Parsing Concurrent XML

Ionut E. Iacob[*]
Department of Computer Science
University of Kentucky
Lexington, KY, USA
eiaco0@cs.uky.edu

Alex Dekhtyar[†]
Department of Computer Science
University of Kentucky
Lexington, KY, USA
dekhtyar@cs.uky.edu

Kazuyo Kaneko
Department of Computer Science
University of Kentucky
Lexington, KY, USA
kkane2@pop.uky.edu

ABSTRACT

Concurrent markup hierarchies appear often in document-centric XML documents, as a result of different XML elements having overlapping scopes. They require significantly different approach to management and maintenance. Management of XML documents composed of concurrent markup has been mostly studied by the document processing community and has attracted attention of computer scientists only recently. In this paper we discuss the architecture of an XML parser for concurrent XML. This parser uses a GODDAG data structure in place of traditional DOM Tree to store concurrent markup on top of the document content and provides a DOM-like API that allows software developers of tools working with concurrent XML documents to use it instead of parsing each individual component with a traditional DOM XML parser. The paper describes the architecture of the parser, data structures and algorithms used and the DOM-like API.

Categories and Subject Descriptors

H.3.2 [**Information Storage and Retrieval**]: Information Storage; E.1 [**Data Structures**]: Trees; I.7 [**Computing Methodologies**]: Document and Text Processing—*Miscellaneous*

General Terms

Management, Algorithms

Keywords

Concurrent XML, overlapping markup, GODDAG, DOM

*Work supported, in part, by the NEH grant RZ-20887-02.
†Work supported, in part, by the NSF grant ITR-0219924 and the NSF grant ITR-0325063.

WIDM'04, November 12–13, 2004, Washington, DC, USA.

1. INTRODUCTION

Concurrent XML markup is almost inevitable in any serious text encoding endeavor, be it medieval English manuscripts [12], biblical texts [9], or any modern printed text that needs to be marked up [14]. Even the most simple document-centric markup involving sentence boundaries and physical line boundaries produces elements with overlapping scopes: sentences start and end at mid-line, preventing proper nesting of line and sentence markup.

This problem had been known to the humanities community for years, having originally been brought up in the context of SGML [13]. Proposed approaches to dealing with concurrent markup were mostly structural, most well-known of them being SGML's CONCUR syntax, and a variety of suggestions on how to use milestones (emtpy XML elements) and element fragmentation to avoid overlapping markup found in Text Encoding Initiative (TEI) Guidelines [14]. These approaches however all suffer from two major drawbacks: (a) reliance on human editors and stemming from it (b) hard-to-query XML.

Database research in the past few years has been mostly concentrated on data-centric XML, where the problem of concurrent markup does not occur. At the same time, the new wave of approaches to management of concurrent XML, pioneered by Durusau and O'Donnell [9, 8] started to rely implicitly on computer scientists to provide adequate software support.

In a nutshell, concurrent XML markup considered in this paper can be thought of as a collection of XML documents *sharing the same content* and the same root element. The documents are not independent, they are merely facets of the same complex encoding of the content, which would not yield a well-formed XML document if put together. At the same time, the editor[1] has to treat this entire collection of markup as a single document. To be able to do this, the editor must be helped by the software capable of processing and managing concurrent XML.

In our prior work, we have started addressing these needs. In [6, 7] we have formally introduced the notion of a *distributed XML document* over a concurrent XML hierarchy and considered the algorithms for *automatically* constructing single XML documents from distributed XML documents and vice versa. In [11] we have extended XPath to process path queries over distributed XML documents and showed that query evaluation in our Extended XPath re-

[1]The author of the encoding.

mains efficient. In [5] we have looked at data structures to store document-centric XML both in main memory and in secondary storage.

This paper addresses another aspect of processing of concurrent XML data: parsing. The attraction of XML is in the availability of standardized, powerful tools for dealing with it: XML DOM parsers take as input text representation of an XML document and produce a DOM tree - an internal model of an XML document that is suitable for the use of a wide range of software applications. To facilitate the use of DOM trees, a standard DOM API[4] is used in all XML DOM parsers.

We draw an immediate parallel between the state-of-the-art in XML processing, and the desired features of concurrent XML processors. Just as standard XML, distributed XML documents have a text representation: a collection of XML documents that share content and root element. Sperberg-McQueen and Huitfield proposed a data structure called GODDAG [15], that can serve as the DOM tree analog for concurrent XML (we have successfully used it as the underlying data model for Extended XPath in [11]). Finally, the ARCHWay project[12] at the University of Kentucky provides us with a wide array of application programs that require access to the concurrent XML documents in order to fully support the work of human editors on the preparation of electronic editions of medieval English manuscripts. The contributions of this paper are, thus, threefold:

- We introduce SACX, Simple API for Concurrent XML: an event-based parser that combines SAX events from the components of the distributed XML document into a single SAX stream.
- We introduce the GODDAG parser for concurrent XML. Built on top of SACX, it converts the event stream into a GODDAG data structure.
- We introduce GODDAG API, the programmer's interface to GODDAG. It includes all the standard features of DOM API. In addition, it provides some functionality, that is specific to the processing needs for distributed XML documents.

The rest of the paper is organized as follows. Section 2 briefly recaps the definitions of distributed XML [6] and GODDAG [15]. In Section 3 we introduce the parsing algorithms for the SACX and GODDAG parsers, and describe briefly the GODDAG API. Finally, Section 4 provides an initial evaluation of the performance of our GODDAG parser.

2. BACKGROUND

2.1 Concurrent XML

A concurrent XML hierarchy as defined in [6] is a collection of DTDs sharing the same root element. Using a *concurrent markup hierarchy* (CMH), a large, complex schema can be broken down into a number of smaller, schemas of lesser complexity. However, the most important benefit of using CMHs is the ability to define and use logical hierarchies of XML elements with no conflicts between markup tags inside the same hierarchy.

Before proceeding, we introduce some notation used throughout the paper. First, all XML document instances in this paper (which we refer to as *documents*) are considered to be well-formed, unless explicitly specified otherwise. For a DTD T we let $elements(T)$ be the set of *element type* names as they appear in Element Type Declarations in T[3]. For a document d we let $elements(d)$ be the set of *element types* (*tag names*) in d [3]. It follows that, a necessary condition for a document d to be valid[3] w.r.t. some DTD T is $elements(d) \subseteq elements(T)$.

For a document d we define functions $start, end$ (which describe the position of a node relative to the document textual content) as
$start, end : nodes(d) \rightarrow \{0, 1, \ldots, |string{-}value(d)|\}$
where, $\forall x \in nodes(d)$:
• $start(t)$ is the character position in $string{-}value(d)$ where $string{-}value(t)$ begins; if $string{-}value(t) = \epsilon$, then $start(t) = start(p)$ where $p \in nodes(d)$ is the first node (in reverse document order) that precedes t such that $string{-}value(p) \neq \epsilon$ or $start(t) = 0$ if no such node p precedes t;
• $end(t)$ is the character position in $string{-}value(d)$ before which $string{-}value(t)$ ends; if $string{-}value(t) = \epsilon$, then $end(t) = end(f)$ where $f \in nodes(d)$ is the first node (in document order) that follows t such that $string{-}value(f) \neq \epsilon$ or $end(t) = |string{-}value(d)|$ if no such node f follows t. For instance, $start(root(d)) = 0$ and $end(root(d)) = |string{-}value(d)|$.

DEFINITION 1. [6] *A* concurrent markup hierarchy CMH *is a tuple* $CMH = \langle \rho, \{T_1, T_2, ..., T_k\} \rangle$ *where:*
• ρ *is an XML element called the* root *of the hierarchy;*
• T_i, $i = \overline{1, k}$ *are DTDs such that:*
(i) $\forall 1 \leq j \leq k, i \neq j,\ elements(T_i) \cap elements(T_j) = \{\rho\}$;
(ii) $\forall t \in elements(T_i) - \{\rho\}$, ρ *is an ancestor of* t *in* T_i.

An example of concurrent markup hierarchy is shown in figure Figure 1 (top box).

DEFINITION 2. [6] *A* distributed XML document D *over a concurrent markup hierarchy* $CMH = \langle \rho, \{T_1, T_2, ..., T_k\} \rangle$ *is a collection of XML documents:* $D = \langle d_1, \ldots, d_k \rangle$ *where (i)* $(\forall 1 \leq i \leq k)$ d_i *is valid w.r.t.* T_i*; (ii)* $string{-}value(d_1) = string{-}value(d_2) = \ldots = string{-}value(d_k)$*, and (iii)* $root(d_1) = root(d_2) = \ldots = root(d_k) = \rho$.

We say that for a distributed document D*,* $string{-}value(D) = string{-}value(d_1)$ *and* $root(D) = root(d_1)$.

A *distributed XML document* (see Figure 1, bottom box)[2] allows us to distribute conflicting markup into separate documents. However, D is not an XML document itself, rather it is a *virtual* union of the markup contained in $d_1, \ldots, d_k$. The problem of creating well-formed XML document instances that incorporate all information in a *distributed document* has been addressed in [6]. The focus of this paper is on building the data structure representation of a distributed XML document, which is used for querying the distributed document [11].

Our next step is to define the abstract data model for distributed XML documents, which plays the same role as DOM trees do for regular XML. For a distributed XML document $D = \langle d_1, \ldots, d_k \rangle$, we will use set notation $d_i \in D$ to specify that d_i is a component document of D. Similarly, we will slightly abuse notation and write $D - d$ to represent a distributed XML document that consists of all components

[2]The text fragment is from Alfred the Great's Boethius manuscript [2].

T_1 = <!ELEMENT r (line)+>
<!ELEMENT line (#PCDATA)>

T_2 = <!ELEMENT r (vline)+>
<!ELEMENT vline (w)+>
<!ELEMENT w (#PCDATA)>

T_3 = <!ELEMENT r (#PCDATA|res)*>
<!ELEMENT res (#PCDATA)>

T_4 = <!ELEMENT r (#PCDATA|dmg)*>
<!ELEMENT dmg (#PCDATA)>

$\mathbf{CMH} = \langle \mathbf{r}, \{\mathbf{T_1}, \ldots, \mathbf{T_4}\}\rangle$

d_1 = "<r><line>gesceaftum unawendendne sin</line><line>gallice sibbe gecynde þa</line></r>"

d_2 = "<r><vline><w>gesceaftum</w><w>unawendendne</w></vline><vline><w>singallice</w>
<w>sibbe</w><w> gecynde</w></vline><vline><w>þa</w></vline></r>"

d_3 = "<r><res>gesceaftum una</res>wendendne s<res>in</res><res>gallice sibbe gecyn</res>de þa</r>"

d_4 = "<r>gesceaftum una<dmg>w</dmg>endendne singallice sibbe gecyn<dmg>de þa</dmg></r>"

$string{-}value(d_1) = .. = string{-}value(d_4)$ = "gesceaftum unawendendne singallice sibbe gecynde þa"

$\mathbf{D} = <\mathbf{d_1}, \ldots, \mathbf{d_4}>$

Figure 1: A concurrent XML hierarchy CMH, and a distributed XML document D.

of D except for d. We also let $nodes(D)$ denote the set $\cup_{i=1}^{k} nodes(d_i)$. Given a node $x \in nodes(D)$, we let $doc_D(x)$ denote the document $d \in D$, such that $x \in nodes(d)$. Given a string s, we denote by $|s|$ the *length* of the string (number of characters in s). We also let $substring(s, i_1, i_2)$ denote the substring of s from position i_1 *up to but not including position* i_2 (here positions start from 0 up to position $|s|-1$), and we let ϵ denote the *empty string*.

2.2 The GODDAG data structure

For representing a distributed XML document we use a *General Ordered-Descendant Directed Acyclic Graph (GODDAG)* data structure proposed in [15]. Informally, a GODDAG for a distributed XML document $D = \langle d_1, \ldots, d_k\rangle$ can be thought of as the graph that unites the DOM trees of individual components of D, by merging the root node and the text nodes. However, because of possible overlap in the scopes of XML elements from different component documents, GODDAGs will feature one more node type, that we call here *leaf node*, not found in DOM trees. In a GODDAG, leaf nodes are children of the text nodes, and they represent a consecutive sequence of content characters that is *not broken by an XML tag in* **any** of the components of the distributed XML document. While each CMH component will have its own text nodes in a GODDAG, the leaf nodes will be shared among all of them. Given a distributed XML document $D = \langle d_1, \ldots, d_k\rangle$, we can compute the set of leaf nodes using the following algorithm:

for each $d \in D$
  for each $t \in text{-}nodes(d)$
    $i = start(t)$
    while $i < end(t)$
      $m = min\{j \mid j > i \wedge \exists d \in D$
          $\exists x \in text{-}nodes(d)$
          $(j = start(x) \vee j = end(x))\}$
      create *leaf node* parented by t and
        with textual content $substring(S, i, m)$
      $i = m$

In other words, *leaf nodes* are obtained by projecting each *start tag* and *end tag* from all component documents of D on the $string{-}content(D)$, at corresponding positions, then taking largest contiguous sequences of content characters not separated by markup to be the scope of individual leaf nodes. For a distributed document D we let $leaf{-}nodes(D)$ represent the set of all *leaf nodes* in D and we extend the domain of functions $string{-}value$, $start$, and end over the $leaf{-}nodes(D)$ set. For *leaf nodes* these functions are defined in the same way as for *text nodes*. We define two new functions, $first{-}leaf, last{-}leaf : nodes(D) \rightarrow leaf{-}nodes(D)$. Given an element, or text node x, these functions return the leftmost and the rightmost (respectively) leaf nodes in the subtree of x. If $string{-}value(x) = \epsilon$, then $first{-}leaf(x), last{-}leaf(x)$ return the first following (respectively the first preceding), in reverse document order, *leaf node* for x (or NIL if such nodes do not exist). We enumerate below some useful properties of leaf nodes.

PROPOSITION 1. *Let $D = \langle d_1, \ldots, d_k\rangle$ be a distributed XML document.*

- *If $l \in leaf{-}nodes(D)$ then $|string{-}value(l)| > 0$.*
- *$string{-}value(D)$ is the concatenation of all $string{-}value(l)$, $l \in leaf{-}nodes(D)$ where the leaves l are taken in document order.*
- *$\forall d \in \{d_1, \ldots, d_k\}$, if $l \in leaf{-}nodes(D)$ then $\exists t \in text{-}nodes(d)$ such that $start(t) \leq start(l) < end(l) \leq end(t)$.*
- *$\forall d \in \{d_1, \ldots, d_k\}$, if $t \in text{-}nodes(d)$ then $\exists l \in leaf{-}nodes(D)$ such that $start(t) \leq start(l) < end(l) \leq end(t)$.*

DEFINITION 3. *Let $D = \langle d_1, \ldots, d_k\rangle$ be a distributed XML document. A GODDAG of D is a directed acyclic graph (N, E) where the sets of nodes N and edges E are defined as follows:*

- $N = \cup_{i=1}^{k}(tree{-}nodes(d_i) - \{root(d_i)\}) \cup leaf{-}nodes(D) \cup \{r\}$
- $E = \cup_{i=1}^{k}\{(r, x) \mid x \in tree{-}nodes(d_i) \wedge root(d_i) \text{ is the parent of } x\} \cup \cup_{i=1}^{k}\{(x, y) \mid x, y \in tree{-}nodes(d_i) - root(d_i) \wedge$

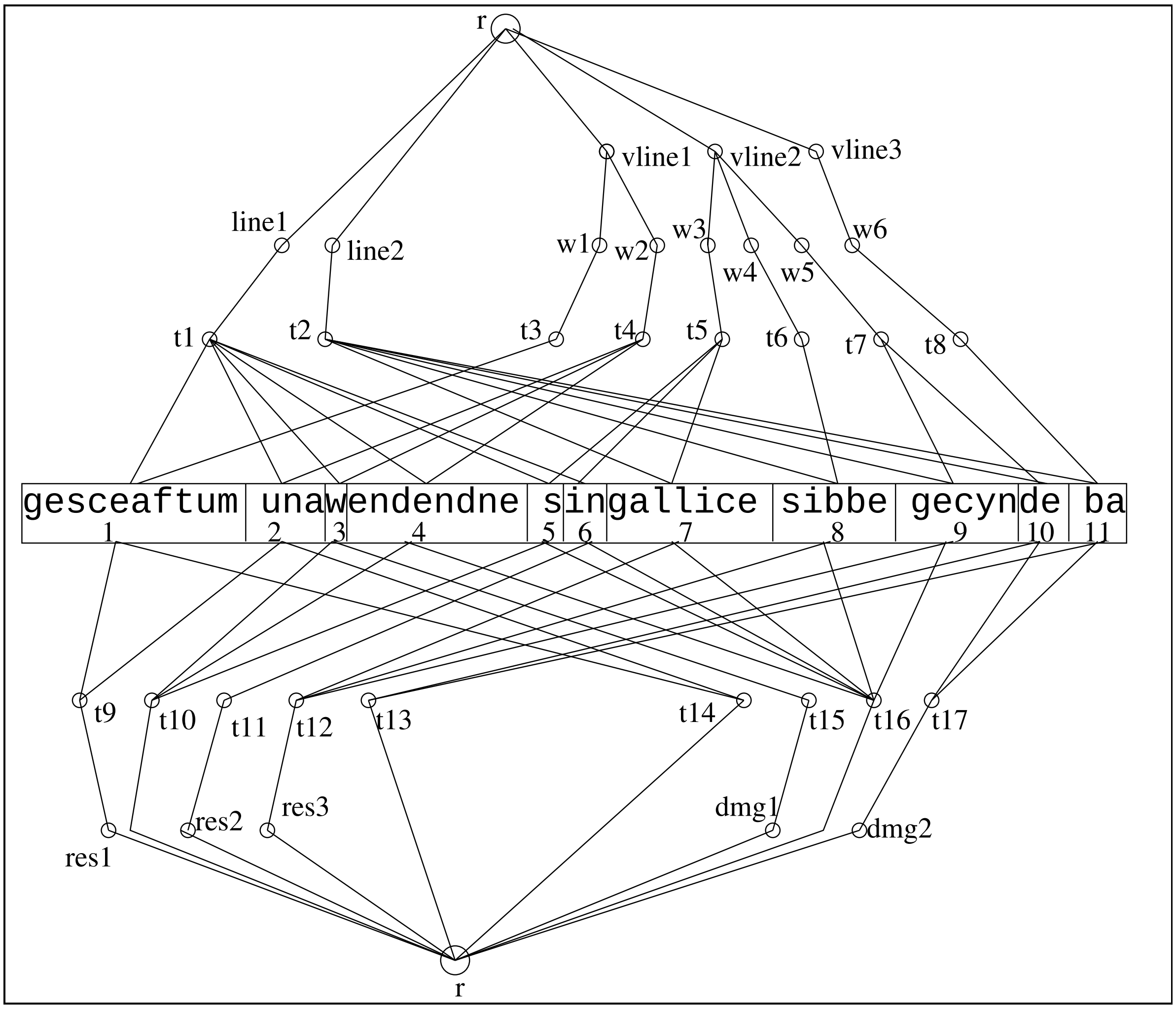

Figure 2: A GODDAG for the distributed document D in Figure 1

$$
\begin{aligned}
& x \textit{ is the parent of } y\} \cup \\
& \cup_{i=1}^{k}\{(x,y) | x \in text{-}nodes(d_i), \\
& \quad y \in leaf{-}nodes(D) \wedge \\
& \quad start(x) \leq start(y) < end(y) \leq end(x)\}
\end{aligned}
$$

A GODDAG of D, basically, joins at the root level and leaf level, of all tree models (DOM trees) of documents in D. Consequently, each node in $nodes(D)$ has $root(D)$ as an ancestor, and each leaf node in $leaf{-}nodes(D)$ has exactly k parents, one for each document in D. Hence, for a leaf $l \in leaf{-}nodes(D)$ we denote as $parent(d_i, l), 1 \leq i \leq k$, the parent of leaf l in $nodes(d_i)$.

The GODDAG of the distributed XML document in Figure 1 is given in Figure 2. Each node of the GODDAG in Figure 2 has a label (a number appended to the node name), solely for the purpose of ease of identification. *Leaf nodes* are represented as bounding boxes around the content substrings and are labelled with numbers 1, 2,...,11. We identify them as "l1",...,"l11". All other nodes are represented as circles. Text nodes are labelled t1, t2, ..., t17, other nodes are labelled by their *node name* and a number (to make distinction between multiple occurrences of the same node name). In order to make the figure clear, we draw the *root node* twice, at the top and bottom of the figure.

3. CONCURRENT XML PARSER

The concurrent XML markup management framework we propose is summarized in Figure 3. In [7] the management of concurrent XML hierarchies is described and in [11] a language and efficient algorithms for querying distributed XML documents represented by a GODDAG data structure are given. This section describes the algorithms for parsing the components of a distributed XML document: the SACX parser, the GODDAG parser, and the GODDAG API.

The concurrent XML parser (SACX) takes as an input a distributed XML document $D = \langle d_1, \ldots, d_k \rangle$, materialized as a set of distinct XML files $d_1, d_2, \ldots, d_n$ sharing the same root element and the same textual content (cf. Definition 2).

The SACX Parser.. Figure 4 shows the general architecture of SACX. The SACX architecture is based on a *pull SAX parser* architecture[3]: all input documents are parsed *in par-*

[3]A classical SAX parser implements a push model: as the parser advances in parsing the input, events are generated

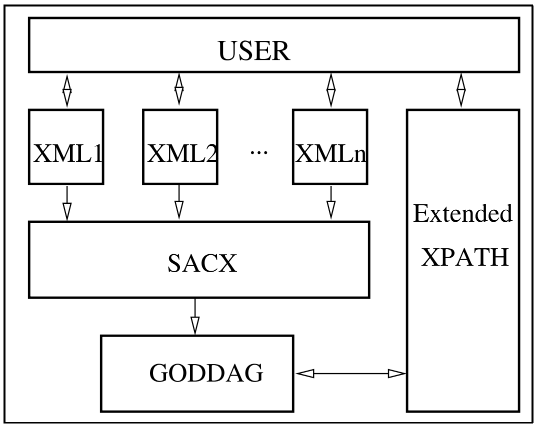

Figure 3: Parsing and querying concurrent XML

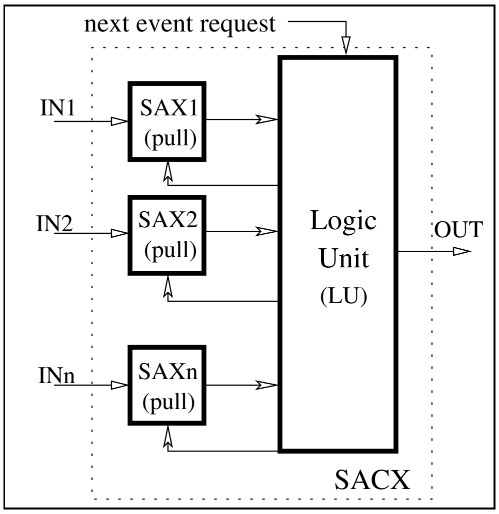

Figure 4: The SACX parser architecture

Input: $D = \langle d_1, \ldots, d_k \rangle$
Output: $SACX$ events

Create a $SACX$ parser
  for each $d_i \in \{d_1, d_2, \ldots, d_n\}$
    create a pull SAX parser SAX_i
    initialize document position $p_i \leftarrow 0$

  initialize global position $p \leftarrow 0$

next() //generates the next event
  while more to parse in $\{d_1, d_2, \ldots, d_n\}$
    for current document d_i in $\{d_1, d_2, \ldots, d_n\}$
      //"startElement", "endElement", and
      //"characters" events are generated here
      if $p_i = p$
        return $SAX_i.parse(d_i)$
      else
        move to the next document in $\{d_1, d_2, \ldots, d_n\}$
    $p' \leftarrow p$ //save the previous position
    $p \leftarrow min_{1 \leq i \leq n}\{p_i\}$
    return a "leaf" event, from p' to p

Figure 5: The SACX parser algorithm

No	DocID	Event type	Content
1	1	element	<r>
2	1	element	<line>
3	1	text	"gesceaftum ... sin"
4	2	element	<r>
5	2	element	<vline>
6	2	element	<w>
7	2	text	"gesceaftum"

Table 1: SACX events (fragment) of parsing the distributed XML document in Figure 1

allel in the sense that all events are generated for a given position in the input documents before moving on to the next position. The types of events generated by the SACX parser, at the external processor request, are as follows (represented as call-back functions):

`startDocument(docID)` – is generated, for each input document, before parsing the document content starts;

`endDocument(docID)` – is generated when parsing of the document identified by `docID` is finished;

`startElement(docID, position, tag)` – is generated by parsing a start-element `tag` in document `docID` at position `position`;

`endElement(docID, position, tag)` – similarly as for `startElement()`, but for an end-element markup;

`characters(docID, position, text)` – is generated for parsing a textual content `text` in document `docID` at position `position`;

`leaf(start, end)` – is generated right after the parser finishes parsing *all* elements and text starting at position `start` and moves the current parsing position at position `end` in the input documents.

The SACX algorithm is given in Figure 5. All positions in the input documents, where start tag, end tag, or text are starting, are scanned in increasing order; for each position all events, corresponding to the elements starting at the respective position, are generated. A "leaf" event is generated each time the scan is moves to the next position. This ensures that each "leaf" event is produced *after* all "character" events containing the start and end positions of the "leaf" event are produced.

Table 1 shows an excerpt of the sequence of SACX events for parsing the distributed XML document in Figure 1. The example contains the events generated while parsing completely all tokens at position 0 in the documents d_1 and d_2. After generating the corresponding events at position 0 in the documents d_3 and d_4, a "leaf" event is generated for the range of the word "gesceaftum" (that is, 0 – 10) and 10 becomes the next scanning position.

The GODDAG Parser and API.. The events generated by the SACX parser are used by the GODDAG parser in

(like starting and ending tags) and pushed to an external processor (using call-back functions). Except for call-back functions, the parser is in control from the beginning to the end of parsing. A pull parser model is based on the external processor control: the next event is generated at the external processor request.

```
interface IDDNode {
  IDDNodeTypes ROOT, ELEMENT, TEXT, LEAF

  //DOM specific fields
  IDDNode parent
  IDDNode firstChild
  IDDNode previousSibling
  IDDNode nextSibling

  //GODDAG specific fields
    //LEAF node specific
  IDDNodeHashtable parentDD
    //ROOT node specific
  IDDNodeHashtable firstChildDD
  integer start
  integer end

  //GODDAG specific methods
  setParentDD(IDDNodeHashtable parent, docID)
  IDDNodeHashtable getParentDD(docID)
  setFirstChildDD(IDDNodeHashtable parent, docID)
  IDDNodeHashtable getFirstChildDD(docID)
          ...........
}
```

Figure 6: The GODDAG node API (fragment)

creating the data structure. A GODDAG data structure extends the standard DOM [4] in the following ways:

(i) the root node has more that one "first child" node: there is one first child node in each hierarchy;

(ii) there is a new type of node, "leaf" node, which is a child of a text node;

(iii) a leaf node has multiple parent nodes, a parent node within each hierarchy;

(iv) each node contains starting and ending position information.

Property (iii) gives the fundamental difference between DOM [4] and GODDAG: the former is a tree while the latter is a graph. A GODDAG data structure is more formally described by an abstraction of the graph node data structure, the `IDDNode` interface (distributed document node interface: see Figure 6). The data structure fields and methods names are rather verbose. There is a new node type (LEAF) and the data fields and methods specific to GODDAGS's root and leaf nodes allow navigation between distributed document's components.

As exemplified in Figure 2 a GODDAG is a union of DOM trees (one tree for each component of the distributed document) united by the root node and the leaf nodes. The root node and the leaf nodes are bridges between individual tree structures and therefore they play an essential role in navigating from one document structure to another. It is of implementation choice how fast to navigate from a given node N in a document structure to the leaf nodes it spans. One option is for N to maintain a pointer to the first leaf on N. This would give $O(1)$ access to the leaf and from there the navigational paths to the other documents structures is open. The price of this option would be an expensive structure for updates. Another option would be to navigate through a path from N down to its first leaf node. This structure would be easier to update but the navigation of the GODDAG structure may be slower.

```
Input: D = ⟨d_1, ..., d_k⟩
Output: GODDAG data structure

  create a SACX parser on input D = ⟨d_1, ..., d_k⟩
  event = SACX.next()   //first event request
  create the GODDAG root out of event

  for each d in {d_1, ..., d_n}
    curNode_d ← root
    prevNode_d ← NULL
    stack_d.push(curNode_d)

  while SACX has more to parse
    event = SACX.next()   //next event request
    d is the document that has generated event

    if event is of "startElement" type
      node ← new GODDAG element node
      stack_d.push(node), node.setParent(curNode_d)
      curNode_d.setChild(node)
      if prevNode_d ≠ NULL
        prevNode_d.setNextSibling(node)
        node.setPreviousSibling(prevNode_d)
      prevNode_d ← NULL, curNode_D ← node
    else if event is of "endElement" type
      prevNode_d ← curNode_d
      curNode_d ← stack_d.pop()
    else if event is of "characters" type
      node ← new GODDAG text node
      node.setParent(curNode_d)
      curNode_d.setChild(node), prevNode_d ← node
    else if event is of "leaf" type
      node ← new GODDAG leaf node
      for each x in {d_1, ..., d_n}
        if prevNode_x is a text node
          node.setParent(x, prevNode_x)
          prevNode_x.setChild(node)

  return root
```

Figure 7: The GODDAGParser algorithm

The algorithm for the parser for the GODDAG structure is shown in Figure 7. It takes as input the output stream of the SACX parser and outputs the GODDAG structure. Informally, the GODDAG is built by concurrent construction of all of its DOM components. Given an element event from the SACX stream, the parser traverses the current state of the DOM tree for the corresponding component of the distributed XML document and sets up the appropriate element node there. When the GODDAG parser observes a *leaf* event, it creates a new leaf node and determines its parents in each component of the distributed XML document. For simplicity, the algorithm skips the details of checking whether or not the first child node for a given node was set. The method `setChild()` should be interpreted as setting the first child of the respective node if the first child node was not already set.

We note here that when the GODDAG parser is run on a distributed XML document that consists of only one component, its output will, virtually, be the DOM tree for that component. More formally,

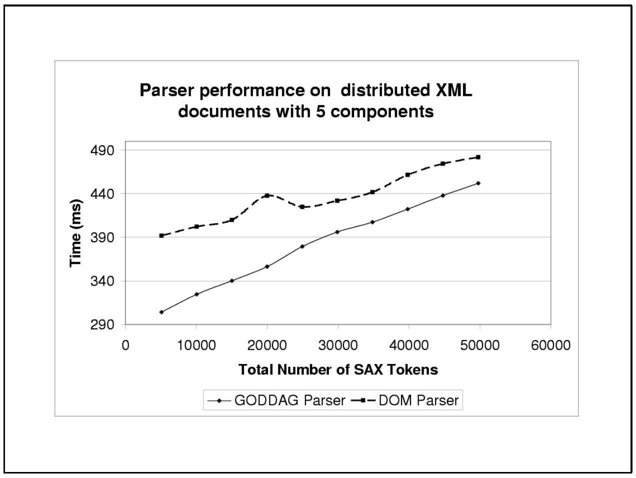

Figure 8: GODDAG parser performances (1)

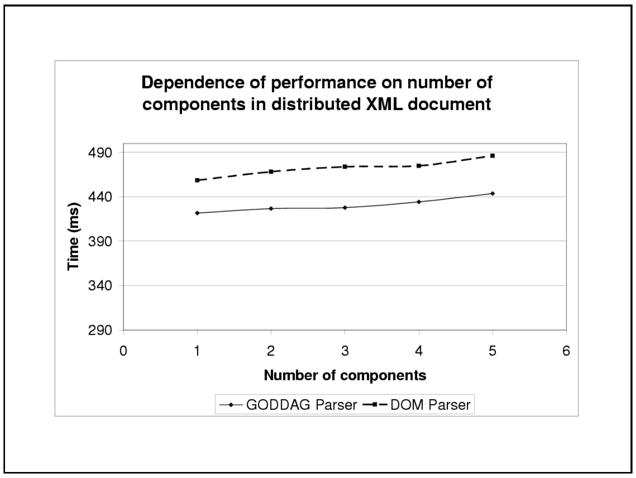

Figure 9: GODDAG parser performances (2)

PROPOSITION 2. *Let $D = \{d\}$ be a distributed XML document with a single component. Let T be a DOM tree for d and let G be the output of the GODDAG parser described in Figure 7. Then, the following holds:*

- *Each text node in G has a single leaf node child.*
- *Let G' be the graph obtained from G by merging each text node with its child leaf node. Then $G' \equiv T$.*

A straightforward analysis of the algorithm in Figure 7 yields the following complexity result:

THEOREM 1. *The time complexity of constructing a GODDAG data structure for a distributed XML document $D = \langle d_1, \ldots, d_k \rangle$ is $O(|d_1| + \ldots + |d_n|)$.*

4. EXPERIMENTAL RESULTS

In this section we describe some preliminary experiments on our implementations of SACX and GODDAG parsers. We have implemented SACX and GODDAG parsers in Java using Xerces Java 2.6.2 XML SAX parser [1] to generate individual SAX streams for the SACX parser.

In our experiments, we have set out to compare the performance of the GODDAG parser to the performance of a standard XML DOM parser on a comparable workload. The dependent variable in our experiments was time. We used two independent variables: size of the distributed XML document and number of components in a distributed XML document. Size was measured in terms of the total number of SAX events (tokens) generated during parsing. For the study of the dependence of time on the document size, we have generated a total of 50 distributed XML documents (we used the same dataset as in [7]), each document consisting of five components. The document sizes ranged from 5000 tokens up to 50000 tokens, with five documents for each size[4]. To test the dependence of performance on the number of components we have generated 25 distributed XML documents of size 50,000 tokens, five documents for each of the number of components from one to five. The individual component sizes were smaller with the increase in the number of components, but the overall "workload" was kept at 50,000 tokens.

As the baseline comparison, we have chosen to use the work of Xerces Java 2.6.2 DOM parser on the same distributed XML documents. Given a distributed XML document, the DOM parser was run for each of its components to produce a DOM tree. The goal of the experiment was *not* to show that the GODDAG parser outperforms the DOM parser – such statement is not very meaningful given different nature of the outputs generated. The objective of the study is to show that the GODDAG parser can be used efficiently by the application programmers to parse and provide access to distributed XML documents. It is, thus, sufficient for us to show that the time it takes the GODDAG parser to produce a GODDAG for a distributed XML is comparable, in general, to the time a standard DOM parser spends on similar workloads (where workload is measured in the number of SAX tokens processed).

The experiments had been conducted on a Dell Optiplex GX 240 computer with a Pentium IV 1.2 Mhz processor, 1 Gb of RAM running Linux Operating System. Some of the results obtained are shown in Figures 8 and 9.

Figure 8 shows the dependence of the performance of the parsers on the size of the distributed XML documents. The results shown are for the 5-component distributed XML documents. Each point on the graph represents the averages of size and time for five documents. As seen from the graph, GODDAG parser is somewhat faster than the DOM parser, with the difference in the performance shrinking as the size of the XML documents grows. We attribute most of the difference in performance to two factors:(a) the DOM parser experiment involved five independent calls to the DOM parser, while the GODDAG parser experiment involved a single call and (b) the GODDAG parser implementation was "light" - it did not include complete DOM functionality, concentrating only on XML element support. While XML documents used in the tests contained only XML elements (no attributes, processing instructions etc), Xerces DOM parser might still have taken time to check for the presence of those features in the documents.

Figure 9 shows the dependence of the performance of the

[4]The actual document sizes varied slightly, and have been averaged over the five documents in the graphs.

parsers on the number of components. Both DOM tree and GODDAG parsers show exactly the same behavior as the number of components of distributed XML documents rises from one to five, while the workload remains at 50,000 tokens: a slight increase in the processing time.

Based on the two experiments conducted, we can conclude that the developed GODDAG parser is efficient enough to be used as the back-end for processing distributed XML in software applications and involve real-time communication with users.

5. CONCLUSIONS

The main objective for our research on management of concurrent XML markup is to develop approaches and build tools for authoring, storage, processing, querying and transformation of complex document-centric XML encodings that occur in numerous humanities (and not only) projects. This paper addresses the heart of our endeavor: the translation of distributed XML documents representing concurrent markup into a an internal data structure and the appropriate API for it to be used by applications programmers. We have chosen our approach to parallel that of standard XML, by providing concurrent XML analogs for SAX and DOM parsers and the DOM API. Our experiments show, that this approach leads to software that is efficient and can be used by software applications in the same way DOM parsers are used.

The work on the full implementation of the GODDAG parser is currently underway. At the same time, the prototype parser described here has already been successfully used to support a document-centric XML editor written for the ARCHWay[12] and Electronic Boethius [10] projects. In [11] we have proposed an extension of XPath for dealing with path expressions over GODDAG. Implementation of the Extended XPath processor on top of the GODDAG API is also currently underway.

6. REFERENCES

[1] Xerces2 Java Parser 2.6.2. http://xml.apache.org/xerces2-j/, 2004. Apache XML project.

[2] Alfred. Boethius. British Library MS Cotton Otho A. vi. Manuscript, folio 36v.

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

[4] M. Champion, S. Byrne, G. Nicol, and L. Wood(Eds.). Document Object Model (DOM) Level 1 Specification. http://www.w3.org/TR/REC-DOM-Level-1/, Oct 1998. World Wide Web Consortium Recommendation, REC-DOM-Level-1-19981001.

[5] A. Dekhtyar, I. Iacob, J. Jarmczyk, K. Kiernan, N. Moore, and D. Porter. Database support for image-based Electronic Editions. In *Proc. Workshop on Multimedia Information Systems (MIS'04)*, pages 147–156, 2004.

[6] A. Dekhtyar and I. E. Iacob. A Framework for Management of Concurrent XML Markup. In *International Workshop on XML Schema and Data Management (XSDM'03)*, pages 311–322. LNCS, 2003.

[7] A. Dekhtyar and I. E. Iacob. A Framework for Management of Concurrent XML Markup. *Data and Knowledge Engineering*, 2004. accepted.

[8] P. Durusau and M. O'Donnel. Declaring Trees: The Future of the Evolution of Markup? In *Proc. Conference on Extreme Markup Languages*, 2002.

[9] P. Durusau and M. B. O'Donnell. Concurrent Markup for XML Documents. In *Proc. XML Europe*, May 2002.

[10] K. Hawley and K. Kiernan. An image-based electronic edition of alfred the great's old english version of boethius's consolation of philosophy. In *Proc., Joint International Conference of the Association for Literary and Linguistic Computing and the Association for Computers and the Humanities (ALLC/ACH)*, pages 91–96, 2003.

[11] I. E. Iacob, A. Dekhtyar, and W. Zhao. XPath Extension for Querying Concurrent XML Markup. Technical Report TR 394-04, University of Kentucky, Department of Computer Science, February 2004. http://www.cs.uky.edu/~dekhtyar/publications/TR394-04.ps.

[12] K. Kiernan, J. Jaromczyk, A. Dekhtyar, D. Porter, K. Hawley, S. Bodapati, and I. Iacob. The ARCHway project: Architecture for research in computing for humanities through research, teaching, and learning. *Literary and Linguistic Computing*, 2004. forthcoming.

[13] A. Renear, E. Mylonas, and D. Durand. Refining our notion of what text really is: The problem of overlapping hierarchies. *Research in Humanities Computing*, 1993. N. Ide and S. Hockey, (Eds.).

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

[15] C. M. Sperberg-McQueen and C. Huitfeldt. GODDAG: A Data Structure for Overlapping Hierarchies, Sept. 2000. Early draft presented at the ACH-ALLC Conference in Charlottesville, June 1999.