内部参照をもつXML文書の双方向変換による編集

木津 幸子　武市 正人　胡 振江

XML に代表される構造化文書処理に双方向変換を用いることが注目されている．大きなデータベースの更新において，関心のある部分情報を抽出したビュー上で該当するデータを更新するビュー更新においてもその有効性が示されている．本稿では，項目間に ID/IDREF による参照関係が存在する XML 文書 (データベース) の系統的な編集の実現のために，ソース (原文書) から抽出したビューの取扱いに関して双方向変換に基づくビュー更新 (view-updating) の観点から，文書内部の依存関係とファイル上の外部表現上の ID/IDREF の参照関係を論じる．XML 文書の双方向変換には，関数型言語 Haskell による XML 変換ライブラリ HaXml を双方向化した Bi-HaXml を用いる．Bi-HaXml は単一順方向の変換を扱う HaXml のプログラムのままで，双方向変換を実現するライブラリであり，各種の XML 文書処理において付加的なプログラムなしで，矛盾のない逆方向変換処理を実現できる特長をもっている．本稿では，双方向変換の過程で現れる文書内部のデータの依存関係と ID/IDREF による参照関係の関連を明らかにして，双方向変換と整合性を保ちつつ，参照関係の更新操作を Bi-HaXml に付加する提案を行う．

1　はじめに

XML [3] は構造データを表現するフォーマットとして幅広く利用されている．XML 文書をデータベースとして用い，変換プログラムにより別の文書を生成する機構を考えると，生成された文書上であるデータを変更したり新しいデータを追加したりしたいときに，整合性を保った編集を実現するのは容易ではない．このような XML の編集を双方向変換によるビュー更新 [1] [2] [4] [5] [6] [7] として捉え，データの変更を直観に矛盾ない形で各文書に反映させる有効な方法が求められる．

われわれはすでに [8] [10] で，双方向変換によるビュー更新のシステムを述べたが，本論文ではこれらに現れる変換処理の過程で生じる文書内の相互依存性と XML 文書における ID/IDREF [3] による内部参照の関係を論じ，内部参照をもつ XML 文書の双方向変換による編集のあり方を提案する．

XML は属性付き木構造であり柔軟な表現力を持つが，木構造を越えた構造を表現する仕組みを用意して実用に供している．ID/IDREF はその一つの仕組みであり，ある木ノードから別の木ノードを参照する機能を用いて，離れた木ノードを指定するといった単純な木構造では表現できないデータをも表現することが可能となっている．このような XML 文書に対しては XML 文書の意味づけが与えられているが，それを双方向に変換する際にはあらためて意味づけを行う必要がある．一方で，われわれが [4] で提案した双方向変換における文書編集においては，項目の追加，削除等の単純なビュー更新だけではなく，順方向変換によって作り出されたビュー内の項目間の依存関係が重要な役割を果たしている．それは，XML データベースのある項目が変換によってビュー上の複数の場所に現れているときに，そのいずれを更新しても，データベース上の項目はもちろんのこと，ビュー上の他の場所にあるものも統一的に更新することができるという機構である．本稿では，外部表現としての XML の ID/IDREF による参照機構と，双方向変換における

Edits on XML Documents with Inner References Using Bidirectional Transformations.

Sachiko Kizu, Masato Takeichi, 東京大学大学院情報理工学系研究科, Graduate School of Information Science and Technology, University of Tokyo.

Zhenjiang Hu, 国立情報学研究所, National Institute of Informatics (NII).

依存関係とを論じるものである．

本稿では，XML 文書の双方向変換のための言語として，関数型言語 Haskell による XML 変換ライブラリ HaXml [9] を双方向化した Bi-HaXml を用いる．Bi-HaXml は通常の HaXml による順方向変換のプログラムとまったく同一のプログラムが逆方向変換をも備えるようにしたもので，逆変換プログラムの開発が必要なく，また，順方向と逆方向の変換に関する一貫性を持ち，信頼性を確保している．Bi-HaXml の詳細を述べることはしないが，双方向変換による依存関係に焦点を当てて概説することとする．Bi-HaXml は，すでに開発した Bi-X [5] に比べて高水準の言語であり，一般の XML 文書処理にも容易に使うことができる．

2 双方向変換による文書編集

双方向変換 [1] [2] [4] [6] [7] による文書編集とは，双方向変換を用いて，変換プログラムで関係付けられた文書間の一貫性を保つように，データの一部分を変更してもその変更を各文書中のすべての出現箇所に反映させるための機構のことをいう．

双方向変換は，ある文書から別の文書への順方向変換と変換された文書からもとの文書への逆方向変換とからなる．もととなる文書をソース，ソースと変換プログラムから生成された文書をビューと呼ぶ．当然のことながら，順方向変換で得られたビューそのままに逆方向変換を適用すれば，もとのソースと同一のものが得られる．一方で，ビュー上で編集を行った場合には，逆方向変換によってソースにその変更を反映させることが期待される．このようなビュー更新の枠組みでは，ソース文書はデータベースといえるほどに膨大な情報を含むものであり，ビューは更新対象となる部分を抽出したものとされる．したがって，ビューの情報だけでもとのソースを復元することはもちろんのこと，ソースの一部にビュー上の更新を反映させるわけにはゆかない．ビューとして抽出した部分さえ特定できないからである．このようなことから，ビューに加えられた変更をソースに反映するための逆方向変換は，もとのソースと変更されたビューを用いて，ビュー上の変更を反映させることになる (図 1).

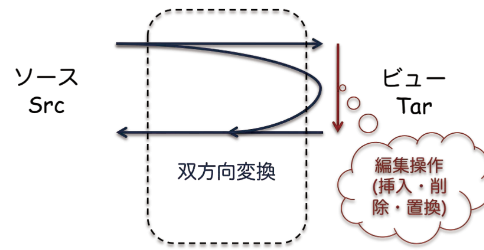

図 1 双方向変換によるビュー更新 (編集)

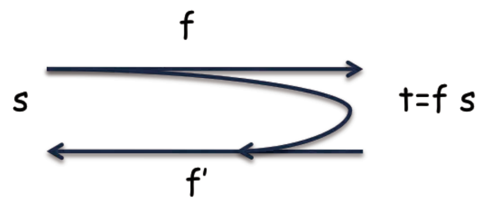

図 2 安定性

以上のような双方向変換は，原理的にはソースのデータ型 Src からビューのデータ型 Tar への順方向の関数 f とその逆方向の関数 f’ の対 (f, f’) によって実現することができる。すなわち，Haskell の表現では

```
type Filter = (Src->Tar, Src->Tar->Src)
f :: Src -> Tar
f’ :: Src -> Tar -> Src
```

となる．関数 f’ の第 1 引数はもとのソースであり，第 2 引数が (変更された可能性のある) ビューである．もちろんのこと，双方向変換 (f, f’) の関数 f と f’ の間には上に述べたような性質を満たす必要があり，その明確な定義が双方向変換による編集処理を定式化することになる．一般には，双方向変換には以下の条件を課すことが望まれる．

安定性 (stability) 任意の s に対して，

```
f’ s (f s) = s
```

この性質はあるソースを順方向変換して得られたビューに何も変更を加えないで逆方向変換を実行すると，もとのソースと等しくなるという性質の要請である (図 2).

保全性 (preservation) 任意の s に対して，

```
f (f’ s t’) = t’ where t’ = h s
```

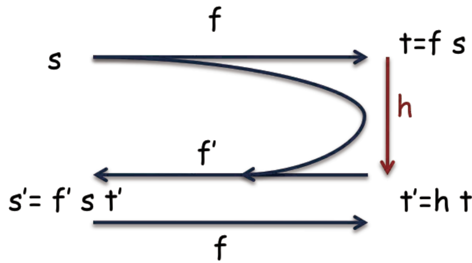

図 3　保全性

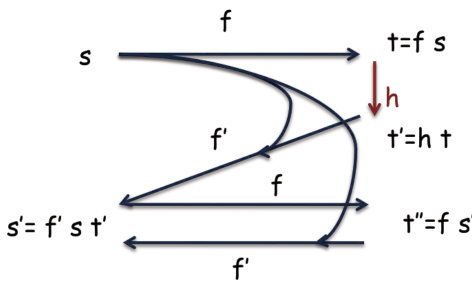

図 4　複製保全性

この性質はビューにどのような変更 h が加えられても，もとのソースをそのまま返すというような意味のない逆変換を排除するための性質である (図 3).

一方で，われわれが用いる双方向変換ではソースのあるデータが順方向変換によってビュー上に複数個現れることがあり，そのうちの一つを変更したものがソースに反映させるだけでなく，ビュー上の他の場所にも反映させるので，上の保全性は成立しないことがある．このような変換は本稿で後述するように，*duplication* という複製であり，このような変換を含む場合には，保全性は次のように定義される (図 4).

複製保全性 (duplication preservation)　任意の s に対して，

```
f' s t'' = s' where
  s' = f' s t'; t' = h (f s); t'' = f s'
```

双方向変換による編集のためには，原理的には以上のような性質をもつ関数 f と f' の対を用いればよいことになるが，われわれの双方向変換では，順方向変換と逆方向変換を一つの関数で定義しようというものである．これによって，逆方向プログラムの作成が不要となり，生産性を向上させるとともに，上にあげた条件を満たす逆方向変換が同時に得られるので，信頼性が高められるという利点がある．実際のところ，順方向変換に応じて，それに対応する正しい逆方向変換を得ることは簡単なことではなく，また，得られた関数の正当性を証明することも容易ではない．関数を定義するときに，双方向性を考慮しておきさえすれば，個別に正当性を示す必要はないということである．

さらに，われわれは通常の単一方向の変換プログラムをそのまま双方向変換プログラムとして使えるようにした．以下に述べる Bi-HaXml は既存の XML 文書処理の言語 (ライブラリ) HaXml [9] をすべて双方向化することにより，HaXml で書かれた変換による XML 文書処理に一切の手を加えずに双方向変換処理にできるという特徴がある．

3　HaXml の概要

HaXml [9] は 1999 年に York 大学 (イギリス) で開発された，いわゆる “ポイントフリー” の XML 文書処理のための領域向け言語であり，Haskell でライブラリとして実現されたものである．ここでは次章にあげる HaXml 双方向化の際に必要となる，HaXml における基本的なことがらを概観する．なおここで述べるものは stable version である HaXml 1.13.3 に即している．

3.1　XML データの型

HaXml では XML データの型を Content と呼び，以下のように定義している．

```
data Content   = CElem Element
               | CString Bool String
data Element   = Elem String
                   [Attribute] [Content]
type Attribute = (String, AttValue)
data AttValue  = AttValue
                   [Either String Reference]
data Reference = ... 省略
```

これは Content が，Content のリストを子供に持つような要素データ CElem であるか，もしくはテキ

ストデータ CString であるかのいずれかであることを示している．また Content は Attribute のリストを持ち，Attribute は属性の名前とその値との組で構成される．

3.2 HaXml のフィルタ

HaXml ではフィルタと呼ばれる Content から [Content] への関数を合成することによってプログラムを構成する．HaXml におけるフィルタは CFilter として，

```
type CFilter = Content -> [Content]
```

と定義されている．

HaXml はポイントフリーであることから，そのプログラム中に変数の定義や参照が存在せず，新たな定義や構造を導入する機構もないので，自ずからプログラム自体が CFilter として機能するものとなる．

HaXml のフィルタには引数にフィルタをとるものととらないものがあり，引数にフィルタをとらないものを基本フィルタ，引数にフィルタをとるものを高階フィルタと呼ぶ．また，基本フィルタと高階フィルタのいずれに対しても，返値の構成方法からの分類として，フィルタの返値リストの各要素が入力の部分木であるような選択フィルタと，返値リストの要素中に入力部分木以外の情報が付け加わった構築フィルタとに分けられる．

- 型による分類
  - **基本フィルタ** 引数にフィルタをとらない
  - **高階フィルタ** 引数にフィルタをとる
- 返値の構成法による分類
  - **選択フィルタ** 返値の木が入力木の部分木からなる
  - **構築フィルタ** 返値の木に入力部分木以外のものが含まれる

以下では基本フィルタ，高階フィルタの順に詳細をみていく．

3.2.1 基本フィルタ

まずはじめに基本フィルタの選択フィルタについてみてみる．最も簡単な基本フィルタには keep と none があり，keep はいかなる入力に対してもそれをシングルトンリスト (要素がひとつだけのリスト) にして返し，none はいかなる入力に対しても空リストを返す．

フィルタの型は Content を引数にとって Content のリストを返す CFilter であるが，実際のところほとんどの基本フィルタは，入力木に対する出力木のリストがシングルトンであるシングルトンフィルタであり，シングルトンフィルタでないフィルタをリストフィルタと呼ぶ．基本フィルタの中で唯一のリストフィルタである children は入力木の子供のリストを返す．

残りの基本フィルタはいずれも，入力がある条件を満たせばそれをシングルトンリストにして返し，そうでなければ空リストを返すという述語フィルタである．述語の種類により 5 つの述語フィルタ elm，txt，tag，attr，attrval が存在し，それぞれ要素データであるか，テキストデータであるか，要素データのラベル名が指定した文字列と一致するか，指定した名前の属性が存在するか，指定した名前と値の組の属性が存在するか，が対応する述語にあたる．基本フィルタの選択フィルタ一覧を表 1 に示す．ただし SL はシングルトンリストを指すものとする．

次に基本フィルタの構築フィルタをみる．この分類には 2 つのフィルタ literal と replaceTag が存在し，literal は入力が何であろうと，指定した文字列を CString の Content としてシングルトンリストにして返し，replaceTag は入力木が要素データであればそのルートノードのラベル名を指定した文字列に変えてそのシングルトンリストを，テキストデータであれば空リストを返す．基本フィルタの構築フィルタ一覧を表 2 に示す．

3.2.2 高階フィルタ

次に高階フィルタをみる．高階フィルタは引数にとるフィルタにより選択フィルタにも構築フィルタにもなりうるが，引数のフィルタが選択フィルタだけならば全体も選択フィルタになる準選択フィルタについてみると，基本的なものには o，cat，with，(?>) の 4 つがあり，ここではそれぞれおおまかな意味を述べる．f ‘o‘ g はまず入力木にフィルタ g を適用してその結果の木それぞれについてフィルタ f を適用し最後にそれらを連結する．cat fs は引数にフィルタ

表 1　基本フィルタの選択フィルタ

フィルタ名	型	返値の分類	入力 Content に対する返値
keep	CFilter	singleton	入力そのままの SL
none	CFilter	singleton	空リスト
children	CFilter	list	入力木の子供のリスト
elm	CFilter	singleton	要素データならその SL
txt	CFilter	singleton	テキストデータならその SL
tag	String -> CFilter	singleton	指定したラベル名ならその SL
attr	String -> CFilter	singleton	指定した名前の属性があればその SL
attrval	(String, String) -> CFilter	singleton	指定した名前と値の組の属性があればその SL

表 2　基本フィルタの構築フィルタ

literal	String -> CFilter	singleton	指定したテキスト木の SL
replaceTag	String -> CFilter	singleton	入力木のラベルを指定文字列に変えた SL

のリストをとり，入力木にそれぞれのフィルタを適用した結果を最後に連結する．f ‘with‘ g はまず入力木にフィルタ f を適用し，そのそれぞれについてフィルタ g を適用した結果が空リストでないものだけを残す．最後に p ?> f :> g は，入力木にフィルタ p を適用した結果が空リストでなければ入力木に f を適用した結果を返値とし，空リストであれば入力木に g を適用した結果を返値とする．これら 4 つの高階フィルタの型を表 3 にまとめる．

最後に，高階フィルタのうち引数がすべて選択フィルタであっても全体が構築フィルタになる高階構築フィルタについてみる．この分類のフィルタは mkElem と mkElemAttr のふたつである．mkElem は文字列とフィルタのリストを引数にとり，指定文字列をルートノードのラベル名として，引数のフィルタリストを cat した結果をその子供としてもつような木をシングルトンリストとして返す．mkElemAttr も同様で，さらに文字列とフィルタの組のリストをとり，指定文字列の名前で，値はフィルタで指定したものをとってきて属性として付け加える．表 4 に高階構築フィルタの型一覧を示す．

以上フィルタを 4 つに分類して，HaXml に定義されている基本的なフィルタを見てきた．各分類ごとにまとめたものを表 5 に示す．

表 5　HaXml フィルタの分類

要素	基本フィルタ	高階フィルタ
(準) 選択フィルタ	keep, none, children, elm, txt, tag, attr, attrval	o, cat, with, (?>)
構築フィルタ	literal, replaceTag	mkElem, mkElemAttr

3.3　HaXml による XML 変換

ここでは HaXml を用いた XML 変換がどのようになされるのかを具体例を通してみていく．まずソース XML として，著者の名前とメールアドレスからなる著者リストをデータベースとして用意し，図 5 に示した．このソース XML から人物名だけをリストアップする HaXml による変換プログラム例を図 6 に，図 5 のソースを図 6 のプログラムで変換した結果を図 7 に示す．

HaXml による変換プログラムとして図 6 に与えられたフィルタは，memberlist というルートノードをつくりだし，その子供として，ソース XML の子供で author という名前を持つノードそれぞれについて member という名前を持つノードをつくり出し，その子供として person ノードの子供の name ノードの子供のテキストデータを用いる，という変換を表して

表 3　高階フィルタの準選択フィルタ

o	CFilter -> CFilter -> CFilter
cat	[CFilter] -> CFilter
with	CFilter -> CFilter -> CFilter
(?>)	CFilter -> ThenElse CFilter -> CFilter

表 4　高階構築フィルタ

mkElem	String -> [CFilter] -> CFilter
mkElemAttr	String -> [(String, CFilter)] -> [CFilter] -> CFilter

```
<authorlist>
  <author>
    <name>Sachiko Kizu</name>
    <email>sachiko@ipl</email>
  </author>
  <author>
    <name>Masato Takeichi</name>
    <email>takeichi@ipl</email>
  </author>
  <author>
    <name>Zhenjiang Hu</name>
    <email>hu@nii</email>
  </author>
</authorlist>
```

図 5　ソース XML 文書の例

いる．

この変換プログラムを見ると，高階フィルタの引数にさまざまなフィルタを組み合わせて使うことで，全体として複雑な変換を表すフィルタをつくり出すことに成功していることがわかる．基本フィルタだけではできる変換も限られるが，基本フィルタを高階フィルタの引数に与えることでできる変換の幅が広がり，さらにフィルタを引数として与えられた高階フィルタ自体もまたフィルタであるので，高階フィルタの引数として与えることができ，さまざまな変換をフィルタとして表現することが可能となる．

このように高階フィルタは，個々のフィルタが表す変換をつなぎ合わせて全体として一つの変換をつくり出すという意味において重要であり，Haskell の得意とする関数合成のうまみを存分に活かしていると言える．

```
mkElem "memberlist"
  [ mkElem "member"
    [ keep /> tag "name" /> txt ]
    `o` (keep /> tag "author")
  ]
```

図 6　HaXml による変換プログラムの例

```
<memberlist>
  <member>Sachiko Kizu</member>
  <member>Masato Takeichi</member>
  <member>Zhenjiang Hu</member>
</memberlist>
```

図 7　図 5 のソースを図 6 のプログラムで変換した結果

4 Bi-HaXml

4.1 HaXml の双方向化

第 2 節で見たように，双方向変換は順方向と逆方向の変換のための関数の対 (f,f') で実現することができるが，すでに述べたように，f と f' を個別に定義するのでは生産性の点からも，また，信頼性を保証する上でも望ましくない．HaXml では，ライブラリに用意されたフィルタの合成で変換を記述するので，HaXml を双方向化した変換プログラムでは，双方向化されたフィルタの合成で変換を記述できれば用いればよい．すなわち，HaXml のフィルタである CFiler

```
type CFilter = Content -> [Content]
```

に対して

```
type Src = Content
type Tar = [Content]
type XFilter = (Src->Tar, Src->Tar->Src)
```

のように定義された XFilter を，双方向化されたフィルタとして用いればよいことになる．こうすれば，適

```
children :: XFilter
children = XFun f where
  f (CElem (Elem n as cs)) = (cs, f')
    where
    f' cs' = CElem (Elem n as cs')
  f s                      = ([], const s)

tag :: String -> XFilter
tag t = XFun f where
  f x@(CElem (Elem n _ _))
    | t == n    = ([x], f')
    | otherwise = ([], const x)
    where
    f' [x'] = x'
    f' t'   = error ("tag Backward: "++show t')
  f x           = ([], const x)

cat :: (Eq a, Eq b) => [XFun a [b]] -> XFun a [b]
cat []  = XFun f where f s = ([], const s)
cat xfs = XFun f where
  f s = (concat tss, f')
    where
    (tss, gs') = unzip (map (\(XFun g) -> g s) xfs)
    ls = map length tss
    f' ts' = k tss tss' gs'
      where
      tss' = unconcat ls ts'
      unconcat [] _        = []
      unconcat (l : ls) ts' = take l ts' : unconcat ls (drop l ts')
      k [] _ _        = s
      k (ts : tss) (ts' : tss') (g' : gs')
        | ts == ts' = k tss tss' gs'
        | otherwise = g' ts'
```

図 8　Bi-HaXml におけるフィルタ定義の一部

切に定義された高階フィルタによって合成されて得られる変換プログラムは自然に XFilter のフィルタとなる．しかし，われわれはさらに効果的な実現法を用いることとした．それは

```
data XFun a b = XFun (a -> (b, b -> a))
```

によって定義される XFun を用いる[†1]ものである．これによって，双方向フィルタ XFilter を新たに

```
type Src = Content
type Tar = [Content]
type XFilter = XFun Src Tar
```

と定義し直した．

Bi-HaXml は以上の考えに基き，HaXml のひとつひとつのフィルタに対して，双方向フィルタ XFilter としての定義を与えたものである．フィルタを XFilter として実装したものの一部を例として図 8 に示す．

XFun 方式の XFilter を用いる利点として，例えば children の逆変換を記述する際に，XFilter の型を順変換 f と逆変換 f' の組とした場合だと，順変換 f と全く同じ定義を逆変換 f' の定義中に再度与える必要があるが，XFun 方式の XFilter だと，順方向の結果を再利用することができ，プログラムのメンテナンスがより容易に行える．またこのことから，cat の定義に見られるように，引数として与えるフィルタごとに順変換で生成されるリストの長さを記憶し，それを逆変換で利用することも可能となる．

†1 もちろん，データ構成子 XFun を用いることなく，単に type a b = a -> (b, b -> a) としてもよいが，プログラム構成上で型による整合性の確認を容易にするために代数データ型を用いている．

4.2 Bi-HaXml による XML の双方向変換

ここでは Bi-HaXml による XML 変換の様子を具体例を用いて解説する．図 5 のソース XML に対して，図 6 で示した HaXml と同じ変換プログラムが Bi-HaXml で記述されている場合にどのような振る舞いを示すか見ていく．

図 9 は Bi-HaXml での XML 変換を支援するためのビューアであり，左側がもとのソース，右側が順方向変換により生成したビューを表す．このビューアにおいては，XML 文書はこのようなツリーで表現される．

このビューア上では，生成したビューに対して 3

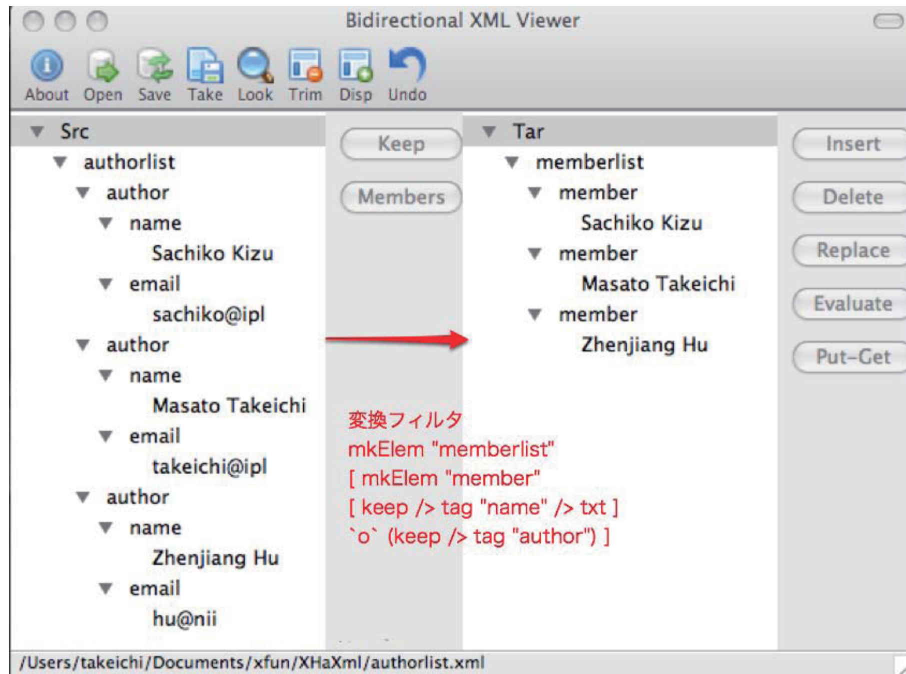

図 9　Bi-HaXml による XML 双方向変換ビュー更新支援ビューア

種類の変更，すなわち，要素の挿入，要素の削除，テキストデータの変更，の操作を加えることができる．それぞれ，ビュー上で変更を加えたい箇所を選択し，ビューの右側に表示されているボタンの，Insert，Delete，Replace をクリックすることで，対応する変更を加えることができる．

さらに，各変更が完了するとプログラムは自動的に逆方向変換を実行してビュー上の変更をソースに反映し，それに続けて順方向変換を実行することでビューが再生成され，ソースとビューとの整合性が保たれる．

5　双方向変換過程における依存関係

HaXml の結合子 union を用いた f ‘union‘ g は，ふたつのフィルタ f と g が同一のデータを受けとり，それぞれで得られた結果の [Content] 型のデータを結合したものを与えるフィルタを表す (図 10)．

この union を双方向化するにあたって，素朴にたとえば，

```
union :: (Eq a, Eq b) =>
  XFun a [b] -> XFun a [b] -> XFun a [b]
union (XFun f) (XFun g) = XFun k where
  k s = (t1++t2, k’)
    where
    (t1, f’)= f s
    (t2, g’)= g s
    k’ = ...
```

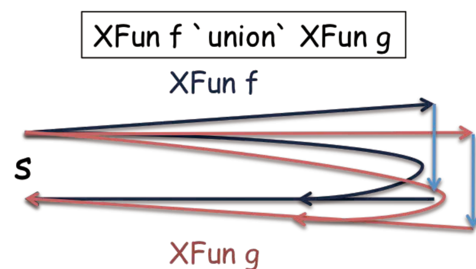

図 10　Bi-HaXml の union 結合子

のように定義することが考えられよう．しかしながら，逆変換 k’ をどのように定めればよいのであろうか．単純には，k’=f’，あるいは k’=g’ とすることも考えられる．しかし，これでは 2 節にあげた双方向変換の複製保全性の性質を満たすことはできない．

k'=f' とすれば，t2 の部分に加えられた変更を逆変換によって s に伝えることができないからである．同様に，k'=g' としてもうまくゆかない．

このように，われわれの対象とする双方向変換においては，同一のデータに対して複数のフィルタが適用される場合に，変換後のビュー上のデータの間に依存関係が生じることになる．実際のところ，われわれの提案する双方向変換によるビュー更新における強力な編集機能はまさにこの機構によるといえる．

Bi-HaXml の結合子 union の定義は，

```
union :: (Eq a, Eq b) =>
    XFun a [b] -> XFun a [b] -> XFun a [b]
union (XFun f) (XFun g) = XFun k where
  k s = (t1++t2, k')
    where
    (t1, f')= f s
    (t2, g')= g s
      l = length t1
      k' t'
        | t2'==t2  = f' t1'
        | t1'==t1  = g' t2'
        | otherwise = error "union Backward"
          where
          t1' = take l t'
          t2' = drop l t'
```

のように，少々複雑なものになっている．そこでは，順方向の変換による t1 と t2 のそれぞれが編集によって変更されたかどうかを確認した上で，変更された側の逆変換を全体の逆変換としているのである．双方が変更される状況が生じたときには，統一的にそれをソースに反映させることはできないのでエラーとしている．実用的には，編集操作のたびに逆変換を適用するという "オンライン編集" を想定して，このようなことが起こらないものとしている．

XML の双方向変換においては，変換の過程で XML 自体の機能で表現できない依存関係が生じることになる．もちろん，このことは XML に限らず，一般的な双方向変換にも現れることである．Bi-HaXml における結合子 union の場合のように，一つのデータをいくつかの順方向変換で共通に用いたときの逆方向変換は，複製保全性を満たすように慎重に定める必要があるといえる．双方向変換言語 X [7] ではこのような構成を *duplication* と呼んでいる．用語だけからは *duplication* と単純な *copy* とは紛らわしいが，ここでもこの違いを明確にして扱う必要がある．*copy* は，もとのデータは同じではあるが，そのうちの一方を変更したとしてももう一方は変わることがないような複製である．Bi-HaXml には，union を一般化して，任意個数のフィルタを同一のソースに適用する結合子 cat(図 8 参照) も用意されている．

実際のところ，Bi-HaXml の union による変換において，引数の 2 つのフィルタによってビュー上に同一のデータが複数個現れたものを，*duplication* の意味を保持したまま外部に XML ファイルとして保存することはできない．すなわち，適当にビューとして見えているテキストをファイルに出力することはできても，その中に現れる複数個の同一のテキストが *duplication* によるものであるのか，*copy* であるのかはもはや区別することができない．

6 ID/IDREF を含む XML 文書の変換

XML 文書には，文書内の相互の引用ができるように ID/IDREF による参照機能が備わっている．前節に述べたような，文書内における依存関係の一つであるということができる．

W3C [3] の仕様によると ID/IDREF は以下の制約を満たさなければならない．

ID ID 型の値はひとつの XML 文書中に二度以上現れてはならない．ひとつの要素型にふたつ以上の ID 属性を指定できない．

IDREF IDREF 型の値は XML 文書中のいずれかの要素の ID 属性の値と合致しなければならない．

ID/IDREF による参照を含んだ XML 文書を定める DTD[3] の例を図 11 に示す．これは人物リスト members と論文リスト papers からなるデータベース database で，各論文 paper 中に現れる著者 author は，各人物 person 中に割り振られた ID pid を IDREF pref から参照する形で指定している．図 11 の DTD に従うデータベースと，そこから生成したビューの例を図 12 に示す．

図 12 におけるソースのようなデータを表現する XML データベースとして，データの一貫性を保つためには，メンバーリスト中と各論文の著者リスト中に同一人物が複数回出現するようなものは望ましくないといえる．いずれかのデータを変更してしまうと整合性が崩れてしまうからである．このようなことを避け，同一のデータはデータベース中に一度しか現れないようにするために ID/IDREF による参照機能が用

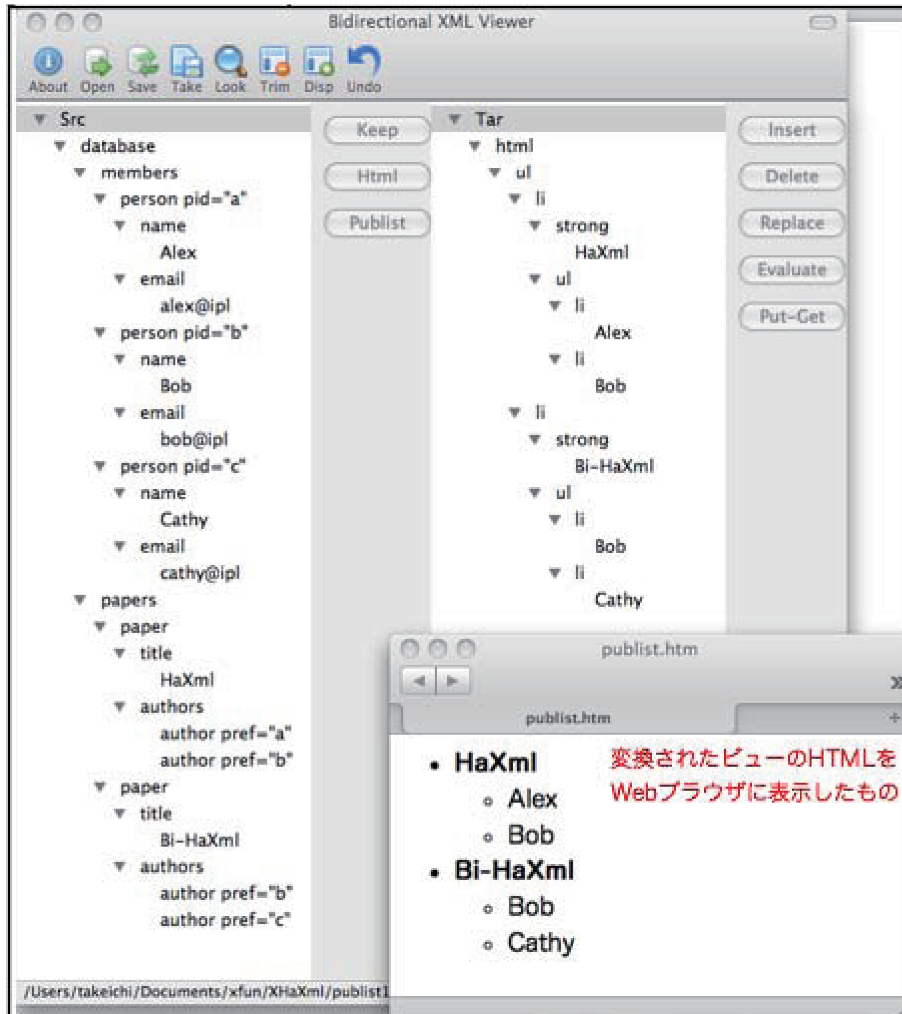

図 12　ID/IDREF による参照を含む XML 文書の変換

```
<!ELEMENT database (members, papers)>

<!ELEMENT members (person+)>
<!ELEMENT person (name, email)>
<!ATTLIST person pid ID #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>

<!ELEMENT papers (paper+)>
<!ELEMENT paper (title, authors)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT authors (author+)>
<!ELEMENT author EMPTY>
<!ATTLIST author pref IDREF #REQUIRED>
```

図 11　ID/IDREF を定める DTD の例

意されている．

このように ID/IDREF による参照を含むソースから図 12 のようなビューを作るためには，もとのソースに対して IDREF のタグのついた要素を対応する ID の要素で置き換える変換 `connectID ("pref","pid")` を施した上で，さらに変換フィルタを適用すればよい．`connectID` はすでに定義されているフィルタ `foldXml`，`iffind`, `keep`, `deep`, `attrval` を使って定義している．このフィルタは双方向変換のために Bi-HaXml で拡張したものである．

```
connectID :: (String, String) -> XFilter
connectID  (refkey, defkey) = XFun k where
 k s = f s
  where
  XFun f =
   foldXml (iffind refkey lookfor keep)
```

```
-- lookfor v lookups element with
-- defkey = v in s
lookfor v = XFun h
 where
 h c@(CElem (Elem n as cs)) =
  ([CElem (Elem n as (t++cs))], const c)
 (t, g') = g s -- g' is thrown away
 XFun g =
  deep
   (attrval (defkey, AttValue [Left v]))
```

しかし，この方法で得られたビュー上のすべての変更をソースに反映させることはできない．上の connectID の定義において，IDREF に対応する ID が実質的には *duplication* ではなく，*copy* になっているからである．ソース上の title から取り出した HaXml や Bi-HaXml は図 12 のビュー上で変更して，それを逆変換によってソースに反映させることはできるが，ソース上で IDREF で ID を参照している author は変更できない．一方で，ビュー上で author に由来する要素である Bob のうちの一つを変更すれば，対応するソース上の ID のついた name 要素が更新され，さらにそれに伴ってビュー上のもう一つの場所に現れているものも更新されるという処理が行われることを期待する向きもあるだろう．

このことに関しては議論のあることが予想される．前節に述べたように，双方向変換による編集過程に現れる依存関係については，本質的に *duplication* として明確に定義される．むしろ，それ以外に双方向変換の満たすべき性質に合致する妥当な定義を見つけることが難しいとさえいえる．しかし，双方向変換による編集における XML 文書の ID/IDREF の取扱いに関しては，それが *duplication* による相互に対称的な依存関係とは異なり，(複数の)IDREF 要素が単一の ID を “参照する” という方向のついた関係である点に注意する必要があるといえる．さらに，XML データベースの設計の背景も考慮する必要があるだろう．図 11 の DTD に示した XML データベースの設計にあたって，図 12 に示されるようなビューから，author 要素を通じてそれに対応する ID の name 要素を変更することを予定しているであろうか？たとえば，上にあげた例のように，ビュー上に現れている Bob の一方を変更したいが，他方はそのままにしたいという場合もあるであろう．これは，ソース上の IDREF 要素が適切ではなく，他の person を参照するような変更が期待される．双方向変換によるビュー更新において，その特長である複製保全性による更新の一貫性はあくまでも ID/IDREF の意味づけとは独立のものであるといえる．

双方向変換における *duplication* の依存関係は ID/IDREF による参照関係よりも一般的である．*duplication* によって依存関係のある要素間に双方に ID/IDREF で参照する (つまり、一つの要素に ID と IDREF をつける) ことによって文書を外部ファイルに保存して，*duplication* をシミュレートすることができる．こうすれば，ファイルを読み込んで，*duplication* による依存関係を復元することも可能である．このことから，双方向変換による更新を実現するフィルターを通じて ID/IDREF の意味づけを行うことができるものと考えられる．現在，Bi-HaXml に定義されている (したがって，HaXml に備わっている) フィルターにはそれに該当するものはないが，それを適切に定めることによって，ID/IDREF を含んだ XML 文書に対する双方向変換によるビュー更新の意味づけを行うことができるといえよう．

7 まとめ

本論文では，双方向変換処理の過程で *duplication* として生じる文書内の相互依存性と，XML 文書における ID/IDREF [3] による内部参照との関係を論じ，内部参照をもつ XML 文書の双方向変換による編集のあり方に対する問題提起を行った．また，ID/IDREF を含む XML 文書に対する双方向変換のあり方のひとつとして，Bi-HaXml のフィルタに追加した connectID による双方向変換を実装を通じて示した．このフィルタは既存のフィルタから構成しているとはいえ，それは HaXml の高階フィルタでは実現することのできない一般的な関数プログラムの構成法によっている．双方向変換を活用するための基本的なフィルタを組み合わせるための高階フィルタ (結合子) を追究することが今後の課題となろう．

謝辞 本研究に関して多くの有益なご指摘を下さった松崎公紀助教，江本健斗研究員，松田一孝研究員，森畑明昌研究員，熊英飛氏，宋輝氏，中野圭介電気通信大学特任助教に感謝いたします．本研究は文部科学省科学研究費補助金基盤研究 (A)19200002 「双方向変換機構とその応用に関する研究」による．

参 考 文 献

[1] Bancilhon, F. and Spyratos, N.: Update Semantics of Relational Views, *ACM Transactions on Database Systems*, Vol. 6, No. 4(1981), pp. 557–575.

[2] Bohannon, A., Foster, J. N., Pierce, B. C., Pilkiewicz, A., and Schmitt, A.: Boomerang: Resourceful Lenses for String Data, *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2008, pp. 407–419.

[3] Bray, T., Paoli, J., Sperberg-McQueen, C. M., and Maler, E.: *Extensible Markup Language (XML) 1.0 Specification (Fifth Edition)*, 2008, http://www.w3.org/TR/xml.

[4] Hu, Z., Mu, S.-C., and Takeichi, M.: A Programmable Editor for Developing Structured Documents based on Bidirectional Transformations, *Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, 2004, pp. 178–189.

[5] Liu, D., Hayashi, Y., Nakano, K., Hu, Z., and Takeichi, M.: Bidirectional XML Transformation with Bi-X, 全国大会講演論文集, 第 *70* 回平成 *20* 年 *(5)*, 社団法人情報処理学会, 2008, pp. 395–396.

[6] Matsuda, K., Hu, Z., Nakano, K., Hamana, M., and Takeichi, M.: Bidirectionalization Transformation based on Automatic Derivation of View Complement Functions, *ACM SIGPLAN International Conference on Functional programming (ICFP)*, 2007, pp. 47–58.

[7] Mu, S.-C., Hu, Z., and Takeichi, M.: An Algebraic Approach to Bidirectional Updating, *The Second Asian Symposium on Programming Language and Systems (APLAS)*, 2004, pp. 2–18.

[8] Nakano, K., Hu, Z., and Takeichi, M.: Consistent Web Site Updating based on Bidirectional Transformation, *10th IEEE International Symposium on Web Site Evolution (WSE 2008)*, 2008, pp. 45–54.

[9] Wallace, M., and Runciman, C.: Haskell and XML: Generic Combinators or Type-based Translation? *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 1999, pp. 148–159.

[10] 武市正人: 双方向変換による高信頼構造化文書処理 (特集 学と産の連携による基盤ソフトウェアの先進的開発), 情報処理, Vol. 49, No. 11(2008), pp. 1265–1270.