A Programmable Editor for Developing Structured Documents based on Bidirectional Transformations

Zhenjiang Hu   Shin-Cheng Mu   Masato Takeichi

Department of Mathematical Informatics, University of Tokyo
7-3-1 Hongo, Bunkyo, Tokyo 113-8656, Japan
{hu,takeichi,scm}@mist.i.u-tokyo.ac.jp

Abstract

This paper presents a novel editor supporting interactive refinement in the development of structured documents. The user performs a sequence of editing operations on the document view, and the editor automatically derives an efficient and reliable document source and a transformation that produces the document view. The editor is unique in its programmability, in the sense that transformation can be obtained through editing operations. The main tricks behind are the utilization of the view-updating technique developed in the database community, and a new bidirectional transformation language that cannot only describe the relationship between the document source and its view, but also data dependency in the view.

Keywords: XML, Presentation-oriented Editor, Document Engineering, Functional Programming, Bidirectional Transformation, View Updating

1   Introduction

XML [BPSM98] has been attracting a tremendous surge of interest as a universal, queryable representation for structured documents. Everyday, a countless number of structured documents in XML are constructed, and so many editors [Sof04] are designed and implemented to support the construction of XML documents. This has in part been stimulated by the growth of the Web and e-commerce, where XML has emerged as the *de facto* standard for representation of structured documents and information interchange. While the existing XML editors are helpful for the *creation* of the documents, they are rather weak to support development of structured documents in the sense they hardly provide powerful mechanism for dynamic *refinement* of the structured documents.

Let us take a close look at the process of using existing editors with an example of construction of an address book. It basically includes three steps: designing a suitable document type, constructing an XML document with the designed type for storing information, and defining a transformation for viewing the document. We may start by defining an address book type (Figure 1), which allows an arbitrary number of people's addresses including a name, some email addresses if there are, and a telephone number. Then, we construct an XML document (Figure 2) of this type to store address information. And finally, we define a

transformation (Figure 3) to display[1] the address book in a friendly way (Figure 4), say by sorting persons according to the last names and adding an name index. Notice the difference between the two XML documents, the original XML document in Figure 2 and the view in Figure 4. Besides difference in their structures, the former has no redundancy information, while the later does; e.g., names appear twice in the view. The result of this development is a structured document with three components: a data type definition, an XML document representing the source data, and a transformation for viewing the data.

While the final documents may be of static and physical form, the documents themselves are a fluid, evolving object. It is also observed that document development follows a lifecycle similar to the development of computer programs, in which the document is iteratively refined. However, the existing editors do not support this interactive refinement (updating) very well:

- First, they treat the three components of structured documents independently, which makes it hard to keep them consistent with each other. Take the address book example, if we want to make a change on the data type by splitting the telephone number (`tel`) into two parts, country code (`ccode`) and local code (`tel`), to share the country code, we may refine the document type definition in Figure 1 to that in Figure 5. This refinement requires corresponding changes on the XML document and the transformation, which is difficult.

- Second, they expect the users to be XML experts knowing DTD, XML, and XSLT for the construction of the three components of structured documents. This may be rather disappointing to those who know very little about XML (for example, whose who possessing merely some basic knowledge of HTML), but still want to create structured documents in their daily work. In fact, more and more people nowadays want to be able to create their structured documents in an user-friendly manner, pretty much like how spreadsheets are created. The intuitive interface of the latter contributes a lot to its popularity.

In this paper, we propose a novel programmable editor that supports interactive refinement during the development

---

[1]To simplify our presentation, we consider the view as another XML data. It should be very straightforward to present this XML data in another format with a suitable stylesheet description.

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

Figure 1: A DTD for the Address Book

```
<addrbook>
  <person>
    <name> Masato Takeichi </name>
    <email> takeichi@acm.org </email>
    <tel> +81-3-5841-7430 </tel>
  </person>
  <person>
    <name> Zhenjiang Hu </name>
    <email> hu@mist.i.u-tokyo.ac.jp </email>
    <email> hu@ipl.t.u-tokyo.ac.jp </email>
    <tel> +81-3-5841-7411 </tel>
  </person>
  <person>
    <name> Shin-Cheng Mu </name>
    <email> scm@mist.i.u-tokyo.ac.jp </email>
    <tel> +81-3-5841-7411 </tel>
  </person>
</addrbook>
```

Figure 2: An XML Document of the Address Book

```
<addrbook>
  <index>
    <name> Zhenjiang Hu </name>
    <name> Shin-Cheng Mu </name>
    <name> Masato Takeichi </name>
  </index>
  <person>
    <name> Zhenjiang Hu </name>
    <email> hu@mist.i.u-tokyo.ac.jp </email>
    <email> hu@ipl.t.u-tokyo.ac.jp </email>
    <tel> +81-3-5841-7411 </tel>
  </person>
  <person>
    <name> Shin-Cheng Mu </name>
    <email> scm@mist.i.u-tokyo.ac.jp </email>
    <tel> +81-3-5841-7411 </tel>
  </person>
  <person>
    <name> Masato Takeichi </name>
    <email> takeichi@acm.org </email>
    <tel> +81-3-5841-7430 </tel>
  </person>
</addrbook>
```

Figure 4: A View of the Address Book in XML

```
<xsl:template match="/addrbook">
  <addrbook>
    <index>
      <xsl:for-each select=''person''>
        <xsl:sort select=''name''/>
        <xsl:value-of select=''name''/>
      </xsl:for-each>
    </index>
    <xsl:for-each select=''person''>
      <xsl:sort select=''name''/>
      <xsl:value-of select=''person''/>
    </xsl:for-each>
  </addrbook>
 </xsl:template>
```

Figure 3: A Transformation in XSLT

```
<!ELEMENT addrbook (ccode, person*)>
<!ELEMENT ccode (#PCDATA)>
<!ELEMENT person (name, email*, tel)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT tel (#PCDATA)>
```

Figure 5: An DTD for the Address Book

of structured documents. Given a sequence of editing operations on the view and a data type definition for the final view, an efficient and reliable structured document with the three basic components can be obtained automatically. The main trick behind this editor is a new bidirectional transformation language describing the relationship among the source data, the view, and the transformation between the source data and the view.

Our main contributions can be summarized as follows.

- We, as far as we are aware, are the first to recognize the importance of the view-updating technique in the design of the editors for interactive development of structured documents. The view-updating technique [BS81, DB82, GPZ88, OT94, Abi99] has been intensively studied in the database community, where modification on the view can be reflected back to the original database. We borrow this technique and use it in the design of our editor with a significant extension not exploited before: editing operations can modify not only the view but also the transformation (from the database to the view).

- We have designed a powerful language for the specification of the relationship between the original data and the view. Comparing to similar languages in [Mee98, GMPS03], our language is more capable of describing data dependency, thanks to a new language construct allowing duplication. The language is powerful enough to describe the editing operations (insert, delete, move, and copy) as well as other important transformations.

- We have successfully implemented our idea in a prototype editor. The editor is particularly interesting in its programmability feature, and a unified presentation-oriented interface for developing the three components through editing operations on the view of the structured documents.

  - *Presentation Oriented*: we provide a uniform view-based editing interface for users to construct and refine their documents. It looks like a tree version of the spreadsheet, which is easy to use.
  - *Programmable*: transformation programs can be constructed though interactive editing operations. In fact, thanks to the bidirectional transformation, the three basic components of structured documents can be automatically derived, after the edition on the view.

The rest of the paper is organized as follows. We start by giving a simple definition of structured documents in functional notations, and demonstrate what our editor can do in Section 2. After defining the bidirectional transformation language that plays an important role in our editor in Section 3, we propose the design principle and implementation technique in Section 4. Related work and conclusions are given in Sections 5 and 6 respectively.

2 Developing Structured Documents

In this section, we introduce the notations that will be used to describe structured documents in this paper, before demonstrating through a concrete example how our editor can support development of structured documents.

```
data Addrbook = Addrbook [Person]
data Person = Person (Name, [Email], Tel)
data Name = Name String
data Email = Email String
data Tel = Tel String
```

Figure 6: Address Book Type in Haskell

```
addrbook = Addrbook
  [ Person ( Name "Masato Takeichi",
             [ Email "takeichi@acm.org" ],
             Tel "+81-3-5841-7430" )
    Person ( Name "Zhenjiang Hu",
             [ Email "hu@mist.i.u-tokyo.ac.jp",
               Email "hu@ipl.t.u-tokyo.ac.jp" ]
             Tel "+81-3-5841-7411" )
    Person ( Name "Shin-Cheng Mu" )
             [ Email "scm@mist.i.u-tokyo.ac.jp" ]
             Tel "+81-3-5841-7411" ) ]
```

Figure 7: Source Data for Address Book in Haskell

2.1 Representation of Structured Documents

A *structured document* is defined by a triple (T, D, X), where T denotes the document type, D the document source that has type T, and X the transformation to map the document source the its view for display. The document displayed to the suer is called the *view*. For instance, the structured document in the introduction specifies T using DTD, D using XML, and X using XSLT.

For conciseness, we choose the Haskell-like[2] notations instead of XML to represent structured documents. Figures 6, 7 and 8 specify the structured document in the introduction in our notation; their correspondence is rather clear except for the transformation which will be explained in detail in Section 3.

Our formulation is significantly simplified from the XML standard in many ways. First, we omitted attributes, which should not be too difficult to cope with by some simple extension. Second, we do not allowed IDRef to refer to other nodes through a unique identifier. This will be one of our future work.

2.2 Generic Representation

Any tree constructed in the way described above is type-checked by the Haskell type system, which is a good thing for the final document. However, for the interactive refinement of the documents, we should allow inconsistency during document development. To this end, we make use of the following generic tree that represents contents of any XML document, independent of all DTDs.

```
data Tree = N String [Tree]
```

The following gives an example of this representation of the the document source in Figure 7.

---

[2]In fact, we could have chosen HaXML [WR99], XDuce [HVP00] or CDuce [BCF03] which are designed for specification of XML documents in functional languages. Instead, we used only the standard Haskell notations, therefore readers need only the basic knowledge of functional programming to understand this paper.

```
      sortX ;
      applyX [] Dup ;
      applyX [1] (modifyRootX "Index" ; Map keepX) ;
      copyX [1] [2, 1] ;
      deleteX ;
      hoistX "Dup"
```

Figure 8: Transformation in X

```
addrbook = N "Addrbook"
  [N "Person"
    [N "Name" [N "Masato Takeichi" []],
     N "Email" [N "takeichi@acm.org" []],
     N "Tel" [N "+81-3-5841-7430" []]],
   N "Person"
    [N "Name" [N "Zhenjiang Hu" []],
     N "Email" [N "hu@mist.i.u-tokyo.ac.jp" [],
                N "hu@ipl.t.u-tokyo.ac.jp" []],
     N "Tel" [N "+81-3-5841-7411" []]],
   N "Person"
    [N "Name" [N "Shin-Cheng Mu" []],
     N "Email" [N "scm@mist.i.u-tokyo.ac.jp" []],
     N "Tel" [N "+81-3-5841-7430" []]]]
```

It seems that the generic representation does not distinguish tag names from texts, since both of them are represented by strings. As a matter of fact, we can think of labels attached to inner nodes as tag names, and labels to leaves as text.

2.3 Development of Structured Documents: An Example

We view the development of structured documents as the process of constructing a triple (T, D, X) meeting the requirements the designer had in mind. Such a development is an interactive refinement process, therefore, it is not reasonable to expect consistency of the document all the time. This is why we introduce the generic form, and its use will be shown in this section.

We illustrate the main idea of our editor by going through the development of the address book in the introduction. From scratch, we start with an empty document with only one node labeled "Root":

```
N "Root" []
```

In the demonstration to follow, we will construct, via interaction with the editor, the triple (T, D, X) like those in Figures 6, 7, and 8, such that the resulting view looks like that in Figure 4.

The node or subtree in focus, on which the user performs editing operations, is selected by a cursor. Here, for simplicity, we use a path to denote the subtree we select. A path is a sequence of positive integers $[a_1, a_2, \ldots, a_n]$, denoting the subtree obtained by going into the a_1-th child of the root, then into the a_2-th child, and so on. For example, [] denotes the root node (or the entire tree), and [1] denotes the first child of the root.

The complete list of operations the user can perform on the focused subtrees is given in Section 4. In this section, we will only make use of the following operations: insert, copy,

duplicate, delete, and editing of the names of tags. We first change the label "Root" to "Addrbook",

```
N "Addrbook" []
```

and, by the insert operation, we insert a name and some contacts information as a subtree of the root (the node at position []), which could be done by inserting nodes one by one.

```
N "Addrbook"
  [N "Person"
    [N "Name" [N "Masato Takeichi" []],
     N "Email" [N "takeichi@acm.org" []],
     N "Tel" [N "+81-3-5841-7430" []]]]
```

We may continue to add another person's contacts by copying the subtree rooted at [1] using the copy operation. The copied tree becomes a sibling of the original:

```
N "Addrbook"
  [N "Person"
    [N "Name" [N "Masato Takeichi" []],
     N "Email" [N "takeichi@acm.org" []],
     N "Tel" [N "+81-3-5841-7430" []]],
   N "Person"
    [N "Name" [N "Masato Takeichi" []],
     N "Email" [N "takeichi@acm.org" []],
     N "Tel" [N "+81-3-5841-7430" []]]]
```

We then change values at the nodes to the second person's name and contacts:

```
N "Addrbook"
  [N "Person"
    [N "Name" [N "Masato Takeichi" []],
     N "Email" [N "takeichi@acm.org" []],
     N "Tel" [N "+81-3-5841-7430" []]],
   N "Person"
    [N "Name" [N "Zhenjiang Hu" []],
     N "Email" [N "hu@mist.i.u-tokyo.ac.jp" []],
     N "Tel" [N "+81-3-5841-7430" []]]]
```

It should be noted that we are editing both the source document and the view, though we are not quite aware of this fact so far. The transformation X, is currently simply the identity function. Now suppose we want to sort persons according to their names, by selecting all the persons and apply a sort transformation on it. The result looks like

```
N "Addrbook"
  [N "Person"
    [N "Name" [N "Zhenjiang Hu" []],
     N "Email" [N "hu@mist.i.u-tokyo.ac.jp" []],
     N "Tel" [N "+81-3-5841-7430" []]],
   N "Person"
    [N "Name" [N "Masato Takeichi" []],
     N "Email" [N "takeichi@acm.org" []],
     N "Tel" [N "+81-3-5841-7430" []]]]
```

What is sorted is the view. The source remains the same, while the transform X now equals the function performs the sorting. In Section 4 we will formally describe how each components of the triple are changed by each operation.

Now we want to make an index of names of people in the address book. To do so, we first make a copy of the address book by a duplicate operation:

```
  N "Dup"
  [N "Addrbook"
     [N "Person"
        [N "Name" [N "Zhenjiang Hu" []],
         ... ],
      N "Person"
        [N "Name" [N "Masato Takeichi" []],
         ...]],
   N "Addrbook"
     [N "Person"
        [N "Name" [N "Zhenjiang Hu" []],
         ... ],
      N "Person"
        [N "Name" [N "Masato Takeichi" []],
         ...]]]
```

and then keep *only* the names by repeatedly deleting the rest of the information, from the duplicated address book (and change the tag `"Addrbook"` to `"Index"`):

```
  N "Dup"
  [N "Index"
     [N "Name" [N "Zhenjiang Hu" []],
      N "Name" [N "Masato Takeichi" []]]
   N "Addrbook"
     [N "Person"
        [N "Name" [N "Zhenjiang Hu" []],
         ... ],
      N "Person"
        [N "Name" [N "Masato Takeichi" []],
         ...]]]
```

It should be remarked that the duplication is one of the most important features of our system. It is different from the copy operation, which we performed just now to add a new person in the address book. Copied data are independent from each other. On the other hand, the duplicate operation indicates that the subtree and its duplicate should be synchronized. In this example, deletion, insertion, or modification of a person's information at one side causes corresponding change on the other side, unless we explicitly inform the editor to perform the editing operations independently.

The keep operation in the above, for example, is such an independent transformation. When it was applied to the subtree at [1] to extract the names, the main address book at [2] remains unchanged. On the other hand, if we insert the following entry

```
  N "Person"
     [N "Name" [N "Shin-Cheng Mu" []],
      N "Email" [N "scm@mist.i.u-tokyo.ac.jp" []],
      N "Tel" [N "+81-3-5841-7411" []]]
```

to the `"Addrbook"` subtree at [2] as its last child, the name "Shin-Cheng Mu" will automatically appear in the index of names, resulting in:

```
  N "Dup"
  [N "Index"
     [N "Name" [N "Zhenjiang Hu" []],
      N "Name" [N "Shin-Cheng Mu" []]],
      N "Name" [N "Masato Takeichi" []]
   N "Addrbook"
     [N "Person"
        [N "Name" [N "Zhenjiang Hu" []],
```

```
         ... ],
      N "Person"
        [N "Name" [N "Shin-Cheng Mu" []],
         ...]],
   N "Person"
     [N "Name" [N "Masato Takeichi" []],
      ...]]]
```

Note also that although the entry is inserted (by the user) as the last child of the ``Addrbook in the view, the resulting view has both the entries under the ``Addrbook'' and the names under the ``Index'' sorted.

Finally, we tell the system that the type of the view should be the following

```
data Addrbook = Addrbook (Index, [Person])
data Index = Index [Name]
data Person = Person (Name, [Email], Tel)
data Name = Name String
data Email = Email String
data Tel = Tel String
```

and our system automatically returns the triple (T, D, X) as in Figures 6, 7 and 8.

We summarize the important features of our programmable editor as follows.

- Our editor is presentation-oriented (view-oriented), with which the developer can directly edit the view, the exact display of the document. This WYSIWYG style is more friendly than existing editors. Those with little knowledge about XML will feel easy to use this system to develop their structured documents.

- Our editor allows simple description of data dependency in the view by the duplicate operation, and provides an efficient solution to keep consistency of the data in the view. As far as we are aware, this is the first structured document editor with local data synchronization.

- Our editor integrates the three components of a structured document in the view displayed to the user. The source data and the transformation are gradually built while the user edits the view, before the user finally imposes a type on the view.

3 A Bidirectional Transformation Language

Our editor is view-oriented, allowing users to develop their structured documents by directly editing the view, while producing the three components of a structured document automatically. This view-oriented environment requires a mechanism to relate the three components with the view. We borrow the view-updating technique [BS81, DB82, GPZ88, OT94, Abi99], which has been intensively studied in the database community. Given a database and a query which produces a view from the database, the view-updating technique is to reflect view modification upon the database. Though the idea is very similar, there are two major difficulties in using this technique in our view-oriented editor.

- Our view may contain local data dependency as seen in Section 2 where the same name appears twice in the view), which hardly happens in a view of database.

- Our view modification should be reflected not only on the source data, but also on the transformation. In other word, the transformation (query) part, which is assumed to be fixed in the existing view-updating technique, should be modifiable in our framework.

In this section, we extend the existing bidirectional transformation languages [Mee98, GMPS03] to a more powerful language for specification of the relationship between source data and view. This transformation language plays a very important role in the design of our editor (see Section 4).

3.1 Bidirectionality

Before explaining our language, we clarify what we mean by being *bidirectional*. Following the convention in [GMPS03], we call the type of source documents C (concrete view) and that of target documents A (abstract view). They are both embedded in Tree but we nevertheless distinguish them for clarity. A transformation x defined in X is associated with two functions. The function $\phi_x :: C \to A$ maps the concrete view to an abstract view, which is displayed and edited by the user. The function $\lhd_x :: C \times A \to C$ takes the edited abstract view and the original concrete view, and returns an updated concrete view. In [GMPS03] they are called *get* and *put* respectively.

We call a transformation x *bidirectional* if the following two properties hold:

PUT-GET : $\phi_x\ (c \lhd_x a) = a$ where $a = \phi_x\ c$
GET-PUT : $c' \lhd_x (\phi_x\ c') = c'$ where $c' = c \lhd_x a$

The GET-PUT property says that if c' is a recently updated concrete view, mapping it to its abstract view and immediately performing the backward update does not change its value. Note that this property only needs to hold for those c' in the range of $\lhd_x$. For an arbitrary c we impose the PUT-GET requirement instead. Let a be the abstract view of c. Updating c with a and taking the abstract view, we get a again.

Unlike in [Mee98, GMPS03], we do not require the PUT-GET property to hold for arbitrary a, nor the GET-PUT property for arbitrary c'. In [Mee98, GMPS03], the two properties are stronger because it is not allowed to duplicate values in the concrete view. Once we introduce duplication into our language, however, an editing action at one location of the abstract view may cause corresponding changes at other locations. Therefore we need an extra ϕ_x to perform the change in the abstract view. The two bidirectional properties above guarantees that no further updating is necessary.

A final remark: a similar pair of properties, that two functions form inverses of each other in restricted ranges, is true of Galois-connected functions. However, the fact that $\lhd_x$ takes two arguments makes it difficult to formulate the bidirectionality properties in terms of a Galois connection.

3.2 The Language X

The syntax of the language X for specifying bidirectional transformation is given in Figure 9. Primitive transformations are denoted by non-terminal B. They can be composed to from more complicated transformations by one of the combinators defined in X. The language looks

$$
\begin{array}{lll}
X & ::= & B \qquad\qquad\quad \{ \text{ primitives } \}\\
  & | & X\ ;\ X \qquad\quad \{ \text{ sequencing } \}\\
  & | & X \otimes X \qquad\quad \{ \text{ product } \}\\
  & | & \text{If } P\ X\ X \qquad \{ \text{ conditional branches } \}\\
  & | & \text{Map } X \qquad\quad \{ \text{ apply to all children } \}\\
  & | & \text{Fold } X\ X \qquad \{ \text{ fold } \}\\
  & & \\
B & ::= & \text{GFun } (f, g) \quad \{ \text{ Galoi function pairs } \}\\
  & | & \text{NFun } f \qquad\quad \{ \text{ a simple function } \}\\
  & | & \text{Dup} \qquad\qquad \{ \text{ duplication } \}
\end{array}
$$

Figure 9: The Language X for Specifying Bidirectional Transformations

very similar to the bidirectional languages proposed in [Mee98, GMPS03]. The most important difference lies in the new language construct Dup, which enables description of data dependency inside the view.

In this section, we will focus on how to use the language to describe transformation of our interest. An important property of the language X is the following theorem, whose proof is omitted due to space limitation.

Theorem 1 (Bidirectionality of X)
Any transformation described in X is bidirectional. $\square$

3.2.1 Primitive Transformations

Rather than giving a fixed set of primitive transformations as in [GMPS03], we adopt a general way to define two classes of primitive transformations — the *bidirectional* primitives (GFun) and the *unidirectional* ones (NFun). Together with the special primitive Dup, they are described below.

Duplication

In the forward direction, the function ϕ_{Dup} generates two copies of its input.

$$\phi_{\text{Dup}}\ c = \texttt{N "Dup"} \ [c, c]$$

In the backward direction, $\lhd_{\text{Dup}}$ checks which of the two copies was touched by the user by comparing them with the original view c, and keeps only the changed one.

$$
\begin{array}{rcll}
c \lhd_{\text{Dup}} (\texttt{N "Dup"}\ [a_1, a_2]) & = & a_2 & \text{if } a_1 = c\\
 & = & a_1 & \text{if } a_2 = c\\
 & = & a_1 & \text{otherwise}
\end{array}
$$

Here we assume that the user performs only one editing action before an updating event is triggered. Therefore, if none of a_1 and a_2 equals c, it must be the case that $a_1 = a_2$, because they result from the same editing action.

The Dup operator is the only means in X to specify value dependency among different parts of the view — when one of the copies is edited by the user, the other should change as well. This is achieved by a backward update $\lhd_{\text{Dup}}$ followed by a forward transform ϕ_{Dup}. The backward phase updates the touched value. The forward phase then overwrites the copies in the abstract view with new values.

The presence of Dup makes ϕ_x, where x uses Dup, a non-total transformation. This is in contrast with [Mee98, GMPS03], where all transformations are bi-total. The definition of $\lhd_{\mathsf{Dup}}$ is mostly about designing a reasonable way to "fill in the missing slots".

Bidirectional Primitive Transformations

A bidirectional primitive GFun (f, g) consists of two functions f and g satisfying:

$$\begin{array}{ll} \text{INV1}: & f \circ g \circ f = f \\ \text{INV2}: & g \circ f \circ g = g \end{array}$$

That is, g is the inverse of f *in the range of f*. The property is satisfied by all Galois-connected pairs of functions, thus the name GFun. The bidirectional semantics of GFun (f, g) is given by

$$\begin{array}{rcl} \phi_{\mathsf{GFun}\ (f,g)} a & = & f\ c \\ c \lhd_{\mathsf{GFun}\ (f,g)} a & = & g\ a \end{array}$$

In words, the abstract view is obtained by applying f to the concrete view, while the concrete view can be obtained by applying g to the abstract view, ignoring the original concrete view. That $\phi_{\mathsf{GFun}\ (f,g)}$ and $\lhd_{\mathsf{GFun}\ (f,g)}$ satisfy the bidirectional property is a direct consequence of INV1 and INV2.

Let us see some useful primitive transformations defined in this way. The simplest transformation is the identity transformation:

$$\mathsf{idX} \;=\; \mathsf{GFun}\ (id, id)$$

which relates two identical data, and is defined by a pair of two identity functions. In this example, the pair of functions are inverse of each other.

Another interesting transformation is defined by

$$\mathsf{sortX} \;=\; \mathsf{GFun}\ (sortT, sortT)$$

which relates the concrete data with the abstract data such that the children of the root in the abstract view are sorted. The function $sortT$ sorts the subtrees of the root, according to the first child value of each subtree. It is clear that $sortT$ is not invertible, but $sortT$ and $sortT$ do satisfy the INV property.

Similarly, we may define other primitive transformations that are useful for manipulating tree locally.

- swapX i j swaps the ith and jth subtrees of the root.

- hoistX n: If the root has label n and a single child v, then the result is v.

- newRoot n makes the current tree the single child of a new root with label n.

- exchangeX exchanges the root with the node of the leftmost child tree that has no child.

- insertHoleX inserts Ω, a special tree denoting a hole, as the leftmost child of the root.

- deleteHoleX deletes the hole appearing as the leftmost child of the root.

- replaceHoleX v replaces the hole with tree v.

Restrictive Primitive Transformations

Not all primitive transformations we wish to have satisfy the INV property. One example is the constX transformation that does not care about the concrete view but only requires the abstract view to be a constant tree. Another example is the numberX transformation that relates the concrete view with the abstract view such that the abstract view shows the number of the children of the root in the concrete view.

We specify these transformations using a single function

$$\mathsf{NFun}\ f$$

only showing how to map the concrete view to the abstract view. The bidirectional semantics of this kind of transformation can be defined as follows.

$$\begin{array}{rcl} \phi_{\mathsf{NFun}\ f} a & = & f\ c \\ c \lhd_{\mathsf{NFun}\ f} a & = & c \end{array}$$

Notice that $c \lhd_{\mathsf{NFun}\ f}$ always returns the original concrete view c, and ignores any change on the abstract view a.

Below are the definitions of the two transformations mentioned above.

$$\begin{array}{rcl} \mathsf{constX}\ v & = & \mathsf{NFun}\ (\lambda x.\, v) \\ \mathsf{numberX} & = & \mathsf{NFun}\ (length \circ children) \end{array}$$

In a sense, transformations defined by NFun f are not really "bidirectional", since all changes on the abstract view are simply ignored. However, it is still very helpful when used together with Dup.

3.3 Transformation Combinators

The set of transformation combinators is useful to construct bigger transformations. An informal explanation of these combinators is given in Figure 10. Most of the combinators are essentially the same as those in [GMPS03]. There are three new combinators, namely duplication, condition, and fold. The duplication combinator is to introduce data dependency inside a document. Different from the reference structure for sharing data, the duplication transformation treats in the same way the duplicated part and the original part. The condition combinator is important to apply different transformations according to the context or information of the local tree, and the fold combinator is useful to specify iterative processing on documents.

3.3.1 Sequencing

Given two bidirectional transformations x_1 and x_2, the transformation $x_1; x_2$ informally means "do x_1, then do x_2". Its bidirectional semantics is given by

$$\begin{array}{rcl} \phi_{x_1;x_2} & = & \phi_{x_2} \circ \phi_{x_1} \\ c \lhd_{x_1;x_2} a & = & c \lhd_{x_1} ((\phi_{x_1}\ c) \lhd_{x_2} a) \end{array}$$

The forward transform $\phi_{x_1;x_2}$ is simply the sequential composition of ϕ_{x_1} and ϕ_{x_2}. To update the concrete view c with a modified abstract view a, we need to know what the intermediate concrete view was. It is computed by $\phi_{x_1}\ c$. The expression $(\phi_{x_1}\ c) \lhd_{x_2} a$ then computes an intermediate abstract view, which is used to update c with $\lhd_{x_1}$.
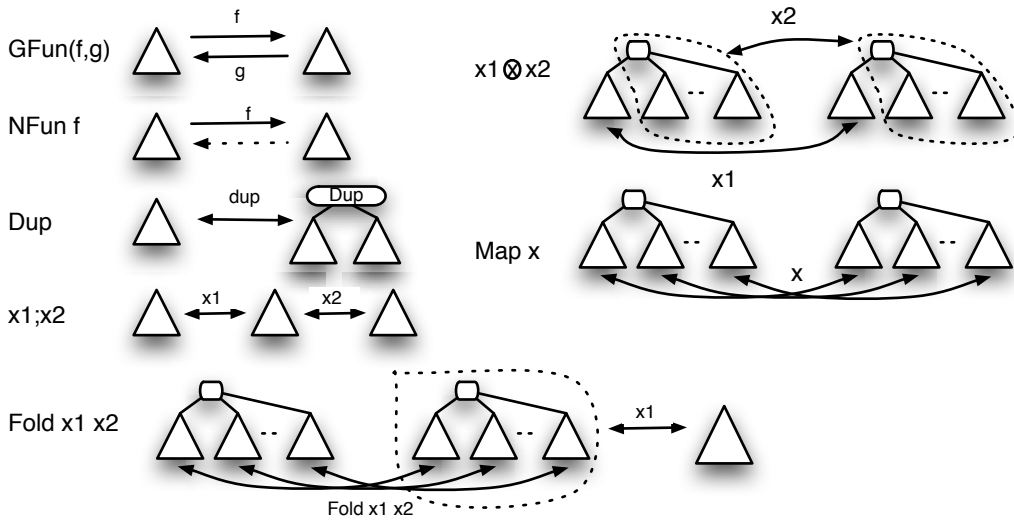
Figure 10: Intuitive Explanation of Transformation Combinators

3.3.2 Product

The product construct $x_1 \otimes x_2$ behaves similar to products in ordinary functional languages, apart from that we are working on trees rather than pairs. The forward transformation is defined by

$$\phi_{x_1 \otimes x_2} \ (\text{N } c \ (c_1 : cs)) = \text{N } a \ (a_1 : as)$$

$$\text{where} \quad \begin{aligned} a_1 &= \phi_{x_1} \ c_1 \\ \text{N } a \ as &= \phi_{x_2} \ (\text{N } c \ cs) \end{aligned}$$

The input tree is sliced into two parts: the left-most child, and the root plus the other children. The transform x_1 is applied to the left-most child, while x_2 is applied to the rest. The result is then combined together. The backward updating is defined by updating the two slices separately.

$$(\text{N } c \ (c_1 : cs)) \lhd_{x_1 \otimes x_2} (\text{N } a \ (a_1, as)) = \text{N } c' \ (c_1' : cs')$$

$$\text{where} \quad \begin{aligned} c_1' &= c_1 \lhd_{x_1} a_1 \\ \text{N } c' \ cs' &= (\text{N } c \ cs) \lhd_{x_2} (\text{N } a \ as) \end{aligned}$$

3.3.3 Conditional Branches

In the forward direction, the combinator $\text{If } p \ x_1 \ x_2$ applies the transform x_1 to the input if the input satisfies the predicate p. Otherwise x_2 is applied.

$$\begin{aligned} \phi_{\text{If } p \ x_1 \ x_2} \ c &= \phi_{x_1} \ c \quad , \text{if } p \ c \\ &= \phi_{x_2} \ c \quad , \text{otherwise} \end{aligned}$$

In the backward direction, we check the root label to determine whether to apply $\lhd_{x_1}$ or $\lhd_{x_2}$ to the modified view.

$$\begin{aligned} c \lhd_{\text{If } p \ x_1 \ x_2} a &= c \lhd_{x_1} a \quad , \text{if } p \ c \\ &= c \lhd_{x_2} a \quad , \text{otherwise} \end{aligned}$$

3.3.4 Map

Given the well-known function map on lists, defined by

$$\begin{aligned} map \ f \ [\,] &= [\,] \\ map \ f \ (a : x) &= f \ a : map \ f \ x \end{aligned}$$

The forward transform of $\text{Map } x$ simply applies the transformation x to all subtrees of the given tree, leaving the root label unchanged.

$$\phi_{\text{Map } x} \ (\text{N } c \ cs) = \text{N } c \ (map \ \phi_x \ cs)$$

The backward updating is defined by updating the subtrees separately,

$$(\text{N } c \ cs) \lhd_{\text{Map } x} (\text{N } c \ as) \quad = \quad \text{N } c \ (zip_{\lhd_x} \ cs \ as)$$

where the function zip is defined as follows.

$$\begin{aligned} zip_{\oplus} \ [\,] \ [\,] &= [\,] \\ zip_{\oplus} \ (a : x) \ (b : y) &= a \oplus b : zip_{\oplus} \ x \ y \end{aligned}$$

3.3.5 Fold

The transform $\text{Fold } x_1 \ x_2$ is defined like a fold on rose trees. The transform x_2 is applied to leaves, x_1 to internal nodes. Its forward transform is defined by

$$\begin{aligned} \phi_{\text{Fold } x_1 \ x_2} \ (\text{N } c \ [\,]) &= \phi_{x_2} \ (\text{N } c \ [\,]) \\ \phi_{\text{Fold } x_1 \ x_2} \ (\text{N } c \ cs) &= \phi_{(\text{Map } (\text{Fold } x_1 \ x_2));x_1} (\text{N } c \ cs) \end{aligned}$$

In the base case, we simply apply x_2 to the leaf. In the recursive case, $\text{Fold } x_1 \ x_2$ is applied to all subtrees of the input tree, before x_1 is applied to the result, thus the use of sequencing.

In the backward direction, we use the cached copy of the concrete view to determine the depth of recursion to go

8

into. Being able to reuse Map and sequencing significantly simplifies the definition.

$$(\mathbb{N}\ c\ [\ ]) \lhd_{\mathsf{Fold}\ x_1\ x_2}\ a\ =\ (\mathbb{N}\ c\ [\ ]) \lhd_{x_2}\ a$$
$$c \lhd_{\mathsf{Fold}\ x_1\ x_2}\ a\ =\ c \lhd_{(\mathsf{Map}\ (\mathsf{Fold}\ x_1\ x_2));x_1}\ a$$

If we expand the second clause of the definition, we get

$$(\mathbb{N}\ c\ cs) \lhd_{\mathsf{Fold}\ x_1\ x_2}\ a\ =\ \mathbb{N}\ c'\ cs'$$

$$\text{where}\quad \begin{aligned} (\mathbb{N}\ c'\ as') &= (\mathbb{N}\ c\ as) \lhd_{x_1}\ a \\ as &= map\ \phi_{\mathsf{Fold}\ x_1\ x_2}\ cs \\ cs' &= zip_{\lhd_{\mathsf{Fold}\ x_1\ x_2}}\ cs\ as' \end{aligned}$$

Like in sequencing, we need an application of $map\ \phi_{\mathsf{Fold}\ x_1\ x_2}$ to create an intermediate value in order to perform $\lhd_{x_1}$. The subtrees are then updated using $zip_{\lhd_{\mathsf{Fold}\ x_1\ x_2}}$.

3.4 Editing as Bidirectional Transformation

With the language X, we are able to define the following important editing operators as bidirectional transformation.

$$\begin{aligned} \mathsf{insertX}\ v\ &=\ \mathsf{insertHoleX}\ ; \\ &\quad (\mathsf{replaceHoleX}\ v) \otimes \mathsf{idX} \\ \mathsf{deleteX}\ &=\ (\mathsf{constX}\ \Omega) \otimes \mathsf{idX}\ ; \\ &\quad \mathsf{deleteHoleX} \\ \mathsf{modifyRootX}\ n\ &=\ \mathsf{insertX}\ (\mathbb{N}\ n\ [])\ ; \\ &\quad \mathsf{exchangeX}\ ; \\ &\quad \mathsf{deleteX} \end{aligned}$$

We may insert some document v as the leftmost child of the root using $\mathsf{insertX}\ v$, or delete the leftmost child using $\mathsf{deleteX}$, or modify the root node information with a new name n using $\mathsf{modifyRootX}\ n$.

Other editing operators like moveX and copyX can be defined via a combination of $\mathsf{insertX}$ and $\mathsf{deleteX}$. Another useful editing operator, keepX, which is to return the leftmost child tree, could be naively realized by a sequence of deletion operations. But we can define it more efficiently by

$$\begin{aligned} \mathsf{keepX}\ &=\ \mathsf{idX} \otimes (\mathsf{constX}\ \Omega)\ ; \\ &\quad \mathsf{hoistX}\ (\mathsf{Root}_\Omega) \end{aligned}$$

where Root_Ω denotes the root node of the Ω tree. In fact, there are many ways of defining an editing transformation; one may go extremely to define them just as basic transformations in terms of NFun with a non-invertible function. Compared with another definition of $\mathsf{insertX'}$ by

$$\begin{aligned} \mathsf{insertX'}\ v\ &=\ \mathsf{NFun}\ f \\ &\quad \text{where}\ f\ (\mathbb{N}\ n\ ts)\ =\ \mathbb{N}\ n\ (v:ts) \end{aligned}$$

which forbids any modification on the view, our definition of $\mathsf{insertX}$ imposes no restriction at all on editing of the view.

So far, a transformation is applied to the whole tree. More often, we want to apply a transformation x to the subtree at path p but leave other parts of the tree unchanged.

$$\begin{aligned} \mathsf{applyX}\ [\ ]\ x\ &=\ x \\ \mathsf{applyX}\ (i:p)\ x\ &=\ \mathsf{swapX}\ 1\ i\ ; \\ &\quad \mathsf{applyX}\ p\ x \otimes \mathsf{idX}\ ; \\ &\quad \mathsf{swapX}\ 1\ i \end{aligned}$$

Note that the applyX behaves as a higher order transformation; it accepts a transformation and return a new transformation as the result.

4 The Programmable Editor

Our editor serves as an view-oriented environment supporting interactive development of structured documents. It allows users to develop structured documents in a WYSIWYG (what you see is what you get) manner, and automatically produces the three components of a structured document.

4.1 Editing Operators

We consider the following editing operators.

$$\begin{aligned} E\ ::=\ &\mathsf{InsertE}\ p\ v \\ |\ &\mathsf{DeleteE}\ p \\ |\ &\mathsf{CopyE}\ p_1\ p_2 \\ |\ &\mathsf{MoveE}\ p_1\ p_2 \\ |\ &\mathsf{FieldEditE}\ p\ l \\ |\ &\mathsf{DuplicateE}\ p \\ |\ &\mathsf{TransformE}\ p\ x \end{aligned}$$

They are standard except for the last two operators. For instance, $\mathsf{InsertE}\ p\ v$ inserts a tree v as the first child of the node at path p, and $\mathsf{FieldEditE}\ p\ l$ modifies the label of the node at path p to l. The last two new editing operators, $\mathsf{DuplicateE}\ p$ and $\mathsf{TransformE}\ p\ x$, are the special features in our editor: $\mathsf{DuplicateE}\ p$ duplicates the tree at path p and the two trees should be kept identical, and $\mathsf{TransformE}\ p\ x$ applies a bidirectional transformation x to the tree at path p.

The state of the editor is a triple

$$\mathcal{S} = (c, x, a)$$

where c and a denote the internal data and the view respectively, and x denotes a bidirectional transformation. Each state $\mathcal{S} = (c, x, a)$ holds the following SYNC property.

$$\begin{aligned} a\ &=\ \phi_x\ c \\ c\ &=\ c \lhd_x\ a \end{aligned}$$

This SYNC property expresses the relationship among the three elements in a state, and the bidirectionality of x ensures an automatic adjustment among the three elements in case one of them is modified. Let (c, x, a) be a given state.

- If c changes to c', the new state is $(c', x, \phi_x\ c)$;

- If x changes to x', the new state is $(c, x', \phi'_x\ c)$.

- If a changes to a', the new state is $(c \lhd_x a', x, \phi_x\ (c \lhd_x a'))$;

We define the following functions for the above adjustments.

$$\begin{aligned} \mathcal{A}_{cx}\ (c, x, a)\ &=\ (c, x, \phi_x\ c) \\ \mathcal{A}_a\ (c, x, a)\ &=\ \mathcal{A}_{cx}\ (c \lhd a, x, a) \end{aligned}$$

The operational semantics of the editing operators is given in Figure 11. Each editing operator is a state transformer. Given the state (c, x, a), the operator $\mathsf{InsertE}\ p\ v$ is to insert a tree v to the view a by a general tree insertion function insert, and to change the path expressions in the transformation x so that the nodes at these paths refer to the same ones. Note that $\mathsf{insP}\ x\ p$ is a function to "increase" some node number in some paths in x. Let $p = p_1 +\!\!+ [a]$, and p' be a path expression in x satisfying $p' = p_1 +\!\!+ [b] +\!\!+ p_2$ and

$$
\begin{aligned}
[\![\mathsf{InsertE}\ p\ v]\!]\ (c,x,a) &= \mathcal{A}_a\ (c,\mathsf{incP}\ x\ p,\mathsf{insert}\ p\ v\ a)\\
[\![\mathsf{DeleteE}\ p]\!]\ (c,x,a) &= \mathcal{A}_a\ (c,\mathsf{decP}\ x\ p,\mathsf{delete}\ p\ a)\\
[\![\mathsf{CopyE}\ p_1\ p_2]\!]\ (c,x,a) &= \mathcal{A}_a\ (c,\mathsf{incP}\ x\ p_2,\mathsf{copy}\ p_1\ p_2\ a)\\
[\![\mathsf{MoveE}\ p_1\ p_2]\!]\ (c,x,a) &= \mathcal{A}_a\ (c,\mathsf{incP}\ (\mathsf{decP}\ x\ p_1)\ p_2,\mathsf{move}\ p_1\ p_2\ a)\\
[\![\mathsf{FieldEditE}\ p\ l]\!]\ (c,x,a) &= \mathcal{A}_a\ (c,x,\mathsf{fieldEdit}\ p\ l\ a)\\
[\![\mathsf{DuplicateE}\ p]\!]\ (c,x,a) &= \mathcal{A}_{cx}\ (c,(x;\mathsf{applyX}\ p\ \mathsf{Dup}),a)\\
[\![\mathsf{TransformE}\ p\ x]\!]\ (c,x,a) &= \mathcal{A}_{cx}\ (c,(x;\mathsf{applyX}\ p\ x)),a)
\end{aligned}
$$

Figure 11: The Operational Semantics of the Editing Operators

$b > a$, then p' will be changed to $p_1 +\!+ [b+1] +\!+ p_2$. Other editing operators like deleteE, copyE, moveE, and fieldEditE are defined similarly. The duplicateE and transformE are two editing operators that change the transformation x. Thanks to the SYNC property of the editor state, their semantics is very clear.

Note the difference between the two editing operations in our editor:

$$\mathsf{InsertE}\ p\ v$$
$$\mathsf{TransformE}\ p\ (\mathsf{insertX}\ v)$$

The former inserts a tree to the view and propagates this change to other places of the view, while the later performs an independent insertion on the view, causing no changes elsewhere. Note also that not any editing sequence is valid in our system. For example, the view produced by a restrictive primitive transformation is not editable by InsertE. However, it can be modified by an independent editing operation.

4.2 Producing Final Structured Documents

This section explains how to produce the three components for a structured document. Recall that in Section 2 the three components of a structured document is a document type, a document source which does not have redundant information, and a transformation that produces the final visible structured document.

The first two elements of the editor state (c,x,a) almost give the source document and the transformation we want to have. What is remained to do is to find a suitable document type to structure c and to make x a transformation accepting typed document sources. The difficult lies in finding the document type. One way is to use the automatic extraction techniques [Chi02, GGR^{+}00] to extract the document type information from c, however this approach is effective only when there is large amount of sample documents, which is not really suitable in our situation.

We adopt another approach. We ask the users to provide a type for the view (see our example in Section 2.3), and we infer types for the document source and the transformation. To do so, we borrow the idea from [PV00], where given a DTD for the XML source data and a query, an inference system derives a tight DTD for the view. Since our transformations are built up upon primitive transformations in terms of GFun (f,g) and NFun f, we can utilize the inference algorithms in [PV00], if the types for functions used in the primitive transformations are given. We hope to design a language to define the functions used in primitive transformations and derive their types automatically in the future.

4.3 Infinite Undo/Redo

A nice side effect of bidirectional transformations in our editor is the ability to implement infinite numbers of operations of undo/redo. The following set of equations indicate that for any editing operations, there always exists another editing operation to recover the state.

$$
\begin{aligned}
&[\![\mathsf{DeleteE}\ p]\!]\ ([\![\mathsf{Insert}\ p\ v]\!]\ s) = s\\
&[\![\mathsf{InsertE}\ p\ (s|p)]\!]\ ([\![\mathsf{InsertE}\ p\ v]\!]\ s) = s\\
&[\![\mathsf{DeleteE}\ p_2]\!]\ ([\![\mathsf{CopyE}\ p_1\ p_2]\!]\ s = s\\
&[\![\mathsf{InsertE}\ p_1\ (s|p_1)]\!]\ ([\![\mathsf{DeleteE}\ p_2]\!]\ ([\![\mathsf{MoveE}\ p_1\ p_2]\!]\ s)) = s\\
&[\![\mathsf{FieldEditE}\ p\ (\mathsf{root}(s|p))]\!]\ ([\![\mathsf{FieldEditE}\ p\ n]\!]\ s) = s\\
&[\![\mathsf{undoX}]\!]\ ([\![\mathsf{DuplicateE}\ p]\!]\ s) = s\\
&[\![\mathsf{undoX}]\!]\ ([\![\mathsf{TransformE}\ p]\!]\ s) = s
\end{aligned}
$$

Here $s|p$ denotes the subtree in the view s at the path p, root v returns the label of the root node of the tree v. undoX is a new editing operation for undoing the last transformation. Its semantics can be defined by

$$[\![\mathsf{undoX}]\!]\ (c,x,a) = \mathcal{A}_{cx}(c,\mathsf{deleLast}\ x,a)$$

where deleLast $(x_1;x_2) = x_1$.

5 Related Work

There are plenty of XML editors [Sof04], which have been designed and implemented for supporting development of structured documents in XML. Most of them, such as XMLSpy [Kim02], develop structured documents in the order of DTD, document content, and presentation. These kinds of tools cannot effectively support interactive document development, as strongly argued by researchers [FQA88, VRL00] in the field of document engineering. Moreover, these tools require developers to have much knowledge about DTD, XML and XSLT. In contrary, our editor provides a single integrated WYSIWYG interface, and requires less knowledge about XML.

The most related system to ours is Poxima [SJ03, Jeu04], a single presentation-oriented generic editor designed for all kinds of XML-documents and presentations. It is very similar to our system; it is also presentation orient and allows description of transformation and computation over view through editing operations. However, for each transformation and computation, users must prepare two functions to explicitly express the two-way transformation. In contrast, we provide a bidirectional language with the view-updating technique, facilitating bidirectional transformation. Another similar system is the TreeCalc system [THK^{+}03], a simple tree version of the spreadsheet system, but it does not support structure modification on the view.

Our representation of the editor state by a triple (the document source, and transformation, and the view) is inspired by the work on view-updating [BS81, DB82, GPZ88,

OT94, Abi99] in the database community, where modification on the view can be reflected back to the original database. We borrow this technique with a significant extension that editing operations can modify not only the view but also the query, which is not exploited before. Since our transformation language does not have the JOIN operator, the problem of the costive propagation of deletion and annotation through views [Pet02] does not happen in our case.

The design of the bidirectional transformation language X learns much from the lens combinators in [GMPS03], where a semantic foundation and a core programming language for bi-directional transformations on tree-structured data are given. The lens combinators cannot describe dependency inside the view. This is not the problem in the context of data synchronization, but has to be remedied in our view-oriented editor. Our language with duplication makes dependency clearly described. Another very much related language is that given by Meertens [Mee98], which is designed for specification of constraints in the design of user-interfaces. Again the language cannot deal with dependency in the view.

Our idea of duplication in X is influenced by the invertible language in [GK03], where duplication is considered as the inverse of equality check and vice versa. In inverse computation, an inverse function computes an input merely from an output, but in bidirectional transformation, a backward updating can use both the output and the old input to compute a new input. Therefore adding duplication to a bidirectional language needs a more involved equality check mechanism. It should be interesting to see if inverse transformation with duplication can implement the view updating, and to compare these two approaches. An attempt has been made in [MHT04].

6 Conclusions

In this paper, we have proposed a presentation oriented editor suitable for interactive development of structured documents. A novel use of the view updating technique in the editor, the duplication construct in our bidirectional language, and the mechanism of changing transformation though editing operations, play a key role in the design of our editor system. The prototyped system implemented based on the system shows the promise of this approach.

This work is still in an early stage, and there is much work to do. Particularly, rather than designing a new bidirectional language, we are interested to see if it is possible to make the existing transformation languages like XSLT to be efficiently bidirectional. By efficiency, we means there is as many editable parts as possible in the view produced by XSLT.

Acknowledgments

References

[Abi99] Serge Abiteboul. On views and XML. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of Database Systems*, pages 1–9. ACM Press, 1999.

[BCF03] V. Benzaken, G. Castagna, and A. Frisch. Cduce: An xml-centric general-purpose language. In *Proceedings of 2003 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2003.

[BPSM98] Tim Bray, Jean Paoli, and C.M. Sperberg-McQueen. Extensible markup language (xml) 1.0. 1998.

[BS81] P. A. Bernstein and N. Spyratos. Updating semantics of relational views. *ACM TODS*, 6(4):557–575, 1981.

[Chi02] Boris Chidlovskii. Schema extraction from xml collections. In *Proceedings of the second ACM/IEEE-CS joint conference on Digital libraries*, pages 291–292. ACM Press, 2002.

[DB82] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM TODS*, 7(3):381–416, 1982.

[FQA88] R. Furuta, V. Quint, and J. André. Interactively editing structured documents. *Electronic Publishing Origination, Dissemination, and Design*, 1(1):19–44, 1988.

[GGR+00] Minos Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim. Xtract: a system for extracting document type descriptors from xml documents. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 165–176. ACM Press, 2000.

[GK03] Robert Glück and Masahiko Kawabe. A program inverter for a functional language with equality and constructors. In Atsushi Ohori, editor, *Programming Languages and Systems. Proceedings*, volume 2895 of *Lecture Notes in Computer Science*, pages 246–264. Springer-Verlag, 2003.

[GMPS03] Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierc, and Alan Schmitt. A language for bi-directional tree transformations. Technical Report Technical Report MS-CIS-03-08, Department of Computer and Information Science University of Pennsylvania, August 2003.

[GPZ88] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM TODS*, 13(4):485–524, 1988.

[HVP00] Haruo Hosoya, Jerome Vouilon, and Banjamin Pierce. Regular expression types for xml. In *Proceedings of 2000 ACM SIGPLAN International Conference on Functional Programming*, pages 11–22. ACM Press, 2000.

11

[Jeu04]   Johan Jeuring. Implementing a generic editor. In *2nd Workshop on Programmable Structured Documents*, February 2004.

[Kim02]   Larry Kim. *The Official XMLSPY Handbook*. John Wiley & Sons, 2002.

[Mee98]   Lambert Meertens. Designing constraint maintainers for user interaction. `http://www.cwi.nl/~lambert`, June 1998.

[MHT04]   S.C. Mu, Z. Hu, and M. Takeichi. An injective language for reversible computation. In *Seventh International Conference on Mathematics of Program Construction (MPC 2004)*, Stirling, Scotland, July 2004. Springer Verlag, LNCS.

[OT94]   Atsushi Ohori and Keishi Tajima. A polymorphic calculus for views and object sharing. In *ACM PODS'94*, pages 255–266, 1994.

[Pet02]   Peter Buneman and Sanjeev Khanna and Wang-Chiew Tan. On Propagation of Deletion and Annotation Through Views. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, Wisconsin, Madison, June 2002.

[PV00]   Yannis Papakonstantinou and Victor Vianu. Dtd inference for views of xml data. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 35–46. ACM Press, 2000.

[SJ03]   Martijn M. Schrage and Johan Jeuring. Xprez: A declarative presentation language for xml. avaliable at `http://www.cs.uu.nl/research/projects/proxima/`, 2003.

[Sof04]   XML Software. A list of xml editors. See http://www.xmlsoftware.com/editors.html, 2004.

[THK+03]   Masato Takeichi, Zhenjiang Hu, Kazuhiko Kakehi, Yasishi Yayashi, Shin-Cheng Mu, and Keisuke Nakano. Treecalc : Towards programmable structured documents. In *JSSST Conference on Software Science and Technology*, September 2003.

[VRL00]   L. Villard, C. Roisin, and N. Layada. A xml-based multimedia document processing model for content adaptation. In *8th International Conference on Digital Documents and Electronic Publishing, LNCS*, September 2000.

[WR99]   Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *ACM SIGPLAN International Conference on Functional Programming*, pages 148–159, Paris, 1999. ACM Press.