A Java Library for Bidirectional XML Transformation

Dongxi Liu, Zhenjiang Hu, Masato Takeichi, Kazuhiko Kakehi
Dept. of Mathematical Informatics, University of Tokyo
{liu,hu,takeichi,kaz,wang}@mist.i.u-tokyo.ac.jp
Hao Wang
Dept. of Creative Informatics, University of Tokyo
wang@ipl.t.u-tokyo.ac.jp

We propose a Java library BiXJ for bidirectional XML transformation. The feature of BiXJ is that when writing one transformation, users can get its corresponding backward transformation for free. The transformations in this library adopt the native XML data model and concentrate on the widely accepted methods of constructing and destructing XML documents. By this design principle, we expect this library is expressive enough, and moreover it does not confuse the users due to providing too many transformations. The difficulty in this work is to seek more suitable semantics for the bidirectional transformations since the semantics in the previous work is too limited to express both useful transformations (e.g., generating multiple elements) and updates . To demonstrate its usability and expressiveness, we have bidirectionalized some typical examples of XQuery and XSLT using 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 data repository. In many cases, an XML document needs to be transformed into another document, which has a structure more convenient for viewing and editing, or contains only interesting data for users. When using the transformed XML document, the applications or users sometimes want to change this document for updating data or correcting some errors, so it is desirable that modifications on the transformed document can be reflected back to the source document without too much cost.

However, the current popular XML transformation languages, such as XSLT [1] and XQuery [2], can perform transformation only in one direction. That is, if a transformed document is generated by XSLT or XQuery, then the modifications on it cannot be reflected back to the source document without using other mechanisms.

This work presents a Java library BiXJ for bidirectional XML transformation. The interface of the transformations (i.e. some Java classes) in this library has two methods relevant to bidirectional transformation: one is to transform the source document into a target document, called *forward transformation*; the other is to transform the updated target document into an updated source document, called *backward transformation*. Thus, in BiXJ, every time users write one transformation, the corresponding backward transformation is gotten automatically. That is, there is no extra effort needed for users to update source documents from modified target documents.

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 used to any case where Java is used to process XML documents, and moreover bidirectional transformation property is desirable.

This style of bidirectional transformation techniques has been proposed in literatures [3, 4]. However, they are both domain specific languages, not general enough for transforming XML documents. The language in [3] is designed for synchronizing tree-structured data, and the language in [4] is mainly used in a tree editor. Due to their purposes

of domain specific applications, these existing languages have many limitations when sued as general purpose XML processing languages. For example, they only allow transformations that generate one target element, while XPath expressions in XQuery and XSLT can return a set of elements; they take data models that are either unordered trees without repeated labels or trees consisting only of nodes without text content, while XML has a more rich data model; they uses their own specific methods to destruct tree-structured data, while XQuery and XSLT have popular XPath expressions for this purpose. In this work, `BiXJ` is designed to extend these existing languages and be used as a general purpose XML processing language. The difficulty is to seek more suitable semantics for bidirectional transformations that can avoid the limitations of the existing work.

In summary, the *main contribution* of this paper is to give a Java library for bidirectional XML transformation, which has the following features:

- Adopts the native XML data model;
- Concentrates on the common and necessary methods of constructing and destructing XML documents. As an example, the `children` and `descendant` axes in XPath are included in `BiXJ`;
- Has a more flexible semantics of bidirectional transformations that allows to update transformation itself;
- Support data dependency in target documents as in [4];
- Be expressive enough to bidirectionalize typical uses of XQuery and XSLT.

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

2 Bidirectional Transformation in Use

Consider the following XML fragment, which contains some book information. The top element,

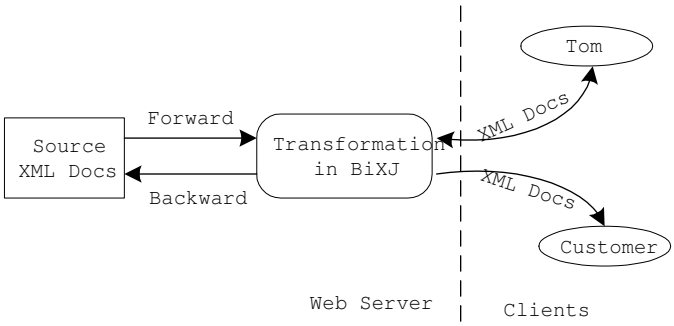

図 1: A Scenario of Bidirectional Transformation

tagged by `books`, contains three child elements, each of which is a `book` element, which in turn contains the information about the book `title`, `author`, published `year` and `publisher`.

```
<books>
  <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</publsher>
  </book>
  <book><title>Computer Graphecs</title>
    <author>Tom</author><year>1999</year>
    <publisher>ACM Press</publisher>
  </book>
</books>
```

Suppose that this XML is stored on a server. The provided Web service allows an author to select the books s/he wrote for checking and correcting error information, or ordinary customers to request the interesting book information for reading. For authors, the server makes forward transformations and returns elements with tag `mybooks`, while for ordinary customers it generates elements tagged by `interestingbooks`.

For example, when Tom wants the title and publisher information of books written by himself, the server should transform forwardly the above source document into the following one:

```
<mybooks>
  <book><title>Computer Programming</title>
    <publisher>Now Century</publisher>
  </book>
  <book><title>Computer Graphecs</title>
    <publisher>ACM Press</publisher>
  </book>
</mybooks>
```

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

Later, a customer query the server for the title, author and publisher information of books published earlier than year 2000. Then the server answers the following information after forward transformation.

```
<interestingbooks>
  <book><title>Computer Graphics</title>
    <author>Tom</author>
    <publisher>ACM Press</publisher>
  </book>
</interestingbooks>
```

Note that this document contains the correct book title now. This scenario is depicted in Figure 1.

3 Overview of BiXJ

Our library is built on JDOM [5], which provides an easy way to process XML document in Java. The class `Element` in JDOM abstracts the elements in XML documents and operations on them. While XML elements in our work take different roles, such as transformation source or result. To make the transformation interface more accurate, we refine class `Element` into three subclasses `SrcElement`, `TgtElement` and `CodeElement`, among which the first two correspond to the source and target element in a transformation, and the third will be introduced later. The transformation interface in `BiXJ` is defined as follows:

```
public interface XAction{
  public TgtElement tranForward(SrcElement sd);
  public SrcElement tranBackward(SrcElement sd,
                                 TgtElement td);
  Public CodeElement dump();
  }
```

In this interface, methods `tranForward` and `tranBackward` perform the forward and backward transformations, respectively. They take same roles as those played by *get* and *put* in [3]. Due to refined classes, when using methods `tranForward` or `tranBackward`, users can know what kind of elements the method will take and return. Type refinement is a common way to express program invariants [6].

Method `dump` is specific to this library. For a transformation object with interface `XAction`, this method can generate an XML fragment with type `CodeElement` to represent the current transformation object. Dually, class `CodeElement` has a method `makeAction` that can interpret a code element as a transformation object. This feature has several interesting applications:

- Code elements can be used as the intermediate code when translating the expressions of high level transformation languages into bidirectional transformation in our library. The syntax of code element is more compact than Java syntax used by transformation classes, so it can simplify the translating procedure. In the later example, code elements will be used to describe the typical transformations in XQuery and XSLT
- Code elements allow users to modify the transformation objects at runtime. After dumping an transformation object into a code element, this element can be processed like an ordinary XML element, and then it can be interpreted as a new transformation object. This feature can help to implement self-adjusting transformations.
- Code elements can be incorporated into data elements to construct Programmable Structured Documents (PSD) [7].
- Code elements can be used as a kind of mobile code. For example, in a cluster of web servers, a runtime transformation object in one server can be dumped and moved to other machine, and then can be interpreted and run again.

Therefore, there are two ways to use this library. In the first way, users can directly use the bidirectional transformation classes just as using ordinary Java classes; in the second way, users can write code element to describe transformation instead of programming in Java classes.

All classes in `BiXJ` are summarized in Figure 2, where x_i is a transformation object, while c_i is its corresponding code element. With code elements, the transformation for Tom can be written as follows:

```
<xseq>
  <xchildren>book</xchildren>
  <xmap>
    <xif>
      <xequals><path>1<path>
        <value>Tom</value>
      <xequals>
      <xid />
      <xhide />
    <xif>
  </xmap>
  <xmap>
    <xseq>
      <xdistribute>2</xdistribute>
      <xzip>
        <xchildren>title</xchildren>
        <xchildren>publisher</xchildren>
      </xzip>
      <xnewroot>book</xnewroot>
      <xcollapse>virtual</xcollapse>
    </xseq>
  </xmap>
  <xnewroot>mybooks</xnewroot>
  <xcollapse>virtual</xcollapse>
</xseq>
```

Note that element `<xequals>` represents a predicate, which tests (in this case) whether the author under book element is Tom.

4 Design Issues

An transformation in our work can be performed in two directions, forward or backward. For forward transformation, we care about its expressiveness, in particular how to model transformations generating multiple elements. While for backward transformation, we will discuss its updating semantics.

4.1 Transformations Generating Multiple Elements

In the interface XAction, method `tranForward` only generates one element. While for the practical transformation languages, like XQuery or XSLT, they are able to return a list of elements as the transformation result. This inconsistency will cause some trouble for users to express the transformations in XQuery or XSLT. Of course, we can modify the interface and let the methods return a list of elements, but this will make the whole library complex.

Here, we use a two step solution to address this problem. At the first step, if a transformation needs to return multiple elements, a virtual parent element, tagged with “virtual”, is introduced to incorporate the resulting elements. At the second step, the virtual element can be collected if it is not the root element. Virtual element collection removes each virtual element, and makes its children appear as the children of its parent element and takes the same position as this virtual element.

For the example in Section 2, after the first step by transformation `<xmap><xchildren>title</xchildren></xmap>`, the document is represented as follows:

```
<books>
  <virtual><title>Computer Programming</title>
  </virtual>
  <virtual><title>Data Structure</title>
  </virtual>
  <virtual><title>Computer Graphecs</title>
  </virtual>
</books>
```

And then, after virtual element collection with `<xcollapse>virtual</xcollapse>`, the element becomes:

```
<books>
  <title>Computer Programming</title>
  <title>Data Structure</title>
  <title>Computer Graphecs</title>
</books>
```

Note that there is some difficulty to combine the above two steps into one step because at the time of generating a virtual element, there is no any information about its parent and its position in its parent according to the method signature in the XAction interface. Moreover, the operation `text()` in XPath returns a string, not an element at all. Based on the mechanism here, this can also be modelled, which will be shown when bidirectionalizing the examples of XSLT and XQuery.

4.2 Updating Semantics

In [8], a translation of view update is required to be both consistent with the view and acceptable. In other words, the translation of view update (i.e. the backward transformation here) should have two properties, that is, reflecting all modifications back to the source document and absence of side effects. However, in our work, the second property for backward transformation is held, while the first fails for some transformations. Below, we will character the

Class Constructors	CodeElement	Description
XSeq(x_0,...,x_n)	<xseq>c_0...c_n</xseq>	Apply transformations x_0,...,x_n sequentially.
XMap(x')	<xmap>c'</xmap>	Apply x' to each child of the source element.
XZip(x_0,...,x_n)	<xzip>c_0...c_n</xzip>	Apply x_0,...,$x_n$to each child of the source element, respectively.
XIf($pred$, x_0, x_1)	<xif>$cpred$ c_0 c_1</xif>	Apply x_0 to the input element if the element holds $pred$, otherwise apply x_0.
XID()	<xid />	Identify transformation.
XConst($elmobj$)	<xconst>elm</xconst>	Constant transformation with elm as result.
XHide()	<xhide />	Hide the input element.
XModifyName(nm)	<xmodifyname>nm </xmodifyname>	Modify the tag of the input element to nm.
XNewRoot(nm)	<xnewroot>nm</xnewroot>	Make the input element as the child of new node with tag nm.
XDistribute(n)	<xdistribute>n </xdistribute>	Make n copies of the input element, put them under node ‘‘virtual’’.
XChildren(nm)	<xchildren>nm </xchildren>	Keep the children of the input element with name nm, and change its name to ‘‘virtual’’.
XDescendant(nm)	<xdescendant>nm </xdescendant>	New a ‘‘virtual’’ element with its children the descendant of the input element with name nm.
XCollapse(nm)	<xcollapse>nm </xcollapse>	Remove all nodes with name nm in the input element and lift their children one level up.
XActionNFun()		Used as the parent class of all un-invertible transformations.

図 2: Classes in BiXJ

second property and discuss the reason why the first fails.

By using methods `tranForward` and `tranBackward` in a transformation class, the property of absence of side effects can be stated as follows, which is `GETPUT` law in [3].

$$\texttt{x.tranBackward}(\texttt{sd}, \texttt{x.tranForward}(\texttt{sd})) = \texttt{sd}$$

where `sd` is the source document, and `x` is a transformation object having interface XAction. That is, after transforming `sd` into a target document, we transform it back immediately, then the same `sd` is gotten.

Like the `PUTGET` law in [3], the consistent property in [8] can be described as follows in our work:

$$\texttt{x.tranForward}(\texttt{x.tranBackward}(\texttt{sd}, \texttt{td})) = \texttt{td}$$

where `sd` is the source document and `td` is gotten by modifying the original target document `x.tranForward(sd)`. However, such property is too strict and unreasonable sometimes. For the example in Section 2, if Tom modifies his name into his full name “Tom Bill”, then the existing `tranForward` will produce nothing from the updated source document because there is no book with author name “Tom”, that is the above law property is violated, though the modifications have been correctly done on `sd`. This case really happens when we test the encoding result of XQuery.

Another thing that makes the above law fail is that sometimes the modifications on a target document should be reflected back onto the used transformation rather than the source document. Still using the example in Section 2, if Tom changes the tag “mybooks” in the target document into “books of Tom”, the source document should be keep unchanged, while the forward transformation should be updated so that Tom will see his modification really take place when he does forward transformation again.

Here, we propose several choices to address the above problem. The first one is to impose a restrict modifications on the target document. For example, the names “Tom” and “mybooks” cannot be allowed to be modified. Actually, the work in [3] simply takes this policy. The second one is to use the following more flexible laws:

$$\texttt{x}'\texttt{.tranForward}(\texttt{x.tranBackward}(\texttt{sd}, \texttt{td})) <: \texttt{td}$$

in which $\texttt{td}' <: \texttt{td}$ means all sub-

trees of $\mathtt{td}'$ appears in $\mathtt{td}$, where $\mathtt{td}' = \mathtt{x}'.\mathtt{tranForward}(\mathtt{x}.\mathtt{tranBackward}(\mathtt{sd}, \mathtt{td}))$.
The most important feature of this law is that when doing forward transformation again, the used transformation $\mathtt{x}'$ is different from that used in previous time, i.e. transformation $\mathtt{x}$. The difficult of using this law lies in seeking the updated transformation $\mathtt{x}'$, which is not a trivial work, especially when we consider an un-invertible XML transformation as a degraded case of bidirectional transformation. The third one is a combination of the first two, and its principal is to try to reflect as many modifications as possible back to source documents or transformations if the $\mathtt{x}'$ is easily to find, otherwise restrictions on modification are imposed. This work takes the third policy.

5 The Implementation of BiXJ

According to the discussions in Section 4, the transformation written in this library are divided into two categories. For the first category, their methods `tranBackward` reflect all modifications on target documents back to source documents or transformations; while for the second category, the target documents are restricted to modify. For presentation, we suppose the element is defined as follows abstractly:

$$\begin{aligned} element &::= <tag>\{element, ..., element\} \\ &\quad | <tag>\{string\} \mid () \\ tag &::= string \end{aligned}$$

Note that `()` is a special element, and in implementation, it is just a *null* element reference.

5.1 Transformation Combinators

Transformation combinators take as arguments other transformations, which is used to build new transformations from already defined transformations. The classes of transformation combinators mainly include `XSeq, XMap, XZip` and `XIf`. And, which category the transformations expressed in these classes belong to is determined by the transformation arguments they take. Below, we will describe each of them in brief.

`XSeq`: Suppose x_0, x_1, ..., x_n are some transformation objects and $x = \mathtt{new\ XSeq}(\{\mathtt{x_0}, \mathtt{x_1}, ..., \mathtt{x_n}\})$. Then the behavior of x is defined as follows:

$$\begin{aligned} x.\mathtt{tranForward}(sd) &= d_n \\ \mathtt{where}, d_n &= x_n.\mathtt{tranForward}(d_{n-1}) \\ d_{n-1} &= x_{n-1}.\mathtt{tranForward}(d_{n-2}) \\ &... \\ d_0 &= x_0.\mathtt{tranForward}(sd) \\ x.\mathtt{tranBackward}(sd, td) &= d'_0 \\ \mathtt{where}, d'_{n-1} &= x_n.\mathtt{tranBackward}(d_{n-1}, td) \\ &... \\ d'_0 &= x_0.\mathtt{tranBackward}(sd, d'_1) \end{aligned}$$

`XMap`: Suppose x' is a transformation object, $x = \mathtt{new\ XMap}(\mathtt{x}')$ and $sd = \mathtt{<tag>}\ \{sd_0, ..., sd_n\}$. Then the behavior of x is defined as follows:

$$\begin{aligned} x.\mathtt{tranForward}(sd) &= d \\ \mathtt{where}, d &= <\mathtt{tag}>\{d_0, ..., d_n\} \\ d_0 &= x'.\mathtt{tranForward}(sd_0) \\ &... \\ d_n &= x'.\mathtt{tranForward}(sd_n) \\ x.\mathtt{tranBackward}(sd, td) &= d' \\ \mathtt{where}, <\mathtt{tag}'>\{d''_0, ..., d''_n\} &= td \\ d' &= <\mathtt{tag}'>\{d'_0, ..., d'_n\} \\ d'_0 &= x'.\mathtt{tranBackward}(sd_0, d''_0) \\ &... \\ d'_n &= x'.\mathtt{tranBackward}(sd_n, d''_n) \end{aligned}$$

Note that td, the modified target document from d, allows changes in its tag and its children, but not insertion or deletion of elements yet. In practice, if d_i is `()`, then it does not take place in the content of d, so some trick is needed to align the contents of td and sd since they have different lengths.

`XZip`: Suppose x_0, x_1, ..., x_n are some transformation objects, $x = \mathtt{new\ XZip}(\{\mathtt{x_0}, \mathtt{x_1}, ..., \mathtt{x_n}\})$ and $sd = \mathtt{<tag>}\{sd_0, ..., sd_n\}$. Then the behavior of x is defined as follows:

$$\begin{aligned} x.\mathtt{tranForward}(sd) &= d \\ \mathtt{where}, d &= <\mathtt{tag}>\{d_0, ..., d_n\} \\ d_0 &= x_0.\mathtt{tranForward}(sd_0) \\ &... \\ d_n &= x_n.\mathtt{tranForward}(sd_n) \\ x.\mathtt{tranBackward}(sd, td) &= d' \\ \mathtt{where}, <\mathtt{tag}'>\{d''_0, ..., d''_n\} &= td \\ d' &= <\mathtt{tag}'>\{d'_0, ..., d'_n\} \\ d'_0 &= x_0.\mathtt{tranBackward}(sd_0, d''_0) \\ &... \\ d'_n &= x_n.\mathtt{tranBackward}(sd_n, d''_n) \end{aligned}$$

Our implementation is more flexible than this definition. It allows the transformations x_i and contents sd_i have different length. If the former is longer, then the extra transformations are ignored; if the latter is longer, then the remaining contents are processed by identity transformations.

`XIf`: Suppose x_0 and x_1 are two transformation objects and $x = \mathtt{new\ XIf}(\mathtt{pred}, \mathtt{x_0}, \mathtt{x_1})$. Then the be-

havior of x is defined as follows:

$$\begin{array}{rcl} x.\texttt{tranForward}(sd) & = & d \\ \texttt{where}, d & = & x_0.\texttt{tranForward}(sd) \\ & & \quad \texttt{if}\ pred.\texttt{qualify}(sd) \\ d & = & x_1.\texttt{tranForward}(sd) \\ & & \quad \texttt{otherwise} \\ x.\texttt{tranBackward}(sd, td) & = & d' \\ \texttt{where}, d' & = & x_0.\texttt{tranBackward}(sd, td) \\ & & \quad \texttt{if}\ pred.\texttt{qualify}(sd) \\ d' & = & x_1.\texttt{tranBackward}(sd, td) \\ & & \quad \texttt{otherwise} \end{array}$$

In this definition, *pred* is an object with interface `XPredicate`, which has a method `qualify` to judge whether an element satisfy a predicate. We have implemented the most commonly used predicates like `XLessThan`, `XGreaterThan`, `XEquals`, `XHasChild` and `XWithTag`, as well as predicate operators `XAnd`, `XOr` and `XNot`. There is some trouble to prove `XIf` satisfies the last law in 4. The idea is that if *pred* is dependent on some data content, which also appears in target document, then *pred* must be adjusted according to the modifications on these content if they are changed. By this way, we can get the new transformation x' used by the law.

5.2 Bidirectional Transformations

The transformations in this category will reflect all modifications on the target documents back to the source documents or the transformations themselves. Some of these transformations are primitively defined, while others are defined based on the transformation combinators.

`XID`: Suppose $x = \texttt{new XID}()$. Then the behavior of x is defined as follows:

$$\begin{array}{rcl} x.\texttt{tranForward}(sd) & = & sd \\ x.\texttt{tranBackward}(sd, td) & = & td \end{array}$$

`XConst`: Suppose $x = \texttt{new XConst}(elm)$, where *elm* is an element. Then the behavior of x is defined as follows:

$$\begin{array}{rcl} x.\texttt{tranForward}(sd) & = & elm \\ x.\texttt{tranBackward}(sd, td) & = & sd\ \texttt{with} \\ & & \quad x' = \texttt{new XConst}(td) \end{array}$$

Note that after backward transformation, the source document keeps unchanged but the transformation is changed into x'.

`XHide`: Suppose $x = \texttt{new XHide}()$. Then the behavior of x is defined as follows:

$$\begin{array}{rcl} x.\texttt{tranForward}(sd) & = & () \\ x.\texttt{tranBackward}(sd, ()) & = & sd \end{array}$$

`XModifyName`: Suppose $x = \texttt{new XModifyName}(nm)$ and $sd = <tag>\{contents\}$. Then the behavior of x is defined as follows:

$$\begin{array}{rcl} x.\texttt{tranForward}(sd) & = & <nm>\{contents\} \\ x.\texttt{tranBackward}(sd, td) & = & <tag>\{contents'\},\ \texttt{with} \\ & & \quad x' = \texttt{new XModifyName}(nm') \\ & & \quad \texttt{where}, <nm'>\{contents'\} = td \end{array}$$

`XNewRoot`: Suppose $x = \texttt{new XNewRoot}(nm)$. Then the behavior of x is defined as follows:

$$\begin{array}{rcl} x.\texttt{tranForward}(sd) & = & <nm>sd \\ x.\texttt{tranBackward}(sd, td) & = & sd', \texttt{with} \\ & & \quad x' = \texttt{new XNewRoot}(nm') \\ & & \quad \texttt{where}, <nm'>\{sd'\} = td \end{array}$$

`XDistribute`: Suppose $x = \texttt{new XDistribute}(n)$, where n is a natural number. Then the behavior of x is defined as follows:

$$\begin{array}{rcl} x.\texttt{tranForward}(sd) & = & <\texttt{virtual}>\{sd_0, ..., sd_{n-1}\} \\ & & \quad \texttt{where}\ sd_i = sd \\ x.\texttt{tranBackward}(sd, td) & = & sd' \\ & & \quad \texttt{where}, <nm>\{td_0, ..., td_{n-1}\} = td \\ & & \quad sd' = \texttt{merge}(sd, \{td_0, ..., td_{n-1}\}) \end{array}$$

Note `merge` is an auxiliary function, which takes the source document *sd* and a list of modified copies of *sd*, and then merges all modifications on each copy and returns an updated document. If several modifications on different copies occur at the same place with respect to the old source document, then they are put together and separated by symbol #, which indicates the users to reconcile the conflicts. `XDistribute` is more general than `Dup` in [4], which is used to maintain data dependency relation in target documents. Moreover, due to using `merge` function, so `XDistribute` allows to update source document in a batch style rather than the interactive style adopted by `Dup`, so it is suitable for network environments.

`XChildren`: Suppose $x = \texttt{new XChildren}(nm)$. This transformation is a derived transformation, which is defined as follows:

$$\begin{array}{rcl} x.\texttt{tranForward}(sd) & = & tran.\texttt{tranForward}(sd) \\ x.\texttt{tranBackward}(sd, td) & = & tran.\texttt{tranBackward}(sd, td) \\ \texttt{where}, tran & = & \texttt{new XSeq}(\{x_0, x_1\}) \\ x_0 & = & \texttt{new XMap}(x_2) \\ x_1 & = & \texttt{new XModifyName}(\text{“}\texttt{virtual}\text{”}) \\ x_2 & = & \texttt{new XIf}(pred, x_3, x_4) \\ x_3 & = & \texttt{new XID}() \\ x_4 & = & \texttt{new XHide}() \\ pred & = & \texttt{new XWithTag}(nm) \end{array}$$

We also implement another commonly used axis, the `descendant` axis, in class `XDescendant`.

`XCollapse`: Suppose $x = \texttt{new XCollapse}(nm)$. Informally, when using x to transform a document forwardly, it removes each element node with name *nm* and lifts its children one level up, which is done

by transformation `XLift`. This transformation is also a derived transformation, which is defined as follows:

$$\begin{aligned} x.\texttt{tranForward}(sd) &= tran.\texttt{tranForward}(sd) \\ x.\texttt{tranBackward}(sd, td) &= tran.\texttt{tranBackward}(sd, td) \\ \texttt{where}, tran &= \texttt{new XFold}(\{x_0, x_1\}) \\ x_0 &= \texttt{new XLift}(nm) \\ x_1 &= \texttt{new XID}() \end{aligned}$$

This definition uses a transformation combinator `XFold`, which has the same definition as in [4]. Briefly, it applies x_0 to the intermediate nodes in an XML data model and x_1 to the leaf node, respectively, in a bottom-up manner.

5.3 Restrictive Bidirectional Transformations

The transformations in this category only generate read-only target documents. Since there is no modification on the target documents, the backward transformation is degraded to jut keep the original source document unchanged. The classes of these transformation can be inherited from class `XActionNFun`, which has implemented the trivial backward transformations, so users just need to implement the forward transformations.

This category of transformations can serve two purposes. First, some forward transformations may not be invertible, so the target documents should be read-only. For example, if some data in target document is the number of child elements in a source document, then these data should not be modified. Second, this feature can help to incorporate other Java XML transformation code into the bidirectional transformation framework.

6 Bidirectionalizing XQuery and XSLT

In order to test the expressiveness and the usability of the bidirectional transformations introduced above, we use them to bidirectionalize some typical examples in XQuery and XSLT, which are both popular and widely used XML processing languages.

All examples in this section use XML file "lib.xml" as the source document, which is partially listed in Figure 3. In this document, the data of interest by users are the engineering books written by Tom with price less than 50 dollar. In the following , these interesting book information will be transformed into different formats by XQuery and XSLT, respectively.

```
<lib>
  <name>TU Lib</name>
  <shelf>
    <category>Engineering</category>
    <cabinet>
      <book>
        <title>Data Structure</title>
        <author>Tom</author>
        <price>33</price>
        <year>2004</year>
        <publisher>
          <name>TU Press</name>
          <addr>US</addr>
        </publisher>
      </book>
      ......
    </cabinet>
    ......
  </shelf>
  <shelf>
    <category>Science</category>
    .......
  </shelf>
</lib>
```

図 3: An XML Document of Lib

```
<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>
```

図 4: An XQuery Expression

6.1 Bidirectionalization of XQuery Expression

The structure of XQuery expressions generally takes FLWR form. Here, we will implement the XQuery expression in Figure 4, which involves `for`, `where` and `return` expressions. This expression

generates an element with name “Books-of-Tom”, in which there is a list of elements `book`, which then contains its title, price and publisher name.

The implementation code in our library is listed in Figure 5. The whole script consists of a sequence of transformation code wrapped by `xseq`. A simple way of writing this script is to finish this script sequence step by step, that is, adding a new transformation to the end of the existing script and looking at the transformation result, and then repeating this procedure until the expected result is gotten. However, it seems a tedious work. In the future, we will give algorithms to transform the code of high level XML processing languages into `BiXJ` code. Here, we informally introduce some experiences learned when writing such script.

The XQuery expression in Figure 4 consists of two kinds of subexpressions: one kind is used in constructing the target element and the other is used to destruct the source element. The first kind of expressions includes three element constructors: `Books-of-Tom`, `book` and `press`. For them, they are encoded as the following form:

`<xnewroot>`*element name*`</newroot>`
`<xcollapse>virtual</xcollapse>`

And, they are generally put at the end of script sequence resulting from the bidirectionalization of their contents. The second kind expressions include `XPath` expressions. To encode XPath children axis, script `xchildren` can be used, probably with the help of `xmap` or `zxip` because the context elements in our work are always wrapped by a virtual node. `xmap` takes the similar role as `for` expression in `XQuery`. For `text()` operation in XPath expression, we modify the names of context elements into “virtual”, so that the text contents can be gotten after collecting virtual tags. This is a trick since the transformation interface only allows to return elements, not strings.

6.2 Bidirectionalization of XSLT

Another encoding strategy worth mentioning is that each transformation in the script sequence generates one element, which will be consumed by the only successive transformation. However,

```
<xseq>
  <xchildren>shelf</xchildren>
  <xmap>
    <xif>
      <xequals>
        <path>0</path>
        <value>Engineering</value>
      </xequals>
      <xid/> <xhide/>
    </xif>
  </xmap>
  <xmap><xchildren>cabinet</xchildren></xmap>
  <xcollapse>virtual</xcollapse>
  <xmap><xchildren>book</xchildren></xmap>
  <xcollapse>virtual</xcollapse>
  <xmap>
    <xif>
      <xand>
        <xequals>
          <path>1</path><value>Tom</value>
        </xequals>
        <xlessthan>
          <path>2</path><value>50</value>
        </xlessthan>
      </xand>
      <xid/><xhide/>
    </xif>
  </xmap>
  <xmap>
    <xseq>
      <xdistribute>3</xdistribute>
      <xzip>
        <xchildren>title</xchildren>
        <xchildren>price</xchildren>
        <xseq>
          <xchildren>publisher</xchildren>
          <xmap><xchildren>name</xchildren>
          </xmap>
          <xcollapse>virtual</xcollapse>
          <xmap>
            <xmodifyname>virtual</xmodifyname>
          </xmap>
          <xnewroot>press</xnewroot>
          <xcollapse>virtual</xcollapse>
        </xseq>
      </xzip>
      <xnewroot>book</xnewroot>
      <xcollapse>virtual</xcollapse>
    </xseq>
  </xmap>
  <xnewroot>Books-of-Tom</xnewroot>
  <xcollapse>virtual</xcollapse>
</xseq>
```

図 5: Bidirectionalization of XQuery Expression

```
<xsl:stylesheet xmlns:xsl=...>
  <xsl:template match="/">
    <html>
      <title>Books-of-Tom</title>
      <body>
        <table>
          <tr>
            <th>title</th>
            <th>price</th>
            <th>press</th>
          </tr>
          <xsl:apply-templates />
        </table>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="/lib">
    <xsl:apply-templates select="shelf[category = 'Engineering']"/>
  </xsl:template>
  <xsl:template match="shelf">
    <xsl:apply-templates select="cabinet"/>
  </xsl:template>
  <xsl:template match="cabinet">
    <xsl:apply-templates
      select="book[author = 'Tom' and price &lt; '50']"/>
  </xsl:template>
  <xsl:template  match="book">
    <tr>
      <td><xsl:value-of select="title"/></td>
      <td><xsl:value-of select="price"/></td>
      <td><xsl:value-of select="publisher/name"/></td>
    </tr>
  </xsl:template>
</xsl:stylesheet>
```

図 6: An XSLT Expression

in XQuery expressions, a context element can be consumed by several parallel successive transformations. For example, the three parallel subexpressions under element $<$ `book` $>$ consume one input element `$b`. In such case, `xdistribute` can be used to copy the input element to the numbers required by the successive transformations, and then `xzip` applies each transformation independently to each copy.

The style sheet of XSLT generally is made up of a list of templates, which are connected by `apply-templates`. The style sheet we will implement is shown in Figure 6, which includes five templates, with the purpose of transforming the data of interest in the source document into a HTML file.

The implementation code is in Figure 7. The code is divided into three parts for readability. The top level code corresponds to the first template in Figure 6; the code x_0 corresponds to the second, third and fourth templates, which extracts the interesting book information from the source document; the code x_1 does the same thing as the last template to construct table rows.

The bidirectionalizing procedure starts from dealing with the top template, and when meeting with an `apply-templates`, then puts here the bidirectionalizing result of the applied template. For each template, we follow the same rules as used in implementing the XQuery expression. For example, in the top template, tag `html` is used to construct the result document, so we have

`<xnewroot>html</xnewroot>`

`<xcollpase>virtual</xcollapse>`

in the end of script; then, tag `<title>` and `<body>` are considered as the parallel processing of the same input element, so `xdistribute` is used to duplicate the input, and `xzip` applies the corresponding transformation to each copy.

7 Related Work

This work takes similar bidirectional transformation style as those work [3, 4]. The languages in these existing work are both domain specific. So they have some limitations to be used as the general XML processing languages. For example, the language in [3] take an un-ordered tree data model with no repeated labels, so some critical combinators, like `xfork`, seem not to be applicable over XML transformation directly. While `BiXJ` extends the techniques they introduced to the general XML processing area.

In database area, there are also work to do XQuery updating. For example, the work in [9] transforms updates on query tree into SQL updates, and then uses such SQL expressions to update the source data. Obviously, this technique is not suit-

```
-------------Top level code-------------
<xseq>
  <xdistribute>2</xdistribute>
  <xzip>
    <xconst>
      <title>Books-of-Tom</title>
    </xconst>
    <xseq>
      <xdistribute>2</xdistribute>
      <xzip>
        <xconst>
          <tr>
            <th>title</th>
            <th>price</th>
            <th>press</th>
          </tr>
        </xconst>
        <xseq>x0  x1</xseq>
      </xzip>
      <xnewroot>table</xnewroot>
      <xcollapse>virtual</xcollapse>
      <xnewroot>body</xnewroot>
      <xcollapse>virtual</xcollapse>
    </xseq>
  </xzip>
  <xnewroot>html</xnewroot>
  <xcollapse>virtual</xcollapse>
</xseq>
--------------Code x0-------------------
<xchildren>shelf</xchildren> <xmap>
  <xif>
    <xequals>
      <path>0</path>
      <value>Engineering</value>
    </xequals>
    <xid/>
    <xhide/>
  </xif>
</xmap> <xmap><xchildren>cabinet</xchildren></xmap>
<xcollapse>virtual</xcollapse>
<xmap><xchildren>book</xchildren></xmap>
<xcollapse>virtual</xcollapse> <xmap>
  <xif>
    <xand>
      <xequals>
        <path>1</path>
        <value>Tom</value>
      </xequals>
      <xlessthan>
        <path>2</path>
        <value>50</value>
      </xlessthan>
    </xand>
    <xid/>
    <xhide/>
  </xif>
</xmap>
```

```
--------------Code x1-------------------
<xmap>
  <xseq>
    <xdistribute>3</xdistribute>
    <xzip>
      <xseq>
        <xchildren>title</xchildren>
        <xmap>
          <xmodifyname>virtual</xmodifyname>
        </xmap>
        <xnewroot>td</xnewroot>
        <xcollapse>virtual</xcollapse>
      </xseq>
      <xseq>
        <xchildren>price</xchildren>
        <xmap>
          <xmodifyname>virtual</xmodifyname>
        </xmap>
        <xnewroot>td</xnewroot>
        <xcollapse>virtual</xcollapse>
      </xseq>
      <xseq>
        <xchildren>publisher</xchildren>
        <xmap>
          <xchildren>name</xchildren>
        </xmap>
        <xcollapse>virtual</xcollapse>
        <xmap>
          <xmodifyname>virtual</xmodifyname>
        </xmap>
        <xnewroot>td</xnewroot>
        <xcollapse>virtual</xcollapse>
      </xseq>
    </xzip>
    <xnewroot>tr</xnewroot>
    <xcollapse>virtual</xcollapse>
  </xseq>
</xmap>
```

図 7: Bidirectionalization of XSLT Expression

able for updating native XML repositories. And also, it cannot be used to update the view defined by XSLT.

The work [10] studies the problem of bidirectionalizing HaXML [11] and shows any transformations in HaXML can be compiled into a bidirectional transformation. In work [12], the authors give an injective language `Inv` to implement view updating, and due to injective, so each program is invertible. While they are still not used in general XML processing.

8 Conclusion

In this paper, we solve the problem of view updating in general XML processing area. The proposed solution is a Java library `BiXJ` for bidirectional XML transformation. By this library, when users write one transformation, the backward transformation can be gotten for free. So there is no extra effort or separated mechanisms needed for users to update the source document after the view is modified. We have demonstrated the expressiveness and usability of `BiXJ` by bidirectionalizing the typical examples of two popular XML processing languages XQuery and XSLT.

In the future, we will develop algorithms that can translate XLST or XQuery expressions into the code of `BiXJ` automatically.

参考文献

[1] W3C Draft. XSL Transformations (XSLT) Version 2.0 . http://www.w3.org/TR/xslt20/, 2005.

[2] W3C Draft. XML Query (XQuery) . http://www.w3.org/XML/Query, 2005.

[3] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2005.

[4] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 2004.

[5] J. Hunter and B. McLaughlin. JDOM Project. http://www.jdom.org.

[6] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, 2003.

[7] D. Liu, Z. Hu, and M. Takeichi. An environment for maintaining computation dependency in XML documents. In *Proceedings of ACM Symposium on Document Engineering*, 2005. To appear.

[8] F. Bancilhon and N. Spyratos. Update semantics of relational views. *TODS*, 6:557–575, 1981.

[9] V. Braganholo, S. Davidson, and C. Heuser. From XML view updates to relational view updates: old solutions to a new problem. In *Proceedings of International Conference on Very Large Databases (VLDB)*, 2004.

[10] Z. Hu, K. Emoto, S. Mu, and M. Takeichi. Bidirectionalizing tree transformations. In *Workshop on New Approaches to Software Construction (WNASC 2004)*, 2004.

[11] Malcolm Wallace and Colin Runciman. Haskell and XML: generic combinators or type-based translation? In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, 1999.

[12] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An algebraic approach to bidirectional updating. In *Second ASIAN Symposium on Programming Languages and Systems (APLAS 2004)*, 2004.