A Framework for Management of Concurrent XML Markup [*]

Alex Dekhtyar, Ionut E. Iacob

Department of Computer Science
University of Kentucky
Lexington, KY 40506
dekhtyar@cs.uky.edu; eiaco0@cs.uky.edu

Abstract. The problem of concurrent markup hierarchies in XML encodings of works of literature has attracted attention of a number of humanities researchers in recent years. The key problem with using concurrent hierarchies to encode documents is that markup in one hierarchy is not necessarily well-formed with respect to the markup in another hierarchy. The proposed solutions to this problem rely on the XML expertise of the editors and their ability to maintain correct DTDs for complex markup languages. In this paper, we approach the problem of maintenance of concurrent XML markup from the Computer Science perspective. We propose a framework that allows the editors to concentrate on the semantic aspects of the encoding, while leaving the burden of maintaining XML documents to the software. The paper describes the formal notion of the concurrent markup languages and the algorithms for automatic maintenance of XML documents with concurrent markup.

1 Introduction

The problem of concurrent markup hierarchies has recently attracted the attention of a number of humanities researchers [13, 6, 15]. This problem typically manifests itself when a researcher must encode in XML a large document (book, manuscript, printed edition) with a wide variety of features. A *concurrent hierarchy* is formed by a subset of the elements of the markup language used to encode the document. The elements within a hierarchy have a clear nested structure. When more than one such hierarchy is present in the markup language, the hierarchies are called concurrent.

A typical example of concurrent hierarchies is the XML markup used to encode the physical location of text in a printed edition: book, page, physical line, vs. the markup used to encode linguistic information about the text: sentence,

[*] This work has been supported in part by NSF ITR grant 0219924. In addition, the work of the second author has been supported in part by NEH grant RZ-20887-02. The manuscript image [1] appearing in this paper was digitized for the Electronic Boethius project by David French and Kevin Kiernan and is used with permission of the British Library Board.

phrase, word, letter. The key problem with using concurrent hierarchies to encode documents is that markup in one hierarchy is not necessarily well-formed with respect to the markup in another hierarchy.

The study of concurrent XML hierarchies for encoding documents is related to the problem of manipulation and integration of XML documents. However, most of the research on XML data integration addresses the problem of integrating heterogeneous, mostly data-centric XML provided by various applications ([4, 9, 10, 8]). In our case, the data to be integrated has a common denominator: the document content, and the XML encodings are document-centric. Also, the features of the document to be marked up are not (in most cases) heterogeneous, but they might be conflicting in some instances.

Management of concurrent markup has been approached in a few different ways. The Text Encoding Initiative (TEI) Guidelines [13] suggest a number of solutions based on the use of *milestone elements* (empty XML elements) or *fragmentation* of the XML encoding. Durusau and O'Donnell [6] propose a different approach. They construct an explicit DTD for each hierarchy present in the markup. Then they determine the "least common denominator" in the markup — the units of content inside which no overlap occurs, in their case, words. They associate attributes indicating the XPath expression leading to the content of each word element for each hierarchy. Other scholars have proposed the use of non-XML markup languages that allow concurrent hierarchies [7].

In their attempts to resolve the problem of concurrent hierarchies, both [13] and [6] rely on the human editor to (i) introduce the appropriate solution to the XML DTD/XSchema, and (ii) follow it in the process of manual encoding of the documents. At the same time, [6] emphasizes the lack of software support for the maintenance of the concurrent markup, which makes, for example, adhering to some of the TEI solutions a strenuous task. While some recent attempts have been made to deal with the problem of concurrent markup from a computer science perspective [14, 15], a comprehensive solution has yet to be proposed.

This paper attempts to bridge the gap between the apparent necessity for concurrent markup and the lack of software support for it by proposing a framework for the creation, maintenance and querying the concurrent XML markup. This framework relies on the following:

- Separate DTDs for hierarchies;
- Use of a variant of fragmentation with virtual join suggested by TEI Guidelines [13] to represent full markup;
- Automatic maintenance of markup;
- Use of a database as XML repository.

The ultimate goal of the proposed framework is to free the human editor from the effort of dealing with the validity and well-formedness issues of document encoding and to allow him or her to concentrate on the meaning of the encoding. This goal is achieved in the following way. Durusau and O'Donnell [6] note the simplicity and clarity of DTDs for individual concurrent hierarchies, as opposed to a unified DTD that incorporates all markup elements. Our approach allows the editor to describe a collection of such simple DTDs without having to worry about the need to build and maintain a "master" DTD. At the same time, existence of concurrent DTDs introduces the need for specialized software to support the editorial process drive it by the semantics of the markup. This software must allow the editor to indicate the positions in the text where the markup is to be inserted, select the desired markup, and take record the results.

In this paper we introduce the foundation for such software support. In Section 2 we present a motivating example based on our current project. Section 3 formally defines the notion of a collection of concurrent markup languages. In Section 4 we present three key algorithms for the manipulation of concurrent XML markup. The Merge algorithm builds a single *master* XML document from several XML encodings of the same text in concurrent markup. The Filter algorithm outputs an XML encoding of the text for an individual markup hierarchy, given the master XML document. The Update algorithm incrementally updates the master XML document given an atomic change in the markup.

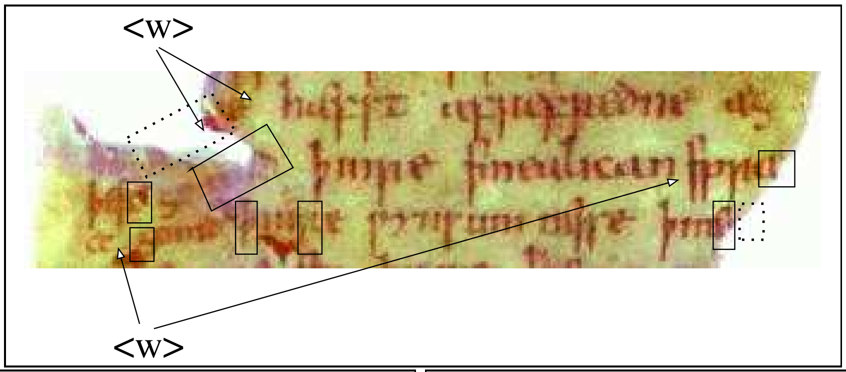
This paper describes the work in progress. A major issue not addressed in here is the database support for multiple concurrent hierarchies in our framework. This problem is the subject of ongoing research.

2 Motivating Example

Over the past few years researchers in the humanities have used XML extensively to create readable and searchable electronic editions of a wide variety of literary works [11, 12, 6]. The work described in this paper originated as an attempt to deal with the problem of concurrent markup in one such endeavor, The ARCHWay Project, a collaborative effort between Humanities scholars and Computer Scientists at the University of Kentucky. This project is designed to produce electronic editions of Old English manuscripts. In this section, we illustrate how concurrent markup occurs in ARCHWay.

Building electronic editions of manuscripts. Electronic editions of Old English manuscripts [11, 12]combine the text from a manuscript (both the transcript and the emerging edition), encoded in XML using an expressive array of features (XML elements), and a collection of images of the surviving folios of the manuscript. The physical location of text on the surviving folios, linguistic information, condition of the manuscript, visibility of individual characters, paleographic information, and editorial emendations are just some of the features that need to be encoded to produce a comprehensive description of the manuscript. Specific XML elements are associated with each feature of the manuscript.

Concurrent hierarchies and conflicts. Most of the features have explicit *scopes*: the textual content (of the manuscript) that the feature relates to, be it the text of a physical line, or a line of verse or prose, or manuscript text that is missing due to a damage in the folio. Unfortunately, the scopes of different features often overlap, resulting in non-well-formed encoding (we call such a situation a *conflict*).



```
hu þu me hæfst afrefredne ægþer
ge mid þinre smealican spræce,
ge mid þinre wynsum nesse þines
(0)
```

```
<line no="22">hu þu me hæfst
        afrefredne æg</line>
<line no="23">þer ge mid
        þinre smealican spræ</line>
<line no="24">ce, ge mid þinre
        wynsumnesse þines </line>
(i)
```

```
<w>hu</w> <w>þu</w> <w>me</w>
<w>hæfst</w> <w>afrefredne</w>
<w>ægþer</w> <w>ge</w> <w>mid</w>
<w>þinre</w> <w>smealican</w>
<w>spræce</w>, <w>ge</w> <w>mid</w>
<w>þinre</w> <w>wynsum nesse</w>
<w>þines</w>
(ii)
```

```
hu <res><restxt>þu m</restxt></res>e hæfst
afrefredne ægþ<dmg>er</dmg> ge <dmg>mid
</dmg> þinre smealican spr<dmg>æ</dmg>ce,
<dmg>g</dmg>e mid
<dmg>þ</dmg>in<dmg>r</dmg>e wynsum
nesse þin<dmg>e</dmg><res><restxt>s
</restxt></res>
(iii)
```

Fig. 1. A fragment of King Alfred's Boethius manuscript [1] and different XML encodings.

Consider a fragment of *folio 38 verso* of British Library Cotton Otho A vi [1] (King Alfred's Boethius manuscript) shown in **Fig.1**. The text of the three lines depicted on this fragment is shown in the box marked (0) in **Fig.1**. The remaining boxes in **Fig.1** show the following markup for this fragment: (i) information about physical breakdown of the text into lines (`<line>` element); (ii) information about the structure of the text (`<w>` element encodes words), (iii) information about the damage and text obscured by the damage (`<dmg>` and `<rstxt>` tags)[1].

Some of the encodings of this fragment are in conflict. The solid boxes over parts of the image indicate the scope of the `<dmg>` elements and the dotted boxes indicate the scope of the `<rstxt>` elements. In addition, we indicate the positions of some of the `<w>` tags. Damage and restoration markup overlaps words in some places: the damaged text includes the end of one word and the beginning of the next word. In addition to that, some words start on one physical line and continue on another.

[1] The encodings are simplified. We have removed some attribute values from the markup to highlight the structure of each encoding.

Resolving markup conflicts. The TEI Guidelines [13] suggest a number of possible ways to resolve conflicts. These methods revolve around the use of empty milestone tags and the fragmentation of markup. We illustrate the proposed suggestions in **Fig.2** on the example of the markup conflict between the <`w`> and <`line`> elements at the end of line 22. The first suggested way (**Fig.2**.(a)) uses *milestone (empty) elements*. In this case the editor determines the pairs of tags that may be in conflict, and for each such pair declares at least one tag as empty in the DTD/XSchema. The other two ways (**Fig.2**.(b),(c)) are variants of the *fragmentatation* technique: one of the conflicting elements is split into two parts by the other one (in **Fig.2** we choose to split <`w`> element). Simple fragmentation, however, may be confusing: encoding in **Fig.2**.(b) creates the impression that "æg" and "þer" are two separate words. To alleviate this problem, a variety of conventions based on the use of attributes can be proposed to indicate that a specific element encodes a *fragment*. **Fig.2**.(c) shows one such convention that uses a "glue" attribute `Id`. This implied attribute will get the same value for all fragments of the same encoding.

Key drawback. The answer lies not only in alleviating the markup conflict problem: a more general problem of *maintenance* of markup in situations where conflicts are a frequent occurrence must be addressed. Up to this point, such maintenance resided in the hands of human editors who were responsible for specific encoding decisions to prevent markup conflicts. This tended to generate a variety of gimmick solutions in the markup language, such as introduction of tags whose sole purpose was to overcome a specific type of conflict, but which, in the process made the DTD/XSchema of the markup language complex and hard to maintain. Our approach, described in the remainder of this paper allows the software to take over the tasks of markup maintenance, simplifying the work of editors.

```
<line no="22"/> <w>hu</w> <w>þu</w> <w>me</w> <w>hæfst</w>
<w>afrefredne</w>
<w>æg<line no="23"/>þer</w> <w>ge</w> <w>mid</w>
```
(a) Milestone elements.

```
<line no="22"> ..... <w>æg </w> </line>
<line no="23"><w>þer</w> ..... </line>
```
(b) Fragmentation.

```
<line no="22"> ..... <w id="1">æg </w></line>
<line no="23"><w id ="1">þer</w> ..... </line>
```
(c) Fragmentation with virtual join (variant with "glue" attribute).

Fig. 2. Resolving markup conflicts.

3 Concurrent XML Hierarchies

In this section we formally define the notion of the *collection of concurrent markup hierarchies.* Given a DTD D, we let $elements(D)$ denote the set of all markup elements defined in D. Similarly, we let $elements(d)$, where d is an XML document, denote the set of all element tags contained in document d.

Definition 1. *A* concurrent markup hierarchy CMH *is a tuple* $CMH =< S, r, \{D_1, D_2, ..., D_k\} >$ *where:*
• S *is a string representing the document content;*
• r *is an XML element called the* root *of the hierarchy;*
• D_i, $i = \overline{1,k}$ *are DTDs such that:*
(i) r *is defined in each* D_i, $1 \leq i \leq k$, *and* $\forall 1 \leq i, j \leq k$, $i \neq j$
$elements(D_i) \bigcap elements(D_j) = \{r\}$;
(ii) $\forall 1 \leq i \leq k$, $\forall t \in elements(D_i)$ r *is an ancestor of* t *in* D_i.

In other words, the collection of concurrent markup hierarchies is composed of textual content and a set of DTDs sharing the same root element and no other elements.

Definition 2. *Let* $CMH =< S, r, \{D_1, D_2, ..., D_k\} >$ *be a concurrent markup hierarchy. A* distributed XML document dd *over* CMH *is a collection of XML documents:* $dd =< d_1, d_2, ..., d_k >$ *where* $(\forall 1 \leq i \leq k)$ d_i *is valid w.r.t.* D_i *and* $content(d_1) = content(d_2) = ... = content(d_k) = S$[2].

The notion of a distributed XML document allows us to separate conflicting markup into separate documents. However, dd is not an XML document itself, rather it is a *virtual* union of the markup contained in $d_1,\ldots,d_k$. Our goal now is to define XML documents that incorporate in their markup exactly the information contained in a distributed XML document. We start by defining a notion of a *path* to a specific character in content.

Definition 3. *Let* d *be an XML document and let* $content(d) = S$. *Let* $S = c_1c_2\ldots c_M$. *The* path to ith character in d *denoted* $path(d,i)$ *or* $path(d,c_i)$ *is the* sequence *of XML elements forming the path from the root of the DOM tree of* d *to the content element that contains* c_i.

Let D *be a DTD and let* $elements(D) \cap elements(d) \neq \emptyset$, *and let the root of* d *be a root element in* D. *Then, the* path to ith character in d w.r.t. D, *denoted* $path(d,i,D)$ *or* $path(d,c_i,D)$ *is the subsequence of all elements of* $path(d,i)$ *that belong to* D.

Following XPath notation, we will write $path(d,i)$ and $path(d,i,D)$ in a form $a1/a2/\ldots/a_s$. We notice that $path(d,i,D)$ defines the projection of the path to ith character in d onto a specific DTD. For example, if $path(d,i) = col/fol/pline/line/w/dmg$ and D contains only elements <col>, <pline> and <w>, then $path(d,i,D) = col/pline/w$. We can now use paths to content characters to define "correct" single-document representations of the distributed XML documents.

[2] $content(doc)$ denotes the text content of the XML document doc.

Definition 4. *Let d^* be an XML document and let D be a DTD, such that $elements(d^*) \cap elements(D) \neq \emptyset$ and the root of d^* is a root element in D. Then, the set of filters of d^* onto D, denoted $Filters(d^*, D)$ is defined as follows:*

$$\begin{aligned} Filters(d^*, D) = \{d | & content(d) = content(d^*), \\ & elements(d) = elements(d^*) \cap elements(D) \\ & \text{and } (\forall 1 \leq i \leq |content(d)|) path(d^*, i, D) = path(d, i)\} \end{aligned}$$

Basically, a filter of d^* on D is any document that contains only elements from D that preserves the paths to each content character w.r.t. D. If we are to combine the encodings of all d_is of a distributed document dd in a single document d^* we must make sure that we can "extract" every individual document d_i from d^*.

Definition 5. *Let $dd =< d_1, d_2, \ldots d_k >$ be a distributed XML document over the collection of markup hierarchies $CMH =< S, r, \{D_1, \ldots, D_k\} >$. A set of* mergers *of dd denoted $Mergers(dd)$ is defined as*

$$\begin{aligned} Mergers(dd) = \{d^* | & elements(d^*) \subseteq \bigcup_{j=1}^{k} elements(D_j) \\ & \text{and } (\forall 1 \leq i \leq k) d_i \in Filters(d^*, D_i)\} \end{aligned}$$

Given a distributed XML document dd, we can represent its encoding by constructing a single XML document d^* from the set $Mergers(dd)$. d^* incorporates the markup from all documents $d_1, \ldots, d_k$ in a way that (theoretically) allows the restoration of each individual document from d^*. A document $d^* \in Mergers(dd)$ is called a *minimal merger* of dd iff for each content character c_j, $path(d^*, c_j)$ consists exactly of the elements from all $path(d_i, c_j)$, $1 \leq i \leq k$.

4 Algorithms

Section 3 specifies the properties that the "right" representations of distributed XML documents (i.e., XML markup in concurrent hierarchies within a single XML document) must have. In this section we provide the algorithms for building such XML documents. In particular, we address the following three problems:

- Merge: given a distributed XML document dd, construct a minimal merger d^* of dd. We will refer to the document constructed by our Merge algorithm as the *master XML document* for dd.
- Filter: given a master XML document for some distributed document dd and one of the concurrent hierarchies D_i, construct the document d_i.
- Update: given a distributed XML document dd, its master XML document d^* and a simple update of the component d_i of dd, that changes it to d'_i, construct (incrementally) the master XML document d' for the distributed document $dd' =< d_1, \ldots, d'_i, \ldots, d_k >$.

Fig.3 illustrates the tasks addressed in this section and the relationship between them and the encoding work of editors. In the proposed framework, the editors are responsible for defining the set $\{D_1, \ldots, D_k\}$ of the concurrent hierarchies and for specifying the markup for each component of the distributed document *dd*. The MERGE algorithm then *automatically* constructs a single master XML document d^*, which represents the information encoded in all components of *dd*. The master XML document can then be used for archival or transfer purposes. When an editor wants to obtain an XML encoding of the content in a specific hierarchy, the Filter algorithm is used to extract the encoding from the master XML document. Finally, we note that MERGE is a global algorithm that builds the master XML document from scratch. If a master XML document has already been constructed, the Update algorithm can be used while the editorial process continues to update incrementally the master XML document given a simple (atomic) change in one of the components of the distributed XML document. Each algorithm is discussed in more detail below. Note that the theorems in this section are given without proofs. The proofs can be found in [5].

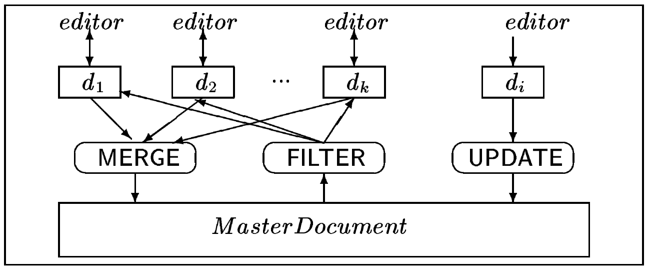


Fig. 3. The framework solution.

4.1 MERGE Algorithm

The MERGE algorithm takes as input *tokenized* versions of the component documents $d_1, \ldots, d_k$ of the distributed document *dd* and produces as output a single XML document that incorporates all the markup of $d_1, \ldots, d_k$. The algorithm resolves the overlap conflicts using the *fragmentation with a "glue" attribute* approach described in Section 2. A special attribute `link` is added to all markup elements that are being split, and the value of this attribute is kept the same for all markup fragments.

The algorithm uses the Simple API for XML (SAX)[3] for generating tokens. SAX callbacks return three different types of token strings: (i) start tag token string (ST), (ii) content token string (CT), (iii) end tag token string (ET). If `token` is the token returned by the SAX parser, then we use `type(token)` to denote its type (ST, CT, ET) as described above and `tag(token)` to denote the tag returned by SAX (for ST and ET tokens).

The MERGE algorithm works in two passes. On the first pass, the input documents are parsed *in parallel* and an ordered list is built of ST and ET tokens for the creation of the master XML document. The second pass of the algorithm scans the token list data structure built during the first pass and outputs the text of the master XML document.

The main data structure in the MERGE algorithm is *tokenListSet*, which is designed to store all necessary markup information for the master XML document. Generally speaking, *tokenListSet* is an array of token lists. Each array position corresponds to a position in the content string of the input XML documents. In reality, only the positions at which at least one input document has ST or ET tokens have to be instantiated. For each position i, $tokenListSet[i]$ denotes the ordered list of markup entries at this position. At the end of the first pass of the MERGE algorithm, for each i, $tokenListSet[i]$ will contain the markup elements to be inserted in front of ith character of the content string in the master XML document *exactly in the order they are to be inserted.* The second pass of the MERGE algorithm is a straightforward traversal of *tokenListSet*, which for each position outputs all the tokens and then the content character.

Fig.4 contains the pseudocode for the MERGE algorithm. The algorithm iterates through the positions in the content string of the input documents. For each position i, the algorithm first collects all ET and ST tokens found at this position. It then determines the correct order in which the tokens must be inserted in the master XML document, and resolves any overlaps by inserting appropriate end tag and start tag tokens at position i and adding the `link` attribute to the start tag tokens. In the algorithm `push(Token,List)` and `append(Token,List)` add `Token` at the beginning and at the end of `List` respectively.

Theorem 1. *Let* $dd =< d_1, \ldots, d_k >$ *be a distributed XML document. Let* d^* *be the output of* MERGE$(d_1, \ldots, d_k)$. *Then* d^* *is a* minimal merger *of* dd.

4.2 FILTER Algorithm

The FILTER algorithm takes as input an XML document d^* produced by the MERGE algorithm and a DTD D, filters out all markup elements in d^* that are not in D and merges the fragmented markup.

In one pass the algorithm analysis the ordered sequence of tokens provided by a SAX parser and performs the following operations:
- removes all ST and ET tokens of markup elements not in D;
- from a sequence ST, [CT], ET, [CT], ..., ST, [CT], ET of tokens for a fragmented element in D, removes the "glue" attributes and outputs the first ST token, all possible intermediate CT tokens and the last ET token in the sequence;
- all other tokens are output without change in the same order they are received from the SAX parser.

The pseudo-code for FILTER appears in **Fig.5**. The following theorem states that FILTER correctly reverses the work of the MERGE algorithm.

```
Algorithm MERGE($d_1, \ldots, d_k$)
  //PASS I
  initialize $tokenListSet$
  for $cpos$ = 1 to sizeof(content($d_1$))
    //Determine the correct nesting
    //of all tags that end at current position
    move all end tag tokens
    in $tokenListSet[cpos]$ to $EndTokenList$
    Build list of tokens from
    $d_1, \ldots, d_k$ at position $i$
    collect $tokenListSet[i]$ from $d_1, \ldots, d_k$
    //Find correct order of tokens,
    //resolve overlapping conflicts
    $pos = cupos$-1
    while Not Empty($EndTokenList$)
    for (each unmarked start tag in
        $tokenListSet[pos]$)
      if (start tag is in $EndTokenList$)
        push(end tag, $tokenListSet[cpos]$)
        delete(end tag, $EndTokenList$)
        mark(start tag)
      else
        add "glue attribute"= currentID
            to start tag entry
        push(matching end tag, $tokenListSet[cpos]$)
        append(start tag, $tokenListSet[cpos]$)
        mark(start tag)
      $pos = pos$ -1
```

```
// PASS II
$marker = 0$
for (each position entry $pos$
    in $tokenListSet$)
  output content in $contBuffer$
    from $marker$ to $pos$
  $marker = pos$
  output tokens at position $pos$
    in $tokenListSet$
```

Fig. 4. The MERGE algorithm

Theorem 2. *Let* $dd =< d_1, \ldots, d_k >$ *be a distributed XML document, and* d^* *be the output of* MERGE(dd). *Then* $(\forall 1 \leq i \leq k)$, FILTER$(d^*, D_i) = d_i$.

4.3 UPDATE Algorithm

The UPDATE algorithm updates the master XML document (see **Fig.3**) with the new markup element. It takes as the input two integers, $from$ and to, the starting and ending positions for the markup in the content string and the new markup element, TAG. Due to possible need to fragment the new markup this process requires some care. The goal of the algorithm is to introduce the new markup into the master XML document in a way that minimizes the number of new fragments. The algorithm uses the DOM model [2] for the XML document and performs the insertion of the node in the XML document tree model. In this model, for an element with mixed content, the text is always a leaf. Then $from$ and to will be positions in some leaves of the document tree. Let $FROM$ and TO be the parent nodes of the text leaves containing positions $from$ and to respectively. We denote by LCA the lowest common ancestor of nodes $FROM$ and TO. Let $AFROM$ be child of LCA that is the ancestor of $FROM$, and let ATO be the child of LCA that is the ancestor of TO (see **Fig.6**).

The UPDATE algorithm traverses the path $FROM \rightarrow \ldots \rightarrow AFROM \rightarrow LCA \rightarrow ATO \rightarrow \ldots \rightarrow TO$ and inserts TAG nodes with glue attributes as

```
Algorithm FILTER($d$, $D$)
  $glueTagSet$ = <empty>
  start parsing document $d$
  while (more tokens)
    $token$ = nextToken()
    if ($token$ is CT)
      output $token$
      continue
    else if ($token$ is ST)
      if (tag($token$) $\in$ $glueTagSet$ OR
          tag($token$) $\notin$ $D$)
        continue
      if (tag($token$) has glue attributes)
        remove glue attributes
        put $token$ in $glueTagSet$
      output $token$
    else if ($token$ is ET)
      if (tag($token$) $\notin$ $D$)
        continue
      if (tag($token$) $\in$ $glueTagSet$ AND
          not last token for tag($token$))
        continue
      output $token$
```

```
Algorithm UPDATE($from$, $to$, $TAG$)
  find $LCA$
  //start inserting nodes
  for (each NODE in the path
      from FROM to AFROM)
    insert a node TAG with glue attributes
        as a parent for all siblings of NODE,
        at the right of NODE
  insert a node TAG with glue attributes
    as a parent of all nodes between
        AFROM and ATO
  for (each NODE in the path
      from ATO to TO)
    insert a node TAG with glue attributes
        as a parent for all siblings of NODE,
        at the left of NODE
```

Fig. 5. The FILTER and UPDATE algorithms

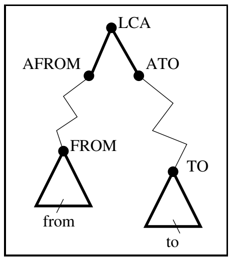


Fig. 6. The XML document tree model used in UPDATE algorithm

needed. The pseudo-code description of the algorithm is shown in **Fig.5**. The following theorem says that the result of UPDATE allows for correct recovery of components of the distributed document.

Theorem 3. *Let $dd =< d_1, \ldots, d_k >$ be a distributed XML document and d^* be the output of* MERGE(dd). *Let $TAG \in elements(D_i)$, $(from, to, TAG)$ be an update request and d'_i be a well-formed result of marking up the content between $from$ and to positions. Then,* $\mathsf{FILTER}(\mathsf{UPDATE}(d^*, (from, to, TAG)), D_i) = d'_i$.

5 Future Work

This paper introduces the general framework for managing concurrent XML markup hierarchies. There are three directions in which we are continuing this research. First, we are working on providing the database support for the maintenance of concurrent hierarchies. Second, we are studying the properties of the proposed algorithms w.r.t. the size of the markup generated, optimality of the markup and computational complexity, and efficient implementation of the algorithms. Finally, we are planning a comprehensive comparison study of a variety of methods for support of concurrent hierarchies.

References

1. British Library MS Cotton Otho A. vi, fol. 38v.
2. Document Object Model (DOM) Level 2 Core Specification. http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/, Nov 2000. W3C Recommendation.
3. Simple API for XML (SAX) 2.0.1. http://www.saxproject.org, Jan 2002. SourceForge project.
4. Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Incremental maintenance for materialized views over semistructured data. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 38–49, 24–27 1998.
5. Alex Dekhtyar and Ionut E. Iacob. A framework for management of concurrent XML markup. Technical Report TR 374-03, University of Kentucky, Department of Computer Science, June 2003. http://www.cs.uky.edu/~dekhtyar/publications/TR374-03.concurrent.ps.
6. P. Durusau and M. B. O'Donnell. Concurrent Markup for XML Documents. In *Proc. XML Europe*, May 2002.
7. C. Huitfeldt and C. M. Sperberg-McQueen. TexMECS: An experimental markup meta-language for complex documents. http://www.hit.uib.no/claus/mlcd/papers/texmecs.html, February 2001.
8. Ioana Manolescu, Daniela Florescu, and Donald Kossmann Kossmann. Answering XML queries over heterogeneous data sources. pages 241–250.
9. Wolfgang May. Integration of XML data in XPathLog. In *DIWeb*, pages 2–16, 2001.
10. Wolfgang May. Lopix: A system for XML data integration and manipulation. In *The VLDB Journal*, pages 707–708, 2001.
11. W.B. Seales, J. Griffioen, K. Kiernan, C. J. Yuan, and L. Cantara. The Digital Atheneum: New Technologies for Restoring and Preserving Old Documents. *Computers in Libraries*, 20(2):26–30, February 2000.
12. E. Solopova. Encoding a transcript of the beowulf manuscript in sgml. In *Proc. ACH/ALCC*, 1999.
13. C. M. Sperberg-McQueen and L. Burnard(Eds.). Guidelines for Text Encoding and Interchange (P4). http://www.tei-c.org/P4X/index.html, 2001. The TEI Consortium.
14. C. M. Sperberg-McQueen and C. Huitfeldt. GODDAG: A Data Structure for Overlapping Hierarchies, Sept. 2000. Early draft presented at the ACH-ALLC Conference in Charlottesville, June 1999.
15. A. Witt. Meaning and interpretation of concurrent markup. In *Proc., Joint Conference of the ALLC and ACH*, pages 145–147, 2002.