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 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. Section 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,

```
<books>
</books>
```

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.

```
```

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>
</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.

```xml
<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:

```java
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 a 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>
</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:

<xnewroot>book</xnewroot>
<xcollapse>virtual</xcollapse>
</xseq>

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

<xnewroot>mybooks</xnewroot>
<xcollapse>virtual</xcollapse>
</xseq>

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
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].

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

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:

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

where sd is the source document and td is gotten by modifying the original target document x \cdot \text{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:

\[
x' \cdot \text{tranForward}(x \cdot \text{tranBackward}(sd, td)) <: td
\]

in which td’ <: td means all sub-
trees of \( td' \) appears in \( td \), where
\[
td' = x'.\text{tranForward}(x.\text{tranBackward}(sd,td)).
\]
The most important feature of this law is that when doing forward transformation again, the used
transformation \( x' \) is different from that used in previous time, i.e., transformation \( x \). The difficult of
using this law lies in seeking the updated transformation \( x' \), 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 \( x' \) 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 \( \text{tranBackward} \) 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:

\[
\begin{align*}
\text{element} & := \ <\text{tag}>\ \{\text{element},...,\text{element}\} \\
& \quad | \ <\text{tag}>\ \{\text{string}\} \\
\text{tag} & := \ \text{string}
\end{align*}
\]

Note that \( () \) is a special element, and in implement-
ation, 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 \( \text{XSeq} \), \( \text{XMap} \), \( \text{XZip} \) and \( \text{XIf} \). And, which category the transformations expressed in
these classes belong to is determined by the trans-
formation arguments they take. Below, we will de-
scribe each of them in brief.

\( \text{XSeq} \): Suppose \( x_0, x_1, ..., x_n \) are some transform-
ations objects and \( x = \text{new XSeq}\{x_0, x_1, ..., x_n\} \).

Then the behavior of \( x \) is defined as follows:
\[
\begin{align*}
x.\text{tranForward}(sd) & = \ d_n \\
\text{where, } d_n & = \ x_n.\text{tranForward}(sd_{n-1}) \\
d_{n-1} & = \ x_{n-1}.\text{tranForward}(sd_{n-2}) \\
& \quad \quad \vdots \\
d_0 & = \ x_0.\text{tranForward}(sd)
\end{align*}
\]

\[
\begin{align*}
x.\text{tranBackward}(sd,td) & = \ d'_n \\
\text{where, } d'_n & = \ x_n.\text{tranBackward}(sd_{n-1},td) \\
d'_{n-1} & = \ x_{n-1}.\text{tranBackward}(sd_{n-2},td) \\
& \quad \quad \vdots \\
d'_0 & = \ x_0.\text{tranBackward}(sd,td)
\end{align*}
\]

\( \text{XIf} \): Suppose \( x_0 \) and \( x_1 \) are two transforma-
tions objects and \( x = \text{new XIf}\{\text{pred},x_0,x_1\} \). Then the be-

behavior of $x$ is defined as follows:

$$x._{\text{tranForward}}(sd) = d$$
$$x._{\text{tranBackward}}(sd, td) = d'$$

where, $d = x_0._{\text{tranForward}}(sd)$

if $\text{pred}.qualify(sd)$

$d = x_1._{\text{tranForward}}(sd)$

otherwise

where, $d' = x_0._{\text{tranBackward}}(sd, td)$

if $\text{pred}.qualify(sd)$

$d' = x_1._{\text{tranBackward}}(sd, td)$

otherwise

In this definition, $\text{pred}$ is an object with interface $\text{XPredicate}$, which has a method $\text{qualify}$ to judge whether an element satisfy a predicate. We have implemented the most commonly used predicates like $\text{XLessThan}$, $\text{XGreaterThan}$, $\text{XEquals}$, $\text{XHasChild}$ and $\text{XWithTag}$, as well as predicate operators $\text{XAnd}$, $\text{XOr}$ and $\text{XNot}$. There is some trouble to prove $\text{XIf}$ satisfies the last law in 4. The idea is that if $\text{pred}$ is dependent on some data content, which also appears in target document, then $\text{pred}$ 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 combinator.

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

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

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

$$x._{\text{tranForward}}(sd) = elm$$
$$x._{\text{tranBackward}}(sd, td) = \text{sd with } x' = \text{new XConst(td)}$$

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

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

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

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

$$x._{\text{tranForward}}(sd) = <\text{nm}>{\text{contents}}$$
$$x._{\text{tranBackward}}(sd, td) = <\text{tag}>{\text{contents}}', \text{ with } x' = \text{new XModifyName(nm')}$$

where, $<\text{nm'}>{\text{contents'}} = td$

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

$$x._{\text{tranForward}}(sd) = <\text{nm}>sd$$
$$x._{\text{tranBackward}}(sd, td) = sd', \text{ with } x' = \text{new XNewRoot(nm')}$$

where, $<\text{nm'}>{\text{sd'}} = td$

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

$$x._{\text{tranForward}}(sd) = <\text{virtual}>({\text{sd}}, \ldots, {\text{sd}_{n-1}})$$
$$x._{\text{tranBackward}}(sd, td) = \text{sd'}$$

where, $<\text{nm}>{\text{td}}, \ldots, {\text{td}_{n-1}} = td$

$$sd' = \text{merge}(sd, {\text{td}}, \ldots, {\text{td}_{n-1}})$$

Note $\text{merge}$ 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 $\text{merge}$ 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 $x = \text{new XChildren(nm)}$. This transformation is a derived transformation, which is defined as follows:

$$x._{\text{tranForward}}(sd) = \text{trans}_0._{\text{tranForward}}(sd)$$
$$x._{\text{tranBackward}}(sd, td) = \text{tr}_{\text{trans}_0}. _{\text{tranBackward}}(sd, td)$$

where, $\text{trans}_0 = \text{new XSeq}({{x_0}, x_1})$

$$x_0 = \text{new XMap(x_2)}$$

$$x_1 = \text{new XModifyName("virtual")}$$

$$x_2 = \text{new XIf}(\text{pred}, x_3, x_4)$$

$$x_3 = \text{new XID()}$$

$$x_4 = \text{new XHide()}$$

$$\text{pred} = \text{new XWithTag(nm)}$$

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

**XCollapse**: Suppose $x = \text{new XCollapse(nm)}$. 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:
\[
\begin{align*}
  x.\text{tranForward}(sd) &= \text{tran}.\text{tranForward}(sd) \\
  x.\text{tranBackward}(sd, td) &= \text{tran}.\text{tranBackward}(sd, td) \\
  \text{where, tran} &= \text{new XFold}\{x_0, x_1\} \\
  x_0 &= \text{new XLift}(nm) \\
  x_1 &= \text{new XID}()
\end{align*}
\]

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, re-
spectively, in a bottom-up manner.

5.3 Restrictive Bidirectional Transforma-
tions

The transformations in this category only gen-
erate read-only target documents. Since there is
no modification on the target documents, the back-
ward transformation is degraded to just 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 im-
plement 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 tar-
get document is the number of child elements in
a source document, then these data should not be
modified. Second, this feature can help to incorpo-
rate other Java XML transformation code into the
bidirectional transformation framework.

6 Bidirectionalizing XQuery and
XSLT

In order to test the expressiveness and the us-
ability of the bidirectional transformations intro-
duced above, we use them to bidirectionalize some
typical examples in XQuery and XSLT, which are
both popular and widely used XML processing lan-
guages.

All examples in this section use XML file
"lib.xml" as the source document, which is partially

```xml
<lib>
  <name>TU Lib</name>
</lib>

<name>TU Lib</name>
</shelf>
  <name>TU Lib</name>
</shelf>
  <name>TU Lib</name>
</shelf>
  <name>TU Lib</name>
</shelf>

<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>
  </book>
</Books-of-Tom>
```

6.1 Bidirectionalization of XQuery Expres-
sion

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

```xml
<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>
  </book>
</Books-of-Tom>
```
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 destroy 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 xzip 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,
in XQuery expressions, a context element can be consumed by several parallel successive transformations. For example, the three parallel sub-expressions 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

```xml
<xsl:stylesheet xmlns:xsl="...">
  <xsl:template match="/">
    <html>
      <title>Books-of-Tom</title>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

图 6: An XSLT Expression

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 combinator, 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>
          <xseq>
            <xchildren>title</xchildren>
            <xmap>
              <xif>
                <xequals>
                  <path>0</path>
                  <value>Engineering</value>
                </xequals>
                <xid/>
              </xif>
            </xmap>
          </xseq>
        </xconst>
        <xseq>
          <xchildren>price</xchildren>
          <xmap>
            <xand>
              <xequals>
                <path>1</path>
                <value>Tom</value>
              </xequals>
            </xand>
            <xlessthan>
              <path>2</path>
              <value>50</value>
            </xlessthan>
            <xif>
              <xhide/>
            </xif>
          </xmap>
        </xseq>
      </xzip>
    </xseq>
  </xzip>
</xseq>
```

---Code $x_1$---

```
<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>
      </xseq>
    </xzip>
  </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 \( \text{Inv} \) 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