A Framework for Management of Concurrent XML Markup ⋆

Alex Dekhtyar *, Ionut E. Iacob

Department of Computer Science, University of Kentucky, Lexington, KY 40506, USA

Abstract

The problem of concurrent markup hierarchies in XML encodings of documents 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. Previously 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.

Key words: XML, Concurrent Markup

1 Introduction

The problem of concurrent markup hierarchies has been known to the text encoding community and the humanities scholars since the days of SGML [1]. Recently, with the switch of Text Encoding Initiative (TEI) Guidelines [2] from SGML to XML, there has been a re-emergence of interest to this problem in the context of XML encodings as evidenced by [2–5]. This problem typically

⋆ A preliminary and partial version of this paper appears in Proceedings of XSDM, Chicago, 2003.

* Corresponding author. Tel: +1-859-257-3062; Fax: +1-859-323-3740
Email address: dekhtyar@cs.uky.edu (Alex Dekhtyar).

manifests itself when a researcher must encode in XML a wide variety of features for a large document (e.g., a book or a manuscript). Some of the features may form so called *concurrent hierarchies*. A 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, 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 ([6–9]). The problem of management of concurrent XML hierarchies differs from XML data integration. In particular, in contrast to typical XML data integration tasks, concurrent XML ranges over exactly the same content, the XML encoding is, for the most part, document-centric, and, most important, markup from different hierarchies may be in conflict.

Management of concurrent markup has been approached in a few different ways. SGML resolved this problem using CONCUR element [2], although its implementation was not widespread among SGML processors. For XML, the Text Encoding Initiative (TEI) Guidelines [2] 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 [3,5] propose some radically different approaches. In [3], an approach dubbed Bottom-Up Vertical Hierarchies (BUVH), they construct an explicit DTD for each hierarchy present in the markup and then determine the "least common denominator" in the markup — the units of content inside which no overlap occurs. Such "least common denominator" can an be individual character or, in their example, a single word. They associate attributes indicating the XPath expression leading to the content of each word element for each hierarchy.

In [5], Durusau and O'Donnell propose a different solution, named Just-in-Time Trees (JITTs). A Just-in-Time Tree of a concurrent XML hierarchy is a document that contains all the markup elements from all hierarchies. JITTs need not be well-formed, in fact, if overlapping markup exists in the XML encoding, its corresponding Just-in-Time Tree will be non-well-formed. While JITTs cannot be directly processed by regular XML parsers, Durusau and

O'Donnell propose a "lazy evaluation" approach: create well-formed XML on the fly from a JITT, given a user query. In addition to Durusau and O'Donnell, other scholars have proposed the use of non-XML markup languages, such as TexMECS that allow concurrent hierarchies [10].

There have been some recent attempts made to deal with the problem of concurrent markup from a computer science perspective [11,4]. In particular, in [11] Sperberg-McQueen and Huitfield proposed GODDAG, an analog of a DOM tree for the concurrent hierarchies. This work goes a long way to establishing the semantics of concurrent hierarchies, but does not address directly the issues of management of concurrent markup.

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. We base our work on the following considerations.

(a) *We want all markup used in the framework to be well-formed XML.* This will give us the ability to transport XML documents from one computer to another, and rely on standard XML parsers for processing.

(b) *We want markup generation and maintenance to become the responsibility of the software.* That is, the user will supply information about the document markup on element-by-element basis, but the concurrent XML management software will provide a level of abstraction that would hide specific details of XML storage and XML generation from the user. At the same time, the user will be provided with the facilities to obtain a wide array of XML documents based on the introduced markup.

(c) *We want to simplify the process of creation of concurrent XML markup for the end user.* To that extent, we will assume that each concurrent hierarchy is represented in the framework by a single DTD (or XSchema). It will be the user's responsibility to create and maintain the DTD/XSchema collection.

(d) *The "lazy evaluation" approach to XML processing suggested by Durusau and O'Donnell makes introduction of markup into the XML document and XML generation two independent operations.* To facilitate this, the information about the XML markup introduced by the user can be stored in some intermediate storage. When the user requests creation of an XML document, that storage is processed and the appropriate XML is generated.

(e) *Separating XML generation from the editorial process allows us to use different "drivers" for generation of XML.* In this paper, we concentrate on using a variant of fragmentation with virtual join suggested by the TEI Guidelines [2] to represent full XML markup in a single XML document. However, other solutions can be incorporated in a similar manner.

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. In the proposed approach, the editor describes a collection of simple DTDs, one for each hierarchy, without having to worry about the need to build and maintain a "master" DTD. Existence of such "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 the ARCHway [12] project, in which the authors are currently involved. 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. Finally, in Section 5 we study the performance of our implementation of the MERGE algorithm.

This paper addresses the problem of formalizing the notion of concurrent XML and the algorithms associated with converting XML documents. A parallel issue of storage and querying of concurrent XML is the subject of ongoing research and is being addressed in separate work. In Section 6 we briefly describe our approach.

2 Conflicting Markup

In recent years researchers in the humanities have used SGML, and later, XML extensively to create readable and searchable electronic editions of a wide variety of literary works [13–17,5]. 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. Image-based electronic editions of Old English manuscripts [15,14,13,12] 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. Some of the manuscripts did not survive in their original form: folios are missing, and existing folios are at times badly damaged by fire, wear and tear and previous restoration attempts. Occasionally, severely damaged folios or pieces of folios are misbound. 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.

In this paper, we use the word *editor* to refer to the person who prepares the XML encoding of a document, such as an electronic edition. The text content of the document is generally stable: few changes in the text (content of the XML) are introduced. The editor proceeds as follows: (s)he analyzes the folio images, notes a specific feature to be included in the XML encoding, finds the affected text in the manuscript transcript and introduces the markup. To differentiate human editors and the software they use to construct such encodings, we call the latter a *tagger tool* [1].

Concurrent hierarchies and conflicts. Most of the features have explicit *scopes*: the textual content (of the manuscript) that the feature relates to. Unfortunately, the scopes of different features often overlap, resulting in *markup conflict* causing non-well-formed encoding. More specifically, we use the term *markup conflict* (or *conflict* for short) to denote a situation when (at least) two different markup elements with overlapping scopes are required by the logic of the encoding. Consider, for example, a fragment of *folio 38 verso* of British Library Cotton Otho A vi [18] (King Alfred's Boethius manuscript) shown in Figure 1. The text of the three lines depicted on this fragment is shown in the box marked (0) in Figure Figure 1. The remaining boxes in Figure 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) [2].

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

[1] Most of the XML editing software available at this point are data-centric editors that construct the skeleton of the XML document first and fill in the content afterwards. This is in contrast to the tagger tools described here that first load text files and then allow editors to introduce markup by highlighting parts of the text and selecting the XML elements to tag them with.

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

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 [18] and different XML encodings.

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.

Resolving markup conflicts. The TEI Guidelines [2] 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 Figure 2 on the example of the markup conflict between the <w> and <line> elements at the end of line 22. The first suggested way (Figure 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. Figure 2.(b), shows the *fragmentation* technique: one of the conflicting elements is split into two parts by the other one (in Figure 2 we choose to split <w> element). A special "glue" attribute is used to indicate that "æg" and "þer" are parts of the same word.

Key drawback. It is not enough to simply alleviate the markup conflict problem in the final XML document: 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. As an example, consider the fragment of the DTD used in the encoding of the manuscripts at the University of Kentucky prior to adoption of concurrent markup shown in Figure 3. We note a few of the peculiarities of this design. First, the tag `<vline>` for marking up lines of verse is defined as a milestone element. This is because verse lines often cross boundaries of physical lines on the manuscript folios encoded with the `<line>` tag. Second, we notice three versions of a tag for encoding words: `<w>`, `<w1>` and `<w2>`. Words in the manuscripts can cross physical line borders and folio borders. Tag `<w>` was used to encode words that are not in conflict with the physical encoding of the manuscript (`<line>` and `<folio>` tags). Tags `<w1>` and `<w2>` were used when the word crossed one of the boundaries above: `<w1>` encoded the first part of the word while `<w2>` encoded the second. In addition, the DTD itself was quite unintuitive: the XML elements could be nested almost arbitrarily inside each other.

This generated problems of two sorts. First, large and complicated DTD of the markup was very hard to maintain and train on. Very often DTDs used by scholars to encode documents are not static: elements are added to them from time to time, and the roles of certain other elements are revised, causing changes in the DTD. Introducing such changes in a correct way into the DTD whose fragment is shown in Figure 3 (the full DTD consisted of close to 100 markup elements) has been a strenuous task: the researcher had to decide for each DTD production whether or not to include the new tag into its right-hand side, and whether such inclusion entailed any more structural changes. As a reuslt, the entire process was extremely error-prone. Second, search queries on the encoding created using milestone elements and split tags such as `<w1>` and `<w2>` had to be programmed in a specialized fashion that had to ensure that, for example, when looking for the "content" of `<vline>` elements, one has to search from milestone to milestone (a traversal of part of the DOM tree of the resulting XML encoding), or, when looking for words, one has to incorporate search for `<w1>`-`<w2>` pair.

A better way. Our approach, described in the remainder of this paper allows editors to resolve concurrent markup in a more elegant way. First, the editors will be tasked with preparation and maintenance of a larger number of DTDs. Each DTD, however, will be much smaller than the DTD discussed above, and will be much more readable. It will also be devoid of gimmicks used to resolve specific conflicts. Second, it allows us to entrust software with the problems of markup storage, maintenance and XML generation. In the next section, we formalize the concept of concurrent markup hierarchies and study some of their properties.

<**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 link="1"**>æg </**w**></**line**>
<**line no="23"**><**w link ="1"**>þer</**w**> ..... </**line**>

(b) 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 XML elements defined in D. Similarly, if d is an XML document, $elements(d)$, denotes the set of all XML element tags contained in document d. Given a DTD D, we say that an element $y \in elements(D)$ is an *ancestor in D* of another element $x \in elements(D)$ if there is a well-formed XML document instance d valid w.r.t. D, such that it contains a node y that is an ancestor node of x in the DOM tree of d (note, that y being an ancestor of x is D does not preclude x from being an ancestor of y).

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, $1 \le i \le k$, *and* $\forall 1 \le i, j \le k$, $i \ne j$

$elements(D_i) \bigcap elements(D_j) = \{r\}$*;*

(ii) $\forall 1 \le i \le k$, $\forall t \in elements(D_i)$ r *is an ancestor of* t *in* D_i.

In other words, a concurrent markup hierarchy (CMH) is composed of textual content and a set of DTDs sharing the same root element and no other elements.

Figure 4 shows an example of a concurrent markup hierarchy. Here, the content is taken from lines 22–24 of [18], and the DTD fragments (showing only the elements used in further examples) are part of the concurrent markup hierarchy built for the ARCHway project from the original DTD (see Figure 3).

```
<!ELEMENT line (#PCDATA|a|abb|accent|add|alt|comment|cvd|del|
                div|dmg|ednote|edsp|emd|enh|exp|fdd|gap|
                gloss|hl|indent|lang|lend|let|oecno|omiss|
                overers|paleog|palimbot|palimtop|pcvd|pline|
                rejoin|res|rest|resx|scribe|suptxt|trpunc|
                trsp|uncn|vlet|vline|vphr|vphr1|vphr2|vwrd|
                vwrd1|vwrd2|w|w1|w2)*>
<!ELEMENT vline EMPTY>
<!ELEMENT w (#PCDATA|a|abb|accent|add|alt|comment|cvd|del|
             div|dmg|ednote|emd|enh|exp|fdd|gap|gloss|
             lang|let|oecno|omiss|overers|paleog|palimbot|
             palimtop|pcvd|pline|rejoin|res|rest|resx|
             suptxt|trsp|uncn|vlet|vline|vwrd)*>
<!ELEMENT w1 (#PCDATA|a|abb|accent|add|alt|comment|cvd|del|
              div|dmg|ednote|emd|enh|exp|fdd|gap|gloss|
              lang|let|oecno|omiss|overers|paleog|palimbot|
              palimtop|pcvd|pline|rejoin|res|rest|resx|
              suptxt|trsp|uncn|vlet|vline|vwrd1)*>
```

Fig. 3. A DTD excerpt

S = “hu þu me hæfst afrefredne ægþer
ge mid þinre smealican spræce,
ge mid þinre wynsumnesse þines”

D_1 := { <!ELEMENT coll (fol)*>
<!ELEMENT fol (line)*>
<!ELEMENT line (#PCDATA)>}
D_2 := { <!ELEMENT coll (vline)*>
<!ELEMENT vline (#PCDATA|hl|w)*>
<!ELEMENT w (#PCDATA)>}
D_3 := { <!ELEMENT coll (dmg)*>
<!ELEMENT dmg (#PCDATA|cvd|fdd|gap|uncn|offset)*>}

Fig. 4. A Concurrent Markup Hierarchy (excerpt) $< S, coll, D_1, D_2, D_3 >$

Distributed XML documents consist of XML encodings of the content of a CHM in the DTDs from it.

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* $content(d_1) = content(d_2) = ... = content(d_k) = S$ [3] .

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

d_1 := "<**coll**> <**line**>hu þu me hæfst afrefredne æg</**line**>
<**line**>þer ge mid þinre smealican spræ</**line**>
<**line**>ce, ge mid þinre wynsumnesse þines </**line**></**coll**>"

d_2 := "<**coll**> <**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**></**coll**>"

d_3 := "<**coll**> hu <**res**>þu m</**res**>e hæfst afrefredne
ægþ<**dmg**>er</**dmg**> ge <**dmg**>mid </**dmg**> þinre smealican spr
<**dmg**>æ</**dmg**>ce, ge mid þinre wynsum nesse þines</**coll**>"

Fig. 5. A distributed XML document $dd =< d_1, d_2, d_3 >$ over CMH in Figure 4

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,...,d_k. An example of a distributed XML document over the CMH in Figure 4 is given in Figure 5.

Our goal now is to define XML documents that incorporate in their markup exactly the information contained in a distributed XML document. We want to represent distributed XML documents as single XML documents, with no markup conflicts, and so that the component XML documents can be easily recovered. For an XML document d we let *nodes(d)* represent the set of nodes in DOM representation of d (not to be confused with $elements(d)$ defined earlier). We denote by $<$ or $>$ the total order relation over the set $nodes(d)$ [19]: for $x \in nodes(d)$, $y \in nodes(d)$, we have $x < y$, if the node x is before node y in d; we have $x > y$ if the node y is before node y in d. For a node $t \in nodes(d)$, we let $element(t)$ denote the corresponding element of t in $elements(d)$. We also let *STposition*$_d$*(t)* denote the value i such that the *starting tag* of t in d is located between the ith and the $(i+1)$th character in $content(d)$ (if the markup is at the end of the content, then we set *STposition*$_d$*(t)* $= |content(d)|$ and if it is at the beginning of the document, then *STposition*$_d$*(t)* $= 0$). Similarly, we let *ETposition*$_d$*(t)* denote the position of the *end tag* of markup t in d relative to $content(d)$. For a node $t \in nodes(d)$ we denote by $content(t)$ the substring of $content(d)$ from $STposition_d(t) + 1$ to $ETposition_d(t)$. If $STposition_d(t) = ETposition_d(t)$, we say that the markup t has no text content, $content(t) = \epsilon$ (here, ϵ denotes the empty string). We call such markup elements *empty*.

Now, we define the notion of a *path*, a useful instrument in describing markup and text content nesting.

Definition 3 *1. 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.

The path to a node $t \in nodes(d)$, *denoted* $path(d,t)$ *is the* sequence *of XML elements forming the path from the root of the DOM tree of* d *to the node* t.

2. 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.

The path to node $t \in nodes(d)$ w.r.t. D, *denoted* $path(d,t,D)$, *is the subsequence of all elements in* D *of* $path(d,t)$.

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 tags and characters to define "correct" single-document representations of the distributed XML documents.

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 the set of XML documents* d *such that*

(1) $content(d) = content(d^*)$ *and* $elements(d) = elements(d^*) \cap elements(D)$;

(2) $(\forall 0 \leq i \leq |content(d)|)\ path(d^*,i,D) = path(d,i)$;

(3) $(\forall t \in nodes(d^*) : element(t) \in elements(D) \wedge content(t) = \epsilon)\ \exists t' \in nodes(d) : element(t) = element(t') \wedge content(t') = \epsilon \wedge STposition_{d^*}(t) = STposition_d(t') \wedge path(d^*,t,D) = path(d,t')$;

(4) $(\forall t \in nodes(d) \exists t_1,t_2 \in nodes(d^*) : element(t) = element(t_1) = element(t_2) \wedge STposition_d(t) = STposition_{d^*}(t_1) \wedge ETposition_d(t) = ETposition_{d^*}(t_2)$ *(here, possibly* $t_1 = t_2$*);*

d_1:	“<r><a>It's<b/><c/><a><d/> rain<a> in Spain.</r>”;
d_2:	“<r><a>It's<b/><c/><a><d/> rain in Spain. </r>”;
d_3:	“<r><a>It's<b/><c/><a><d/> rain <a> in <X>Spa</X>in.</r>”;
d_4:	“<r>I<a>t's<b/><c/><a><d/> rain<a> in Spain.</r>”;
d_5:	“<r><a>It's<a><c/><d/> rain<a> in Spain.</r>”;
d_6:	“<r><a>It's<b/><c/><a><d/> rain in<a> Spain.</r>”;
d_7:	“<r><a>It's<d/><c/><a><b/> rain <a> in Spain.</r>”.

Fig. 6. Filters of XML documents?

(5) $(\forall t_1, t_2 \in nodes(d) : t_1 < t_2 \wedge content(t_1) = content(t_2) = \epsilon) \exists t'_1, t'_2 \in nodes(d^*) : t'_1 < t'_2 \wedge element(t_1) = element(t'_1) \wedge element(t_2) = element(t'_2) \wedge content(t'_1) = content(t'_2) = \epsilon \wedge STposition_d(t_1) = STposition_{d^*}(t'_1) \wedge ETposition_d(t_2) = ETposition_{d^*}(t'_2)$.

In a nutshell, a filter of d^* on D is any XML document encoding the same content with only the elements from D ((1)), that preserves the paths to each content character and each node w.r.t. D ((2), (3)). In addition, for every XML element in the filter, some element of the same type starts and some element of the same type ends in d^* at its starting and ending positions in the filter ((4)). Finally, the order in which empty elements occur in d^* is preserved in its filter ((5)). The propositions below show some properties of filters. A filter does not create new markup and does not change the *start tag* and *end tag* positions for markup present in the filter. However, a filter might "join" consecutive markup of the same type. Before stating the propositions, we show some examples of filters.

Example 5 *Let*

d^* = “<r><a>It's<b/><c/><a><d/> rain <a> in <X>Spa</X> in.</r>”.

Let D be a DTD with $elements(D) = \{r, a, b, c, d\}$. Consider the XML documents shown in Figure 6. Then:
$d_1, d_2 \in Filters(d^*, D)$. In d_2 the last two occurrenes of <a> in d^* have been merged.
$d_3 \notin Filters(d^*, D)$ because $X \notin elements(D)$ (contradicts Definition 4 (1));
$d_4 \notin Filters(d^*, D)$ because $path(d^*, 0, D) \neq path(d_4, 0)$, i.e., the first character of the content is not in the scope of <a> (contradicts Definition 4 (2));
$d_5 \notin Filters(d^*, D)$ because $path(d^*, c, D) \neq path(d_5, c)$ (element c is inside element a in d_5, but is outside a in d^*) and $b \notin nodes(d_5)$ (contradicts Definition 4 (3));
$d_6 \notin Filters(d^*, D)$ because the positioning of the end of the second a and the beginning of the third a elements in d_6 and in d^* are different. (contradicts Definition 4 (4));

$d_7 \notin Filters(d^*, D)$ because $b < d$ in d^*, but $b > d$ in d_7 (contradicts Definition 4 (5)).

The two propositions below state that filters of XML documents preserve all empty elements and do not introduce any extra markup.

Proposition 6 *(Filters preserve empty elements).*

Let $d \in Filters(d^*, D)$. *Then*

$|\{t | t \in nodes(d^*) \wedge element(t) \in D \wedge content(t) = \epsilon\}| = |\{t | t \in nodes(d) \wedge content(t) = \epsilon\}|$.

PROOF. Let $t \in nodes(d^*)$, $content(t) = \epsilon$ and $element(t) \in D$. By Definition 4 (3), there is a node $t' \in d$, $element(t') = element(t)$ and $content(t') = \epsilon$. Conversely, let $p \in nodes(d)$, $content(p) = \epsilon$. From Definition 4 (5) it follows that there is a $p' \in nodes(d^*)$ so that $content(p') = \epsilon$ and $element(p') = element(p)$. This proves the proposition. □

Proposition 7 (*Filters do not introduce extra markup)*

Let $d \in Filters(d^*, D)$. *Then*

$$|\{t | t \in nodes(d^*) \wedge element(t) \in D\}| \geq |\{t | t \in nodes(d)\}|.$$

PROOF. We prove by contradiction. Suppose that $|\{t | t \in nodes(d^*) \wedge element(t) \in D\}| < |\{t | t \in nodes(d)\}|$. Then from Proposition 6 it follows that $|\{t | t \in nodes(d^*) \wedge element(t) \in D \wedge content(t) \neq \epsilon\}| < |\{t | t \in nodes(d) \wedge content(t) \neq \epsilon\}|$. This contradicts Definition 4 (2) since there is at least one character in $content(d)$ for which the path is not conserved. Hence the assumption that $|\{t | t \in nodes(d^*) \wedge element(t) \in D\}| < |\{t | t \in nodes(d)\}|$ is false. □

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^*. The main challenge in constructing such a representation is eliminating the *markup conflicts* (so that the resulting document is a well-formed XML document). We formally define a conflict between two XML elements in different documents of a distributed XML document as follows.

Definition 8 *Let* $dd = < d_1, d_2, \ldots d_k >$ *be a distributed XML document over the collection of markup hierarchies* $CMH = < S, r, \{D_1, \ldots, D_k\} >$. *Let* $x \in nodes(d_i)$, $content(x) \neq \epsilon$, $y \in nodes(d_j)$, $content(y) \neq \epsilon$, *for some* $i \neq j$. x *has a* conflict *with* y *if one of the following happens:*

(1) $STposition_{d_i}(x) < STposition_{d_j}(y) < ETposition_{d_i}(x) < ETposition_{d_j}(y)$.

(2) $STposition_{d_j}(y) < STposition_{d_i}(x) < ETposition_{d_j}(y) < ETposition_{d_i}(x)$.

Besides *conflicts* as defined above, there might be situations where two markup elements (from different documents) start or end at the same position. Even though this is not a conflict in the sense of Definition 8, a fragmentation might occur if the order of markup is not handled carefully. These are *potential conflicts* and they are handled easily by appropriate sorting of markup from different documents.

Definition 9 *Let* $dd =< d_1, d_2, \ldots d_k >$ *be a distributed XML document over the collection of markup hierarchies* $CMH =< S, r, \{D_1, \ldots, D_k\} >$. *Let* x, y *be markup tags in* d_i, d_j *respectively* $(i \neq j)$. x *has a* potential conflict *with* y *if one of the following holds:*

(1) $STposition_{d_i}(x) = STposition_{d_j}(y)$

(2) $ETposition_{d_i}(x) = STposition_{d_j}(y)$

(3) $STposition_{d_i}(x) = ETposition_{d_j}(y)$

(4) $ETposition_{d_i}(x) = STposition_{d_j}(y)$.

We now introduce a notion of a *merger* of a distributed XML document. Intuitively, it is a well-formed XML document that "preserves" the encodings contained in all individual components of a distributed document.

Definition 10 *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*

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

As mentioned above, we want any merger d^* of dd to incorporate the markup from all documents $d_1, \ldots, d_k$ in a way that (theoretically) allows the restoration of each individual document from d^*. In practice, we demand that each individual component d_i of dd is a filter of d^* on D_i. A merger contains all the markup in the distributed document and, by virtue of the properties of filters, can resolve markup conflicts by using fragementation. The following two propositions mirror Propositions 6 and 7: they show that a merger preserves all the empty elements and does not remove any markup found in the distributed XML document.

Proposition 11 *(Mergers preserve empty elements)*

Let $dd =< d_1, d_2, \ldots d_k >$ be a distributed XML document and let $d^* \in Mergers(dd)$. Then

$|\{t|t \in nodes(d^*) \wedge content(t) = \epsilon\}| = \sum_{i=1}^{k} |\{t|t \in nodes(d_i) \wedge content(t) = \epsilon\}|$.

PROOF. We have $d_i \in Filters(d^*, D_i)$, $1 \leq i \leq k$. Then by Proposition 6 it follows that: $|\{t|t \in nodes(d^*) \wedge content(t) = \epsilon \wedge element(t) \in D_i\}| = |\{t|t \in nodes(d_i) \wedge content(t) = \epsilon\}|$. Then, $|\{t|t \in nodes(d^*) \wedge content(t) = \epsilon\}| = \sum_{i=1}^{k} |\{t|t \in nodes(d^*) \wedge content(t) = \epsilon \wedge element(t) \in D_i\}| = \sum_{i=1}^{k} |\{t|t \in nodes(d_i) \wedge content(t) = \epsilon\}|$. □

Proposition 12 (*Mergers do not remove markup*)

Let $dd =< d_1, d_2, \ldots d_k >$ be a distributed XML document over the collection of markup hierarchies $CMH =< S, r, \{D_1, \ldots, D_k\} >$ and let $d^* \in Mergers(dd)$. Then $|nodes(d^*)| \geq \sum_{i=1}^{k} |nodes(d_i)| - (k-1)$.

PROOF. Since $d_i \in Filters(d^*, D_i)$, $1 \leq i \leq k$, by Proposition 7 it follows that: $|\{t|t \in nodes(d^*) \wedge element(t) \in D_i\}| \geq |\{t|t \in nodes(d_i)\}|$. With the observation that all documents d_i, $1 \leq i \leq k$ share the same root, we have: $|nodes(d^*)| = \sum_{i=1}^{k} |\{t|t \in nodes(d^*) \wedge element(t) \in D_i\}| - (k-1) \geq \sum_{i=1}^{k} |nodes(d_i)| - (k-1)$. □

4 Algorithms

Section 3 specifies the properties that the "right" representations of distributed XML documents (i.e., single XML documents containing concurrent markup) 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 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 $dd' =< d_1, \ldots, d'_i, \ldots, d_k >$.

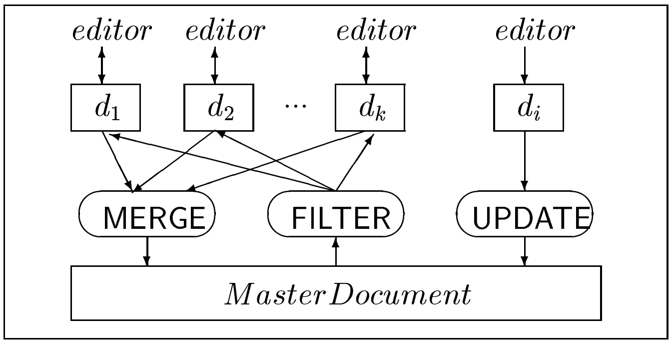


Fig. 7. The framework solution.

Figure 7 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.

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)[20] 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

```
Algorithm MERGE(d_1, ..., d_k)
  //PASS I
  tokenListSet = empty
  for cpos = 1 to sizeof(content(d_1))
    buffer content characters in contBuffer
    //Determine the correct nesting of all tags that end at cpos
    move all end tag tokens in tokenListSet[cpos] to EndTokenList
    build list of tokens from d_1, ..., d_k at position i
    collect tokenListSet[i] from d_1, ..., d_k
    //Find correct order of tokens, resolve overlapping conflicts
    pos = cpos-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" 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. 8. The MERGE algorithm

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

```
d* =‘‘<coll> <line link="1"> <w>hu</w> <res><w>þu</w>
                <w link="2">m</w></res><w link="2">e</w>
                <w>hæfst</w> <w>afrefredne</w>
             </line>
                <w link="3"> <line link="1">æ</line></w>
             <line> <w link="3">þ<dmg>er</dmg></w>
                <w>ge</w> <w><dmg>mid</dmg></w> <w>þinre</w>
                <w>smealican</w> <w link="4">spr<dmg>æ </dmg></w>
             </line>
             <line><w link="4">ce</w>, <w>ge</w> <w>mid</w>
                   <w>þinre</w> <w>wynsumnesse</w> <w>þines</w>
            </line></coll>’’
```

Fig. 9. The output of $\mathsf{MERGE}(dd)$ for the distributed XML document dd in Figure 5

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 characters.

Figure 8 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.

Figure 9 shows the output of $\mathsf{MERGE}(dd)$ for the distributed XML document dd in Figure 5. As seen from it, MERGE introduces four fragmentations in the output XML.

Theorem 13 *Let $dd =< d_1, \ldots, d_k >$ be a distributed XML document. Let d^* be the output of* $\mathsf{MERGE}(d_1, \ldots, d_k)$. *Then d^* is a* merger *of dd.*

PROOF. The MERGE algorithm decides the markup order at a given position in the document if more tags from at least two input documents occur at the same position (to avoid potential conflicts) and introduces markup fragmentation (to resolve conflicts and hence to preserve XML document well-formedness).

The proof if based on four key observations about how MERGE works: (i) the output document is constructed using only element (ST, ET) and content (CT) tokens parsed out from the input documents, (ii) the element tokens ST and ET are output exactly at the same positions in the output document con-

tent as in the original documents (stable sort w.r.t. tokens relative positions in each document), (iii) additional ET and ST tokens needed for fragmentation are inserted (in this order) at the same position in the document content, and (iv) the content tokens (CT) are output in the same order as in the original input documents. From (i) it follows that:
$elements(d^*) = \bigcup_{i=1}^{k} elements(d_i) \subseteq \bigcup_{i=1}^{k} elements(D_i) \quad (1)$
(Note that fragmentation doesn't introduce new elements, but more elements from $elements(d_i),\ 1 \leq i \leq k$).
From (iv) it follows directly that MERGE doesn't change the content of the output document (given that the input documents have the same content):
$content(d^*) = content(d_1) = ... = content(d_k)$
From (ii) and (iii) it follows that MERGE preserves the path to each character in the document content w.r.t. the elements of each input document (fragmentation splits the content into multiple manifestations of the same tag) and the path to each markup node without text content. Also no other markup besides tags in $elements(d_i),\ 1 \leq i \leq k$ are introduced. Consequently:
$d_i \in Filters(d^*, D_i),\ (\forall 1 \leq i \leq k)$
Then from (1), it follows that d^* is a merger for dd. □

A simple observation of the fact that MERGE produces fragmentation based on the *end tag* markup at a given position over all input documents gives the following result:

Proposition 14 *The number of fragments in the output of MERGE algorithm does not depend on the order of the input documents.*

The MERGE algorithm produces a *merger* of the input distributed XML document. The following theorem shows that MERGE's time complexity is linear in both text content and number of nodes over all documents in the distributed XML document.

Theorem 15 *Let $dd =< d_1, \ldots, d_k >$ be a distributed XML document. The time complexity of MERGE(dd) algorithm is $O((\sum_{i=1}^{k} |nodes(d_i)|)|content(d_1)|)$.*

PROOF. The MERGE algorithm gets the input XML documents as an array of tokens (markup tokens or text content tokens). At each position in the text content (there are $|content(d_1)| + 1$ such positions), markup that start/end at that position might conflict other markup. MERGE introduces fragmentation whenever necessarily to avoid conflicts: there are at most $(\sum_{i=1}^{k} |nodes(d_i)|$ fragments to be created per each conflict position. This makes the total number of fragments to be possibly introduced at most $(\sum_{i=1}^{k} |nodes(d_i)|)|content(d_1)|$. Since scanning the input documents takes $O(\sum_{i=1}^{k} |nodes(d_i)| + k|content(d_1)|)$, then, overall, the time complexity is $O((\sum_{i=1}^{k} |nodes(d_i)|)|content(d_1)|)$. □

```
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) ∈ glueTagSet OR tag(token) ∉ 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) ∉ D)
        continue
      if (tag(token) ∈ glueTagSet AND not last token for tag(token))
        continue
      output token
```

Fig. 10. The FILTER algorithm

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 Figure 10. The following theorem states that FILTER correctly reverses the work of the MERGE algorithm.

Theorem 16 *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$(d^*, D_i) = d_i$.

PROOF. Let $d_i' =$ FILTER(d^*, D_i). Then $content(d_i') = content(d_i)$ and

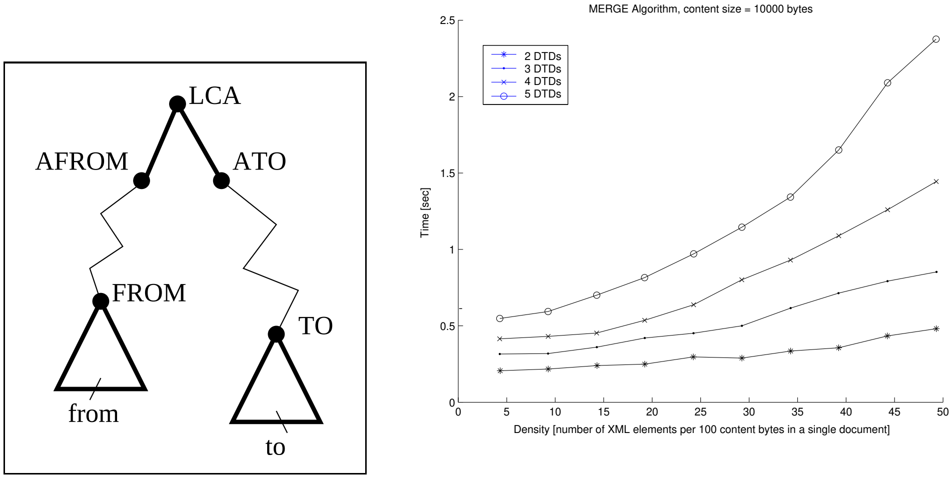
Algorithm $\mathsf{UPDATE}(from, to, TAG)$
  find FROM, TO, LCA, AFROM, ATO
  //start inserting nodes
  for (each NODE in the path from FROM to AFROM)
    insert a node TAG with glue attributes
        as a parent for all siblings of NODE, at the right of NODE
  insert a node TAG with glue attributes
        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. 11. The UPDATE algorithm

$\forall t \in nodes(d'_i)$, $element(t) \in D_i$.
We show that d'_i contains all markup in d_i at the same position as in d_i. Let $t \in nodes(d_i)$ (hence $element(t) \in D_i$). Then, as a result of $\mathsf{MERGE}(dd)$, there is a sequence of nodes $t_1, t_2, \ldots t_n \in d^*, 1 \le n \le |content(d^*)|$ so that $element(t) = element(t_1) = \ldots = element(t_n)$ and $STposition_{d_i}(t) = STposition_{d^*}(t_1), ETposition_{d^*}(t_1) = STposition_{d^*}(t_2), \ldots, ETposition_{d^*}(t_n) = ETposition_{d_i}(t)$ and each markup uses the same *glue attribute*. $\mathsf{FILTER}(d^*, D_i)$ joins all markup $t_1, t_2, \ldots t_n$ in a single node $t' \in d'_i$ so that $element(t') = element(t)$ and $STposition_{d'_i}(t') = STposition_{d^*}(t_1) = STposition_{d_i}(t)$, and $ETposition_{d'_i}(t') = ETposition_{d^*}(t_n) = ETposition_{d_i}(t)$.
We show now that d_i contains all markup in d'_i at the same position as in d'_i, by using a converse argument for the above. Let $t' \in d'_i$. Since d'_i is a result of a filter operation, then $element(t') \in D_i$ and t' has no *glue attributes*. Moreover, since FILTER doesn't remove markup (except for joining consecutive markup linked by a glue attribute), then d^* must contain a markup sequence as described above. Similarly, since MERGE doesn't remove markup and the only markup created by it are fragments joined by the same glue attribute, it follows that d_i (the only document that contains elements from D_i) has a node $t \in nodes(d_i)$, so that $element(t) \in D_i$ and $STposition_{d_i}(t) = STposition_{d'_i}(t')$ and $ETposition_{d_i}(t) = ETposition_{d'_i}(t')$. □

Proposition 17 *The time complexity of FILTER(d,D) algorithm is $O(|nodes(d)| + |content(d)|)$.*

PROOF. The FILTER algorithm scans each position in $content(d)$ and for each ST or ET token performs a constant number of operations: test for glue attributes, removing glue attributes, or joining markup. All these add up to $O(|nodes(d)| + |content(d)|)$ time complexity. □



(a) (b)

Fig. 12. (a) Illustrating the work of the UPDATE algorithm; (b)Test 1: Dependence of performance of the MERGE algorithm on markup density.

4.3 UPDATE Algorithm

The UPDATE algorithm updates the master XML document (see Figure 7) 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 [21] 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 Figure 12).

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 needed. The pseudo-code description of the algorithm is shown in Figure 11. The following theorem states that the result of UPDATE allows for correct recovery of components of the distributed document.

Theorem 18 *Let* $dd =< d_1, \ldots, d_k >$*;* d^* *be the output of* MERGE(dd)*. Let* $TAG \in elements(D_i)$*,* $(from, to, TAG)$ *be an update request and* d'_i *be the*

Content Size	Avg. Num. of Tags			Avg. Num. of Conflicts		
[bytes]	Sparse	Med. Dens.	Dense	Sparse	Med. Dens.	Dense
1000	294.8	1285.4	2522.8	96.2	85.8	96.8
2000	479.6	2476.4	4977.6	188.2	197.2	160.6
3000	679.4	3692.4	7438	313.2	294.2	284
4000	905.4	4903	9900.8	329.2	360	365.6
5000	1092.8	6108.8	12350	445.6	451.6	555.2
6000	1307	7306	14803.4	540.2	597.6	593.6
7000	1522.8	8507.2	17256.4	646.6	721.6	753.6
8000	1707	9705.8	19727	774.4	828.4	805.8
9000	1921.2	10930.8	22175	830.6	900.6	921.4
10000	2144	12142.2	24644.2	1031.4	991	909

Table 1
Datasets used for Test 2.

well-formed result of tagging the content between $from$ *and* to *positions. Then,*

$\mathsf{FILTER}(\mathsf{UPDATE}(d^*, (from, to, TAG)), D_i) = d'_i$.

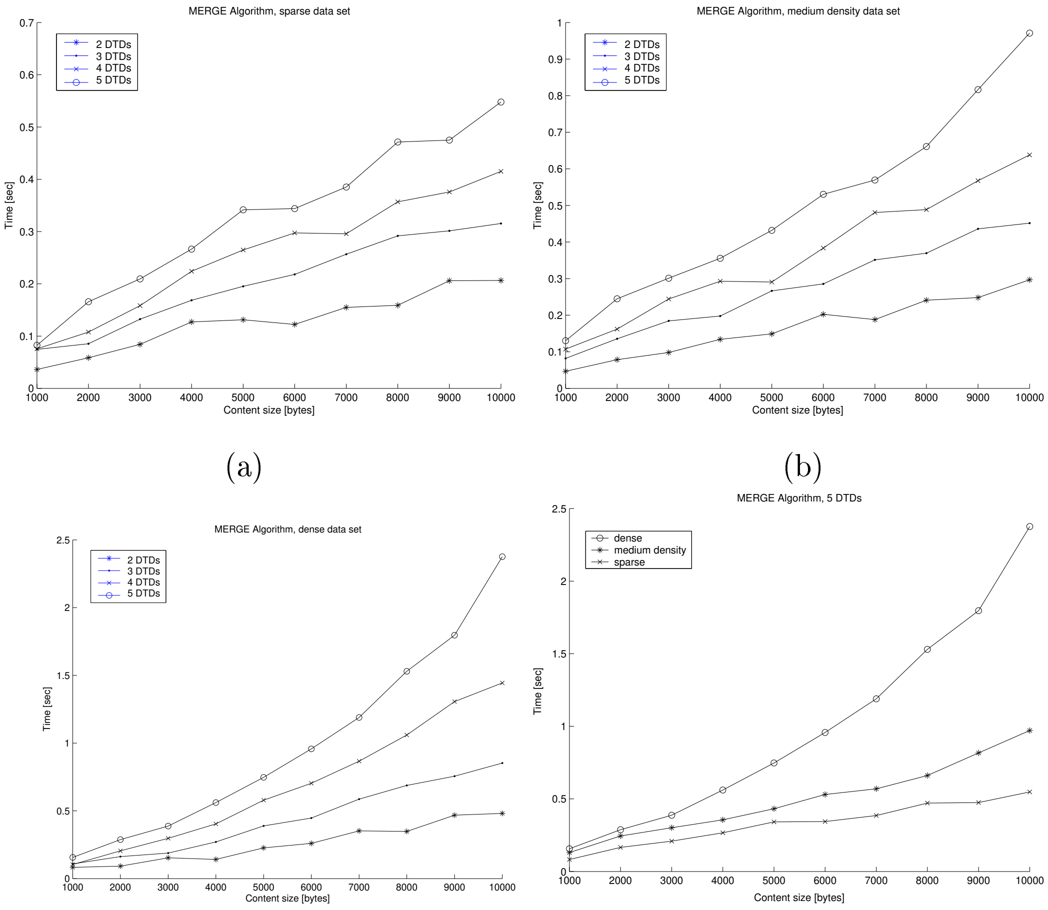
PROOF. The UPDATE algorithm is a markup insertion (using DOM model for the XML document) combined with fragmentation (to preserve well-formedness). Hence, if d'^* is the output of $\mathsf{UPDATE}(d^*, (from, to, TAG))$ then d'^* is a merger of $dd' =< d_1, \ldots, d'_i, \ldots, d_k >$ where the markup corresponding to TAG might be fragmented using *glue attributes*. Then $\mathsf{FILTER}(d'^*, D_i)$ removes all markup elements not in D_i and joins adjacent markup with glue attributes. Hence $\mathsf{FILTER}(\mathsf{UPDATE}(d^*, (from, to, TAG)), D_i) = \mathsf{FILTER}(d'^*, D_i) = d'_i$. □

5 Evaluating Performance of the Merging Algorithm

In this section we describe the results of testing the performance of the MERGE algorithm described in previous section.

The tests were conducted in the following manner. We have designed an XML generator that, given a DTD, a root element, number of tags and length of a content string generates an XML document with these parameters. The XML document may not necessarily be valid w.r.t. the DTD, but the encoding does not violate any DTD constraints. Rather, the cause of invalidity is the possible incompleteness of the markup. Such XML documents model the process by which editors introduce document-centric markup – only the final version of the document is guaranteed to be valid.

In our tests we have used the same DTD repeated five times, with tags renamed in each copy. The performance of the MERGE algorithm on a specific problem can be affected by the following parameters:



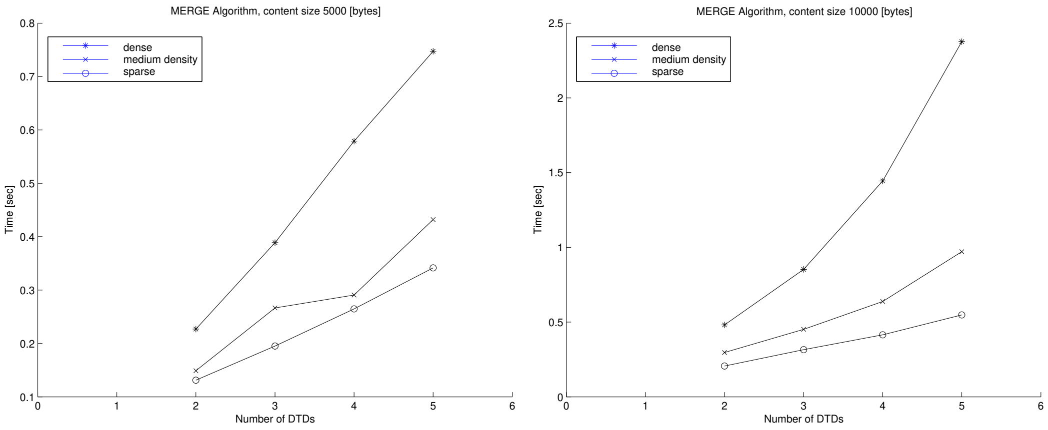
(a) (b) (c) (d)

Fig. 13. Test 2: Dependence of performance of the MERGE algorithm on content size for (a) sparse, (b) medium density and (c) dense datasets. Running times for all three datasets for 5 DTDs are compared side-by-side in (d).

- *Number of DTDs in the CMH*;
- *Size of the content string* of the input distributed XML document. We measured the size in the number of characters (bytes);
- *Markup density* in individual components of the input distributed XML document. It was measured in the number of XML elements per 100 bytes of content.

We have ran two separate experiments:

- Test 1: study dependence of the performance of the algorithm on the density of markup. The content size for this experiment was fixed at 10000 bytes, and the density ranged from 5 elements per 100 bytes to 50 elements per 100 bytes. The tests were ran on the distributed XML documents with 2, 3, 4 and 5 components.



(a) (b)

Fig. 14. Test 2: Dependence of performance of the MERGE algorithm on the number of DTDs for content size of (a) 5000 and (b) 10000 bytes.

- Test 2: study dependence of the performance of the algorithm on the size of the content. Three datasets called *sparse*, *medium density* and *dense* were generated, with the density of markup of 5, 25 and 50 elements per 100 bytes respectively. In each dataset, distributed XML documents were generated for content sizes of 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000 and 10000 bytes. The tests were ran on the distributed XML documents with 2, 3, 4 and 5 components.

In both tests, for each point of the graph, we have generated five different distributed XML documents. We measured the performance of the MERGE algorithm on each of the five distributed XML documents and plotted the average time.

The MERGE algorithm was implemented in Java, and run on a Dell Optiplex GX240 PC with 1.4Ghz Pentium 4 processor and 256 Mb main memory. On each run, the input files were first read into main memory. From there, the SAX parser tokenized the input and provided it for the MERGE algorithm. Only the running time of the MERGE algorithm proper was counted and is reported in the graphs. The time for parsing reading data from disk had been excluded. Below, we briefly discuss the results of the tests.

Figure 12 contains the results of the Test 1 series for all numbers of DTDs. As seen from the graph, for two and three DTDs, the growth of running time is linear in the density of markup, whereas for four and five DTDs, it appears to be quadratic.

Figure 13 shows the main results of Test 2 for each of the three data sets used (graphs (a), (b) and (c)). Graph (d) compares the 5 DTD results for all three datasets. Table 1 summarizes the three datasets used in Test 2. For each dataset and for each content size we indicate the average number of elements in the distributed XML documents (among the five documents generated) and the average number of markup conflicts the MERGE algorithm had to resolve (the data is provided for the distributed XML documents consisting of five components). Looking at the graphs on Figure 13 we observe that for the sparse and medium density datasets, the running time grows linearly with the size of the content. For the dense dataset, we were able to observe quadratic running time behavior only for the 5 DTD CMH. A general observation is that on all datasets, the algorithm performed well in real-time, finishing most of the tests in under 1 second.

Figure 14 inverts the data from Figure 13 to show the dependence of running time on the number of DTDs in the CMH. Graph (a) shows this dependence for the size of content equal to 5000 bytes, while graph (b) — for 10000 bytes. It appears from these two graphs that the dependence of running time on the number of DTDs (distributed XML document components in the input) is quadratic at 10000 bytes for the dense and medium dense dataset, and is linear elsewhere.

The overall conclusion we draw from the tests is two-fold:

(1) The MERGE algorithm as implemented in this work is applicable in practice.
(2) The running time of the MERGE algorithm shows linear dependency on various parameters (markup density, content size, number of DTDs) on smaller values, and gradually changes into quadratic dependency as the values grow. This behavior is consistent with the result of Theorem 15.

6 Conclusions

In this paper we introduce the general framework for managing concurrent XML markup hierarchies. In particular, we formally define notions of concurrent markup hierarchies and distributed XML documents over them. The key problem studied here is that of representation of distributed XML documents as single well-formed XML documents. Via the notions of *filters* and *mergers*, we defined what is means for a single XML document to *faithfully represent* a distributed XML document. We have provided efficient algorithms for constructing such representations from distributed XML documents (MERGE), extracting distributed document components from them (FILTER) and incrementally updating them when new markup is introduced (UPDATE).

Acknowledgments

This article is based on work supported, in part, by the National Science Foundation under Grant No. 0219924. The work of the second author is also supported, in part, by The Electronic Boethius Project funded by a Collaborative Research Award from the National Endowment for the Humanities (NEH grant RZ-20887-02) and the Andrew W.Mellon Foundation, and sponsored by The British Library and the Bodleian Lirary, Oxford, who are providing digital images of the relevant documents. The manuscript image [18] 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. The authors would like to thank Kevin Kiernan for the introduction to the problem of concurrent hierarchies and inspiration. We would also like to thank Dorothy C. Porter for useful comments, and Jerzy W. Jaromczyk all the students working on the ARCHway project for useful discussions.

References

[1] A. Renear, E. Mylonas, D. Durand, Refining our notion of what text really is: The prob lem of overlapping hierarchies, Research in Humanities Computing (Editors: N. Ide and S. Hockey).

[2] C. M. Sperberg-McQueen, L. Burnard, Guidelines for Text Encoding and Interchange (P4), http://www.tei-c.org/P4X/index.html, the TEI Consortium (2001).

[3] P. Durusau, M. B. O'Donnell, Concurrent Markup for XML Documents, in: Proc. XML Europe, 2002.

[4] A. Witt, Meaning and interpretation of concurrent markup, in: Proc., Joint Conference of the ALLC and ACH, 2002, pp. 145–147.

[5] P. Durusau, M. O'Donnell, Declaring trees: The future of the evolution of markup?, in: Proc. Conference on Extreme Markup Languages, 2002.

[6] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, J. L. Wiener, Incremental maintenance for materialized views over semistructured data, in: Proc. of VLDB, 1998, pp. 38–49.

[7] W. May, Integration of XML data in XPathLog, in: DIWeb, 2001, pp. 2–16.

[8] W. May, Lopix: A system for XML data integration and manipulation, in: The VLDB Journal, 2001, pp. 707–708.

[9] I. Manolescu, D. Florescu, D. Kossmann, Answering XML queries over heterogeneous data sources, in: Proc. of VLDB , Roma, Italy, 2001, pp. 241–250.

[10] C. Huitfeldt, 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).

[11] C. M. Sperberg-McQueen, C. Huitfeldt, GODDAG: A Data Structure for Overlapping Hierarchies, ACH-ALLC Conference, Charlottesville (June 1999).

[12] K. Kiernan, J. Jaromczyk, A. Dekhtyar, D. Porter, K. Hawley, S. Bodapati, I. Iacob, The ARCHway project: Architecture for research in computing for humanities through research, teaching, and learning, Literary and Linguistic Computing (forthcoming).

[13] K. Kiernan, A. Prescott, E. Solopova, D. French, L. Cantara, M. Ellis, C. Yuan, I. Iacob, Electronic Beowulf, http://www.uky.edu/∼kiernan/eBeowulf/guide.htm (2003).

[14] E. Solopova, Encoding a transcript of the Beowulf manuscript in SGML, in: Proc. ACH/ALCC, 1999.

[15] W. Seales, J. Griffioen, K. Kiernan, C. J. Yuan, L. Cantara, The Digital Atheneum: New Technologies for Restoring and Preserving Old Documents, Computers in Libraries 20 (2) (2000) 26–30.

[16] Canterbury tales project, http://www.cta.dmu.ac.uk/projects/ctp/, De Monfort University (1999).

[17] C. Sperberg-McQueen, D. Seaman, A TEI-based tag set for manuscript transcription, Digital Scriptorium.

[18] British Library MS Cotton Otho A. vi, fol. 38v.

[19] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, Extensible Markup Language (XML) 1.0 (Second Edition), http://www.w3.org/TR/REC-xml, W3C, REC-xml-20001006 (Oct 2000).

[20] Simple API for XML (SAX) 2.0.1, http://www.saxproject.org, SourceForge project (Jan 2002).

[21] A. L. Hors, P. L. Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, S. Byrne, DOM, http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/ (Nov 2000).