classes=c:uml!Class{name=cn,isAbstract=false}}
// patV = s:rdbms!Schema{tables=t:rdbms!Table{name=cn}}

updateWithBy patS patV
  index sc st (index tc tt (replace patS patV))        ⟹  b
    delete assoc                                       ⟹  ua
      condition p.classes->exists(sc|sc.name=sn)         }
        (do nothing) enforce sc:uml!Class{name=sn} ;     }
      condition p.classes->exists(tc|tc.name=tn)         }
        (do nothing) enforce tc:uml!Class{name=tn} ;     }  ur
      enforce assoc:uml!Association{name=an,             }
        source=sc:uml!Class{},target=tc:uml!Class{}}     }
```

Figure 11: Translation of rule AssocToFKey

An XMU transformation can be compiled into an XMU core program, which can further be executed according to the semantics defined in Figure 12. It is worthwhile emphasizing that we do not claim that the core language we proposed is complete. In fact, XMU core should be further refined and extended to cover more application scenarios (such as delta-based BXs). The reminder of this section will discuss the semantics of the core language in detail.

5.1 Variable, Environment and Pattern

A variable v in XMU core is viewed as an identifier associated with a type. A variable type can be a primitive type (e.g., integer) or an index type (if this variable will be pointed to an object). An environment γ is a set of mappings from variables to values. Γ denotes the set of all environments. We assume that $\gamma(v)$ returns the value assigned to v, and $\gamma[v := c]$ adds a mapping from v to c. Besides, $\gamma(v) = \bot$ means that v is unassigned in γ.

In XMU core, model patterns are used to find matches and to create model fragments. XMU core adopts the conventional semantics of pattern matching and instantiation that is widely used in existing model transformation languages (such as QVT). We mainly explain some properties in this subsection.

We term an environment γ a match of a pattern *pat* if all the variables occurring in *pat* (namely pattern variables, denoted by $vars(pat)$) are assigned in γ, and if the values assigned to pattern variables satisfy all the constraints implied by *pat* (as explained in [1], a pattern can be viewed as some constraints). Given an initial environment γ, $pat(\gamma, M)$ returns all matches of a pattern *pat* in a model M based on γ. Each returned environment γ' satisfies the condition $\forall v (v \notin vars(pat) \vee \gamma(v) \neq \bot \implies \gamma'(v) = \gamma(v))$. Instantiating a pattern *pat* based on an environment γ (denoted as $M = pat.new(\gamma)$) is to create a model fragment M that matches *pat* using the information provided by γ. Obviously, pattern matching and instantiation satisfy the following properties:

$$
\begin{array}{c}
\gamma' \in pat(\gamma, M) \implies M = (pat.new(\gamma') \triangleright M) \\
M' = pat.new(\gamma) \triangleright M \implies \gamma \in pat(\gamma, M')
\end{array}
$$

For simplicity, given variable v, if the type of v is the index type and $\gamma(v) = \bot$, then we assume that γ is automatically replaced by $\gamma[v := idx]$ during pattern instantiation, where idx a fresh and unused object index.

5.2 Unidirectional Statements

A unidirectional statement can convert a source model into an updated source model without considering the view model. Unidirectional statements will never be executed in the forward transformation. Given a unidirectional statement u, $\mathbb{U}^{\gamma}_{[\![u]\!]} : \mathbb{S} \rightarrow \mathbb{S} \times \Gamma$ denotes the semantics of u under the environment γ. For an expression e, the notation $\lambda\gamma\ e \cdot val, \gamma'$ means given environment γ, e evaluates to value val and a new environment γ'. If we do not care about the new environment, we may say $\lambda\gamma\ e \cdot val$. The semantics of unidirectional statements is discussed as follows.

`enforce` pat_S ensures that a match of pat_S exists in the result. If the given environment is not a match of pat_S, this statement creates a new instance of pat_S and merge it into the source model (i.e., $\mathbb{U}_{[\![\texttt{enforce}\ pat_S]\!]}$).

`delete` v removes the object referenced by variable v from the model (i.e., $\mathbb{U}_{[\![\texttt{delete}\ v]\!]}$). After the object removal, the indices mapped onto the removed object cannot be mapped onto other objects again. `delete` v $feature$ e removes an attribute value or a link from the model (i.e., $\mathbb{U}_{[\![\texttt{delete}\ v\ feature\ e]\!]}$). We also view attribute values as links for simplicity. For the node/link deletion, if the node/link does not exist in the input, the output is identical to the input.

$$\begin{array}{l}
\mathbb{F}^{\gamma}_{\llbracket b_1\,;b_2\rrbracket}(S) \equiv \texttt{let}\ V_1 = \mathbb{F}^{\gamma}_{\llbracket b_1\rrbracket}(S),\ V_2 = \mathbb{F}^{\gamma}_{\llbracket b_2\rrbracket}(S),\ V' = V_1 \uplus_{\emptyset} V_2\ \texttt{in}\\
\quad \texttt{if}\ S = \mathbb{B}^{\gamma}_{\llbracket b_1\rrbracket}(S,V') = \mathbb{B}^{\gamma}_{\llbracket b_2\rrbracket}(S,V')\ \texttt{then}\ V'\ \texttt{else}\ \bot\ \texttt{endif}\\
\mathbb{B}^{\gamma}_{\llbracket b_1\,;b_2\rrbracket}(S,V) \equiv \texttt{let}\ S_1 = \mathbb{B}^{\gamma}_{\llbracket b_1\rrbracket}(S,V),\ S_2 = \mathbb{B}^{\gamma}_{\llbracket b_2\rrbracket}(S,V),\ S' = S_1 \uplus_S S_2,\\
\quad V_1 = \mathbb{F}^{\gamma}_{\llbracket b_1\rrbracket}(S_1),\ V_2 = \mathbb{F}^{\gamma}_{\llbracket b_2\rrbracket}(S_2),\ V' = V_1 \uplus_{\emptyset} V_2,\\
\quad V_{c1} = core_1\ S\ V, V_{c2} = core_2\ S\ V,\ V_c = V_{c1} \uplus_{\emptyset} V_{c2}\ \texttt{in}\\
\quad \texttt{if}\ core_1\ S_1\ V = core_1\ S'\ V \wedge core_2\ S_2\ V = core_2\ S'\ V\\
\quad\quad \wedge S' = \mathbb{B}^{\gamma}_{\llbracket b_1\rrbracket}(S',V) \wedge S' = \mathbb{B}^{\gamma}_{\llbracket b_2\rrbracket}(S',V) \wedge V_{c1} = core_1\ S\ V_c\\
\quad\quad \wedge V_{c2} = core_2\ S\ V_c \wedge V_1 \uplus_{\emptyset} V_2 \simeq V_c\ \texttt{then}\ S'\ \texttt{else}\ \bot\ \texttt{endif}
\end{array}$$

$$\begin{array}{l}
\mathbb{F}^{\gamma}_{\llbracket \texttt{replace}\ pat_S\ pat_V\rrbracket}(S) \equiv\\
\quad \texttt{if}\ \{\gamma'\} = pat_S(\gamma,S)\ \texttt{then}\ pat_V.new(\gamma')\ \texttt{else}\ \bot\ \texttt{endif}\\
\mathbb{B}^{\gamma}_{\llbracket \texttt{replace}\ pat_S\ pat_V\rrbracket}(S,V) \equiv \texttt{if}\ pat_S(\gamma,S) = \{\gamma'\}\\
\quad \wedge |pat_V(\gamma',V)| = |pat_V(\gamma,V)| = 1\ \texttt{then}\ S\\
\quad \texttt{else if}\ \{\gamma'\} = pat_V(\gamma,V) \wedge pat_S(\gamma,S) = \emptyset\\
\quad\quad \texttt{then}\ pat_S.new(\gamma') \rhd S\ \texttt{else}\ \bot\ \texttt{endif endif}
\end{array}$$

$$\begin{array}{l}
\mathbb{F}^{\gamma}_{\llbracket \texttt{index}\ v_S\ v_V\ b\rrbracket}(S) \equiv \texttt{let}\ i_v = \texttt{index}_V(\gamma(v_S)),\ V = \mathbb{F}^{\gamma}_{\llbracket b\rrbracket}(S)\ \texttt{in}\ V\\
\quad \texttt{where}\ i_v\ \text{is mapped onto}\ object_V(\gamma(v_V))\ \text{in}\ V\\
\mathbb{B}^{\gamma}_{\llbracket \texttt{index}\ v_S\ v_V\ b\rrbracket}(S) \equiv \texttt{let}\ i_s = \texttt{index}_S(\gamma(v_S),\gamma(v_V)),\ S' = \mathbb{B}^{\gamma}_{\llbracket b\rrbracket}(S,V)\ \texttt{in}\ S'\\
\quad \texttt{where}\ i_s\ \text{is mapped onto}\ objects_{S'}(\gamma(v_S))\ \text{in}\ S'
\end{array}$$

$$\begin{array}{l}
\mathbb{F}^{\gamma}_{\llbracket \texttt{matchS}\ pat_S\ b\rrbracket}(S) \equiv \texttt{if}\ \{\gamma'\} = pat_S(\gamma,S)\ \texttt{then}\ \mathbb{F}^{\gamma'}_{\llbracket b\rrbracket}(S)\ \texttt{else}\ \bot\ \texttt{endif}\\
\mathbb{B}^{\gamma}_{\llbracket \texttt{matchS}\ pat_S\ b\rrbracket}(S,V) \equiv \texttt{if}\ \{\gamma'\} = pat_S(\gamma,S)\ \texttt{then let}\ S' = \mathbb{B}^{\gamma'}_{\llbracket b\rrbracket}(S,V)\ \texttt{in}\\
\quad \texttt{if}\ \{\gamma'\} = pat_S(\gamma,S')\ \texttt{then}\ S'\ \texttt{else}\ \bot\ \texttt{endif}\\
\quad \texttt{else}\ \bot\ \texttt{endif}
\end{array}$$

$$\begin{array}{l}
\mathbb{F}^{\gamma}_{\llbracket \texttt{matchV}\ pat_S\ b\rrbracket}(S) \equiv \texttt{let}\ V = \mathbb{F}^{\gamma}_{\llbracket b\rrbracket}(S)\ \texttt{in}\\
\quad \texttt{if}\ \{\gamma'\} = pat_V(\gamma,V) \wedge V = \mathbb{F}^{\gamma'}_{\llbracket b\rrbracket}(S)\ \texttt{then}\ V\ \texttt{else}\ \bot\ \texttt{endif}\\
\mathbb{B}^{\gamma}_{\llbracket \texttt{matchV}\ pat_S\ b\rrbracket}(S,V) \equiv \texttt{let}\ \{\gamma_1,\gamma_2,...\} = pat_V(\gamma,V)\ \texttt{in}\\
\quad \texttt{if}\ \exists! i(\mathbb{B}^{\gamma_i}_{\llbracket b\rrbracket}(S,V) = S' \neq \bot)\ \texttt{then}\\
\quad\quad \texttt{if}\ \exists! i(\mathbb{B}^{\gamma_i}_{\llbracket b\rrbracket}(S',V) \neq \bot)\ \texttt{then}\ S'\ \texttt{else}\ \bot\ \texttt{endif}\\
\quad\quad \texttt{else}\ \bot\ \texttt{endif}
\end{array}$$

$$\begin{array}{l}
\mathbb{X}^{\gamma}_{\llbracket P(e_1,\dots,e_n)\rrbracket} \equiv \texttt{let}\ \gamma' = \{v_i := val_i | \lambda\gamma\ e_i \cdot val_i\}\ \texttt{in}\ \mathbb{X}^{\gamma'}_{\llbracket b\rrbracket}\\
\quad \texttt{where}\ b\ \text{is the body statement of}\ P
\end{array}$$

$$\begin{array}{l}
getBranch(\gamma,f,b,S) \equiv \texttt{let}\ V = \mathbb{F}^{\gamma}_{\llbracket b\rrbracket}(S)\ \texttt{in}\\
\quad \texttt{if}\ \lambda\gamma\ f(S,V)\cdot true\ \texttt{then}\ V\ \texttt{else}\ \bot\ \texttt{endif}\\
\mathbb{F}^{\gamma}_{\llbracket \texttt{ifThenElse}\ f\ b_1\ b_2\rrbracket}(S) \equiv \texttt{let}\ V_2 = getBranch(\gamma,\neg f,b_2,S)\ \texttt{in}\\
\quad \texttt{let}\ V_2' = \texttt{if}\ \lambda\gamma\ f(S,V_2)\cdot true\ \texttt{then}\ \bot\ \texttt{else}\ V_2\ \texttt{endif in}\\
\quad\quad \texttt{if}\ V_2' = \bot\ \texttt{then}\ getBranch(\gamma,f,b_1,S)\ \texttt{else}\ V_2'\ \texttt{endif}\\
\mathbb{B}^{\gamma}_{\llbracket \texttt{ifThenElse}\ f\ b_1\ b_2\rrbracket}(S,V) \equiv \texttt{if}\ \lambda\gamma\ f(S,V)\cdot true\ \texttt{then}\\
\quad \texttt{let}\ S_1' = \mathbb{B}^{\gamma}_{\llbracket b_1\rrbracket}(S,V)\ \texttt{in}\\
\quad\quad \texttt{if}\ \lambda\gamma\ f(S_1',V)\cdot true\ \texttt{then}\ S_1'\ \texttt{else}\ \bot\ \texttt{endif}\\
\quad \texttt{else let}\ S_2' = \mathbb{B}^{\gamma}_{\llbracket b_2\rrbracket}(S,V)\ \texttt{in if}\ \lambda\gamma\ f(S_2',V)\cdot false\ \texttt{then}\\
\quad\quad \texttt{if}\ getBranch(\gamma,f,b_1,S_2') = \bot\ \texttt{then}\ S_2'\ \texttt{else}\ \bot\ \texttt{endif}\\
\quad\quad \texttt{else}\ \bot\ \texttt{endif}\\
\quad \texttt{endif}
\end{array}$$

$$\begin{array}{l}
\mathbb{F}^{\gamma}_{\llbracket \texttt{ifThenAdaption}\ f\ b\ u\rrbracket}(S) \equiv getBranch(\gamma,f,b,S)\\
\mathbb{B}^{\gamma}_{\llbracket \texttt{ifThenAdaption}\ f\ b\ u\rrbracket}(S,V) \equiv\\
\quad \texttt{if}\ \lambda\gamma\ f(S,V)\cdot true\ \texttt{then let}\ S' = \mathbb{B}^{\gamma}_{\llbracket b\rrbracket}(S,V)\ \texttt{in}\\
\quad\quad \texttt{if}\ \lambda\gamma\ f(S',V)\cdot true\ \texttt{then}\ S'\ \texttt{else}\ \bot\ \texttt{endif}\\
\quad \texttt{else let}\ (S_a,\gamma_a) = \mathbb{U}^{\gamma}_{\llbracket u\rrbracket}(S)\ \texttt{in}\\
\quad\quad \texttt{if}\ \lambda\gamma\ f(S_a,V)\cdot true\ \texttt{then let}\ S_a' = \mathbb{B}^{\gamma}_{\llbracket b\rrbracket}(S_a,V)\ \texttt{in}\\
\quad\quad\quad \texttt{if}\ \lambda\gamma\ f(S_a',V)\cdot true\ \texttt{then}\ S_a'\ \texttt{else}\ \bot\ \texttt{endif}\\
\quad\quad \texttt{else}\ \bot\ \texttt{endif}\\
\quad \texttt{endif}
\end{array}$$

$$\begin{array}{l}
\mathbb{X}_{\llbracket \texttt{updateWithBy}\ pat_S\ pat_V\ b\ u_c\ u_r\rrbracket} \equiv \mathbb{X}_{\llbracket \texttt{ifThenAdaption}\ f\ b^*\ u_a\rrbracket}\\
\mathbb{F}_{\llbracket b^*\rrbracket}\gamma(S) \equiv \texttt{let}\ \{\gamma_1,...,\gamma_n\} = pat_S(\gamma,S),\ V_i = \mathbb{F}^{\gamma_i}_{\llbracket b\rrbracket}(S),\\
\quad V' = V_1 \uplus_{\emptyset} ... \uplus_{\emptyset} V_n\ \texttt{in}\\
\quad\quad \texttt{if}\ S = \mathbb{B}^{\gamma_i}_{\llbracket b\rrbracket}(S,V')\ \texttt{then}\ V'\ \texttt{else}\ \bot\ \texttt{endif}\\
\mathbb{B}^{\gamma}_{\llbracket b^*\rrbracket}(S,V) \equiv \texttt{let}\ \{\gamma_1,...,\gamma_n\} = pat_S(\gamma,S),\ S_i = \mathbb{B}^{\gamma_i}_{\llbracket b\rrbracket}(S,V),\\
\quad S' = S_1 \uplus_S ... \uplus_S S_n, V_i = \mathbb{F}^{\gamma_i}_{\llbracket b\rrbracket}(S),\ V' = V_1 \uplus_{\emptyset} ... \uplus_{\emptyset} V_n\ \texttt{in}\\
\quad \texttt{if}\ V_i = \mathbb{F}^{\gamma_i}_{\llbracket b\rrbracket}(S') \wedge S' = \mathbb{B}^{\gamma_i}_{\llbracket b\rrbracket}(S',V) \wedge S_i = \mathbb{B}^{\gamma_i}_{\llbracket b\rrbracket}(S,V')\\
\quad\quad \wedge \{\gamma_1,...,\gamma_n\} = pat_S(\gamma,S')\ \texttt{then}\ S'\ \texttt{else}\ \bot\ \texttt{endif}\\
isAligned(\gamma_l,\gamma_r) \equiv \big(\forall v(\gamma_l(v) \neq \bot \wedge \gamma_r(v) \neq \bot \implies \gamma_l(v) = \gamma_r(v))\big)\\
\quad \texttt{where}\ \text{the value of}\ v\ \text{is considered as}\ \bot\ \text{when}\ v\ \text{is an unused index}
\end{array}$$

$$\begin{array}{l}
\mathbb{U}^{\gamma}_{\llbracket \texttt{enforce}\ pat_S\rrbracket}(S) \equiv (pat.new(\gamma) \rhd S, \gamma)\\
\mathbb{U}^{\gamma}_{\llbracket \texttt{delete}\ \ v\rrbracket}(S) \equiv (S - objects_S(\gamma(v)), \gamma)\\
\mathbb{U}^{\gamma}_{\llbracket \texttt{delete}\ \ feature\ e\rrbracket}(S) \equiv (S - l, \gamma')\\
\quad \texttt{where}\ \lambda\gamma\ e\cdot val,\gamma' \wedge l \equiv (objects_S(\gamma(v)), val) \wedge typeOf(l) = feature
\end{array}$$

$$\begin{array}{l}
\mathbb{U}^{\gamma}_{\llbracket u_1\,;u_2\rrbracket}(S) \equiv \texttt{let}\ (S',\gamma') = \mathbb{U}^{\gamma}_{\llbracket u_1\rrbracket}(S)\ \texttt{in}\ \mathbb{U}^{\gamma'}_{\llbracket u_2\rrbracket}(S')\\
\mathbb{U}^{\gamma}_{\llbracket \texttt{condition}\ e\ u_1\ u_2\rrbracket}(S) \equiv \texttt{if}\ \lambda\gamma\ e\cdot true,\gamma'\ \texttt{then}\ \mathbb{U}^{\gamma'}_{\llbracket u_1\rrbracket}(S)\\
\quad \texttt{else if}\ \lambda\gamma\ e\cdot false,\gamma'\ \texttt{then}\ \mathbb{U}^{\gamma'}_{\llbracket u_2\rrbracket}(S)\ \texttt{else}\ \bot\ \texttt{endif}\\
\quad \texttt{endif}
\end{array}$$

Figure 12: Formal Semantics of XMU Core

`condition` e u_1 u_2 is a conditional statement, which executes u_1 or u_2 according to the result of expression e (i.e., $\mathbb{U}_{\llbracket \texttt{condition}\ e\ u_1\ u_2\rrbracket}$).

$u_1\,;u_2$ executes u_1 and u_2 sequentially (i.e., $\mathbb{U}_{\llbracket u_1\,;u_2\rrbracket}$).

5.3 Bidirectional Statements

A bidirectional statement b is interpreted as a BX $\mathbb{X}^{\gamma}_{\llbracket b\rrbracket}$, which represents the bidirectional semantics of b under environment γ. $\mathbb{X}^{\gamma}_{\llbracket b\rrbracket}$ consists of a forward semantics $\mathbb{F}^{\gamma}_{\llbracket b\rrbracket}$ and a backward semantics $\mathbb{B}^{\gamma}_{\llbracket b\rrbracket}$ (under the same environment γ).

$b_1\,;b_2$ merges b_1 and b_2 using $\uplus$. We embed explicit checks of equations (8)-(12) in the definitions of $\mathbb{B}^{\gamma}_{\llbracket b_1\,;b_2\rrbracket}$ and $\mathbb{F}^{\gamma}_{\llbracket b_1\,;b_2\rrbracket}$ to ensure their well-behavedness. The core function of $\mathbb{X}^{\gamma}_{\llbracket b_1\,;b_2\rrbracket}$ is the merging of the core functions of $\mathbb{X}^{\gamma}_{\llbracket b_1\rrbracket}$ and $\mathbb{X}^{\gamma}_{\llbracket b_2\rrbracket}$.

`replace` pat_S pat_V ensures that a match of pat_S is paired with a match of pat_V. For $\mathbb{B}^{\gamma}_{\llbracket \texttt{replace}\ pat_S\ pat_V\rrbracket}$, this statement instantiates a match of pat_S based on a match of pat_V if the source match is absent. In the forward semantics $\mathbb{F}^{\gamma}_{\llbracket \texttt{replace}\ pat_S\ pat_V\rrbracket}$, this statement instantiates a match of pat_V based on a match of pat_S. The core function of $\mathbb{X}^{\gamma}_{[\![\texttt{replace}\ pat_S\ pat_V]\!]}$ extracts the unique match of pat_V out from the view model based on γ.

`index` $v_S\ v_V\ b$ may append new indices to the result of b. The core function of $\mathbb{X}^{\gamma}_{[\![\texttt{index}\ v_S\ v_V\ b]\!]}$ is the core function of $\mathbb{X}^{\gamma}_{[\![b]\!]}$. As mentioned in Section 4, we employ an index function $\texttt{index}(s, v)$ to compute object indices. In $\mathbb{B}^{\gamma}_{[\![\texttt{index}\ v_S\ v_V\ b]\!]}$, the index function is written as $\texttt{index}_S(s, v)$, which computes an updated-source index based on s and v. In the same backward transformation, any two invocations of $\texttt{index}_S(s_1, v_1)$ and $\texttt{index}_S(s_2, v_2)$ must satisfy the following runtime constraints: 1) $v_1 = v_2 \Rightarrow s_1 = s_2$; 2) $s_1 \neq s_2 \implies \texttt{index}_S(s_1, v_1) \neq \texttt{index}_S(s_2, v_2)$; 3) $\texttt{index}_S$ should not generate an index that is already used in the original source model. In $\mathbb{F}^{\gamma}_{[\![\texttt{index}\ v_S\ v_V\ b]\!]}$, the index function is denoted as $\texttt{index}_V(s)$, which computes a view index based on s. In the same forward transformation any two invocations of $\texttt{index}_V(s_1)$ and $\texttt{index}_V(s_2)$ must satisfy the following constraint $s_1 \neq s_2 \implies \texttt{index}_V(s_1) \neq \texttt{index}_V(s_2)$. These constraints ensure that 1) the same view objects are always created based on the same source objects and that 2) index functions do not cause index collision. Index functions may be implemented in many ways. They can (but not necessarily) be bidirectional transformations. As shown in Section 4, we can concatenate the actual parameters of an index function to compute the result index. In our future work, we plan to investigate other possible implementation of index functions.

`ifThenElse` $f\ b_1\ b_2$ is a bidirectional statement that selects b_1 or b_2 according to condition f, where f is a boolean function about the source and view model. We adapt the semantics of the `case` statement in BiGUL [20] (i.e., a two-branch `case` statement) for the semantics of this statement. The core function of $\mathbb{X}^{\gamma}_{[\![\texttt{ifThenElse}\ f\ b_1\ b_2]\!]}$ returns either the result of $\mathbb{X}^{\gamma}_{[\![b_1]\!]}$'s core function or the result of $\mathbb{X}^{\gamma}_{[\![b_2]\!]}$'s core function, depending on condition f.

`matchS` $pat_S\ b$ finds a match of pat_S and then evaluates b based on this match (i.e., $\mathbb{F}^{\gamma}_{[\![\texttt{matchS}\ pat_S\ b]\!]}$ and $\mathbb{B}^{\gamma}_{[\![\texttt{matchS}\ pat_S\ b]\!]}$). Especially, in the backward semantics, the match of pat_S must also exist in the updated source model. Similarly, `matchV` $pat_V\ b$ finds a match of pat_V and then evaluates b based on this match (i.e., $\mathbb{F}^{\gamma}_{[\![\texttt{matchV}\ pat_S\ b]\!]}$ and $\mathbb{B}^{\gamma}_{[\![\texttt{matchV}\ pat_S\ b]\!]}$). In the backward semantics, there may be multiple matches of pat_V found, but only one match is expected to be successfully used in the evaluation of b. If no or many matches of pat_V can be used, then the statement returns $\perp$. The core functions of $\mathbb{X}^{\gamma}_{[\![\texttt{matchS}\ pat_S\ b]\!]}$ and $\mathbb{X}^{\gamma}_{[\![\texttt{matchV}\ pat_V\ b]\!]}$ are the core function of $\mathbb{X}^{\gamma'}_{[\![b]\!]}$, where γ' is the match of pat_S or pat_V that is found based on γ.

$P(e_1, \ldots, e_n)$ denotes the rule call statement. It prepares a new environment for the callee rule and executes the body statement (supposing that b is the body of rule P). The core function of this statement is the core function of b.

Finally, consider the semantics of `updateWithBy`. Before going on, we define a *conditional statement with adaption* (i.e., `ifThenAdaption` $f\ b\ u$), which will be used to construct the semantics `updateWithBy`. `ifThenAdaption` is also adapted from BiGUL [20], so it is a well behaved BX. Informally, in the backward direction, `ifThenAdaption` executes bidirectional statement b if condition f holds; otherwise, this statement executes unidirectional statement u to change the given source model into a new source that can make f hold, and then executes b with the new source. In the forward direction, `ifThenAdaption` ignores u and behaves like `ifThenElse` statement whose *else*-branch is $\perp$.

As for `updateWithBy` $pat_S\ pat_V\ b\ u_c\ u_r$, informally, in the backward direction, it aligns the matches of pat_S with the matches of pat_V, and executes the bidirectional statement b (to synchronize the aligned source and view matches) and unidirectional statements u_r (to destroy misaligned source matches) and u_c (to create matches of pat_S for misaligned view matches) according to the alignment result. In the forward direction, this statement finds all matches of pat_S and executes b to construct the view model for each match.

We start from a simple case in which source and view models (i.e., S and V) satisfy the following *alignment condition*: each match of pat_S can be uniquely aligned with a match of pat_V, and each match of pat_V can be uniquely aligned with a match of pat_S (we use $isAligned(\gamma_l, \gamma_r)$, defined in Figure 12, to determine whether a source match and a view match can be aligned). In fact, in this case, a view match is uniquely determined by a source match (and vice versa). Furthermore, only bidirectional statement b will be executed for each pair of source and view matches that are aligned. We view each execution of b as a derived component BX. Because the view match is fully determined by the corresponding source match, every component BX is actually derived from a single source match. Hence, in the simple case, the semantics of `updateWithBy` can be viewed as a composition of a set $\{\mathbb{X}^{\gamma_i}_{[\![b]\!]}\}$ of derived component BXs, where γ_i is a match of pat_S. We formulate this semantics as $(\mathbb{F}_{[\![b^*]\!]}, \mathbb{B}_{[\![b^*]\!]})$ in Figure 12. In $\mathbb{F}_{[\![b^*]\!]}$ and $\mathbb{B}_{[\![b^*]\!]}$, we explicitly check equations (8)-(12) and the alignment condition.

Now we consider the complex case where some source/view matches may be misaligned. Based on the informal semantics above, u_c and u_r will be executed to destroy the unaligned source matches and to construct new source matches that can be paired with unaligned view matches, respectively. In short, we execute u_c and u_r to convert the original source model into a new one that will satisfy the *alignment condition*. Hence, given a certain source model S, we can specify the semantics of `updateWithBy` as follows: $\mathbb{X}^{\gamma}_{[\![\texttt{updateWithBy}\ pat_S\ pat_V\ b\ u_c\ u_r]\!]} \equiv \mathbb{X}^{\gamma}_{[\![\texttt{ifThenAdaption}\ f\ b^*\ u_a]\!]}$, where f is the *alignment condition*, u_a is derived from u_c and u_r. In backward transformation, u_a can be derived as follows: 1) for each misaligned match γ_{s_i} of pat_S, execute u_r to obtain $S_{s_i} \equiv \mathbb{U}^{\gamma_{s_i}}_{[\![u_r]\!]}(S)$; 2) for each misaligned match γ_{v_i} of pat_V, execute u_c to obtain $S_{v_i} \equiv \mathbb{U}^{\gamma_{v_i}}_{[\![u_c]\!]}(S)$; 3) merge all S_{s_i} and S_{v_i}. In forward direction, we simply ignore u_a since it is useless.

5.4 Round-trip Properties of XMU Core

The key to defining a BX language is to assure the well-behavedness of this language. All bidirectional statements defined in XMU core can be proven to be well behaved. In this section, we presents the proof sketch of the well-behavedness of XMU core as the evidence of the correctness of our approach.

- $\mathbb{X}^{\gamma}_{[\![\texttt{replace}\ pat_S\ pat_V]\!]}$: After executing $\mathbb{B}^{\gamma}_{[\![\texttt{replace}\ pat_S\ pat_V]\!]}$, we can ensure that there exists a unique match of pat_S in the updated source model which can be found in the forward execution to construct the view model (i.e., PutGet*). $\mathbb{B}^{\gamma}_{\llbracket \texttt{replace}\ pats_S\ pat_V \rrbracket}$ also states that if there is already a match of $pats_S$ in the original source model, this statement does nothing (i.e., PutTwice). After running $\mathbb{F}^{\gamma}_{\llbracket \texttt{replace}\ pats_S\ pat_V \rrbracket}$, for the unique match of $pats_S$, a view instance is created which can be paired with the match of $pats_S$ during the backward execution (i.e., GetPut).

- $\mathbb{X}^{\gamma}_{\llbracket \texttt{index}\ v_S\ v_V\ b \rrbracket}$: If $\mathbb{X}^{\gamma}_{\llbracket b \rrbracket}$ is well behaved, then this statement is also well behaved because it does not change the output of $\mathbb{X}^{\gamma}_{\llbracket b \rrbracket}$ (though it appends new object indices).
- $\mathbb{X}^{\gamma}_{\llbracket \texttt{ifThenElse}\ f\ b_1\ b_2 \rrbracket}$: This statement is well behaved because it is adapted from the `case` statement in BiGUL (i.e., a `case` statement with two branches and without adaption). In BiGUL, the `case` statement has been verified [20].
- $\mathbb{X}^{\gamma}_{\llbracket \texttt{matchS}\ pats_S\ b \rrbracket}$: In $\mathbb{B}^{\gamma}_{\llbracket \texttt{matchS}\ pats_S\ b \rrbracket}$, a match of $pats_S$, namely γ', is found first and then $\mathbb{B}^{\gamma'}_{\llbracket b \rrbracket}$ is executed. After executing $\mathbb{B}^{\gamma'}_{\llbracket b \rrbracket}$, $\mathbb{B}^{\gamma}_{\llbracket \texttt{matchS}\ pats_S\ b \rrbracket}$ also checks that the match γ' of $pats_S$ still exists in the updated source. Hence, PutGet* and PutTwice hold when $\mathbb{X}^{\gamma'}_{\llbracket b \rrbracket}$ is well behaved. In $\mathbb{F}^{\gamma}_{\llbracket \texttt{matchS}\ pats_S\ b \rrbracket}$, a match of $pats_S$, namely γ', is found first and then $\mathbb{F}^{\gamma'}_{\llbracket b \rrbracket}$ is executed. Hence, GetPut holds when $\mathbb{X}^{\gamma'}_{\llbracket b \rrbracket}$ is well behaved.
- $\mathbb{X}^{\gamma}_{\llbracket \texttt{matchV}\ pat_V\ b \rrbracket}$: The backward semantics (i.e., $\mathbb{B}^{\gamma}_{\llbracket \texttt{matchV}\ pat_V\ b \rrbracket}$) ensures PutTwice. As discussed previously, $\mathbb{B}^{\gamma}_{\llbracket \texttt{matchV}\ pat_V\ b \rrbracket}$ finds a match (namely γ') of pat_V to evaluate b. In this process, it does not change the information related to source in the original environment γ. Hence, executing $\mathbb{F}^{\gamma}_{\llbracket b \rrbracket}$ results in a view that is isomorphic to the result of $\mathbb{F}^{\gamma'}_{\llbracket b \rrbracket}$. Hence, PutGet* must hold when b is well behaved. Regarding GetPut, in $\mathbb{F}^{\gamma}_{\llbracket \texttt{matchV}\ pat_V\ b \rrbracket}$, the statement executes b to create a view V that contains exactly one match of pat_V (namely γ'). If we execute the backward transformation immediately, we will find a unique match in V that must be γ'. Hence, GetPut holds.
- $\mathbb{X}^{\gamma}_{\llbracket P(e_1,\ldots,e_n) \rrbracket}$: If the body statement of P, namely b, is well behaved, then the rule call statement is well behaved. It is because that a rule call statement simply constructs a new environment based on γ and then executes b with the new environment.
- $\mathbb{X}^{\gamma}_{\llbracket b_1 ; b_2 \rrbracket}$: This statement is well behaved because this statement is actually realized by using the model-merging-based BX combinator $\uplus$ (see Theorem 3.1).
- $\mathbb{X}^{\gamma}_{\llbracket \texttt{updateWithBy}\ pats_S\ pat_V\ b\ u_c\ u_r \rrbracket}$: This statement is interpreted as $\mathbb{X}^{\gamma}_{\llbracket \texttt{ifThenAdaption}\ f\ b^*\ u_a \rrbracket}$, where `ifThenAdaption` statement is also adapted from the `case` statement in BiGUL. Hence, we only have to prove $\mathbb{X}^{\gamma}_{\llbracket b^* \rrbracket}$ is well behaved. According to the definition of $\mathbb{X}^{\gamma}_{\llbracket b^* \rrbracket}$, we learn that $\mathbb{X}^{\gamma}_{\llbracket b^* \rrbracket}$ is a composition of $\mathbb{X}^{\gamma_1}_{\llbracket b \rrbracket}$, $\mathbb{X}^{\gamma_2}_{\llbracket b \rrbracket}$, ..., $\mathbb{X}^{\gamma_n}_{\llbracket b \rrbracket}$ (they are combined by $\uplus$), where $\{\gamma_1, \ldots, \gamma_n\} = pats_S(\gamma, S)$. The definition itself is well behaved. However, this definition of $\mathbb{X}^{\gamma}_{\llbracket b^* \rrbracket}$ is actually related to the source matches $\gamma_1, \ldots, \gamma_n$. To ensure $\mathbb{X}^{\gamma}_{\llbracket b^* \rrbracket}$ is well behaved, we must ensure that $\gamma_1, \ldots, \gamma_n$ are not destroyed during the backward transformation (since in the forward transformation, the source model is not changed). In $\mathbb{B}^{\gamma}_{\llbracket b^* \rrbracket}$, we append an explicit check at the end of the execution. Overall, the round-trip properties of $\mathbb{X}^{\gamma}_{\llbracket \texttt{updateWithBy}\ pats_S\ pat_V\ b\ u_c\ u_r \rrbracket}$ are satisfied.

6 TOOL SUPPORT AND EXAMPLES

We implemented a prototype tool of XMU on Eclipse platform. The tool employed Eclipse Modeling Framework [27] as the internal data representation. The language facilities, such as code editor, interpreter and launcher, were realized based on EMFText [26] and Xtext [28]. In our tool implementation, we also extend the syntax of XMU by providing some syntax sugars, such as the `otherwise` branch for `switch-case`. At present, we applied the dynamic check of equations (8)-(12) to ensure the well-behavedness of a BX.

To evaluate and demonstrate XMU, we implemented several benchmark BX programs[2] using our tool support. Those BX programs include (but not limited to) the follow programs: 1) the classic transformation `UML2RDBMS` between UML class diagrams and RDBMS models, which has been intensively studied in the model transformation community (e.g., Zan et al., [35] presented a partial implementation based on BiFluX); 2) the conversion between a hierarchical state machine and a flatten state machine (`HSM2FSM`), which was studied in [9]; 3) the book mark example, which was studied in [25] and [18]; 4) the nested section example, which was studied in [2]; 5) the address book examples, which were studied in [25]. Due to space limitation, we cannot present all details of these examples. A full list of example BX programs (including the source code and transformation results) is available on our website. To the best of our knowledge, our implementation of `UML2RDBMS` is the first fully implemented *putback*-based version, which is partially presented in Section 4.

7 CONCLUSION AND FUTURE WORK

In this paper, we proposed a *putback*-based language XMU for bidirectional model transformation and a model-merging-based BX combinator $\uplus$ to address the ambiguity and the shared node issue. XMU enables us to define a BX in the form of a backward transformation. According to the semantics presented in Figure 12, we can derive a unique forward transformation from this backward transformation to eliminate ambiguities. The model-merging-based BX combinator allows multiple conversions of the same node and preserves the result of each conversion if there is no conflict.

There are some open issues that must be investigated in the future to turn XMU into a more efficient, user-friendly and practical BX language. First, we believe that XMU core may be refined and enhanced to improve its semantics and expressiveness. Second, XMU directly checks equations (8)-(12) at runtime to ensure the successful application of $\uplus$. However, doing so may be very time consuming. We plan to investigate how to efficiently check the validity of the BX combination. Third, we will explore some useful implementation mechanisms of index functions and integrate them into our tool support. Fourth, we will investigate how to support delta-based, incremental, and symmetric BX, and how to integrate XMU core with the existing transformation languages. Finally, we will conduct more case studies to evaluate and apply our approach.

REFERENCES

[1] 2015. *Meta Object Facility (MOF) 2.0 Query/View/Transformation specification V1.2.* Technical Report formal/15-02-01. Object Management Group.

[2] Davi M J Barbosa, Julien Cretin, Nate Foster, Michael J Greenberg, and Benjamin C Pierce. 2010. Matching lenses: alignment and view update. In *Proc. of ICFP'10.* 193–204.

[3] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. 2011. JTL: a bidirectional and change propagating transformation language. In *Proc. of SLE'11*. Springer, 183–202.

[4] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. 2009. Bidirectional Transformations: A Cross-Discipline Perspective. In *Proc. of ICMT'09 (LNCS)*, Richard F. Paige (Ed.), Vol. 5563. Springer, 260–283.

[5] Zinovy Diskin. 2008. Algebraic models for bidirectional model synchronization. In *Proc. of MODELS'08*. Springer, 21–36.

[6] Zinovy Diskin, Romina Eramo, Alfonso Pierantonio, and Krzysztof Czarnecki. 2016. Incorporating Uncertainty into Bidirectional Model Transformations and their Delta-Lens Formalization. In *Proc. of BX'16*, J. Gibbons A. Anjorin (Ed.).

[7] Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann, and Fernando Orejas. 2011. From state-to delta-based bidirectional model transformations: the symmetric case. In *Proc. of MODELS'11*. Springer, 304–318.

[8] Hartmut Ehrig, Karsten Ehrig, Claudia Ermel, Frank Hermann, and Gabriele Taentzer. 2007. Information preserving bidirectional model transformations. In *Proc. of FASE'07*. Springer, 72–86.

[9] Romina Eramo, Alfonso Pierantonio, and Gianni Rosa. 2015. Managing Uncertainty in Bidirectional Model Transformations. In *Proc. of SLE'15*. ACM, 49–58.

[10] Sebastian Fischer, ZhenJiang Hu, and Hugo Pacheco. 2015. The essence of bidirectional programming. *Science China Information Sciences* 58, 5 (2015), 1–21.

[11] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem. *ACM Trans. Program. Lang. Syst.* 29, 3, Article 17 (May 2007).

[12] Holger Giese and Robert Wagner. 2009. From model transformation to incremental bidirectional model synchronization. *Softw. & Syst. Modeling* 8, 1 (2009), 21–43.

[13] Xiao He, ChangJun Hu, ZhiYi Ma, and WeiZhong Shao. 2014. A bidirectional-transformation-based framework for software visualization and visual editing. *Science China Information Sciences* 57, 5 (2014), 1–23.

[14] Xiao He, Zhenjiang Hu, and Yi Liu. 2017. Towards Variability Management in Bidirectional Model Transformation. In *Proc. of COMPSAC'17*. 224–233.

[15] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, Yingfei Xiong, Susann Gottmann, and Thomas Engel. 2013. Model synchronization based on triple graph grammars: correctness, completeness and invertibility. *Softw. & Syst. Modeling* 14, 1 (2013), 241–269.

[16] Daniel Jackson. 2012. *Software Abstractions: logic, language, and analysis*. MIT press.

[17] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. 2006. ATL: a QVT-like transformation language. In *Proc. of OOPLSA'06*. ACM, 719–720.

[18] Shinya Kawanaka and Haruo Hosoya. 2006. biXid: a bidirectional transformation language for XML. In *ACM SIGPLAN Notices*, Vol. 41. ACM, 201–214.

[19] Hsiang-Shang Ko and Zhenjiang Hu. 2018. An Axiomatic Basis for Bidirectional Programming. In *Proc. of POPL'18*. ACM.

[20] Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. 2016. BiGUL: a formally verified core language for putback-based bidirectional programming. In *Proc. of PEPM'16*. 61–72.

[21] Yngve Lamo, Florian Mantz, Adrian Rutle, and Juan de Lara. 2013. A declarative and bidirectional model transformation approach based on graph co-spans. In *Proc. of PPDP'13*. ACM, 1–12.

[22] Marius Lauder, Anthony Anjorin, Gergely Varró, and Andy Schürr. 2012. Bidirectional model transformation with precedence triple graph grammars. In *Proc. of ECMFA'12*. Springer, 287–302.

[23] Nuno Macedo and Alcino Cunha. 2016. Least-change bidirectional model transformation with QVT-R and ATL. *Softw. & Syst. Modeling* 15, 3 (2016), 783–810.

[24] Hugo Pacheco, Zhenjiang Hu, and Sebastian Fischer. 2014. Monadic combinators for Putback style bidirectional programming. In *Proc. of PEPM'14*. ACM, 39–50.

[25] Hugo Pacheco, Tao Zan, and Zhenjiang Hu. 2014. Biflux: A bidirectional functional update language for XML. In *Proc. of PPDP'14*.

[26] EMFText project. 2018. EMFText. http://www.emftext.org/index.php/EMFText

[27] Eclipse Modeling Framework project. 2018. Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf/

[28] Xtext project. 2018. Xtext. http://www.eclipse.org/Xtext/

[29] Oszkár Semeráth, Csaba Debreceni, Ákos Horváth, and Dániel Varró. 2016. Incremental Backward Change Propagation of View Models by Logic Solvers. In *Proc. of MODELS'16*. ACM, 306–316.

[30] Hui Song, Gang Huang, Franck Chauvel, Yingfei Xiong, Zhenjiang Hu, Yanchun Sun, and Hong Mei. 2011. Supporting runtime software architecture: A bidirectional-transformation-based approach. *Journal of Systems & Software* 84, 5 (2011), 711–723.

[31] Perdita Stevens. 2010. Bidirectional model transformations in QVT: semantic issues and open questions. *Softw. & Syst. Modeling* 9, 1 (2010), 7–20.

[32] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. 2012. A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Softw. & Syst. Modeling* 13, 1 (2012), 239–272.

[33] Yingfei Xiong, Hui Song, Zhenjiang Hu, and Masato Takeichi. 2013. Synchronizing concurrent model updates based on bidirectional transformation. *Softw. & Syst. Modeling* 12, 1 (2013), 89–104.

[34] Yijun Yu, Yu Lin, Zhenjiang Hu, Soichiro Hidaka, Hiroyuki Kato, and Lionel Montrieux. 2012. Maintaining invariant traceability through bidirectional transformations. In *Proc. of ICSE'12*. IEEE, 540–550.

[35] Tao Zan, Hugo Pacheco, and Zhenjiang Hu. 2014. Writing Bidirectional Model Transformations As Intentional Updates. In *Proc. of ICSE'14*. ACM, 488–491.

[36] Zirun Zhu, Yongzhe Zhang, Hsiang-Shang Ko, Pedro Martins, João Saraiva, and Zhenjiang Hu. 2016. Parsing and Reflective Printing, Bidirectionally. In *Proc. of SLE'16*. ACM, 2–14.