Bidirectional Scripting for Structured Documents

Shin-Cheng Mu, Zhenjiang Hu, Masato Takeichi
Department of Mathematical Informatics
University of Tokyo

In a *programmable* structured document, some parts of the document may be computed from other parts. For example, a table-of-contents may be computed by the body of an article. We would wish, for example, that when the table-of-content is edited, the article body be changed accordingly. The computed result, however, is usually static and not editable. Using the *birdirectional updating* techniques we developed, we describe how to automatically reflect the changes made to the result back to the source. The scripts in a structured documents therefore reacts bidirectionally to the editing actions of the user. In this paper we present our algorithm to solve global constraints in a structured document, which is implemented in a prototype editor, XDoc.

1 Introduction

With the growth of the Internet, structured documents like HTML/XML [3] has become a common format for data representation and exchange. In various occasions, one would want to create structured documents conforming to certain constraints. For example, the document shall conform to some particular format (type); the value or structure of one particular node shall always in some way be related to the value or structure of another node. One way to achieve so is to have the document be generated automatically by some program. One then checks whether the program always produce trees satisfying the constraints.

In [15], we proposed the idea of a *programmable structured document*. The idea is similar to, but more general than, allowing scripting languages in a structured document, such that some parts of the document can be computed from other parts. Our canonical example is like in Figure 1. Shown in the figure is an XML document representing an address book. The node `body` contains, as its children, a list of subtrees each representing an entry in the address book. Before the body, however, we wish to have a small index of all names. The node `index`, as specified in the `code` attribute, calls a function `toc` using the node `body` as the input (specified in the `src` attribute.) The function `toc` is defined in a standard prelude in the language `Inv` [11] to be discussed in Section 3.1, but any scripting language satisfying the *bidirectionality* property in Section 3.3 will do. After evaluation, the node `index` expands to:

```
<index code="toc" src="body">
  <name>Zhenjiang Hu</name>
  <name>Shin-Cheng Mu</name>
  <name>Masato Takeichi</name>
</index>
```

The use of a program ensures that when the body changes, the index, after re-evaluation, is always consistent with the body. In general, a computed node or its parent may be referred to by another node as its input, one node may be referred to by more than one node, and computed node may make use of more than one node as its input as well.

In our project we were developing an XML editor that displays the evaluated document to the user. A natural question arise: when the user edits an evaluated subtree, is it possible to reflect the changes back to the referred input? In the example above, when the user alters, inserts, or deletes a name in the index, we may want to change the name of the person in the entry, insert a fresh entry, or delete the corresponding entry accordingly. If the referred tree has a subtree which is again computed from another tree, the updating has to be propagated backwards too. Is there a way to conveniently specify such updating, while guarantee the correctness?

To achieve this goal we have developed various

```
<addrbook>
  <index code="toc" src="body" />
  <body name="body">
   <person>
    <name>Zhenjiang Hu</name>
    <email>hu@mist.i.u-tokyo.ac.jp</email>
    <email>hu@ipl.t.u-tokyo.ac.jp</email>
   </person>
   <person>
    <name>Shin-Cheng Mu</name>
    <email>scm@mist.i.u-tokyo.ac.jp</email>
   </person>
   <person>
    <name>Masato Takeichi</name>
    <email>takeichi@acm.org</email>
   </person>
  </body>
</addrbook>
```

図 1: An XML address book.

theories and techniques [10, 11, 8], and now we are designing a prototype editor realising our ideas. This paper extends our previous results to the scenario of a programmable structured document. We start with presenting the prototype editor, XDoc, in Section 2. The current scripting language of choice, Inv, will be reviewed in Section 3, where the important concept of bidirectionality will also be discussed. We then talk about how to make use of bidirectionality in the editor, and present a concise algorithm to solve global bidirectional constraints from local ones in Section 4.

2 Using XDoc

After launching XDoc the user is presented with a window like that in Figure 2(a). In this example, shown in the window is the document in Figure 1 after evaluation. In the information pane in the bottom of the window is shown that the selected node, `index`, has a label `toc` and calls the function `toc` with the node labelled `body` as its input. If the node were referred to by another node, such information would also be shown in the pane. Otherwise the information pane might simply say that the selected node is a plain node or a leaf.

If we select the second `person` node and click the Delete button, the selected subtree will be deleted. The action also triggers a corresponding deletion in the index, as shown in Figure 2(b). On the other hand, if we select an item in the index and press the Insert button, a pop-up dialogue would appear, allowing the user to type in a subtree. The subtree (in this example, `<name>Keisuke Nakano</name>`) would be inserted to where the selection is. Insertion into the index, furthermore, also triggers a corresponding insertion in the body, as seen in Figure 2(c). Since the code specifies no constraints on the input, the newly inserted entry in the body has `_undef` as its tag name, indicating that the value is unspecified. We may either allow the user to fill in the value, or derive the tag name `person` using some global type constraint.

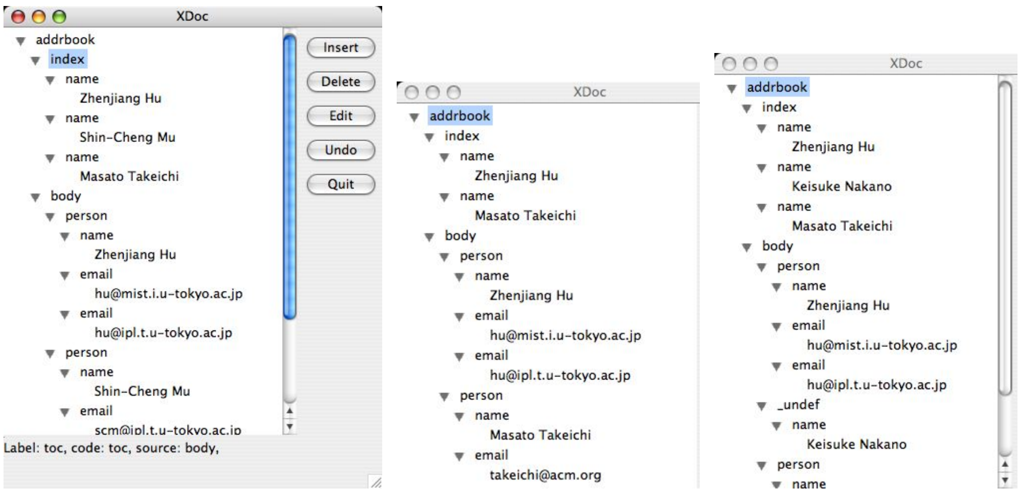
The example above illustrates two important points: firstly, having code in the document helps to maintain consistency between parts. Secondly, editing the referred node and the computed node are both allowed, and changes in one of them might trigger corresponding changes in another.

The task of allowing the user to alter both the source/target values, and recover corresponding target/source values is called *bidirectional updating*. In the next few sections we will describe our work on the bidirectional updating problem, and how it is applied to build XDoc.

3 A Bidirectional Scripting Language

To build XDoc, we had to solve one crucial subproblem. Consider a function $f :: A \to B$, $a \in A$, and $b = f\ a$. Now assume that b is slightly altered to b', how do we find an appropriate pair of value $(a', b'') \in (A \times B)$, such that a' and b'' is still related by the function f, while a' and b'' are "close enough" to a and b'? This is called the *bidirectional updating* problem. The name was first coined by [7], although their definition is slightly different from ours due to different area of applications.

We have gone through a long way to tackle the problem, and in this section we will give a brief summary. In [10], we defined a functional programming language, Inv, in which the programmers are allowed to define injective (while possibly partial and non-surjective) functions only. In [11], the language is given an extended semantics, such that



Label: toc, code: toc, source: body,

(a) (b) (c)

図 2: The prototype editor XDoc.

$$
\begin{aligned}
X ::= {} & X^{\smile} \mid \mathit{nil} \mid \mathit{cons} \mid \mathit{node} \\
& \mid \delta \mid \mathit{dupNil} \mid \mathit{duStr}\ \mathit{String} \\
& \mid X;X \mid \mathit{id} \mid X \cup X \\
& \mid X \times X \mid \mathit{assocr} \mid \mathit{assocl} \mid \mathit{swap} \\
& \mid \mu(V\colon X_V)
\end{aligned}
$$

図 3: The syntax of Inv.

when given an input b not in the domain of a function $f :: B \to A$, the extended semantics looks for a value a in the range for which there exists a "reasonable" value b' in B such that $f\ b' = a$ and b and b' are close enough. Take the inverse of f and we get what we want.

3.1 The Language Inv

In this section we will briefly introduce Inv. Only a subset of Inv relevant to the discussions later are introduced, and only the injective semantics is covered. The reader is referred to [11] for a complete discussion.

The syntax of Inv is given in Figure 3. We abuse the notation a bit by using X_V to denote the union of X and the set of variable names V. The semantics of each Inv construct and the domain of values it deals with are given in Figure 4. A relation of type $A \to B$ is a set of pairs whose first components have type A and second components type B, while a function[1] is one such that a value in A is mapped to at most one value in B. A function is injective if all values in B are mapped to at most one value in A as well. Every Inv program is an injective function from *Val* to *Val*. The range of *Val* includes unit, pairs, lists, and trees. A list is built by constructors *nil* and *cons*, where the input of *nil* is restricted to unit type. One can also produce a fresh empty list or a string using *dupNil* or *dupStr*. The constructor *node* produces a tree from a pair consisting of a label and a list of subtrees. We use a *Str s* to simulate texts and *Node tag ts* to simulate an XML element.

The function *id* is the identity function, the unit of composition. The semicolon (;) is overloaded both as functional composition and as an Inv construct. It is defined by $(f;g)\ a = g\,(f\ a)$. Union of functions is simply defined as set union. To avoid non-determinism, however, we require in $f \cup g$ that f and g have disjoint domains. To ensure injectivity, we require that they have disjoint ranges as well. The product $(f \times g)$ is a function taking a pair and applying f and g to the two components respectively. We make composition bind tighter than product. Therefore $(f;g \times h)$ means $((f;g) \times h)$.

The fixed-point of F, a function from Inv expres-

[1]For convenience, we refer to possibly partial functions when we say "functions".

sions to Inv expressions, is denoted by μF. We will be using the notation $(V : \mathit{expr})$ to denote a function taking an argument V and returning expr.

The *converse* of a relation R is defined by

$$(b, a) \in R^{\circ} \equiv (a, b) \in R$$

The *reverse* operator $^{\smile}$ corresponds to converses on relations. Since all functions here are injective, their converses are functions too. The reverse of *cons*, for example, decomposes a non-empty list into the head and the tail. The reverse of *nil* matches only the empty list and maps it to the unit value. The reverse operator distributes into composition, products and union by the following rules, all implied by the semantics definition $[\![f^{\smile}]\!] = [\![f]\!]^{\circ}$:

$$\begin{aligned}
[\![(f;g)^{\smile}]\!] &= [\![g^{\smile}]\!]; [\![f^{\smile}]\!] \\
[\![(f \times g)^{\smile}]\!] &= [\![(f^{\smile} \times g^{\smile})]\!] \\
[\![(f \cup g)^{\smile}]\!] &= [\![f^{\smile}]\!] \cup [\![g^{\smile}]\!] \\
[\![f^{\smile\smile}]\!] &= [\![f]\!] \\
[\![(\mu F)^{\smile}]\!] &= [\![\mu(X : (F\, X^{\smile})^{\smile})]\!]
\end{aligned}$$

The δ operator generates an extra copy of its argument. Written as a set comprehension, we have $\delta_A = \{(n, (n, n)) \mid n \in A\}$, where A is the type δ gets instantiated to. We restrict A to atomic types (integers, strings, and unit) only, and from now on use variable n and m to denote values of atomic types. To duplicate a list, we can always use $\mathit{map}\, \delta; \mathit{unzip}$, where *map* and *unzip* are to be introduced in the sections to come. Taking its reverse, we get:

$$\delta_A{}^{\smile} = \{((n, n), n) \mid n \in A\}$$

That is, $\delta^{\smile}$ takes a pair and lets it go through only if the two components are equal. Similarly, the reverse of *dupNil* drops the left component of a pair only of it is an empty list, while the reverse of *dupStr s* throws away the string only if we know its value.

3.2 Example Functions

A number of list processing functions can be defined using the fixed-point operator. The standard functions *foldr*, *map*, and *unzip* (transposing a list of pair to a pair of lists) can be defined exactly as the point-free counterpart of their usual definitions:

$$\begin{aligned}
\mathit{foldr}\, f\, g &= \mu(X : \mathit{nil}^{\smile}; g \cup \\
&\qquad \mathit{cons}^{\smile}; (\mathit{id} \times X); f) \\
\mathit{map}\, f &= \mathit{foldr}\; ((f \times \mathit{id}); \mathit{cons})\; \mathit{nil} \\
\mathit{unzip} &= \mu(X : \mathit{nil}^{\smile}; \delta; (\mathit{nil} \times \mathit{nil}) \cup \\
&\qquad \mathit{cons}^{\smile}; (\mathit{id} \times X); \mathit{trans}; \\
&\qquad (\mathit{cons} \times \mathit{cons}))
\end{aligned}$$

In Inv there is no higher-order functions. However, *foldr* and *map* can be seen as macros.

With *unzip* we can define a generic duplication operator. Let dup_a be a type-indexed collection of functions, each having type $a \to (a \times a)$:

$$\begin{aligned}
\mathit{dup}_{\mathit{String}} &= \delta \\
\mathit{dup}_{(a \times b)} &= (\mathit{dup}_a \times \mathit{dup}_b); \mathit{trans} \\
\mathit{dup}_{[a]} &= \mathit{map}\; \mathit{dup}_a; \mathit{unzip} \\
\mathit{dup}_{\mathit{Tree}} &= \mu(X : \mathit{node}^{\smile}; (\mathit{dup}_{\mathit{String}} \times \\
&\qquad (\mathit{map}\, X; \mathit{unzip})); \mathit{trans}; \\
&\qquad (\mathit{node} \times \mathit{node}))
\end{aligned}$$

In particular, to duplicate a list we shall duplicate each element and unzip the resulting list of pairs. In the discussion later we will omit the type subscript.

$$\begin{aligned}
\mathit{Val} &::= \mathit{Str}\ \mathit{String} \mid () \mid (\mathit{Val} \times \mathit{Val}) \\
&\quad\; \mid \mathit{List}\ \mathit{Val} \mid \mathit{Tree}\ \mathit{Val} \\
\mathit{List}\ a &::= [\,] \mid a : \mathit{List}\ a \\
\mathit{Tree}\ a &::= \mathit{Node}\ a\ (\mathit{List}\ (\mathit{Tree}\ a))
\end{aligned}$$

$$\begin{aligned}
[\![\mathit{nil}]\!]\; () &= [\,] \\
[\![\mathit{cons}]\!]\; (a, x) &= a{:}\, x \\
[\![\mathit{node}]\!]\; (a, x) &= \mathit{Node}\; a\; x \\
[\![\mathit{id}]\!]\; a &= a
\end{aligned}$$

$$\begin{aligned}
[\![\mathit{swap}]\!]\; (a, b) &= (b, a) \\
[\![\mathit{assocr}]\!]\; ((a, b), c) &= (a, (b, c))
\end{aligned}$$

$$[\![\mathit{assocl}]\!]\; (a, (b, c)) = ((a, b), c)$$

$$\begin{aligned}
[\![\delta]\!]\; a &= (a, a) \\
[\![\mathit{dupNil}]\!]\; a &= (a, [\,]) \\
[\![\mathit{dupStr}\; s]\!]\; a &= (a, s)
\end{aligned}$$

$$\begin{aligned}
[\![f; g]\!]\; x &= [\![g]\!]\; ([\![f]\!]\; x) \\
[\![f \times g]\!]\; (a, b) &= ([\![f]\!]\; a, [\![g]\!]\; b) \\
[\![f \cup g]\!] &= [\![f]\!] \cup [\![g]\!], \\
&\text{if } \mathit{dom}\, f \cap \mathit{dom}\, g = \mathit{ran}\, f \cap \mathit{ran}\, g = \emptyset \\
[\![f^{\smile}]\!] &= [\![f]\!]^{\circ} \\
[\![\mu F]\!] &= [\![F\; \mu F]\!]
\end{aligned}$$

図 4: Injective semantics of Inv constructs.

In Inv one can define injective functions. A non-injective function of type $A \to B$, on the other hand, is simulated by an injective function $A \to (A \times B)$, where a copy of the input is returned in the output. This is also the approach taken in XDoc. Every expression in the `code` attribute is supposed to have type $a \to (a, [\mathit{TreeVal}])$. The input is copied to the output, while $[\mathit{TreeVal}]$ is the list of subtrees to become the children of the current node.

The function *children* defined below takes a tree and returns the list of its children, together with the original tree. It makes use of *dup* to copy the list of children.

$$\mathit{children} = \mathit{node}^{\smile}; (\mathit{id} \times \mathit{dup}); \mathit{assocl}; (\mathit{node} \times \mathit{id})$$

The function *toc* takes an input tree and returns a copy of the tree, together with the first child of every children of the input. It is defined similar to *children*, apart from calling *extract* instead of *dup*.

$$\mathit{toc} = \mathit{node}^{\smile}; (\mathit{id} \times \mathit{extract}); \mathit{assocl}; (\mathit{node} \times \mathit{id})$$

The function *getFst* defined below takes an input tree and returns a copy of the tree together with its first child. For example,

getFst `<a><b/><c/></a>`
  = (`<a><b/><c/></a>`, `<b/>`)

The function *extract* applies *getFst* to every tree in the given list, and use *unzip* to collect all the results together.

$$\begin{aligned} \mathit{extract} &= \mathit{map}\ \mathit{getFst}; \mathit{unzip} \\ \mathit{getFst} &= \mathit{node}^{\smile}; (\mathit{id} \times (\mathit{cons}^{\smile}; (\mathit{dup} \times \mathit{id}))) \\ &\quad (\mathit{id} \times (\mathit{assocr}; (\mathit{id} \times \mathit{swap}); \mathit{assocl})); \\ &\quad \mathit{assocl}; (((\mathit{id} \times \mathit{cons}); \mathit{node}) \times \mathit{id}) \end{aligned}$$

3.3 Bidirectionality

So far, we have only described the injective semantics. The extended semantics of Inv is described in [11]. Rather than going through the extended semantics, we will merely describe what it satisfies.

For every Inv expression x having type $S \to V$, we assume the existence of two functions: $\mathit{get}_x :: S \to V$ defines the transformation from the source to an result (also called a *view*), while $\mathit{put}_x :: (S \times V) \to S$ takes the original source and an edited view, and returns an updated source.

Definition 1 (Bidirectionality) A pair of functions $\mathit{get}_x :: S \to V$ and $\mathit{put}_x :: (S \times V) \to S$ is called *bidirectional* if they satisfy the following two properties:

GET-PUT-GET :
$\mathit{get}_x\ (\mathit{put}_x\ s\ v) = v$ where $v = \mathit{get}_x\ s$
PUT-GET-PUT :
$\mathit{put}_x\ s'\ (\mathit{get}_x\ s') = s'$ where $s' = \mathit{put}_x\ s\ v$

The GET-PUT-GET property says that updating s with v and taking its view, we get v again, provided that v was indeed resulted from s — for general v this property may not hold. The PUT-GET-PUT property says that if s' is a recently updated source, mapping it to its view and immediately performing the backward update does not change its value. This property only needs to hold for those s' in the range of put_x . The two properties together ensures that when the user alters the view, we need to perform only one *put* followed by one *get*. No further updating is necessary.

4 Scripting in a Document

The previous section described Inv as a language for specifying transformations from a tree to another tree. In this section we will talk about how to embed Inv programs in documents such that subtrees may refer to each other, and how to perform bidirectional updating in this scenario.

4.1 Documents with Bidirectional Scripts

Let us call a tree node with `code` and `src` attributes a *computed* node. If node pointed to by a `src` attribute of a computed node, we call it a *referred* node. If a tree node is neither computed nor referred, we call it a *plain* node. Consider the tree below, which will be referred to as *deps* later (for brevity we assume that every node is given a label identical to its name):

```
<deps>
 <n1>BC</n1>
 <n3>A<n2 code="children" src="n1"/>D</n3>
 <n4 code="children" src="n3"/>
 <n5 code="children" src="n2"/>
</deps>
```

Nodes `n2`, `n4`, and `n5` are computed, while `n1`, `n2`, `n3` are referred.

It will turn out that it is reasonable and necessary to assume some structural constraints on the tree. A computed node, being computed itself, will not have another computed node as its decedent — that is, we do not deal with program-generating programs. A computed node may also be a referred node, and on the path from a computed node to the root there may be one or more referred nodes. However, a computed node shall not have a referred node as its decedent. It is a necessary restriction to avoid non-termination resulting from trying to extract components from a tree that is not evaluated yet. We do not lose generality this way – to have access to a decedent of a computed node, we can always refer to its parent and extract the parts we want in the code. Finally, the tree together with the src-to-target arcs shall form a directed acyclic graph – no circularity is allowed.

4.2 Global Get and Put

Given a possibly unevaluated tree, some nodes may contain `code` and `src` attributes. The task is to define a function *getAll* that (re)evaluates these nodes. The tricky part here is that a referred node itself, or some of its subtrees, may need to be computed. Therefore we need to refer to the node in the evaluated tree. The function *buildG* takes a fully-evaluated tree t', and the to-be-evaluated tree t. It traverses through t, and looks up the referred nodes in t' when necessary. Its completed definition is long, but it basically has the following structure:

```
buildG                  :: Val → Val → Val
buildG t′ (Str s)       = Str s
buildG t′ (Node tag ts)
 | isComputed tag  =
   let f   = getCode tag
       src = locate t′ (getSrc tag)
       ts′ = get_f src
   in Node tag ts′
 | otherwise  = Node tag (map (buildG t′) ts)
```

If the current node is a plain node, *buildG* simply processes the subtrees recursively. Otherwise it tries to evaluate the tree. Functions *getCode* and *getSrc* merely scan through the list of attributes in *tag*, extract, and parse `code` and `src`. The function *locate* scans the tree (t') and looks for the node with the given label (we actually allow a list of sources but here we assume that a computed node refers to only one source). We then call get_f to compute the subtrees. How do we get the fully-evaluated tree t', then? To do so we build a circular program, in which the result of *buildG* is fed back to itself.

```
getAll   :: Val → Val
getAll t = let t′ = buildG t′ t in t′
```

The function *getAll* takes a tree and performs a global *get*.

Now assume that the user performed some editing on the tree. The modified node may be a computed node, a referred node, or a plain node. We also want to define a global *put* function to update the referred nodes according to the change. We define:

```
putAll    :: Val → Val
puttAll t = let t′ = buildP t′ (findPaths t) t in t′

buildP :: Val → (Label → Path) → Val → Val
buildP t′ fnd (Str s)         = Str s
buildP t′ fnd (Node tag ts)
 | isReferred tag  =
   let Node tt us = invite (fnd (getTar tag))
       f          = getCode tt
       ts′        = put_f (Node tag ts, us)
   in Node tag (merge t′ fnd ts ts′)
 | otherwise  = Node tag (map (buildP t′ fnd) ts)
```

The function *buildP* is defined in a way essentially similar to *buildG*. To keep the function lazy enough and avoid non-termination, however, we cannot perform a global search by *locate*. Instead we first scan through the tree and build a *Label* → *Path* mapping using t, and extract the need tree according to the path, without evaluating other parts of t'. The function *findPaths* builds the label-to-path mapping, where a path is simply a sequence of integers, each indexing the list of children starting from 0. For example, *findPaths deps* yields a function such that

$$\mathit{findPaths}\ \mathit{deps}\ \text{“}\mathtt{n1}\text{”} = [0]$$

$$\textit{findPaths deps } \text{“}\texttt{n2}\text{”} = [1, 1]$$
$$\textit{findPahts deps } \text{“}\texttt{n4}\text{”} = [2]$$
$$\vdots$$

and so on. The function *invite* takes a path and extracts the node at that path. For example,

$$\textit{invite}\ [1, 1]\ \textit{deps} =$$

```
<n2 code="children" src="n1"/>
```

An important thing is that *invite* has to be lazy enough: it shall not touch the nodes not on the path. Furthermore, adding to the complication is the fact what we cannot safely assume that the path of a node in t stays the same in t'! Extra care is needed to make the old path work in the new tree. That is a very tedious technical detail we will omit here.

Instead of get_f, in *buildP* we apply put_f to a pair of the current referred node *Node tag ts* and the computed node *us*. The result ts', however, cannot simply overwrites *ts*. Indeed, ts' may have been altered (some trees deleted or inserted, some tags changed); however, *ts* may have also been altered too, due to propagation of changes from other parts of the tree. We need a *merge* function which compares *ts* and ts' and builds a forest compatible with changes in both. Furthermore, in the subtrees of *ts* there may be other referred trees. In such cases *merge* would again call *buildP* to do the updating (that is why it needs t' and *fnd* as its argument). The actual definition of *merge* involves more technical parts of the Inv semantics and will be omitted here.

After each atomic editing action (insertion, deletion, or change of tag), the editor calls *putAll* followed by *getAll* to update the tree. Thanks to bidirectionality, one get followed by one put is sufficient. This is stated as a theorem below:

Theorem 1 *With the definition above, if bidirectionalty holds for all transformations in the document, we have:*

$$\textit{getAll} \cdot \textit{putAll} \cdot \textit{getAll} = \textit{getAll}$$

4.3 Examples

Recall the tree *deps* in the beginning of Section 4, which evaluates to (again for brevity we omit the `code` attributes, which are all set to `"children"`):

```
<deps>
 <n1>BC</n1>
 <n3>A<n2 src="n1">BC</n2>D</n3>
 <n4 src="n3">
   A<n2>BC</n2>D
 </n4>
 <n5 src="n2">BC</n5>
</deps>
```

Now assume that the user inserted a new element under `n4` between `B` and `C`, making the subtree look like:

```
 <n4 src="n3">
   A<n2>BEC</n2>D
 </n4>
```

The system will then perform a *putAll*. Since `n4` refers `n3`, the change will be propagated to `n3`. A subtree of `n3`, moreover, is computed from `n1`. The updated tree is therefore:

```
<deps>
 <n1>BEC</n1>
 <n3>A<n2 src="n1">BEC</n2>D</n3>
 <n4 src="n3">
   A<n2>BEC</n2>D
 </n4>
 <n5 src="n2">BC</n5>
</deps>
```

After the *putAll* we need a global *getAll*. The children of node `n1` is copied to `n2`, referred by `n5`. Therefore `n5` also gets updated:

```
<deps>
 <n1>BEC</n1>
 <n3>A<n2 src="n1">BEC</n2>D</n3>
 <n4 src="n3">
   A<n2>BEC</n2>D
 </n4>
 <n5 src="n2">BEC</n5>
</deps>
```

This finishes the updating. Thanks to bidirectionality, a *putAll* followed by a *getAll* gives us a stable tree. No further updating is necessary.

5 Conclusion and Related Work

We have presented XDoc, an editor for structured documents with embedded scripting program to compute parts of the document from other parts. Allowing parts of the documents to be generated helps to ensure that the document follows some certain format, and the contents are always consistent. The use of a bidirectional language, Inv, provides a convenient mechanism to keep the consistency between the computed and the referred subtrees. When either is altered, the other can be updated automatically.

Although Inv is essentially as powerful as the reversible Turing machine, writing large programs in a point-free functional language is a tiresome task. It shall be easy to give Inv a point-wise syntax so the programmer may concentrate on the algorithmic side of the code rather than distributing and swapping values in pairs. Further more, we are also developing a higher-level lanugage, X [8], providing tree-specific operations. The language X can be embedded into Inv.

Proxima [14] is a highly configurable editor which also allows "derived values" in documents. One specifies the structure of the document in attribute grammar, and the editor would know how to parse, display, edit the document, as well as updating the derived values. Furthermore, Proxima also allows non-structural editing, which greatly enhances the flexibility and usability of the editor.

The bidirectional updating problem can be traced back to the view-updating techniques in database community [2, 4, 6, 12, 1]. In the context of data synchronisation, it was identified by [7]. In [7, 5], a semantic foundation and a programming language (the "lenses") for bidirectional transformations are given. They form the core of the data synchronisation system Harmony [13]. Another very much related language was given by Meertens [9] to specify constraints in the design of user-interfaces. Due to their intended applications, less efforts were put on describing either element-wise or structural dependency inside the view.

Acknowledgements

This research is partly supported by the e-Society Infrastructure Project of the Ministry of Education, Culture, Sports, Science and Technology, Japan.

参考文献

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

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

[3] T. Bray, J. Paoli, C. M. Sperberg-Macqueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition), October 2000. `http://www.w3.org/TR/REC-xml`.

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

[5] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *The 32nd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL 2005)*, pages 233–246, Long Beach, California, 2005. ACM Press.

[6] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems*, 13(4):486–524, December 1988.

[7] M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. A language for bi-directional tree transformations. Technical Report, MS-CIS-03-08, University of Pennsylvania, August 2003.

[8] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Proceedings of ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation*, Verona, Italy, August 2004. ACM Press.

[9] L. Meertens. Designing constraint maintainers for user interaction. `ftp://ftp.kestrel.edu/pub/papers/meertens/dcm.ps`, 1998.

[10] S.-C. Mu, Z. Hu, and M. Takeichi. An injective language for reversible computation. In *Seventh International Conference on Mathematics of Program Construction*, number 3125 in Lecture Notes in Computer Science. Springer-Verlag, July 2004.

[11] S.-C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bi-directional updating. In W.-N. Chin, editor, *The Second Asian Symposium on Programming Language and Systems*, number 3302 in Lecture Notes in Computer Science, pages 2–20. Springer-Verlag, November 4-6, 2004.

[12] A. Ohori and K. Tajima. A polymorphic calculus for views and object sharing. In *Proceedings of the 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 255–266. ACM Press, 1994.

[13] B. C. Pierce, A. Schmitt, and M. B. Greenwald. Bringing harmony to optimism: an experiment in synchronizing heterogeneous tree-structured data. Technical Report, MS-CIS-03-42, University of Pennsylvania, March 18, 2004.

[14] M. M. Schrage. *Proxima - A presentation-oriented editor for structured documents*. PhD thesis, Utrecht University, The Netherlands, 2004.

[15] M. Takeichi, Z. Hu, K. Kakehi, Y. Hayashi, S.-C. Mu, and K. Nakano. TreeCalc:towards programmable structured documents. In *The 20th Conference of Japan Society for Software Science and Technology*, September 2003.