A Java Library for Bidirectional XML Transformation

Dongxi Liu Zhenjiang Hu Masato Takeichi
Kazuhiko Kakehi Hao Wang

We propose a Java library BiXJ for bidirectional XML transformation. A bidirectional transformation generates target XML documents from source XML documents in forward transformations, and updates source documents in backward transformations by reflecting back modifications on target documents. The benefit of using BiXJ is that users can get the corresponding backward transformation automatically just by writing one forward transformation. BiXJ has addressed several limitations of the existing bidirectional transformation languages, and can be used for general purpose XML processing. For example, bidirectional XPath expressions written in BiXJ can be used to locate and extract data from XML documents. To validate the usability and expressiveness of BiXJ, we have bidirectionalized some typical examples of XQuery and XSLT with this library. The results of these experiments are promising.

1 Introduction

XML is widely used as the de facto standard format of data exchange or repository in the Internet environment. XML documents often need to be transformed for different reasons. For example, an XML file is transformed into HTML format for displaying in web browsers, or transformed into a small XML file containing only interesting data for users. In some cases, the target XML documents generated by transformations are probably modified by users to update some data or to correct some errors, and it is desirable that these modifications on the target documents can be reflected back into source documents.

However, the current popular XML transformation languages, such as XSLT [1] and XQuery [2], perform transformation only in one direction, generating target documents from source documents. As a result, modifications on target documents cannot be reflected back into the source documents without using other separate mechanisms.

This work presents a Java library BiXJ for bidirectional XML transformation. A bidirectional transformation program can be executed in two directions: forward direction and backward direction. The forward transformation transforms a source XML document into a target document, while the backward transformation transforms the target document (probably modified) together with the original source document into the updated source document. After the backward transformation, the modifications in the target document will be reflected back into the source document. When writing BiXJ transformations, users just need to consider how to generate the expected target documents from source documents, which is a similar programming paradigm as that of using XSLT and XQuery. With BiXJ, no extra efforts or other separate mechanisms are needed for users to update source documents after target documents are modified. The only work they need to do is to execute backward their BiXJ programs.

In BiXJ, each language construct is defined with
two meanings: one for forward transformations and the other for backward transformations. This style of bidirectional transformation techniques has been proposed in the literature [3][4]. However, the languages in [3][4] are both domain-specific. The language in [3] is designed for synchronizing tree-structured data, and the language in [4] is mainly used in an editor for editing tree-structured data. Because of their domain-specific purposes, these languages have several limitations when they are used as general purpose XML processing languages.

First, these languages do not take the standard XML data model [5], which is a tree with ordered labeled nodes or non-labeled data nodes (i.e., text contents). The language in [3] takes a tree-structured data model that only allows unordered labeled nodes without repeated labels, and the language in [4] uses a model which does not allow non-labeled data nodes.

Second, a transformation in these languages can only generate one target XML element, which is too restrictive for XML transformations. For example, if a book element contains one title element and one or more author elements, then extracting the author information from this book element will return a sequence of author elements. XQuery and XSLT allow element sequences as their transformation results.

Third, these languages use their own specific methods to destruct tree-structured data, which are probably unnatural for users who have been familiar with the existing XML transformation languages. For example, the construct hoist in [3] and hoistX in [4] return a child element if the source document contains only this element as its child. However, in XML transformation, a widely accepted way is to use XPath [6] to locate and extract data from XML documents, which is used in both XQuery and XSLT.

Fourth, the view updating semantics of these languages is too restrictive. This semantics gives conditions on whether a bidirectional transformation is well-behaved [3]. That is, whether it can correctly reflect modifications on target documents back into source documents. Under this restrictive semantics, to guarantee the well-behavedness of bidirectional transformations, some reasonable modifications on target data or the expressiveness of transformation languages have to be restricted. This problem is discussed more in Section 4.

In this work, BiXJ is designed to extend the expressiveness and usability of these existing languages, so that it can be used for general purpose XML processing. In order to design such library, we have to address the first three limitations discussed above. However, we are in a dilemma because the fourth limitation does not allow us to make too much extensions, otherwise the transformations written in the extended language are probably not well-behaved. Our approach in this work is to seek a more flexible view updating semantics for bidirectional transformations, and then under this semantics to extend the existing languages. To demonstrate its expressiveness and usability, we have used BiXJ to bidirectionalize some typical examples of XQuery and XSLT.

Since this library is written in Java, it can be applied in any case where Java is used to process XML documents, and moreover bidirectional transformations are expected. For example, Java is one of the most popular languages to implement web service [7], so with this library, a Java program can generate the interesting XML data for clients, and also it can update the original source XML data on web servers after receiving the modified data from clients. On the other hand, the transformation constructs in this library can also be represented as XML elements and then interpreted using the corresponding Java classes at runtime. This provides a convenient way for users to write BiXJ programs if they do not know Java or their programs are not used together with Java programs. The Java interface of BiXJ transformations and their XML representations will be described in Section 3.

Updating operations on target documents generally include insertions, deletions and modifications [8]. In this work, we only consider modifications. We allow modifications both on text contents and element tags, while in [8], only modifications on text contents are allowed.

The main contribution of this paper is the design and implementation of a Java library BiXJ for bidirectional XML transformation, which addresses the limitations of the existing bidirectional transformation languages from the following aspects:

- BiXJ uses the standard XML data model and allows to construct or destruct XML documents in a similar way as that in current XML
transformation languages. For example, the child and descendant axes of XPath are supported in BiXJ, and with them we have demonstrated how to make XPath expressions bidirectional.

- We define a more flexible semantics of bidirectional transformations, which underlies the well-behavedness of BiXJ transformations without restricting its expressiveness and reasonable modifications on target documents.
- The transformations written in BiXJ can be updated during backward executions because some modifications on target documents should be reflected back into transformations rather than into source documents.
- BiXJ provides flexible ways to write transformations, i.e., in Java or in XML, and it is expressive enough to bidirectionalize typical use cases of XQuery and XSLT.

The remainder of this paper is organized as follows. Section 2 gives a practical scenario of using bidirectional XML transformation. Section 3 overviews the library BiXJ. Section 4 discusses some issues in designing BiXJ. Section 5 presents the definitions of the BiXJ transformations. Section 6 gives examples of bidirectionalizing typical XQuery and XSLT transformations. Section 7 discusses the related work and Section 8 concludes the paper and gives the future work.

2 A Scenario for Bidirectional Transformations

The XML document in Figure 1 will be used as the source data for the scenario. The root element, tagged by books, contains three child book elements. Each book element contains the child elements title, author, year and publisher.

Suppose that this XML document is stored on a web server. This server allows authors to select the books they wrote for checking and correcting error information, or ordinary customers to request the information of their interesting books. For the requests from authors, the server performs forward transformations and returns elements with the tag mybooks, while for the requests of ordinary customers it generates elements tagged by interestingbooks.

For example, if the author Tom wants the title and publisher information of his books, the server should reply with the following document as the result of transforming forward the source document:

```
<mybooks>
  <book>
    <title>Computer Programming</title>
    <author>Tom</author><year>2003</year>
    <publisher>Now Century</publisher>
  </book>
  <book>
    <title>Data Structure</title>
    <author>Peter</author><year>2005</year>
    <publisher>Great Press</publisher>
  </book>
  <book>
    <title>Computer Graphics</title>
    <author>Tom</author><year>1999</year>
    <publisher>ACM Press</publisher>
  </book>
</mybooks>
```

Unfortunately, Tom finds several errors in the above document: “Now” in the publisher element of the first book should be “New”, and “Graphics” in the title element of the second book should be “Graphics”. After correcting these errors, Tom sends this changed document back to the server and the server will perform a backward transformation to update the source document.

Later, a customer asks the server to list the ti-
public interface XAction{
    public List<TgtElement> tranForward(List<SrcElement> sd);
    public List<SrcElement> tranBackward(List<SrcElement> sd, List<TgtElement> td);
    public CodeElement dump();
}

Fig. 3 The Java Interface of BiXJ

3 Overview of BiXJ

Our library is built on JDOM [9], which provides an easy way to process XML documents in Java. The class Element in JDOM abstracts the elements in XML documents and operations on them. In this work, we refine the class Element into three subclasses SrcElement, TgtElement and CodeElement to help users to understand the roles of elements in transformation. The first two subclasses correspond to the source and target elements, respectively, and the third subclass will be introduced later.

The interface of BiXJ transformations is defined in Figure 3. In this interface, the methods tranForward and tranBackward perform the forward and backward transformations, respectively. However, unlike the corresponding get and put in [3], these two methods take lists of elements as their source and target data.

The method dump is specific to this library. For a runtime object of type XAction, this method returns an XML fragment of type CodeElement that represents this transformation object. In BiXJ, transformations can be updated during backward executions, where the state of transformation objects (i.e. some fields) are changed. The dump method is used to output the updated transformations to users. In Section 5, we will see some constructs in BiXJ that update transformations in backward executions, such as XConst. In addition, the class CodeElement has a method (not introduced in this paper) to interpret a code element as a transformation object during runtime.

We provide two ways to represent BiXJ transformations, i.e., in Java classes or in code elements. Our design purpose is that Java classes will be used by Java programmers when writing Java programs to implement bidirectional XML transformations, and code elements will be used by users who do not know Java or their applications are not Java applications. We can imagine the following interesting applications for the second representation:

- Code elements can be used as the intermediate code when translating the expressions of high level transformation languages into bidirectional transformations in BiXJ. The XML representation is more compact, so the translation result is more readable. In the examples of Section 6, code elements are used to describe the typical use cases of XQuery and XSLT.
- Code elements provide a means of modifying transformation objects at runtime. After a transformation object is dumped into a code element, this element will be processed like an ordinary XML element, and then it can be interpreted as a new transformation object again. This technique can help implement self-adjusting transformations. As we will see, the implementation of xmap uses this technique to update its argument transformation.
- Code elements can be incorporated into XML documents to construct Programmable Structured Documents (PSD) [10]. For example, suppose that there is a book document containing a child element for the table of contents and one or more child chapter elements, and the table of contents comprises all chapter titles. In PSD framework, the table of contents comprises all chapter titles.
Table 1 Classes in BiXJ

<table>
<thead>
<tr>
<th>Class Constructor</th>
<th>Code Elements</th>
<th>Description (Forward Meaning)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSeq($x_1, ..., x_n$)</td>
<td>&lt;xseq&gt;$c_1$...$c_n&lt;/xseq&gt;</td>
<td>Applies $x_1$, ...,$x_n$ sequentially.</td>
</tr>
<tr>
<td>XMap($x'$)</td>
<td>&lt;xmap&gt;$c'&lt;/xmap&gt;</td>
<td>Applies $x'$ to each element in the source data.</td>
</tr>
<tr>
<td>XZip($x_1, ..., x_n$)</td>
<td>&lt;zip&gt;$c_1$...$c_n&lt;/zip&gt;</td>
<td>Applies $x_1$, ...,$x_n$ to the corresponding child of the source data, which must be an element.</td>
</tr>
<tr>
<td>XIf(pred, $x_1$, $x_2$)</td>
<td>&lt;if&gt;$c_{pred} c_1 c_2&lt;/if&gt;</td>
<td>Applies $x_1$ to the source data if this data satisfies $pred$, otherwise applies $x_2$.</td>
</tr>
<tr>
<td>XID()</td>
<td>&lt;xid /&gt;</td>
<td>Identity transformation.</td>
</tr>
<tr>
<td>XConst(elmobj)</td>
<td>&lt;xconst&gt;$elm&lt;/xconst&gt;</td>
<td>Constant transformation with $elmobj$ as its result.</td>
</tr>
<tr>
<td>XHide()</td>
<td>&lt;xhide /&gt;</td>
<td>Returns the empty value ($\lambda$).</td>
</tr>
<tr>
<td>XModifyName(nm)</td>
<td>&lt;xmodifyname&gt;$nm&lt;/xmodifyname&gt;</td>
<td>Modifies the tag of the source element to $nm$.</td>
</tr>
<tr>
<td>XNewRoot(nm)</td>
<td>&lt;xnewroot&gt;$nm&lt;/xnewroot&gt;</td>
<td>Returns an element with $nm$ as its tag and the source data as its content.</td>
</tr>
<tr>
<td>XDistribute(n)</td>
<td>&lt;xdistribute&gt;$n&lt;/xdistribute&gt;</td>
<td>Makes $n$ copies of the source element.</td>
</tr>
<tr>
<td>XChildren()</td>
<td>&lt;xchildren /&gt;</td>
<td>Returns all child elements of the source element.</td>
</tr>
<tr>
<td>XDescendant()</td>
<td>&lt;xdescendant /&gt;</td>
<td>Returns all descendant elements of the source element.</td>
</tr>
<tr>
<td>XChildrenNm(nm)</td>
<td>&lt;xchildrennm&gt;$nm&lt;/xchildrennm&gt;</td>
<td>Returns all child elements with the tag $nm$ in the source element.</td>
</tr>
<tr>
<td>XActionNFun()</td>
<td>&lt;xchildrennm&gt;</td>
<td>Used as the parent class for all non-invertible transformations.</td>
</tr>
</tbody>
</table>

Fig. 4 The Transformation for Tom

book will be represented as an embedded expression for computing chapter titles. BiXJ can be used to write such expressions to synchronize the titles in the table of contents and chapters.

- Code elements can be used as a kind of mobile code. For example, in a cluster of web servers, a runtime transformation object on one server can be dumped and moved to other machines, and then interpreted and run again.

All classes in BiXJ are summarized in Table 1, where $x$ denotes a transformation object, and $c$ is its corresponding code element; $pred$ represents an object for conditions and $c_{pred}$ is its XML representation. The formal definitions of these classes will be given in Section 5. With code elements, the transformation of publishing data for Tom in Section 2 is given in Figure 4. The element $xequals$ represents a predicate, which tests (in this case) whether the author under the $book$ element is Tom. Its parameter $<path>1</path>$ indicates that the content of the second child element in its source data is tested.

4 Design Issues

Any transformation in BiXJ can be executed in two directions: forward or backward. For forward transformations, we care about the expressiveness of BiXJ, while for backward transformations, we need to concern its view updating semantics, which is to determine whether a backward transformation...
correctly reflects modifications in target documents back into source documents. In this section, we discuss the view updating semantics for bidirectional transformations in BiXJ.

4.1 View Updating Semantics

In [3], the well-behavedness of bidirectional transformations is guaranteed by two properties: the first one, characterized by the GETPUT law, says the backward transformation of an unchanged target document should not change the source document, that is, a well-behaved bidirectional transformation is side-effect free; the second one, characterized by the PUTGET law, says the forward transformation of an updated source document should get the same modified target document, that is, a well-behaved bidirectional transformation should reflect all modifications in the target document back into the source document.

However, the second property is found restrictive for BiXJ. In what follows, we first characterize the first property in BiXJ transformations and then discuss the reasons why the second property is restrictive.

With methods \textbf{tranForward} and \textbf{tranBackward}, the property of side-effect free can be stated as follows, which is the GETPUT law in [3]:

\[ x.\text{tranBackward}(sd, x.\text{tranForward}(sd)) = sd \]

where \(sd\) is the source data, and \(x\) is a transformation object with the interface \textbf{XAction}.

The property stated by PUTGET law in [3] can be described as follows in BiXJ:

\[ x.\text{tranForward}(x.\text{tranBackward}(sd, td)) = td \]

where \(sd\) is the original source data and \(td\) is the modified target data obtained by modifying the data generated by \(x.\text{tranForward}(sd)\). However, this property is too restrictive. In the following, we will give four cases where this property is violated by BiXJ transformations.

The first case is relevant to conditional transformations, such as \textbf{xif} in Figure 4. Suppose the example for Tom in Section 2 is revised to include the author information in the target document. In this example, if Tom modifies all his names in the target data into his full name “Tom Bill” and updates the source document on the web server, then the forward transformation \(x.\text{tranForward}\) will produce an empty target document from the updated source data because this source data does not include books with the author “Tom” any more. Hence, the above property is violated, though the modifications have been correctly reflected into \(sd\).

The second case is that sometimes the modifications on target documents should be reflected back into transformations rather than into source documents. Still using the example in Section 2, if Tom changes the tag \textit{mybooks} in the target data into \textit{Books-of-Tom}, the source data should be kept unchanged. Rather, the transformation \texttt{<xnewroot>mybooks</xnewroot>} in Figure 4 should be updated as \texttt{<xnewroot>Books-of-Tom</xnewroot>}, so that Tom can see his modification really takes place when the web server executes this updated transformation for him again.

In the third case, the violation is caused by data dependency in target data, which has been recognized in [4]. For example, if an element in a source document is duplicated in the target document, then these two replicas are mutually dependent, and only modifying one replica will cause the PUTGET law violated. However, the proposed laws in [4], \textit{PUTGETPUT} and \textit{GETPUTGET}, are not strong enough to guarantee that all reasonable modifications can be reflected back into the source data in a well-behaved transformation.

The fourth case is relevant to non-invertible functions. For example, suppose there is a transformation that sums two integers in source data in the forward direction and keeps the source data unchanged in the backward direction. In this case, changing the sum in the target data will violate the PUTGET law since this modification is abandoned in the backward transformation.

A way of solving the above problems is to restrict possible modifications on target documents or exclude those problematic language constructs. For example, the string “Tom”, the tag \textit{mybooks} and the sum in the above examples are not allowed to change or the duplication operators (i.e., \textit{xidistribute} in BiXJ and \textit{Dup} in [4]) should be excluded. Obviously, this approach is too restrictive.

In our new view updating semantic, we still restrict modifications, but we only restrict modifications on the result of non-invertible functions, such as the sum of two integers.

The definition of our view updating semantics depends on the differences between two documents with the same structure, such as an original source
document and the source document after updating it. The differences between two documents are represented as a multiset of pairs, and each pair includes two different text strings, which are either element tags or text contents. A pair represents a modification, that is, its first component is changed to the second one. We write \( \text{diff}(od,md) \) for the differences between the original document \( od \) and its modified document \( md \).

In addition, in order to precisely specify modifications, we suppose each text content or tag in the source data and code elements has an associated unique identifier. These identifiers are only used for studying the view updating semantics, and they are transparent to transformations and updating operations on target documents. That is, the identifier of each string is unchanged during transformations and updating target documents. Thus, through the multiset of difference pairs and the associated identifiers on strings, we know not only the modified strings in a document, but also their positions in this document. Two modifications are said equal when they make the same changes on the same strings and also these strings have the same identifiers.

For the example in Section 2, suppose \( sd \) is the original source document on the web server and \( sd' \) is the updated source document. Then \( \text{diff}(sd, sd') = \{ ("Now Century", "New Century") \} \), \( ("Computer Graphics", "Computer Graphics") \}, where \( i \) and \( j \) are the associated identifiers. Since transformations in BiXJ can be represented as XML elements, the differences between two documents can also be applied to two transformations.

The view updating semantics in this work is defined as follows: Suppose \( sd \) is a source document, \( x \) a transformation object with the interface \( \text{XAction} \), \( td \) a target document of \( sd \) transformed by \( x \) (i.e., \( XAction(sd) = td \)), \( td' \) is the result of modifying \( td \) and the modified parts in \( td' \) do not include the results of non-invertible functions. Then the transformation \( x \) is well-behaved if the following condition holds:

\[
\text{diff}(sd, sd') \cup \text{diff}(c, c') = \text{diff}(td, td')
\]

where \( sd' = x.\text{tranBackward}(sd, td') \), and \( c \) and \( c' \) are the XML representations of \( x \) and the updated \( x \) by this backward transformation, respectively.

A sharp reader may argue that if there are conflicts in modifications in \( td' \), the above equation will not hold because not all modifications can be reflected back successfully. For the example in Section 2, there are two books written by Tom in the target data, and if Tom changes the tag \( \text{book} \) in his first book into \( \text{book-1} \), and that in his second book into \( \text{book-2} \), then these two modifications cause a conflict when updating the transformation \(\langle xnewroot \rangle \text{book} \langle/ xnewroot \rangle\) in Figure 4, and one of the modifications has to be abandoned. Note that the above equation depends on the normal terminated execution of \( \text{tranBackward} \). In BiXJ, some runtime technique helps solve this problem. If there are conflict modifications on target data, the execution of \( \text{tranBackward} \) will be aborted due to exceptions raised in \( \text{xmap} \) or \( \text{xmap} \). It can be checked that the above equation holds for all successful BiXJ transformations.

### 5 The Implementation of BiXJ

In this section, we will give the detailed definition of BiXJ transformations listed in Table 1. These transformations can be divided into two kinds: basic transformations and transformation combinators. Some other interesting transformations can be derived from the existing transformations defined in or implemented by BiXJ. As examples of derived transformations, we will demonstrate how to make XPath expressions bidirectional.

The syntax of XML elements is defined below, where the ending tags of elements are omitted and their contents are put into brackets to save space.

\[
\begin{align*}
\text{element} & ::= \langle \text{tag}\rangle[\text{element}, . . . , \text{element}] \\
\text{tag} & ::= \text{string}
\end{align*}
\]

The value \( () \) represents an empty sequence. We will use \( sd, td \) or \( d \) for sequences of elements, respectively, and \( e \) for a single element.

#### 5.1 Basic Transformations

A basic transformation generally performs one particular operation on source documents. For example, such an operation can be renaming the tag of a source element.

\( \text{XID: Let } x = \text{new XID}() \).

\[
\begin{align*}
XAction(sd) &= td \\
x.\text{tranForward}(sd) &= \text{new XID}() \\
x.\text{tranBackward}(sd, td) &= \text{new XID}()
\end{align*}
\]
In this transformation, the object $x$ is changed with its parameter $e$ replaced by $e'$ after backward executions. This effect is implicit in this definition, but it can be observed when we use the method dump to output $x$ in its XML format. When this updated $x$ is used to perform a forward transformation, the target data will be $e'$ rather than $e$. In this transformation and some following transformations, the element $e$ should be understood as a singleton list $[e]$.

**XHide:** Let $x = \text{new XHide}()$.

\[
\begin{align*}
x.\text{tranForward}(sd) &= () \\
x.\text{tranBackward}(sd, () &= sd
\end{align*}
\]

This transformation is to hide the source data, so its target data is the empty value $()$.

**XModifyName:** Let $x = \text{new XModifyName}(nm)$ and $e = <\text{tag}>[d]$.

\[
\begin{align*}
x.\text{tranForward}(e) &= <nm>[d] \\
x.\text{tranBackward}(e, nm'[d']) &= <\text{tag}>[d']
\end{align*}
\]

After backward executions, the object $x$ is updated with its parameter $nm$ replaced by $nm'$ and the contents of the source data is also updated. This transformation takes a single element as its source data. It can be passed to the transformation combinator XMap to transform an element list.

**XNewRoot:** Let $x = \text{new XNewRoot}(nm)$.

\[
\begin{align*}
x.\text{tranForward}(sd) &= nm>[sd] \\
x.\text{tranBackward}(sd, nm'[sd']) &= sd'
\end{align*}
\]

This transformation puts a new root tag $nm$ onto the source data in the forward direction. In the backward direction, beside updating the source data, this transformation also updates the object $x$ by replacing its parameter $nm$ with $nm'$.

**XDistribute:** Let $x = \text{new XDistribute}(n)$, where $n$ is a natural number.

\[
\begin{align*}
x.\text{tranForward}(e) &= [e, ..., e] \\
&\text{(}$n$\text{ copies of } e) \\
x.\text{tranBackward}(e, td') &= \text{merge}(e, td') \\
\text{where } td' &= [e_1', ..., e_n']
\end{align*}
\]

In this definition, $\text{merge}$ is an auxiliary function, which returns an updated source data by combining all modifications on all copies of the source data. If several copies contain different modifications at the same place, then these are conflict modifications.

In this case, $\text{merge}$ will raise an exception and the execution is aborted. XDistribute is more general than Dup in [4], which is used to maintain data dependency relation in target documents. Moreover, by using $\text{merge}$ function, XDistribute allows to update source data in a batch style rather than the interactive style adopted by Dup, so it is suitable for updating XML documents in a network environment.

**XChildren:** Let $x = \text{new XChildren}()$ and $e = <\text{tag}>[e_1, ..., e_n]$.

\[
\begin{align*}
x.\text{tranForward}(e) &= [e_1, ..., e_n] \\
x.\text{tranBackward}(e, [e_1', ..., e_n']) &= <\text{tag}>[e_1', ..., e_n']
\end{align*}
\]

This transformation corresponds to the child axis in XPath. We will bidirectionalize XPath expressions using this transformation and some transformation combinators later. We also implement another commonly used axis, the descendant axis, in the class XD descendant.

### 5.2 Transformation Combinators

Transformation combinators are used to build new transformations from already defined transformations.

**XSeq:** Let $x = \text{new XSeq}([x_1, ..., x_n])$.

\[
\begin{align*}
x.\text{tranForward}(d_0) &= d_n \\
x.\text{tranBackward}(d_0, d'_n) &= d_0 \\
\text{where } d_i &= x_i.\text{tranForward}(d_{i-1}) \\
d'_i &= x_i.\text{tranBackward}(d_{i-1}, d'_{i}) \\
(1 &\leq i \leq n)
\end{align*}
\]

XSeq takes a list of transformation objects as its argument. These argument transformations are applied sequentially in executions. The object $x$ is updated if some of its argument transformations are updated during backward executions. Note that the source data of $x_i$ comes from the forward execution of $x_{i-1}$. Hence, backward executions of XSeq involves executing forward its transformation arguments to compute the intermediate source data. The backward transformations XMap and XZip also need the forward executions of their arguments.

**XMap:** Let $x = \text{new XMap}(x')$ and $sd = [e_1, ..., e_n]$.

\[
\begin{align*}
x.\text{tranForward}(sd) &= td \\
x.\text{tranBackward}(sd, td') &= [e_1', ..., e_n']
\end{align*}
\]
where
\[
\begin{align*}
  t_d &= td_1 + \cdots + td_n \\
  td_i &= x'.\text{transForward}(e_i) \\
  (td'_1, \ldots, td'_n) &= \text{split}(td', \{[td_1], \ldots, [td_n]\}) \\
  e'_i &= x'.\text{transBackward}(e_i, td'_i) \\
  (1 \leq i \leq n)
\end{align*}
\]

In this definition, the operator \(+\) concatenates two lists. In the backward direction, the target data \(td'\) is divided into \(n\) sublists using the operator \(\text{split}\), and the \(i\)th sublist has the length \(|td_i|\). Since \(td_i\) is a value computed during forward executions, the backward executions of \(\text{XMap}\) need to execute forward \(x'\) to compute the value \(|td_i|\). Updating this transformation is a bit complex. Suppose that after backward transformation of \(td'_i\) \((1 \leq i \leq n)\), the transformation object \(x'\) is updated to \(x_i\). Then, we can generate a new transformation element by using \(\text{merge}(c'_i, [c'_1, \ldots, c'_i])\), where \(c'\) and \(c'_i\) are the XML representations of \(x'\) and \(x_i\), respectively. This transformation element will be interpreted as a transformation object, and then used to replace the old object \(x'\). Note that if \(\text{merge}\) raises an exception due to conflicts, the backward executions of \(\text{XMap}\) will be aborted.

**XZip:** Let \(x = \text{new XZip}([x_1, \ldots, x_n])\) and \(e = \text{<tag} [e_1, \ldots, e_n]\).

\[
\begin{align*}
  x.\text{transForward}(e) &= \text{<tag} [td] \\
  x.\text{transBackward}(e, d') &= \text{<tag} [e'_1, \ldots, e'_n] \\
\end{align*}
\]

where
\[
\begin{align*}
  td &= td_1 + \cdots + td_n \\
  td_i &= x.\text{transForward}(e_i) \\
  \text{<tag} [d']:d' &= \text{split}(td', \{[td_1], \ldots, [td_n]\}) \\
  (td'_1, \ldots, td'_n) &= \text{split}(td', \{[td_1], \ldots, [td_n]\}) \\
  e'_i &= x.\text{transBackward}(e_i, td'_i) \\
  (1 \leq i \leq n)
\end{align*}
\]

This transformation is used to change the contents of the source element by executing its argument transformations. Each argument transformation is applied to the corresponding child element in the source document. The actual implementation of **XZip** is more flexible than this definition. It allows different lengths for the following two sequences, the argument transformations of **XZip** and the contents of the source data \(e\). If the former is longer, then the extra argument transformations are ignored; if the latter is longer, then the extra contents are processed by identity transformations.

**XIf:** Let \(x = \text{new XIf}(pred, x_1, x_2)\).

\[
\begin{align*}
  x.\text{transForward}(sd) &= x_1.\text{transForward}(sd) \\
  &\quad \text{if} \ pred.\text{qualify}(sd) \\
  &= x_2.\text{transForward}(sd) \\
  &\quad \text{otherwise} \\
  x.\text{transBackward}(sd, td) &= x_1.\text{transBackward}(sd, td) \\
  &\quad \text{if} \ pred.\text{qualify}(sd) \\
  &= x_2.\text{transBackward}(sd, td) \\
  &\quad \text{otherwise}
\end{align*}
\]

In this definition, \(pred\) is an object with the interface **XPredicate**, which has a method **qualify** to judge whether the source data satisfies some condition. We have implemented several commonly used predicates, including **XTrue**, **XLessThan**, **XGreaterThan**, **XEquals**, **XHasChild** and **XWithTag**, and three predicate operators **XAnd**, **XOr** and **XNot**. The meaning of each predicate is obvious. For example, **XWithTag(nm)** judges whether the source data is an element with the tag \(nm\).

5.3 Bidirectional XPath

Some transformations need not to be defined primitively. Rather, they can be defined with the existing transformations. In this section, we give two examples, which shows how to make XPath expressions bidirectional using the transformations defined before.

**XChildrenNm:** Let \(x = \text{new XChildrenNm}(nm)\).

\[
\begin{align*}
  x.\text{transForward}(e) &= \text{trans.\text{transForward}(e)} \\
  x.\text{transBackward}(e, td) &= \text{trans.\text{transBackward}(e, td)} \\
\end{align*}
\]

where
\[
\begin{align*}
  \text{trans} &= \text{new XSeq}([x_1, x_2]) \\
  x_1 &= \text{new XChildren()} \\
  x_2 &= \text{new XMap}(x_3) \\
  x_3 &= \text{new XIf}(pred, x_4, x_5) \\
  x_4 &= \text{new XID()} \\
  x_5 &= \text{new XHide()} \\
  pred &= \text{new XWithTag}(nm)
\end{align*}
\]

This transformation corresponds to **child::nm** in XPath.

**XPathStep:** Let \(x = \text{new XPathStep}(nm, pred)\).

\[
\begin{align*}
  x.\text{transForward}(e) &= \text{trans.\text{transForward}(e)} \\
  x.\text{transBackward}(e, td) &= \text{trans.\text{transBackward}(e, td)} \\
\end{align*}
\]

where
\[
\begin{align*}
  \text{trans} &= \text{new XSeq}([x_1, x_2]) \\
  x_1 &= \text{new XChildrenNm(nm)} \\
  x_2 &= \text{new XMap}(x_3) \\
  x_3 &= \text{new XIf}(pred, x_4, x_5) \\
  x_4 &= \text{new XID()} \\
  x_5 &= \text{new XHide()} \\
\end{align*}
\]

This transformation corresponds to the XPath step **child::nm[pred]**. It can be represented as fol-
With this transformation, the XPath expression
\[ /nm_1[pred_1]/\ldots/nm_n[pred_n] \]
is encoded as the following bidirectional transformation:

```xml
<xseq>
  <xmap>
    <xpathstep>
      <name>nmap</name> pred_1
    </xpathstep>
    ...  
    <xpathstep>
      <name>nmap</name> pred_n
    </xpathstep>
  </xmap>
</xseq>
```

### 5.4 Degraded Bidirectional Transformations

BiXJ does not support modifications on the target data generated by non-invertible functions. That is, users should not modify these data, and even if they make some modifications, these modifications will not be reflected back. In BiXJ, we provide an abstract class `XActionNFun` for implementing non-invertible functions, which has implemented a trivial backward transformation by just returning the original source data. Hence, when implementing a non-invertible function, we just need to define a subclass of `XActionNFun` and implement this function in the forward transformation method of this subclass.

In addition to implementing non-invertible functions, the abstract class `XActionNFun` can also be used to incorporate other existing XML transformation Java code into the bidirectional transformation framework if we do not care the backward transformations of these code.

### 6 Examples

In order to test the expressiveness and usability of the bidirectional transformations in BiXJ, we use them to bidirectionalize some typical examples from XQuery and XSLT. The resulting BiXJ code implements the same transformations in the forward direction.

All examples in this section use an XML file “lib.xml” as the source document, which is partially listed in Figure 5. For this document, we assume that users are interesting in the engineering books written by Tom with a price less than 50.

#### 6.1 Bidirectionalization of XQuery Expressions

XQuery expressions generally take the FLWR form. The interesting book information for users can be obtained by using the XQuery expression in Figure 6, which involves `for`, `where` and `return` expressions. This expression generates an element with the tag `Books-of-Tom`, containing a list of `book` elements. And each `book` element has three child elements: `title`, `price` and `press`. The `press` element contains the publisher name.

The BiXJ script in Figure 7 implements the same transformation as the above XQuery expression. The whole script consists of a sequence of transformations wrapped by `xseq`. A simple way of writing this script is to finish this task step by step, that is, by adding a new transformation to the end of the
<Books-of-Tom>{
  for $l in doc("lib.xml")/lib return
  for $s in $l/shelf[category="Engineering"]
    return
  for $c in $s/cabinet return
    for $b in $c/book
      where $b/author = "Tom" and $b/price<50
        return
      <book>
        $b/title,
        $b/price,
        <press> {
          $b/publisher/name/text()}
      </press>
    </book>
}</Books-of-Tom>

Fig. 6 An XQuery Expression

existing script, checking the transformation result, and repeating this procedure until the expected result is obtained. However, it seems a tedious task. In the future, we would like to develop algorithms to translate the programs of high level XML processing languages into BiXJ code. In the following, we informally introduce some experiences learned when writing such script.

The XQuery expression in Figure 6 consists of two kinds of subexpressions: one is to construct the target element and the other is to destruct the source element.

The first kind of expressions includes three element constructors for Books-of-Tom, book and press elements. They are encoded according to the following principle: If the source data already contains the content expected by a constructor, then we just modify the name of the source data with the element name specified in this constructor, otherwise we encode this constructor in the following form:

<newroot>element name</newroot>
<zip>code for constructing content</zip>

In Figure 7, the constructor for press element is encoded by simply modifying the name of the source data to press, and the other two constructors are encoded using the above form.

The second kind of expressions includes XPath expressions. An XPath expression is a sequence of path steps, and each step consists of an axis, a node test and a qualifier. To encode the child axis and

Fig. 7 Bidirectionalization of The XQuery Expression
name node test, the transformation \texttt{xchildrennm} is used, probably with the help of \texttt{xmap} to process an element list. The transformation \texttt{xmap} plays the similar role as the \texttt{for} clause in XQuery. The qualifier in an XPath step, such as \texttt{category = 'Engineering'}, is encoded by an \texttt{xif} following the corresponding axis and node test. As a comparison, in the next section, XPath expressions will be encoded in the derived transformation \texttt{xpathstep}.

6.2 Bidirectionalization of XSLT Expressions

The style sheet of XSLT generally is made up of a list of templates, which are connected by \texttt{apply-templates}. The style sheet in Figure 8 generates the same interesting book information as the XQuery expression in Figure 6. This style sheet includes five templates and transforms the data of interest into an HTML file.

The BiXJ code in Figure 9 implements the same transformation as the above XSLT style sheet. The code is divided into three parts for readability. The code in Figure 9 corresponds to the first template in Figure 8; the code \texttt{x1} in Figure 10 extracts the interesting books from the source document, corresponding to the second, third and fourth templates; the code \texttt{x2} in Figure 11 does the same thing as the last template, which is to construct table rows.

The bidirectionalizing procedure starts with the first template. When meeting with an \texttt{apply-templates} in a template, we put here the bidirectionalizing result of the applied template. For each template, we almost follow the same rules as used in bidirectionalizing XQuery expressions. The exception is that if an element constructor contains the contents not computed by \texttt{apply-templates}, then we use \texttt{xconst} to construct this element directly. For example, the code for constructing the \texttt{title} element in Figure 9 belongs to this case.

Fig. 8 An XSLT Expression

7 Related Work

This work takes a bidirectional transformation style similar to [3][4]. As discussed in Section 1, they have several limitations to be used as general XML transformation languages. In this work, BiXJ has addressed their limitations and is used for general purpose XML processing.

In the database area, there is also some work to do XQuery updating. For example, the work in [8] transforms updates on query tree into SQL updates, and then uses the traditional view updating techniques to update relational databases. Obviously, this approach is not suitable for updating native XML repositories. In addition, it cannot be used to update the view defined by XSLT, either.

The work [11] studies the problem of bidirectionalizing HaXML [12] and shows that any transfor-
Fig. 9 Bidirectionalization of The XSLT Expression

Fig. 10 The BiXJ Code $x_1$

Fig. 11 The BiXJ Code $x_2$

information in HaXML can be compiled into a bidirectional transformation. In work [13], the authors give an injective language $\text{Inv}$ to implement view updating, and due to injectivity, so each program is invertible. However, they are still not used in general purpose XML processing. For example, they do not support bidirectional XPath.

8 Conclusion

In this paper, we solve the problem of view updating for general purpose XML processing. The proposed solution is a Java library BiXJ for bidirectional XML transformation. By this library, given a forward transformation, the backward transformation can be obtained for free. Hence, no extra efforts or separate mechanisms are needed for users to update source documents after target documents are modified. We have demonstrated the expressiveness and usability of BiXJ by bidirectionalizing some typical examples of two popular XML processing languages XQuery and XSLT.

In the future, we will develop algorithms that can translate XSLT or XQuery expressions into the code of BiXJ automatically, and consider other updating operations on target documents.

9 Acknowledgment

Thanks to the PSD project members in the University of Tokyo for stimulating discussion on this work. This work is partially supported by Comprehensive Development of e-Society Foundation Software Program of the Ministry of Education, Culture, Sports, Science and Technology, Japan. We are also grateful to the anonymous reviewers for their detailed and helpful comments and sugges-
Fig. 11 The BiXJ Code $x_2$

References