BiFluX: A Bidirectional Functional Update Language for XML

Tao Zan, Hugo Pacheco and Zhenjiang Hu

XML formats are widely used for information exchange and data processing. A common use is to store information in the same way as relational databases, and many XML query languages such as XQuery have been developed for this purpose. De facto XML processing languages are traditionally unidirectional, in the sense that programmers write transformations from source formats to view formats, but backward updating, i.e., translating view modifications into the original XML database, is not considered. To cover this gap, existing XML bidirectional languages propose particular linguistic mechanisms that aid programmers in writing a forward XML transformation, such that a consistent backward transformation can be derived for free. Unfortunately, as the classical view-update problem for relational databases, backward transformations are generally not unique and existing bidirectional systems simply provide one out of many possible, which may not match the programmer's expectations of a view update.

This paper introduces BiFluX, a novel bidirectional XML update language inspired on Flux, a functional XML update language. BiFluX follows a pragmatically different approach from existing bidirectional languages: instead of writing (simpler) XML queries, programmers write backward transformations by specifying how a view format can be used to update a source database, for which there is a unique query. The novel approach allows programmers to transparently control how view updates can be reflected back to an XML database, while offering bidirectionality.

1 Introduction

Bidirectional transformations [7] [11], transformations that execute both forwards and backwards, aim at enforcing some notion of consistency between two data domains. Since data evolves naturally over time and is often replicated among different applications, this pattern reoccurs with various shapes in many computer science disciplines, such as databases, model transformations, data synchronization or user interfaces.

Originated from the well-known *view updating* problem in the database community [1], one of the most popular bidirectional transformation idioms are *lenses* [9]. A lens focuses on a special asymmetric case: a forward transformation *get* defines a view of a source database that usually contains more information, and a backward transformation *put* defines a view update strategy that synchronizes an original source database with a modified view. Naturally, these transformations shall not be arbitrary and are expected to satisfy two instrumental *well-behavedness* properties:

$$put\ s\ (get\ s) = s \qquad \textsc{PutGet}$$

$$get\ (put\ s\ v) = v \qquad \textsc{GetPut}$$

The PutGet property means that a lens is *acceptable* and all views can be reflected to the source: applying the forward transformation after putting back any view shall return the same view.

BiFluX: A Bidirectional Functional Update Language for XML

Tao Zan, Department of Informatics, The Graduate University for Advanced Studies.

Hugo Pacheco, Information Systems Architecture Research Division, National Institute of Informatics.

Zhenjiang Hu, Information Systems Architecture Research Division, National Institute of Informatics.

The GetPut property means that a lens is *stable*: putting back a view immediately after getting it shall yield the original source.

During the last decade, several *bidirectional programming languages* [9] [10] [3] [4] [14] [15] [12] have been developed to aid programmers in writing bidirectional programs that resemble writing a forward *get* transformation, and provide automatic mechanisms to derive a suitable backward *put* transformation for free. Such languages permit writing bidirectional programs that are easier to maintain, but fail to adequately address the inherent ambiguity problem of bidirectional transformations [5]. As most interesting cases of *get* transformations are not injective, there may exist many possible *put* transformations that combined with *get* form a well-behaved lens. Despite of this fact, existing bidirectional languages are only designed to satisfy fundamental well-behavedness principles and only consider one particular synchronization strategy that may not match the programmer's expectations for particular updates. This non-uniqueness often leads to unpredictable bidirectional behavior as these languages give little insight to users about which particular backward synchronization strategy is chosen among the myriad possible.

An insightful observation is that while one *get* transformation may be combined with several *put* transformations, only one *get* transformation is possible per *put* under the corresponding consistency properties. This primacy of *put* has been shown in [8] and encourages an excitingly powerful approach to specifying bidirectional lens programs by their backward transformations, with the distinctive advantage of being capable of completely expressing all aspects of a bidirectional transformation, while retaining the maintainability of writing a single program.

A logical conclusion is then to focus the design of a bidirectional language entirely on the programming of *put*. Nevertheless, this is not a trivial task since not all said *put* transformations satisfy the bidirectional laws. Moreover, $put : S \to V \to S$ is evidently harder to write than $get : S \to V$, as programmers cannot think solely in source-to-view terms but must consider more complex update strategies that synchronize view updates with existing sources. Therefore, the programmer shall be guided and restrained accordingly by a practical *put* programming approach.

In previous work [16] we have explored the design of a put-based language whose building blocks are injective $put\ s$ transformations (of type $V \to S$), for any source s. This motivates a specification style dual to that of get-based languages, by composing put-based combinators from view to source, but that allows users to control in a predictable way the behavior of the backward transformations that they write.

Notwithstanding, writing a program in a (get-based or put-based) bidirectional language typically implies describing the concrete steps that convert source databases into smaller views. This makes it impractical to express queries over eventually large databases, since one must explicitly describe how it ignores unrelated parts of the source.

In this paper, we investigate another approach: to flip the definition of *put* and develop a language of $put\ v$ functions (of type $S \to S$), for some view v. This parameterization on views motivates a *bidirectional update language* (in contrast to bidirectional transformation languages) in which programmers write bidirectional updates that modify an original database to embed some view information. Type-preserving updates are simpler to write than type-changing transformations in that one only needs to specify how the view information changes a small part of the source, leaving the remaining data fixed.

We demonstrate the feasibility of the novel approach by proposing the BiFluX language for the

bidirectional updating of XML databases. BiFluX is a bidirectional variant of Flux [6], a functional XML update language. We lift unidirectional Flux updates to bidirectional BiFluX updates that carry an additional notion of view. Reading BiFluX programs as *put* transformations will motivate several language extensions and demand a careful choice of semantics and extra static conditions on updates to ensure that view information is indeed put back to the source, thus building well-behaved lenses.

The rest of the paper is organized as follows. Section 2 gives a taste of our BiFluX bidirectional update language with a simple lens transformation written in two different styles. We give an overview of the features of the BiFluX language in Section 3 and demonstrate them with various practical examples in Section 4. Section 5 concludes and discusses directions for future work.

2 Motivation Example

This section begins by illustrating a typical example of a bidirectional transformation written in a standard bidirectional programming language. We then exemplify how the same transformation can be expressed in BiFluX, giving a flavor of the novel bidirectional updating style.

2.1 Bidirectional programming

Let us start by defining an address book regular expression type according to the following DTD.

```
<!DOCTYPE addrbook [
<!ELEMENT addrbook (person*)>
<!ELEMENT person (name,email+,tel)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT tel (#PCDATA)>
]>
```

An address book contains a list of persons, each one possessing a name, a non-empty list of emails and a telephone number. The following XML document encodes a database of three persons conforming to this type definition:

```
<addrbook>
  <person>
    <name>Tao Zan</name>
    <email>zantao@nii.ac.jp</email>
    <email>zantao007@gmail.com</email>
    <tel>+81-3-5841-7430</tel>
  </person>
  <person>
    <name>Hugo Pacheco</name>
    <email>hpacheco@nii.ac.jp</email>
    <email>hpacheco@di.uminho.pt</email>
    <tel>+81-3-5841-7411</tel>
  </person>
  <person>
    <name>Zhenjiang Hu</name>
    <email>hu@mist.i.u-tokyo.ac.jp</email>
    <email>hu@ipl.t.u-tokyo.ac.jp</email>
    <tel>+81-3-5841-7411</tel>
  </person>
</addrbook>
```

Now imagine that we want to summarize our source address book into a simpler view address book where people only have one email, and no telephone numbers, according to the following DTD. The purpose for this may be for instance to exhibit a minimal address book in an email application that has no concern for telephone addresses.

```
<!DOCTYPE addrbook [
<!ELEMENT addrbook (person*)>
<!ELEMENT person (name,email)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
]>
```

For this purpose, we would like to write a bidirectional transformation between the two source and view schemas, providing us a forward transformation that gets a simplified address book from a more complete address book and a backward transformation that synchronizes a simplified address book with an original address book that contains more information about each person. One possible way is to write this bidirectional transformation in the Boomerang string lens language [3]:

```
let ALPHA = [A-Za-z]+
let DOMAIN = [A-Za-z|.|-]+
let EMAIL = ALPHA . "@" . DOMAIN
let TEL = "+81" . "-" . [0-9] ."-"
        . [0-9]{4} ."-" . [0-9]{4}
```

```
let emails = (del ", " . del EMAIL)*
let comp = key ALPHA . copy ","
        . copy EMAIL . emails . copy ", "
        . ins TEL "+81-0-0000-0000"
let comps = copy "" | <comp> . ("\n" . <comp>)*
```

Since string lenses were not specifically designed for XML processing, to avoid distracting the readers from the essential we assume that the XML formats are flattened into a simpler format of strings where fields are separated by commas and people by newlines; an XML-compliant string lens would simply involve more parsing and pretty-printing boilerplate code. For example, an arbitrary source database would look like:

```
name1, email11, email12, tel1
name2, email2, tel2
```

The first four lines of our string lens define the regular expressions for alphabetical data, emails and telephone numbers. The lens *comp* processes each person (per each line): it copies the name to the view, then copies the first email while deleting remaining emails (*emails*) and at last deletes the telephone number; the separating commas are dealt with accordingly. A sequence of lines is handled by *comps*.

The *get* transformation of this lens for our source database then produces the following view address book.

```
<addrbook>
  <person>
     <name>Tao Zan</name>
     <email>zantao@nii.ac.jp</email>
  </person>
  <person>
    <name>Hugo Pacheco</name>
    <email>hpacheco@nii.ac.jp</email>
  </person>
  <person>
    <name>Zhenjiang Hu</name>
    <email>hu@mist.i.u-tokyo.ac.jp</email>
  </person>
</addrbook>
```

When writing lenses, programmers are invited to think in terms of the forward transformation, but in our running lens there are some more subtle details mixed together into a single lens design, namely **key**, angle brackets **<comp>** and **ins**. These additional combinators are used to provide extra information on how to derive a corresponding backward transformation strategy. Consider for example the modified view address book where we reorder Zhenjiang (while updating his email) and Tao and insert a new person John Doe:

```
<addrbook>
  <person>
    <name>Zhenjiang Hu</name>
    <email>hu@nii.ac.jp</email>
  </person>
  <person>
     <name>Tao Zan</name>
     <email>zantao@nii.ac.jp</email>
  </person>
  <person>
    <name>John Doe</name>
    <email>foo@bar.com</email>
  </person>
</addrbook>
```

The *put* transformation of our string lens, when invoked with this updated view and the original source, will produce the following updated address book:

```
<addrbook>
  <person>
    <name>Zhenjiang Hu</name>
    <email>hu@nii.ac.jp</email>
    <email>hu@ipl.t.u-tokyo.ac.jp</email>
    <tel>+81-3-5841-7411</tel>
  </person>
  <person>
     <name>Tao Zan</name>
     <email>zantao@nii.ac.jp</email>
     <email>zantao007@gmail.com</email>
     <tel>+81-3-5841-7430</tel>
  </person>
  <person>
    <name>John Doe</name>
    <email>foo@bar.com</email>
    <tel>+81-0-0000-0000</tel>
  </person>
</addrbook>
```

This *put* transformation has matched each person in the source with each person in the view by their names, and retrieved their additional source information for Zhenjiang and Tao that had contacts in

the original source, but creating a default telephone number for John Doe. In the lens program, the angled brackets `<comp>` identify lines (persons) as entries that can be reordered in the source and view, and `key` declares the unique key on which they must match to identity the same person; `ins` specifies the default telephone for newly created contacts.

2. 2 Bidirectional updating

The same bidirectional transformation can be written as a bidirectional update procedure in BiFluX. Instead of writing a get-based lens program, the programmer specifies a *put* update strategy directly as an update on the original address book format. Figure 1 shows the BiFluX code for our address book example.

To give an intuition, our update starts by defining the root procedure *addrbook* that accepts as arguments the original source and updated view. Then it focuses on a list of persons in the source address book according to the path $\$saddrbook/addrbook/person$ and a list of persons in the view address book according to the path $\$addrbook/addrbook/person$. Persons in the two lists are matched using an explicit matching condition, in this example the name of persons in both lists. Finally, if a person in the view matches a person in the source, its updated view email ($\$email$) must replace the first email ($email[1]$) in the source database; if a new unmatched person appears in the updated view, then a new contact with a default telephone is inserted in the source.

The main difference in comparison to the previous string lens is that the emphasis is now on writing a *put* transformation instead of a *get* transformation. This will allow a much more flexible and intuitive control over backward synchronization strategies, by making several put design choices explicit in the design of a bidirectional update; the unique forward query can be essentially derived from the source path expressions that connect selected source data with the view. Throughout this paper we will present more examples that attempt to practically corroborate this assumption. Moreover, since programmers write such *put* transformations in an higher-level update language, they only specify which parts of the source are to be updated with the updated view information. For instance, in the string lens version we had to explicitly delete trailing emails and the telephone of each person, whereas the bidirectional update version just concentrates on the updated information and leaves the remaining parts unchanged by definition.

3 A Bidirectional Update Language

In this section, we introduce our BiFluX language by describing its high-level, natural language syntax similar to other update languages like SQL, XQuery and (naturally) Flux. The complete syntax for BiFluX updates is shown in Figure 2.

3. 1 Flux-like syntax

At the outset, bidirectional BiFluX source updates look just like regular Flux source updates. Variables *Var* are written \$x, \$y, etc. The syntactic classes *Expr* and *Path* denote ordinary XQuery expressions and XPath paths, respectively. Statements *Stmt* include let-binding, conditionals, sequential composition, update operations *Upd* and procedures. A procedure has a name and a tuple of argument paths, but it usually takes only two arguments corresponding to the path to the current source focus and a path to the current view focus. (An update program is assumed to start with a predefined root procedure that is invoked with the original source and updated view XML documents, declared in two special reserved variables $\$source$ and $\$view$ that are always in the environment.)

An update operation *Upd* receives a path from the current source focus and, like in Flux, may

```
PROCEDURE addrbook($saddrbook, $addrbook) =
UPDATE $sperson IN $saddrbook/addrbook/person BY
{
  MATCH -> REPLACE email[1] WITH $email
| UNMATCHV -> CREATE VALUE <person> <name/> <email/> <email>foo@default.com</email> <tel/> </person>
}
FOR VIEW person[$name AS name, $email AS email] IN $addrbook/addrbook/person
MATCHING SOURCE BY $sperson/name VIEW BY $name
```

図 1 BiFluX update for the address book example.

```
Stmt    ::=  Upd [WHERE Expr]
         |   IF Expr THEN Stmt ELSE Stmt
         |   Stmt ; Stmt
         |   LET Pat := Expr IN Stmt
         |   CASE Expr OF { Cases }
         |   { Stmt }
         |   PROCEDURE Name '(' Path, ..., Path ')'
Upd     ::=  INSERT (BEFORE | AFTER) PathPat VALUE Expr
         |   INSERT AS (FIRST | LAST) INTO PathPat VALUE Expr
         |   DELETE [FROM] PathPat
         |   KEEP Path AS (FIRST | LAST)
         |   CREATE VALUE Expr
         |   REPLACE [IN] PathPat WITH Expr
         |   UPDATE PathPat BY Stmt
         |   UPDATE PathPat BY VStmt FOR VIEW PathPat [Match]

Cases   ::=  Pat → Stmt
         |   Cases '|' Cases
VStmt   ::=  VUpd '|' VUpd
         |   { VUpd }
VUpd    ::=  MATCH → Stmt
         |   UNMATCHS → Stmt
         |   UNMATCHV → Stmt
Match   ::=  MATCHING BY Path
         |   MATCHING SOURCE BY Path
         |                  VIEW BY Path
PathPat ::=  [Pat IN] Path
         |   Var [AS Type]
Pat     ::=  '(' Pat, ..., Pat ')'
         |   Var [AS Type] | Type
         |   Name[Pat, ..., Pat]
Type    ::=  Name | @Name | Name[Type]
         |   String | () | Type* | Type?
         |   Type+ | (Type)
         |   Type '|' Type
         |   Type, Type
```

図 2 Concrete syntax of BiFluX.

be either *singular* or *plural*. Singular updates are executed once for each node in the result set of the path; plural updates operate on the children of each selected value. Briefly, singular insertions (INSERT BEFORE/AFTER) insert a value before or after each selected node, while plural insertions (INSERT AS FIRST/LAST INTO) insert a value at the beginning or end of the child-list of the selected nodes. Singular deletions (DELETE) delete selected nodes, whereas plural deletes (DELETE FROM) delete only the child-lists of selected nodes. Singular replacements (REPLACE WITH) replace all selected nodes with the same expression, while plural replacements (REPLACE IN) replace the

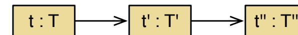

図 3 Update composition in Flux.

corresponding child-lists. The singular UPDATE *Stmt* BY operation applies a statement to each node in the result of *Path*.

3.2 Bidirectional updates

In spite of the fact that BiFluX updates strongly resemble Flux updates, their bidirectionality entails a subtly different semantics. A simple (unidirectional) Flux statement *Stmt* is read as type-

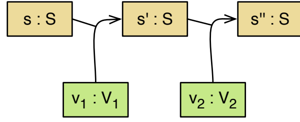

図 4 **Update composition in BiFluX.**

changing transformation: given an input tree t of type T, execute *Stmt* to update t, producing a target tree t' of some other type T'. In BiFluX, a (bidirectional) statement *Stmt* denotes a $put : V \to S \to S$ transformation, i.e., given an updated view of type V, it updates the contents of a source of type S with the updated view information, preserving the source type S. For this reason, we assume that BiFluX programs are defined for fixed source and view schemas, while Flux programs are defined for a fixed source schema and produce a corresponding target schema.

This justifies a different notion of composition. Imagine that we compose two Flux statements $Stmt; Stmt'$ (Figure 3). This can be read as: given an initial tree t of type S, apply statement *Stmt* to update s into an intermediate tree u of type U, followed by applying *Stmt'* to compute the resulting updated tree t with final type T. Since a BiFluX update can not change the source type, two composed statements $Stmt; Stmt'$ update source trees of same type, but embedding different view information (Figure 4): *Stmt* updates an original source s of type S with a view v_1 of type V_1, producing an intermediate source s', and $Stmt'$ updates s' with different view information v_2 of type V_2, producing a resulting updated source s''.

3.3 Conditionals and where conditions

In Flux, conditionals IF *Expr* THEN *Stmt* have a single then-statement, and its semantics simply applies *Stmt* if the condition *Expr* evaluates to true or returns the original source otherwise. McCarthy conditionals IF *Expr* THEN *Stmt* ELSE $Stmt'$ can be defined using composition IF *Expr* THEN *Stmt*; IF $not(Expr)$ THEN $Stmt'$ – if the condition is verified, then the first statement updates the source via *Stmt* and the second statement does nothing, and vice-versa.

However, the same reasoning can not be applied in BiFluX because, to ensure that view information is embedded into the source, an update that does not satisfy the condition must fail (except for empty views). This is the reason why we introduce a primitive if-then-else conditional.

BiFluX considers two classes of where conditions for source-based filtering or view-based filtering. A source-based condition does not refer to view variables and performs like a where conditions in Flux, by filtering only the nodes that satisfy a specify criteria from the source path. On the other hand, a view-based condition does not refer to source variables and imposes a constraint on the nodes from a view path: since we cannot filter out any view information (it must be embedded in the source), the update will fail if the condition is not satisfied.

3.4 Pattern matching

In contrast to Flux, that does not support defining updates by matching tree patterns agains the source data, pattern matching is a key design decision of BiFluX. The typical way to write an update in Flux (and BiFluX) is to define a path on the original source that selects the particular source information to be modified. But to ensure that a bidirectional update is well-behaved, we can not apply the same kind of lossy queries to the view. Therefore, apart from very restrictive injective paths, views in BiFluX must always be decomposed via pattern matching, hence the added optional patterns for source and view paths of update operations.

This choice enables our language to statically

guarantee that view information is embedded into the source, by simply guaranteeing that all view variables in a pattern are put back to the updated source. (Our pattern matching algorithm follows familiar lines of XML transformation languages and is implemented via structural subtyping [13].) To keep pattern matching as simple as possible, we avoid the use of non-linear variables [12] by supporting only patterns that are sequences of variables. Alternative patterns can still be defined by using CASE statements, that behave in the same way as multiple patterns in functional languages such as Haskell.

3.5 Source-view alignment

Thus far, all the update operations in our language perform *in-place* updates; they iterate over a source sequence (defined as path on the source), updating each and every node in the sequence accordingly. To make it a practical bidirectional updating language, we extend BiFluX with a new update operation UPDATE $PathS$ BY $VStmt$ FOR VIEW $PathV$ that supports iterating over a view sequence. Informally, this combinator applies $VStmt$ (what we call a view statement) to each node in the result of a view path $PathV$, generating an updated source sequence that replaces the original source sequence computed by the source path $PathS$. This kind of behavior can not be done in-place. Therefore, we evaluate the source path as a lens from the current source focus to the result of the path, and use lens composition to synchronize the updated sequence with the source focus. (In our implementation, we interpret the source paths of update operations as lenses.)

Furthermore, the view sequence may not have the same number of elements as the original source sequence or original source elements may appear have been reordered in the view. This forces an update strategy to match the elements of the source and view sequences in order to identify the source elements to which view updates must be translated. This *alignment* problem is well-known in bidirectional transformations [3] [2] [15].

We allow users to flexibly specify arbitrary alignment strategies by defining a MATCHING condition that defines a (not necessarily unique) key that intuitively specifies the criteria for source and view elements to match; if a matching condition is not specified, a default positional matching is assumed. Then a view statement $VStmt$ supports three special cases: MATCH for two matching source and view values; UNMATCHV for unmatched view values for which there is no source value; and UNMATCHS for unmatched source values for which there is no view. Each of these cases carries an corresponding statement. For UNMATCHV, since there is no original source, such statement must begin with a special CREATE VALUE $Expr$ update that generates a default source value. For UNMATCHS, the statement must either be of the form DELETE . (that forgets an original source) or begin with a special KEEP . AS FIRST/LAST update (that inserts an "old" source element as the first or last element in the updated source). As an additional restriction to use KEEP, programmers must ensure manually that the kept source value does not satisfy the where conditions on the source. This naturally implies that KEEP can only be used when a guarding condition is applied to the source. The reason for this restriction is that, when additional elements are added to an updated source sequence, the system must clearly know the underlying motives for programmers to want to recover such elements. Note that an automatically derived forward transformation must ignore them as long as they do not belong to the view.

By default, if the conditions are unspecified we assume that unmatched sources are ignored from the updated source and unmatched views generate

default source values generated from the structure of the source type.

4 Bidirectional Updating Examples

In this section, we demonstrate how diverse bidirectional transformation scenarios can be specified as bidirectional updates, in particular using BiFluX. These examples illustrate the spirit of bidirectional updating, and highlight the power and programmability of our bidirectional update language in specifying various update strategies.

4.1 XML transformation

We can also write typical bidirectional XML transformations as bidirectional updates. A famous bidirectional XML transformation language is biXid [12]. Bidirectional programs in biXid specify forward and backward conversions between pairs of XML formats. As an example, consider a bidirectional transformation between the Netscape and XBEL browser bookmark formats, written in biXid as the following biXid transformation.

```
relation top =
  html[head[String],
       body[h1[var t as String], dl[var nc]]]
<->
  xbel[title[var t as String], var xc]
  where
   contents(nc, xc)

relation contents =
  (var nb | var nf)*
<->
  (var xb | var xf)*
  where
    bookmark(nb, xb),
    folder(nf, xf)

relation bookmark =
  dt[a[@href[var url as String],
       var title as String]]
<->
  bookmark[@href[var url as String],
           title[var title as String]]

relation folder =
  dd[h3[var title as String], dl[var nc]]
<->
  folder[title[var title as String], var xc]
  where
    contents(nc, xc)
```

Both bookmark formats are loosely equivalent and contain a general title (`h1` or `title`) and a sequence of bookmarks (`dt` or `bookmark`) or folders (`dd` or `folder`), where folders may recursively contain sequences of bookmarks or folders.

The biXid language relies on pattern matching to decompose the source and target formats, and defines their conversions in a programming by relation paradigm, that makes critical use of ambiguity and non-linear variable bindings. Unlike lenses, biXid transformations do not have a primordial transformation direction and make full use of ambiguity: a document in one side may have multiple representations in the other side, or data in any side may not be related in any way to the opposite side; biXid solves ambiguity by arbitrarily choosing one of many possible representations or generating arbitrary data when necessary. For example, in the *top* relation `head` elements in the Netscape format are not related with XBEL, and so a backward transformation will "loose" the original source head for an arbitrarily generated one. In BiFluX, we can deal with such ambiguity (for the asymmetric case of lenses) by allowing users to choose from many possible representations, recover data from the original source, or create their own default data.

The above biXid transformation can be encoded in BiFluX as shown in Figure 5. Apart from intricacies bound to the disparate styles of both languages, the BiFluX update resembles very much the biXid transformation. The *top* procedure decomposes the source into head and body (with a title $\$h1$ and a sequence $\$nc$ of `dt`s or `dd`s) and the view into a title $\$t$ and a sequence $\$xc$ of `bookmark`s or `folder`s. Since the head in the source is not re-

```
PROCEDURE top($html,$xbel) =
        UPDATE html[head[String], body[$h1 AS h1, dl[$nc AS (dt|dd)*]]] IN $html BY
                { REPLACE IN $h1 WITH $t ; contents($nc,$xc) }
        FOR VIEW xbel[title[$t AS String], $xc AS (bookmark|folder)*] IN $xbel

PROCEDURE contents($nc,$xc) =
        UPDATE $nc BY
        {
                CASE $v OF
                { bookmark[@href[$url AS String], title[$title AS String]]
                          -> REPLACE . WITH <dt> <a href={$url}>{$title}</a> </dt>
                | folder[title[$title AS String], $fxc AS (bookmark|folder)*]
                          -> REPLACE IN h3 WITH $title ; contents(./dl/*,$fxc)
                }
        }
        FOR VIEW $v IN $xc
```

図 5　**BiFluX update for the bookmark example.**

lated to the view, we don't updated it (but the original head is preserved unlike in biXid). (Note that in BiFlux the same behavior would not be possible for the view, as view patterns must fully decompose the view into a set of variables that shall be embedded into the source.) Then *top* replaces the source $\$h1$ with $\$t$ and invokes *contents* to update the remaining sequences.

The *contents* relation in the biXid transformation makes use of non-linear variables, i.e., variables that are instantiated multiple times, to match `dt`s and `dd`s in the source with `bookmark`s and `folder`s in the view, respectively. Since only support linear pattern variables in BiFluX, we handle each view element using a case expression: when it matches the structure of a bookmark, we generate a source `dt` element with the bookmark's url and title; otherwise, if it matches the structure of a folder, we generate a source `dd` element with the folder's title and a `dl` with recursively computed contents by invoking *contents* with the all the elements under the *dl* and the elements from the view folder.

For an execution example, consider a source bookmark tree represented in the Netscape format

```
<html>
 <head>My Bookmarks</head>
 <body><h1>my bookmarks</h1>
  <dl><dt><a href="foo.com">Foo's</a></dt>
   <dd><h3>my folder</h3>
     <dl><dt><a href="stefanzan.com">stefanzan</a>
         </dt></dl>
   </dd>
   <dt><a href="bar.edu">Bar's</a></dt>
  </dl>
 </body>
</html>
```

and a different view bookmark tree represented in the XBEL format:

```
<xbel>
 <title>NII bookmarks</title>
  <folder>
    <title>National Institute of Informatics
    </title>
    <bookmark href="http://www.nii.ac.jp/en">
       <title>English</title>
    </bookmark>
    <bookmark href="http://www.nii.ac.jp">
       <title>Japanese</title>
    </bookmark>
  </folder>
  <folder><title>my folder</title>
          <bookmark href="stefanzan.com">
             <title>stefanzan</title>
          </bookmark>
  </folder>
  <bookmark href="bar.edu">
      <title>Bar's</title>
  </bookmark>
</xbel>
```

Finally, we can run our BiFLuX program to update the source bookmark tree using the view bookmark tree:

```
<html>
 <head>My Bookmarks</head>
 <body><h1>NII bookmarks</h1>
  <dl><dd>
       <h3>National Institute of Informatics</h3>
       <dl>dt>
       <a href="http://www.nii.ac.jp/en">English</a>
       <a href="http://www.nii.ac.jp">Japanese</a>
      </dt>
     </dl>
   </dd>
   <dd><h3>my folder</h3>
     <dl><dt>
       <a href="stefanzan.com">stefanzan</a>
     </dt></dl>
     <dt><a href="bar.edu">Bar's</a></dt>
   </dd>
  </dl>
 </body>
</html>
```

After updating, despite the structure of the view XBEL tree is fully propagated to the source, the updated source bookmark tree is not created afresh from the view, unlike in biXid. For example, the original header “My Bookmarks” is preserved.

4. 2 Flexible alignment

Many bidirectional programming approaches are *state-based*, in the sense that *put* transformations only consider original source and updated view values, as opposed to *operation-based* where some knowledge of the exact changes that led to the update result is also recorded. Since this formulation provides little knowledge about the actually performed updates, *put* must align source and view values in order to guide the propagation of view updates to corresponding source elements. Some existing alignment-based bidirectional lens languages [3] [2] [15] promote parameterizing lens programs with user-specified heuristics for calculating alignments, and devise particular bidirectional transformation primitives that reuse such alignment information to refine *put* behavior. Similarly, our UPDATE *PathS* BY *VStmt* FOR VIEW *PathV* update operation allows users to explicitly declare how to align source and view sequences using arbitrary paths.

For example, suppose that we have a source database that contains two levels of nested structure: a document is constituted by a list of sections, that have a title, a paragraph and a list of subsections, where a subsection has itself a title and a paragraph:

```
<sections>
  <section>
    <title>Grand Tours</title>
      <paragraph>
        The grand tours are major cycling ...
      </paragraph>
    <subsection>
      <title>Giro d'Italia</title>
      <paragraph>
        The Giro is usually held in May ...
      </paragraph>
    </subsection>
  </section>
  <section>
     <title>Classics</title>
     <paragraph>
       The classics are one-day cycling ...
     </paragraph>
     <subsection>
       <title>Milan-San Remo</title>
       <paragraph>
         The Spring classic is held in ...
       </paragraph>
     </subsection>
  </section>
</sections>
```

A view definition consists in deleting all paragraphs and performing some XML-specific structural changes like changing titles from child nodes to attributes. Consider the following updated view document.

```
<secs>
  <sec title='Classics'>
  </sec>
  <sec title='Olympic Competitions'>
    <subsec title='2008 Summer Olympics'/>
    <subsec title='2012 Summer Olympics'/>
  </sec>
```

```
PROCEDURE sections($sections, $secs) =
UPDATE $section IN $sections/sections/section BY
{
  MATCH -> subsections($section/subsection,$subsecs)
| UNMATCHV -> CREATE VALUE <section> <title/> <paragraph>this section is new</paragraph> </section>
}
FOR VIEW sec[$title AS @title, $subsecs AS subsec*] IN $secs/secs/sec
MATCHING SOURCE BY title/text() VIEW BY $title/string()

PROCEDURE subsections($subsections, $subsecs) =
UPDATE $subsection IN $subsections BY
{
  MATCH -> {}
| UNMATCHV -> CREATE VALUE
                    <subsection>
                      <title/>
                        <paragraph>this subsection is new</paragraph>
                     </subsection>
}
FOR VIEW $subsec IN $subsecs
MATCHING SOURCE BY title/text() VIEW BY @title/string()
```

図 6 BiFluX update for the sections example (local subsection alignment).

```
  <sec title='Grand Tours'>
    <subsec title='Milan-San Remo'/>
    <subsec title='Giro d'Italia'/>
  </sec>
</secs>
```

In comparison to the original source: the order of the sections “Grand Tours” and “Classics” is reversed; “Milan-San Remo” is now a subsection of “Grand Tours”; and a new section “Olympic Competitions” has been added to the view.

Figure 6 shows the BiFluX program for this example, that contains two procedures. The root procedure *sections* updates the content of `sections` in the source with the content of `secs` in the view, matched by their titles. If two sections match, we just update their subsections by calling the auxiliary procedure *subsections*, otherwise we create a new source section with a default paragraph. Note that we do not need to update source paragraphs as they are kept unchanged. The *subsections* procedure updates `subsections` with `subsecs` in a similar way.

Executing this update with the above source and updated view documents yields the following updated source:

```
<sections>
  <section>
     <title>Classics</title>
     <paragraph>
       The classics are one-day cycling ...
     </paragraph>
  </section>
  <section>
    <title>Olympic Competitions</title>
    <subsection>
       <title>2008 Summer Olympics</title>
       <paragraph>this section is new<paragraph>
    </subsection>
    <subsection>
       <title>2012 Summer Olympics</title>
       <paragraph>this section is new<paragraph>
    </subsection>
  </section>
  <section>
    <title>Grand Tours</title>
      <paragraph>
        The grand tours are major cycling ...
      </paragraph>
    <subsection>
```

```
LET $subs := $source/sections/section/subsection[title/text() = $subsec/@title/string()] IN
LET $par := if $subs then string-join($subs/paragraph,' ') else 'this section is new' IN
CREATE VALUE <subsection> <title/> <paragraph>{$par}</paragraph> </subsection>
```

図 7 **BiFluX update for the sections example (global subsection alignment).**

```
      <title>Milan-San Remo</title>
      <paragraph>this section is new</paragraph>
    </subsection>
    <subsection>
      <title>Giro d'Italia</title>
      <paragraph>
        The Giro is usually held in May ...
      </paragraph>
    </subsection>
  </section>
</sections>
```

The updated source correctly reorders the sections in the source according to the view, with updated subsections and original retrieved from the source paragraphs for matched sections. However, the behavior is slightly different for subsections: the update has recovered the original paragraph for "Giro d'Italia", but not for "Milan-San Remo", that was originally a subsection of "Classics". This happens because view subsections are only locally aligned with source subsections under the same section.

Therefore, a programmer may want to also retrieve the paragraphs of subsections that have been moved to different sections. He can do so by changing the statement after UNMATCHV in the *subsections* procedure as shown in Figure 7.

The new code now retrieves every `subsection` in the original source that has the same title as the current view `subsec`, and creates a new paragraph by concatenating all paragraphs of selected source subsections with the XPath function $string-join$; the XQuery if-then-else condition returns the a default paragraph if no source subsections are found. Running our example again, we would see "The Spring classic is held in ... " as the paragraph of "Milan-San Remo" instead of a default.

4.3 User control over update strategies

A typical query for defining views of databases is selection, which filters rows satisfying a particular condition. In BiFluX, we can write various update strategies for selection queries.

Suppose that we have a source XML document that contains persons with a name and a city:

```
<people>
  <person>
    <name>Hugo</name><city>Tokyo</city>
  </person>
  <person>
    <name>Sebastian</name><city>Kiel</city>
  </person>
  <person>
    <name>Zhenjiang</name><city>Tokyo</city>
  </person>
</people>
```

Assuming an updated view database with only the names of people who live in 'Tokyo'

```
<fromtokyo>
  <name>Zan</name>
  <name>Zhenjiang</name>
</fromtokyo>
```

we can define a BiFluX program that updates the original database with a new view as shown in Figure 8. The *people* procedure starts by selecting (using a where condition) only the persons in the original source that live in 'Tokyo' and matches them with names of people in the view. Then, it copies matched view tokyolites to the source, creates new view tokyolites with city 'Tokyo' in the source (to satisfy the filtering condition) and unmatched source tokyolites are ignored (by default).

This update will yield an updated source as follows when applied to our source and view examples:

```
<people>
```

```
PROCEDURE people($people, $fromtokyo) =
UPDATE $sperson IN $people/people/person BY
{
  MATCH -> {}
| UNMATCHV -> CREATE VALUE <person> <name/> <city>Tokyo</city> </person>
}
FOR VIEW $vname IN $fromtokyo/fromtokyo/name
MATCHING SOURCE BY name VIEW BY .
WHERE city/text() = 'Tokyo'
```

図 8 **BiFluX update for the people example (delete source tokyolites).**

```
  <person>
    <name>Zan</name><city>Tokyo</city>
  </person>
  <person>
    <name>Sebastian</name><city>Kiel</city>
  </person>
  <person>
    <name>Zhenjiang</name><city>Tokyo</city>
  </person>
</people>
```

The updated source now contains "Zan" and "Zhenjiang" living in "Tokyo", as they come from the updated view, and "Sebastian" living in "Kiel", that existed in the original source and has not been updated (since he did not live in "Tokyo"). Since "Hugo" lived at 'Tokyo' in the original source and does not appear in the updated view, he is consequently deleted.

Alternatively, we could have reasonably wanted to keep "Hugo" in the updated source even if he had been deleted in the view. For instance, if "Hugo" moved to a different city and has been deleted from that city's municipal records. In fact, he could be kept in the updated source, as long as he is moved to a different city, say, "Kyoto".

This alternative behavior is usually difficult to specify using traditional bidirectional language, because they assume fixed update strategies for selection. In BiFluX, we can easily specify it using a KEEP . update operation (Figure 9). In the new version, unmatched source people like "Hugo" will be inserted in the updated source (note that KEEP also requires a position relative to the updated view where such persons are to be inserted), with their city original city (namely "Tokyo") replaced with "Kyoto":

```
<people>
  <person>
    <name>Zan</name><city>Tokyo</city>
  </person>
  <person>
    <name>Sebastian</name><city>Kiel</city>
  </person>
  <person>
    <name>Zhenjiang</name><city>Tokyo</city>
  </person>
  <person>
    <name>Hugo</name><city>Kyoto</city>
  </person>
</people>
```

5 Conclusion

Bidirectional transformations play an important role in synchronizing and exchanging information between different data formats, and have gained significant attention and applicability in the last decade. Despite the currently wide offer on bidirectional programming languages, research on bidirectional transformations has still not matured for deployment at a larger scale and a unifying framework for bidirectional transformations is yet to emerge.

The main reason for this is the still very magical nature of existing solutions, what makes users justifiably less confident to use bidirectional technologies. We believe that a feasible answer to

```
PROCEDURE people2($people, $fromtokyo) =
UPDATE $sperson IN $people/people/person BY
{
  MATCH -> {}
| UNMATCHV -> CREATE VALUE <person> <name/> <city>Tokyo</city> </person>
| UNMATCHS -> KEEP . AS LAST ; REPLACE IN city WITH 'Kyoto'
}
FOR VIEW $vname IN $fromtokyo/fromtokyo/name
MATCHING SOURCE BY name VIEW BY .
WHERE city/text() = 'Tokyo'
```

図 9 BiFluX update for the people example (moved source tokyolites).

these problems lies in put-based bidirectional programming approaches, that enable programmers to potentially specify the complete behavior of bidirectional transformations by directly writing *put* transformations, cutting on the magic and unpredictability at the cost of more responsibility.

In this paper, we have proposed an hybrid form of put-based programming (that we have named bidirectional updating) that invites programmers to write bidirectional transformations not directly as *put* transformations from source to view but as more intuitive and readable programs that specify how to update sources with view information, while preserving the advantages of put-based programming (at a reasonable expressiveness cost). To the best of our knowledge, the design and motivation of bidirectional update languages is new, and unveils a promising style to inspire future bidirectional transformation frameworks.

As a demonstration of the feasibility of our novel approach, we have designed BiFluX, a bidirectional update language for XML data. In this paper we have given a general informal overview of our language through differently motivated bidirectional scenarios. BiFluX is ongoing work and contains much more under the hood. Our prototype implementation already supports all the examples discussed in this paper: it receives source and view XML documents and DTDs, and animates an higher-level BiFluX program by defining its semantics as lower-level put-based lenses. To be able to read BiFluX programs as bidirectional transformations, several static conditions (not fully discussed in this paper) need to be imposed on paths, patterns, statements and updates; in some cases (like view-based where conditions), the derived well-behaved *put* transformations may be partial.

As future work, we plan to clarify the semantics of our language and provide more static guarantees to prove the totality of BiFluX updates for particular domains. We also intend to investigate the design of bidirectional update languages for other data domains, such as relational and graph data.

参 考 文 献

[1] François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems (TODS)*, 6(4):557–575, December 1981.

[2] Davi M.J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: alignment and view update. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 193–204, New York, NY, USA, 2010. ACM.

[3] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: resourceful lenses for string data. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 407–419,

New York, NY, USA, 2008. ACM.

[4] Aaron Bohannon, Benjamin C Pierce, and Jeffrey A Vaughan. Relational lenses: a language for updatable views. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 338–347. ACM, 2006.

[5] Peter Buneman, James Cheney, and Stijn Vansummeren. On the expressiveness of implicit provenance in query and update languages. *ACM Transactions on Database Systems (TODS)*, 33(4):28, 2008.

[6] James Cheney. Flux: functional updates for xml. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 3–14, New York, NY, USA, 2008. ACM.

[7] Krzysztof Czarnecki, J Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Theory and Practice of Model Transformations*, pages 260–283. Springer, 2009.

[8] Sebastian Fischer, Zhengjiang Hu, and Hugo Pacheco. "Putback" is the Essence of Bidirectional Programming. Technical Report GRACE-TR 2012-08, GRACE Center, National Institute of Informatics, December 2012.

[9] J Nathan Foster, Michael B Greenwald, Jonathan T Moore, Benjamin C Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3):17, 2007.

[10] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1-2):89–118, 2008.

[11] Zhenjiang Hu, Andy Schurr, Perdita Stevens, and James F Terwilliger. Dagstuhl seminar on bidirectional transformations (bx). *ACM SIGMOD Record*, 40(1):35–39, 2011.

[12] Shinya Kawanaka and Haruo Hosoya. bixid: a bidirectional transformation language for xml. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, ICFP '06, pages 201–214, New York, NY, USA, 2006. ACM.

[13] Kenny Zhuo Ming Lu and Martin Sulzmann. An implementation of subtyping among regular expression types. In *Programming Languages and Systems*, pages 57–73. Springer, 2004.

[14] Hugo Pacheco and Alcino Cunha. Generic point-free lenses. In *Mathematics of Program Construction*, pages 331–352. Springer, 2010.

[15] Hugo Pacheco, Alcino Cunha, and Zhenjiang Hu. Delta lenses over inductive types. *Electronic Communications of the European Association of Software Science and Technology*, 49, 2012.

[16] Hugo Pacheco, Zhenjiang Hu, and Sebastian Fischer. Combinators for "putback" style bidirectional programming. Technical report, GRACE Center, National Institute of Informatics, July 2013.