A Framework for Management of Concurrent XML Markup

Alex Dekhtyar  Ionut E. Iacob

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.

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
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 ex-
pressive 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 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.
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/XMLSchema. 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 “ag” and “per” 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 \texttt{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 \texttt{line} tag. Second, we notice three versions of a tag for encoding words: \texttt{w}, \texttt{w1} and \texttt{w2}. Words in the manuscripts can cross physical line borders and folio borders. Tag \texttt{w} was used to encode words that are not in conflict with the physical encoding of the manuscript (\texttt{line} and \texttt{folio} tags). Tags \texttt{w1} and \texttt{w2} were used when the word crossed one of the boundaries above: \texttt{w1} encoded the first part of the word while \texttt{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 result, the entire process was extremely error-prone. Second, search queries on the encoding created using milestone elements and split tags such as \texttt{w1} and \texttt{w2} had to be programmed in a specialized fashion that had to ensure that, for example, when looking for the "content" of \texttt{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 \texttt{w1}<\texttt{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.
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 $\text{elements}(D)$ denote the set of all XML elements defined in $D$. Similarly, if $d$ is an XML document, $\text{elements}(d)$, denotes the set of all XML element tags contained in document $d$. Given a DTD $D$, we say that an element $y \in \text{elements}(D)$ is an ancestor in $D$ of another element $x \in \text{elements}(D)$ 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 $\text{CMH}$ is a tuple $\text{CMH} =< S, r, \{D_1, D_2, \ldots, D_k\}>$ where:

- $S$ is a string representing the document content;
- $r$ is an XML element called the root of the hierarchy;
- $D_i$, $i = 1, k$ are DTDs such that:
  (i) $r$ is defined in each $D_i$, $1 \leq i \leq k$, and $\forall 1 \leq i, j \leq k$, $i \neq j$
  $\text{elements}(D_i) \cap \text{elements}(D_j) = \{r\};$
  (ii) $\forall 1 \leq i \leq k$, $\forall t \in \text{elements}(D_i)$ $r$ 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).
S = “hu þu me hæfst arefredne ægehr
ge mið þiørne sméalcan spræce,
ge mið þiørne wynsumnesse þines”

$D_1 := \{ <!ELEMENT coll (fol) *> \}$

$D_2 := \{ <!ELEMENT coll (vline) *> \}$

$D_3 := \{ <!ELEMENT coll (dmg) *> \}$

Fig. 4. A Concurrent Markup Hierarchy (excerpt) $\langle S, coll, D_1, D_2, D_3 \rangle$

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

**Definition 2** Let $CMH = \langle S, r, \{ D_1, D_2, \ldots, D_k \} \rangle$ be a concurrent markup hierarchy. A distributed XML document $dd$ over $CMH$ is a collection of XML documents: $dd = \langle d_1, d_2, \ldots, d_k \rangle$ where ($\forall 1 \leq i \leq k$) $d_i$ is valid w.r.t. $D_i$ and $content(d_1) = content(d_2) = \ldots = content(d_k) = S^3$.

$^3 content(doc)$ denotes the text content of the XML document $doc$. 
\[d_1 := \text{"\langle coll \rangle < line > hu ða me hæfst afrefredne æg\langle /line >\rangle} < line > ber ge mid þinre snealian spræ\langle /line >\rangle < line > ce, ge mid þinre wynsumnesse þines \langle /line >\rangle\langle /coll \rangle\]

\[d_2 := \text{"\langle coll \rangle < w > hu\langle /w > < w > ðu\langle /w > < w > me\langle /w >\rangle < w > hæfst\langle /w > < w > afrefredne\langle /w > < w > æger\langle /w > < w > ge\langle /w > < w > mid\langle /w > < w > þinre\langle /w > < w > snealian\langle /w > < w > spræce\langle /w >, \langle /w > ge\langle /w > < w > mid\langle /w > < w > þinre\langle /w > < w > wynsumnesse\langle /w > < w > þines\langle /w >\langle /coll \rangle\text{"}}

\[d_3 := \text{"\langle coll \rangle lu \langle res \rangle ðu m\langle /res \rangle e hæfst afrefredne ægb\langle /dmg \rangle er\langle /dmg \rangle ge \langle /dmg \rangle mid \langle /dmg \rangle þinre snealian spræce\langle /dmg \rangle ae\langle /dmg \rangle ce, ge mid þinre wynsumnesse þines\langle /coll \rangle\text{"}}

Fig. 5. A distributed XML document \(dd = \langle d_1, d_2, d_3 \rangle\) 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 \(d_1, \ldots, d_k\). 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 \(\text{nodes}(d)\) represent the set of nodes in DOM representation of \(d\) (not to be confused with \(\text{elements}(d)\) defined earlier). We denote by \(<\) or \(>\) the total order relation over the set \(\text{nodes}(d)\) [19]: for \(x \in \text{nodes}(d)\), \(y \in \text{nodes}(d)\), we have \(x < y\), if the node \(x\) is before node \(y\) in \(d\); we have \(x > y\) if the node \(x\) is before node \(y\) in \(d\). For a node \(t \in \text{nodes}(d)\), we let \(\text{element}(t)\) denote the corresponding element of \(t\) in \(\text{elements}(d)\). We also let \(\text{STposition}_d(t)\) denote the value \(i\) such that the starting tag of \(t\) in \(d\) is located between the \(i\)th and the \((i+1)\)th character in \(\text{content}(d)\) (if the markup is at the end of the content, then we set \(\text{STposition}_d(t) = |\text{content}(d)|\) and if it is at the beginning of the document, then \(\text{STposition}_d(t) = 0\)). Similarly, we let \(\text{ETposition}_d(t)\) denote the position of the end tag of markup \(t\) in \(d\) relative to \(\text{content}(d)\). For a node \(t \in \text{nodes}(d)\) we denote by \(\text{content}(t)\) the substring of \(\text{content}(d)\) from \(\text{STposition}_d(t) + 1\) to \(\text{ETposition}_d(t)\). If \(\text{STposition}_d(t) = \text{ETposition}_d(t)\), we say that the markup \(t\) has no text content, \(\text{content}(t) = \epsilon\) (here, \(\epsilon\) 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 \(\text{content}(d) = S\). Let
$S = c_1 c_2 \ldots c_M$. 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 $a_1/a_2/\ldots/a_s$. 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. $(\forall t \in nodes(d^*) : element(t) \in elements(D) \land content(t) = \epsilon) \exists t' \in nodes(d) : element(t) = element(t') \land content(t') = \epsilon \land STposition_{d^*}(t) = STposition_d(t') \land path(d^*, t, D) = path(d, t')$;
4. $(\forall t \in nodes(d) \exists t_1, t_2 \in nodes(d^*) : element(t) = element(t_1) = element(t_2) \land STposition_d(t) = STposition_d^*(t_1) \land ETposition_d(t) = ETposition_d^*(t_2)$ (here, possibly $t_1 = t_2$);
d_1: "<r><a>It's<b/>/</a><c/>><a><d/> rain </a><a> in Spain.</a></r>";

\[d_2: "<r><a>It's<b/>/</a><c/>><a><d/> rain in Spain. </a>";
\]

\[d_3: "<r><a>It's<b/>/</a><c/>><a><d/> rain </a><a> in <X>Spa</X> in.</a></r>";
\]

\[d_4: "<r><a>It's<b/>/</a><c/>><a><d/> rain </a><a> in Spain.</a></r>";
\]

\[d_5: "<r><a>It's<b/>/</a><c/>><a><d/> rain </a><a> in Spain. </a>";
\]

\[d_6: "<r><a>It's<b/>/</a><c/>><a><d/> rain </a><a> in Spain. </a>";
\]

\[d_7: "<r><a>It's<d>/</a><c/>><a><b/> rain </a><a> in Spain.</a></r>".
\]

**Fig. 6. Filters of XML documents?**

\((5) \ (\forall t_1, t_2 \in \text{nodes}(d)) : t_1 < t_2 \land \text{content}(t_1) = \text{content}(t_2) = \epsilon \exists t_1', t_2' \in \text{nodes}(d^*) : t_1' < t_2' \land \text{element}(t_1) = \text{element}(t_1') \land \text{element}(t_2) = \text{element}(t_2') \land \text{content}(t_1') = \text{content}(t_2') = \epsilon \land \text{STposition}_{d^*}(t_1) = \text{STposition}_{d^*}(t_1') \land \text{ETposition}_{d^*}(t_2) = \text{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

\[d^* = "<r><a>It's<b/>/</a><c/>><a><d/> rain </a><a> in \n<X>Spa</X> in.</a></r>".
\]

Let \(D\) be a DTD with \(\text{elements}(D) = \{r, a, b, c, d\}\). Consider the XML document shown in Figure 6. Then:

\(d_1, d_2 \in \text{Filters}(d^*, D)\). In \(d_2\) the last two occurrences of \(<a>\) in \(d^*\) have been merged.

\(d_5 \notin \text{Filters}(d^*, D)\) because \(X \notin \text{elements}(D)\) (contradicts Definition 4 (1));

\(d_4 \notin \text{Filters}(d^*, D)\) because \(\text{path}(d^*, 0, D) \neq \text{path}(d_4, 0)\), i.e., the first character of the content is not in the scope of \(<a>\) (contradicts Definition 4 (2));

\(d_5 \notin \text{Filters}(d^*, D)\) because \(\text{path}(d^*, c, D) \neq \text{path}(d_5, c)\) (element \(c\) is inside element \(a\) in \(d_5\), but is outside \(a\) in \(d^*\)) and \(b \notin \text{nodes}(d_5)\) (contradicts Definition 4 (3));

\(d_6 \notin \text{Filters}(d^*, D)\) 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));
$d_\tau \notin \text{Filters}(d^*, D)$ because $b < d$ in $d^*$, but $b > d$ in $d_\tau$ (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 $d \in \text{Filters}(d^*, D)$. Then

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

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

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

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

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

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

If we are to combine the encodings of all $d_i$s 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 $x \in \text{nodes}(d_i)$, $\text{content}(x) \neq \epsilon$, $y \in \text{nodes}(d_j)$, $\text{content}(y) \neq \epsilon$, for some $i \neq j$. $x$ has a conflict with $y$ if one of the following happens:
(1) \( ST_1(x) < ST_2(y) < ET_1(x) < ET_2(y) \).
(2) \( ST_1(y) < ST_2(x) < ET_1(y) < ET_2(x) \).

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. \( ST_{d_i}(x) = ST_{d_j}(y) \)
2. \( ET_{d_i}(x) = ST_{d_j}(y) \)
3. \( ST_{d_i}(x) = ET_{d_j}(y) \)
4. \( ET_{d_i}(x) = ST_{d_j}(y) \).

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 \( \text{Merger}(dd) \) is defined as

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

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 fragmentation. 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 \( d^* \in \text{Mergers}(dd) \). Then
\[
|\{t | t \in \text{nodes}(d^*) \land \text{content}(t) = \epsilon\}| = \sum_{i=1}^{k} |\{t | t \in \text{nodes}(d_i) \land \text{content}(t) = \epsilon\}|.
\]

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

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 \( d^* \in \text{Mergers}(dd) \). Then
\[
|\text{nodes}(d^*)| \geq \sum_{i=1}^{k} |\text{nodes}(d_i)| - (k - 1).
\]

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

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 \( dd' =< d_1, \ldots, d_i', \ldots, d_k > \).
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 \textsc{merge}(d_1, \ldots, d_k)

//PASS I
\text{tokenListSet} = \text{empty}
\\text{for} \ cpos = 1 \text{ to sizeof(content}(d_1)\text{)}
\quad \text{buffer content characters in contBuffer}
\quad \text{//Determine the correct nesting of all tags that end at cpos}
\quad \text{move all end tag tokens in \text{tokenListSet}[cpos] to \text{EndTokenList}}
\quad \text{build list of tokens from } d_1, \ldots, d_k \text{ at position } i$
\quad \text{collect } \text{tokenListSet}[i] \text{ from } d_1, \ldots, d_k$
\quad \text{//Find correct order of tokens, resolve overlapping conflicts}
\quad pos = cpos-1
\quad \text{while not empty(\text{EndTokenList})}
\quad \quad \text{for (each unmarked start tag in } \text{tokenListSet}[pos])$
\quad \quad \quad \text{if (start tag is in } \text{EndTokenList})$
\quad \quad \quad \quad \text{push(end tag, } \text{tokenListSet}[cpos])$
\quad \quad \quad \quad \text{delete(end tag, } \text{EndTokenList})$
\quad \quad \quad \quad \text{mark(start tag)}$
\quad \quad \quad \text{else}$
\quad \quad \quad \quad \text{add ”glue attribute” to start tag entry}
\quad \quad \quad \quad \text{push(matching end tag, } \text{tokenListSet}[cpos])$
\quad \quad \quad \quad \text{append(start tag, } \text{tokenListSet}[cpos])$
\quad \quad \quad \quad \text{mark(start tag)}$
\quad \quad \quad \quad pos = pos -1$
\quad \quad \text{// PASS II}
\quad \quad marker = 0$
\quad \text{for (each position entry pos in } \text{tokenListSet})$
\quad \quad \quad output content in contBuffer from marker to pos
\quad \quad \quad \text{marker = pos}$
\quad \quad \quad output tokens at position pos in tokenListSet$

Fig. 8. The \textsc{merge} algorithm

the tag returned by SAX (for ST and ET tokens).

The \textsc{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 \textsc{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 \textsc{merge} algorithm, for each \( i \), tokenListSet[\( i \)] will contain the markup elements to be inserted in front of \( i \)th character of the content string.
d* = '‚<coll> <line link="1"> <w>hu</w> <res><w>hu</w>  
  <w> link="2">m</w>/<w>link="2">e</w>  
  <w>hafst</w> <w>a</w>refredne</w> 
</line>  
  <w link="3"> <line link="1">m</line></w>  
  <line> <w link="3">]<dmg>er</dmg></w>  
  <w>ge</w> <w>dmg>mid</w> <w>binre</w>  
  <w>smearcan</w> <w link="4">spr</w>dmg>ae </dmg></w>  
</line>  
  <line><w link="4">ce</w>, <w>ge</w> <w>mid</w>  
  <w>binre</w> <w>wnsumnesse</w> <w>bin</w>es</w>  
</line></coll>'

Fig. 9. The output of 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 token-ListSet, 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 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 = < d₁, ..., dₖ > be a distributed XML document. Let d* be the output of MERGE(d₁, ..., dₖ). 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 is 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:

\[ \text{elements}(d^*) = \bigcup_{i=1}^{k} \text{elements}(d_i) \subseteq \bigcup_{i=1}^{k} \text{elements}(D_i) \quad (1) \]

(Note that fragmentation doesn’t introduce new elements, but more elements from \( \text{elements}(d_i), \ 1 \leq i \leq k \).

From (iv) it follows directly that \text{MERGE} doesn’t change the content of the output document (given that the input documents have the same content):

\[ \text{content}(d^*) = \text{content}(d_1) = \ldots = \text{content}(d_k) \]

From (ii) and (iii) it follows that \text{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 \( \text{elements}(d_i), \ 1 \leq i \leq k \) are introduced. Consequently:

\[ d_i \in \text{Filters}(d^*, D_i), \quad (\forall 1 \leq i \leq k) \]

Then from (1), it follows that \( d^* \) is a merger for \( dd \). □

A simple observation of the fact that \text{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 \text{MERGE} algorithm does not depend on the order of the input documents.

The \text{MERGE} algorithm produces a merger of the input distributed XML document. The following theorem shows that \text{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 \text{MERGE}(dd) algorithm is \( O((\sum_{i=1}^{k} |\text{nodes}(d_i)|)|\text{content}(d_1)|) \).

**PROOF.** The \text{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 \(|\text{content}(d_i)| + 1 \) such positions), markup that start/end at that position might conflict other markup. \text{MERGE} introduces fragmentation whenever necessarily to avoid conflicts: there are at most \( (\sum_{i=1}^{k} |\text{nodes}(d_i)| \) fragments to be created per each conflict position. This makes the total number of fragments to be possibly introduced at most \( (\sum_{i=1}^{k} |\text{nodes}(d_i)|)|\text{content}(d_1)| \). Since scanning the input documents takes \( O((\sum_{i=1}^{k} |\text{nodes}(d_i)|+k|\text{content}(d_1)|) \), then, overall, the time complexity is \( O((\sum_{i=1}^{k} |\text{nodes}(d_i)|)|\text{content}(d_1)|). \ □
Algorithm \textsc{FILTER}(d, D) \\
glueTagSet = \text{empty}  \\
start parsing document \(d\)  \\
while (more tokens)  \\
token = \text{nextToken()}  \\
if (\text{token is CT})  \\
\hspace{1em} \text{output token}  \\
\hspace{1em} \textbf{continue}  \\
\hspace{1em} \textbf{else if} (\text{token is ST})  \\
\hspace{2em} \text{if} (\text{tag(token)} \in \text{glueTagSet} \text{ OR tag(token)} \notin D)  \\
\hspace{3em} \textbf{continue}  \\
\hspace{3em} \text{if} (\text{tag(token)} \text{ has glue attributes})  \\
\hspace{4em} \text{remove glue attributes}  \\
\hspace{4em} \text{put token in glueTagSet}  \\
\hspace{2em} \text{output token}  \\
\hspace{1em} \textbf{else if} (\text{token is ET})  \\
\hspace{2em} \text{if} (\text{tag(token)} \notin D)  \\
\hspace{3em} \textbf{continue}  \\
\hspace{3em} \text{if} (\text{tag(token)} \in \text{glueTagSet} \text{ AND not last token for tag(token)})  \\
\hspace{4em} \textbf{continue}  \\
\hspace{2em} \text{output token}  \\

Fig. 10. The \textsc{FILTER} algorithm

4.2 \textsc{FILTER} Algorithm

The \textsc{FILTER} algorithm takes as input an XML document \(d^*\) produced by the \textsc{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 \textsc{FILTER} appears in Figure 10. The following theorem states that \textsc{FILTER} correctly reverses the work of the \textsc{MERGE} algorithm.

Theorem 16 Let \(dd = \langle d_1, \ldots, d_k \rangle\) be a distributed XML document, and \(d^*\) be the output of \textsc{MERGE}(dd). Then (\(\forall 1 \leq i \leq k\)), \(\textsc{FILTER}(d^*, D_i) = d_i\).

PROOF. Let \(d_i' = \textsc{FILTER}(d^*, D_i)\). Then content\((d_i')\) = content\((d_i)\) and
Algorithm 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 for 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

∀t ∈ nodes(d′ i), element(t) ∈ D i.

We show that d′ i contains all markup in d i at the same position as in d i. Let

\text{t ∈ nodes}(d i) \text{ (hence element}(t) ∈ D i). \text{Then, as a result of MERGE(dd),}

there is a sequence of nodes t 1, t 2, ..., t n ∈ d′ i, 1 ≤ n ≤ |content(d i)| so that
element(t) = element(t 1) = \ldots = element(t n) and STposition_{d i}(t) =

STposition_{d i′}(t 1), ETposition_{d i′}(t 1) = STposition_{d i′}(t 2), \ldots, ETposition_{d i′}(t n) =

ETposition_{d i′}(t) and each markup uses the same glue attribute. FILTER(d′ i, D i)
joins all markup t 1, t 2, ..., t n in a single node t′ i ∈ d′ i so that element(t′ i) =
element(t) and STposition_{d i′}(t′ i) = STposition_{d i′}(t 1) = STposition_{d i′}(t), and

ETposition_{d i′}(t′ i) = ETposition_{d i′}(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′ i ∈ d′ i. Since d′ i is a result of a
filter operation, then element(t′ i) ∈ D i and t′ i has no glue attributes. Moreover,
since FILTER doesn’t remove markup (except for joining consecutive markup
linked by a glue attribute), then d′ i must contain a markup sequence as de-
scribed above. Similarly, since MERGE doesn’t remove markup and the only
markup created by it are fragments joined by the same glue attribute, it fol-

\text{lowt}\text{s that d}_i \text{ (the only document that contains elements from d}_i\text{) has a node}
t ∈ nodes(d i), so that element(t) ∈ D i and STposition_{d i}(t) = STposition_{d i′}(t′ i)
and ETposition_{d i}(t) = ETposition_{d i′}(t′ i).

\text{Proposition 17} The time complexity of FILTER(d, D) algorithm is \text{O}[|nodes}(d)| + |content(d)|].

\text{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
\text{O}(|nodes}(d)| + |content(d)|] time complexity.
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 → ... → AFROM → LCA → ATO → ... → 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
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:

![Table 1](image)

Datasets used for Test 2.

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

\[
\text{FILTER(UPDATE}(d^*, (from, to, TAG)), D_i) = d'_i. 
\]

**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 \(\text{UPDATE}(d^*, (from, to, TAG))\) then \(d'^*\) is a merger of \(d^* = \langle d_1, \ldots, d'_i, \ldots, d_k \rangle\) where the markup corresponding to TAG might be fragmented using glue attributes. Then \(\text{FILTER}(d'^*, D_i)\) removes all markup elements not in \(D_i\) and joins adjacent markup with glue attributes. Hence \(\text{FILTER(UPDATE}(d^*, (from, to, TAG)), D_i) = \text{FILTER}(d'^*, D_i) = d'_i. \)

\(\Box\)
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.
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 Library, 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


