ELECTRONIC PUBLISHING, VOL. 8(4), 295–317 (DECEMBER 1995)

File format for documents containing both logical structures and layout structures

MAKOTO MURATA

Fuji Xerox Information Systems Co., Ltd.
KSP 9A7, 2-1 Sakado 3-Chome,
Takatsu-ku, Kawasaki-shi,
Kanagawa-ken 213, Japan

SUMMARY

A file format for documents containing both the logical and layout structures is presented. Unlike CONCUR of SGML, this format can represent a large class of WYSIWYG documents, such as documents containing footnotes. The key idea is to untangle the logical-layout relationship by introducing embedding nodes and mold nodes. Embedding nodes are nodes representing logical-layout correspondences; mold nodes are temporary dummy nodes, which will later be replaced by layout nodes. Once the logical-layout relationship is untangled, it becomes possible to represent a document by a sequential data stream which can be readily stored in a file. This file format has the following advantages: First, it is compact. No leaf nodes are duplicated; tags for embedding nodes and mold nodes do not require much space; the text formatting output can be omitted. Second, the depth-first traversal of logical structures and that of layout structures are efficient and easy to implement. Files are sequentially read only once, and neither logical structures nor layout structures need to be copied into the main memory. Third, the logical-layout correspondences are explicitly represented by tags for embedding nodes.

KEY WORDS Structured document Logical structure Layout structure File format Stream CONCUR

1 INTRODUCTION

This paper shows a file format for documents containing both the logical and layout structures (hereafter *formatted-processable* documents). Such a file format is highly important for at least two reasons.

First, in the area of information retrieval, queries based on both logical and layout characteristics (e.g., ‘retrieve technical articles that have mathematical expressions in the first page’) have been studied [1,2]. To provide such search capabilities, an information retrieval system [2] reconstructs layout structures from Postscript files and further reconstructs some logical ionformation from these layout structures. Search is then performed with respect to the reconstructed logical information and layout structures. However, if a file format for formatted-processable documents is established, documents containing both the logical and layout structures become readily available and thus information retrieval systems can take full advantage of the logical and layout characteristics.

Second, researchers in humanities are often concerned with both the logical and layout

CCC 0894–3982/95/040295–23
Received 20 Februrary 1995
Revised 16 November 1995

characteristics of manuscripts [1]. For example, a chronicle is logically divided into entries for years. Scholars compare different manuscripts of the same chronicle with respect to these logical entries (rather than pages), since different manuscripts have the same entry on different pages. Thus, electronic representation of chronicles must capture their logical structures. On the other hand, a manuscript is also divided into pages and lines. Linguists cite examples of grammatical structures with respect to pages and lines (rather than logical entries), since they provide precise reference points. Thus, layout structures of chronicles must also be captured. Furthermore, to interpret the text properly, textologists need to know where lines and pages end, since the copying process often introduces mistakes at the end of lines and pages. Thus, the relationship of logical and layout structures of chronicles must be captured as well. Therefore, a file format for formatted-processable documents is strongly required. The TEI project [3] has been pursuing such a format.

SGML [4] provides a mechanism called CONCUR for representing formatted-processable documents [2]. However, CONCUR can handle very simple documents only. It cannot handle documents having footnotes, for example. On the other hand, our file format can handle a large class of WYSIWYG documents. Nevertheless, our file format is similar to CONCUR when applied to very simple documents. In this sense, our file format is an extension of CONCUR. However, since the present work is based on document formatting models and provides useful access algorithms, it should not be considered as merely an extension but rather as a theory behind CONCUR.

The rest of this paper is organized as follows. Section 2 describes the common characteristics of many document models and clarifies the relationship of the logical structure and the layout structure. Section 3 introduces embedding nodes and mold nodes, which are key concepts of this work. Section 4 shows our file format as well as an example document representation. Section 5 shows access algorithms. Section 6 further introduces the omission of text formatting output so as to compact files. Section 7 shows a comparison to related work and suggests some extensions. Section 8 shows the conclusion.

2 DOCUMENT MODEL

Although there are a variety of structured document models [5], basic concepts are similar. The logical characteristics of a document is represented by a tree whose nodes have attributes. This tree is called a *logical structure*. A leaf node in a logical structure is a visible object such as a character, bitmap, graphics frame, etc. Given a logical structure and some associated layout control parameters, document formatters create a *layout structure*. This process is called *formatting*. The layout structure is an attributed tree, which represents the layout characteristics of the document. A node of a layout structure represents a rectangular area. Again, a leaf node in a layout structure is a visible object such as a character, bitmap, graphics frame, etc.

A node A in a logical (layout) structure is *subordinate to* another node B and B is *superior* to A if we can reach from B to A by descending the structure. Node A *precedes* B if A occurs earlier than B in the preorder and B is not subordinate to A.

For our purpose we further study the relationship of the logical structure and the layout

[1] Private communications with David Birnbaum.

[2] CONCUR was originally introduced into SGML as a mechanism for representing formatted-processable documents, since a competing standard, namely ODA, already had a mechanism called FPDA (see Section 7). People later started to use CONCUR for other purposes such as representing multiple logical structures.

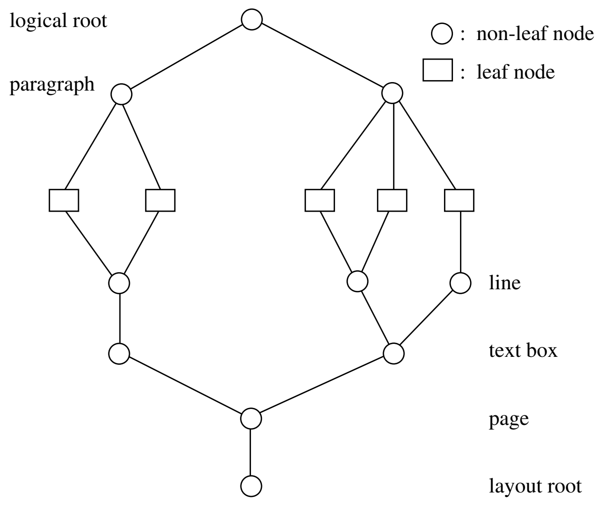

Figure 1. Leaf nodes are shared by logical structure (upper half) and layout structure (lower half)

structure of a document. Although different document formatting models (Interscript[6], DSSSL[7], ODA[8], and others[9,10,11]) differ on many points, after formatting is done, the relationship between the input logical structure and the output layout structure is similar in any formatting model.

2.1 Sharing of leaf nodes

The logical structure and layout structures share leaf nodes; that is, every leaf node belongs to a logical node and a layout node (Figure 1). For example, a character in a document belongs to a logical node such as a paragraph node and also belongs to a layout node such as a line node. A bitmap in a document belongs to a logical node such as a logical artwork node and also belongs to a layout such as a layout artwork node.

Exceptions are duplication, removal, and addition. *Duplication:* what occurs once in the logical structure occurs more than once in the layout structure (e.g., running headers). *Removal:* what occurs in the logical structure does not occur in the layout structure (e.g., omitted annotations). *Addition:* what does not occur in the logical structure occurs in the layout structure (e.g., page numbers). We will discuss these exceptions in Section 7.

2.2 Correspondences

A *layout template* can be specified for a logical node. A document formatter generates a sequence of layout subtrees from this layout template, and lays out the logical subtree rooted by that logical node into the layout subtrees. The logical node is said to *correspond* to the root layout nodes of these layout subtrees, and vice versa. The sequence of these layout nodes is called the *counterpart* of the logical node.

Here are some examples of correspondences. A logical structure is (often implicitly) constrained to be laid out into a layout structure. The logical root corresponds to the layout root; the counterpart of the logical root is the layout root. A paragraph is constrained to

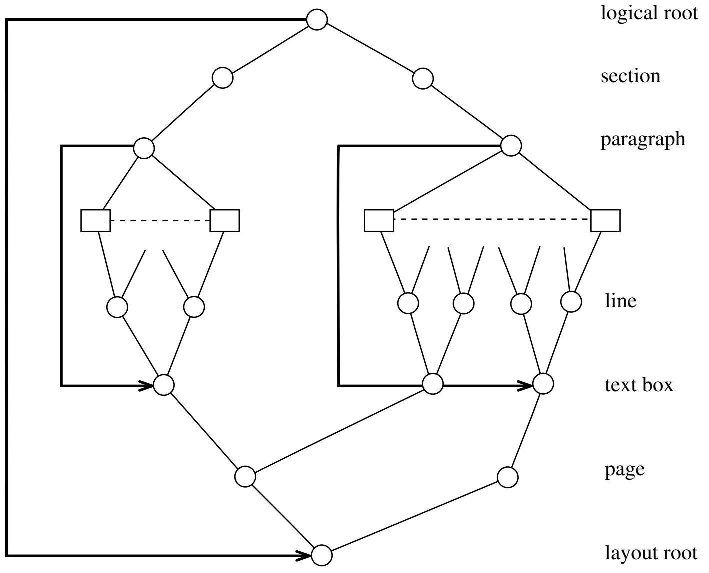

Figure 2. The logical root corresponds to the layout root; paragraphs, to text boxes

be laid out into a sequence of text boxes. The paragraph node corresponds to text boxes; the counterpart of the paragraph node is the sequence of these text boxes. (Figure 2 shows a document containing paragraphs.) A chapter may be constrained to be laid out into a sequence of page nodes; then, this chapter node corresponds to page nodes. A logical table may be constrained to be laid out into a sequence of layout tables (in case that the page break occurs); then, the table logical node corresponds to table layout nodes. A chapter header may be constrained to be laid out into a chapter header area; then, the chapter header logical node corresponds to a chapter header layout node. (Figure 3 shows a document containing chapters, chapter headers, sections, tables, and paragraphs.)

Observe that some logical nodes do not correspond to any layout nodes. For example, a section node does not correspond to any layout nodes, unless this section node has an associated layout template.

Assume that logical nodes *G1* and *G2* corresponds to layout nodes and that *G1* is subordinate to *G2*. Then, the counterpart of *G1* is subordinate to that of *G2*; to be precise, any of the layout nodes corresponding to *G1* is subordinate to one of those corresponding to *G2*. (We later consider an exception called break-out.) In the above example, a paragraph node, a chapter header logical node, and a table logical node are subordinate to a chapter node; the corresponding text boxes, chapter header layout nodes, and table layout nodes are subordinate to the corresponding pages. The chapter node in turn is subordinate to the logical root; the corresponding pages are subordinate to the layout root.

2.3 Lowest-level correspondences and leaf nodes

Correspondences without subordinate correspondences are said to be at the *lowest-level*. To be precise, the correspondence between a logical node and layout nodes is at the lowest-level if none of the subordinate logical nodes correspond to layout nodes. For example, the

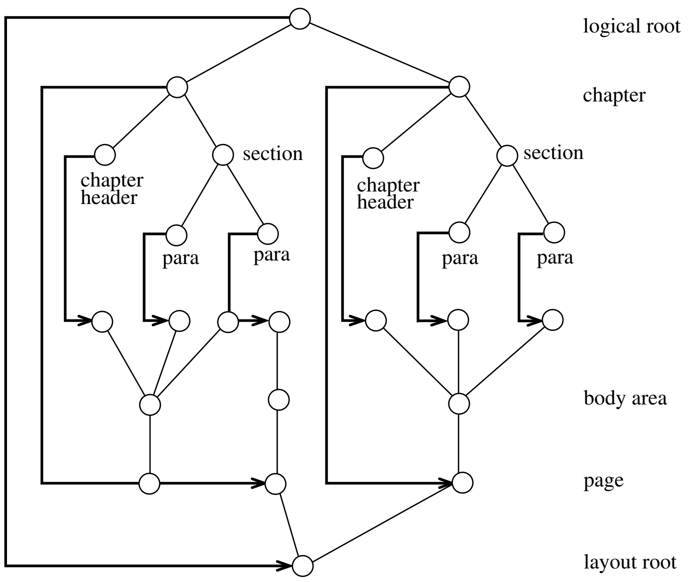

Figure 3. The logical root, chapters, chapter headers, sections, tables, and paragraphs correspond to layout nodes. Subordinates of these logical nodes and layout nodes are omitted

correspondence between a paragraph and text boxes is at the lowest-level, since subordinates of paragraphs (e.g., sentence logical nodes) do not corresond to layout nodes.

If we ascend a logical tree from a leaf node, we first encounter a lowest-level correspondence and then encounter higher-level correspondences: we never encounter a higher-level correspondence first. In other words, if a logical node *G* corresponds to layout nodes and this correspondence is not at the lowest-level, leaf nodes can be subordinate to *G* only through logical nodes having counterparts. In the above example, correspondences between paragraphs and text boxes are at the lowest-level; leaf nodes can be directly subordinate to paragraph nodes. The correspondences between chapter nodes and pages are not at the lowest-level; leaf nodes can be subordinate to chapter nodes only through paragraph nodes, chapter header logical nodes, or table logical nodes.

2.4 Reordering

Formatting often causes *reordering*. That is, the order in which logical nodes occur in the logical structure may be different from the order in which the corresponding layout nodes occur in the layout structure.

For example, consider a document whose logical structure consists of a logical root with paragraph nodes and footnote nodes as subordinates. Paragraph nodes are laid out in the body area of the page, and footnote nodes are laid out in the footnote area. To be precise, text boxes corresponding to paragraph nodes are subordinate to body area (layout) nodes. Those corresponding to footnote nodes are subordinate to footnote area (layout) nodes. If a paragraph *G1* precedes another paragraph *G2* in the logical structure, the counterpart of *G1* also precedes that of *G2*. Otherwise, the text appears out of order. In other words, as long as only paragraph nodes are concerned, there is no reordering. The same thing also

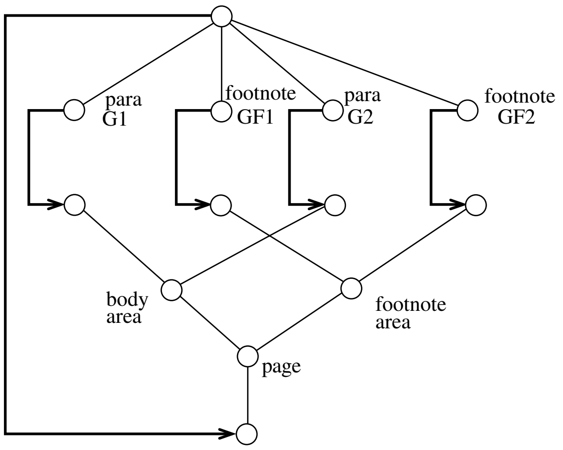

Figure 4. Reordering example

applies to footnotes: if a footnote *GF1* precedes another footnote *GF2*, the counterpart of *GF1* also precedes that of *GF2*.

However, reordering may arise when both paragraph nodes and footnote nodes are used with each other. Consider a paragraph *G1*, a footnote *GF1*, a paragraph *G2*, and a footnote *GF2*, which occur in the logical structure in this order (Figure 4). Assume that *G1* and *G2* are laid out in the body area of the first page, and that *GF1* and *GF2* are laid out in the footnote area of the same page. In the layout structure, the counterpart of *G1* precedes that of *G2*, and the counterpart of *GF1* precedes that of *GF2*. Since the body area layout node precedes the footnote area node, the occurrence order in the layout structure is *G1 G2 GF1 GF2*, and is thus different from the original occurrence order in the logical structure (*G1 GF1 G2 GF2*).

Similar reordering also arises in synchronized bilingual documents. Paragraphs in one language are laid out into right columns and those in the second language are laid out into left columns. Assume that a paragraph *G1* precedes another paragraph *G2* in the logical structure. If *G1* and *G2* are in the same language, the counterpart of *G1* precedes that of *G2*. If *G1* and *G2* are not in the same language, this is not necessarily the case.

Let us generalize this observation. If a logical node *G* corresponds to layout nodes and this correspondence is not at the lowest-level, there is more than one *stream* below *G*. Each stream is a sequence of logical nodes that correspond to layout nodes. Which stream a logical node pertains to is specified by the type or an attribute of that node. There is no reordering as far as logical nodes of the same stream are concerned. That is, if two logical nodes *G1* and *G2* pertain to the same stream and *G1* precedes *G2* in the logical structure, the counterpart of *G1* precedes that of *G2*. Reordering arises when logical nodes pertaining to different streams are involved. That is, provided that *G1* and *G2* pertain to different streams, even if *G1* precedes *G2* in the logical structure, the counterpart of *G1* does not necessarily precede that of *G2*.

Exceptions are bibliographies, indexes, and glossaries, where their entries are sorted in an alphabetical order, for example, and thus the order in which they apppear in the logical structure is irrelevant. We will discuss these exceptions in Section 7.

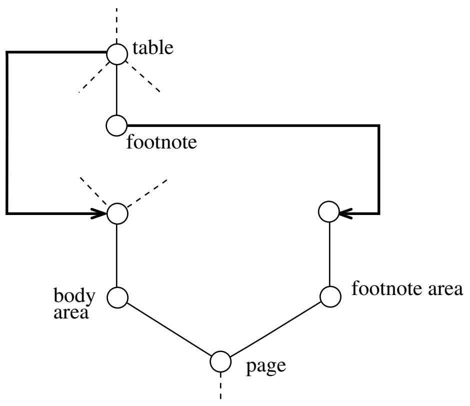

Figure 5. Break-out example

2.5 Break-out

Footnotes and their variations (marginal notes, etc.) cause break-out [3] [11]. Assume that a table logical node has a subordinate footnote node, which is laid out into the footnote area (Figure 5). The table logical node corresponds to a table layout node, and the footnote node corresponds to a footnote text box. The footnote text box is *not* subordinate to the table layout node but is subordinate to the footnote area node. The footnote node is said to *break out* from the table logical node. The footnote node does not pertain to streams within the table logical node. Rather, it pertains to a stream of a further superior logical node (e.g., a chapter node), and is laid out in conjunction with other footnotes. A footnote within a paragraph also breaks out: the footnote text boxes corresponding to the footnote node are not subordinate to the text boxes corresponding to the paragraph node. Figure 6 shows a document that has two paragraphs *G1* and *G2*, which in turn have footnotes *GF1* and *GF2*, respectively. Note that these footnotes cause both break-out and reordering.

We assume that no footnote notes (including variations) exist under other footnote nodes, because no accepted writing styles permit footnotes references within footnotes. This assumption simplifies the procedure for the depth-first traversal of layout structures (see 5.2).

Having introduced break-out, we have to revise the definition of *lowest-level*. The correspondence between a logical node and layout nodes is said to be at the *lowest-level* if none of the subordinate logical nodes pertain to streams under that logical node. In the above example, the correspondence between *G1* and its counterpart are at the lowest-level, because *GF1* does not pertain to a stream under *G1*. Therefore, characters can be subordinate to *G1* without intervening logical nodes.

3 EMBEDDING NODES AND MOLD NODES

In preparation, we untangle the logical and layout structures by introducing embedding nodes and mold nodes. For each correspondence between a logical node and a sequence of layout nodes, we introduce an embedding node and a sequence of mold nodes.

An *embedding node* (named after embedding [10]) represents a correspondence, and references a logical node and a sequence of layout nodes. An embedding node has attributes

[3] In Interscript [6] break-out is called left-over.

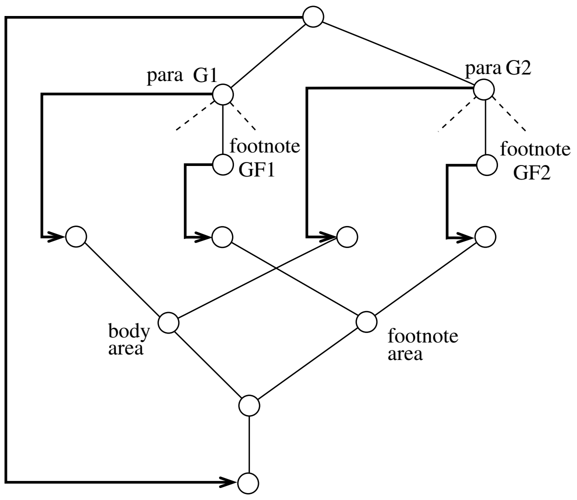

Figure 6. Footnotes within paragraphs cause break-out and reordering

‘stream’ and ‘subordinate_streams’. The ‘stream’ attribute specifies the name of the stream of the referenced logical node. The attribute ‘subordinate_streams’ specifies the names of the streams under the referenced logical node. Embedding nodes intervene between logical nodes: the logical nodes referenced by embedding nodes are no longer referenced by their original superior logical nodes; instead, these superior nodes now reference the embedding nodes.

A *mold node* [6] is a temporary dummy node, which will later be replaced by a layout node. The layout node that will replace a mold node is called its *prime node*. A mold node is introduced for each of the layout nodes (except the layout root) that correspond to logical nodes. Each mold node has an attribute ‘stream’. This attribute specifies the name of the stream of the logical node that corresponds to the prime node of the mold node. With the introduction of mold nodes, the layout structure is fragmented: the layout nodes that are prime nodes of mold nodes are no longer referenced by their original superior layout nodes; instead, these superior nodes now reference the mold nodes.

Having introduced embedding nodes and mold nodes, a document can be considered as a hierarchy of embedding nodes. Each embedding node references a logical subtree and a sequence of layout subtrees. This logical subtree in turn references subordinate embedding nodes.

Observe that mold nodes and embedding nodes temporarily dissolve reordering and break-out. They are implicitly represented by pairs of mold nodes and layout nodes.

Figure 7 shows embedding nodes and mold nodes introduced into the document shown in Figure 6. The logical root, paragraph *G1*, paragraph *G2*, footnote *GF1*, and footnote *GF2* correspond to layout nodes. Embedding nodes *E0*, *E1*, *E2*, *EF1*, and *EF2* are introduced for these correspondences. *E0* references the logical root and the layout root. *E1* references *G1* and the corresponding text box. And so forth. Note that the logical root references *G1* and *G2* through *E1* and *E2*, respectively. *G1* references *GF1* through *EF1*; *G2* references *GF2* through *EF2*. The prime nodes of mold nodes *M1*, *M2*, *MF1*, and *MF2* are the layout nodes referenced by *E1*, *E2*, *EF1*, and *EF2*, respectively. These layout nodes are no longer referenced by their original superior layout nodes.

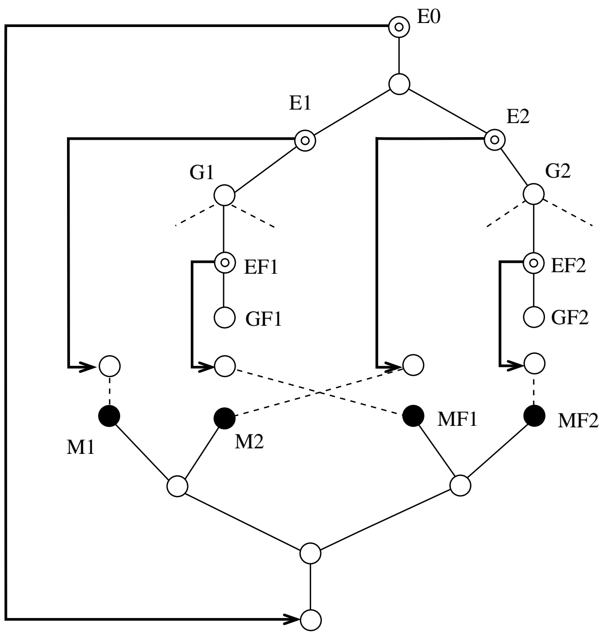

Figure 7. Embedding and mold nodes introduced into the document shown in Figure 6

4 FILE FORMAT

Now, we are ready to introduce our file format. In other words, we provide a data stream representing a document augmented with embedding nodes and mold nodes. Although this data stream is in character encoding, we can easily provide a data stream in binary encoding. Hereafter examples are based on the document shown in Figure 7 and are put in brackets.

First, we introduce tokens, which are constituents of the data stream. A *token* is either an embedding start tag, embedding end tag, logical start tag, logical end tag, layout start tag, layout end tag, mold tag, or a leaf descriptor. These tags are similar to those in SGML [4]. A *embedding start tag/embedding end tag* pair represents an embedding node. An embedding start tag specifies the 'stream' attribute and the 'subordinate_streams' attribute of this embedding node. A *logical start tag/logical end tag* pair represents a logical node. A *layout start tag/layout end tag* pair represents a layout node. A *mold tag* represents a mold node and specifies the 'stream' attribute of the mold node. A *leaf descriptor* is a sequence of characters or bytes representing a leaf node. Any method can be used to obtain leaf descriptors from leaf nodes. Typically, the leaf descriptor for a character leaf node is the character itself. A *place holder* is a temporary dummy token representing an embedding node. A place holder will later be replaced by a fragment representation (defined below) of the embedding node.

A *fragment representation* of an embedding node *E* is a sequence of tokens such that the following four conditions are satisfied.

Condition 1 The first in the token sequence is an embedding start tag, and the last in the sequence is an embedding end tag. This embedding start and end tag pair represents *E*. No other embedding start or end tags occur in the token sequence. (In a fragment representation of embedding node *E0*, the first and the last tokens are `<embedding stream="" subordinate_streams="body footnote">` and `</embedding>`, respectively. No other embedding start or end tags occur.)

Condition 2 The logical start tags, logical end tags, place holders, and leaf descriptors in the token sequence collectively represent the logical subtree referenced by *E*. (In a fragment representation of *E0*, tokens:

```
<(log)root>
  ?E1?
  ?E2?
</(log)root>
```

occur in this order, where 1) `<(log)root>` and `</(log)root>` are a start and an end tag for the logical root, respectively, and 2) `?E1?` and `?E2?` are place holders temporarily representing *E1* and *E2*, respectively.)

Condition 3 The layout start tags, layout end tags, mold tags, and leaf descriptors in the token sequence collectively represent the layout subtrees referenced by *E*. (In a fragment representation of *E0*, tokens:

```
<(lay)root>
  <(lay)page>
    <(lay)body_area>
      <(lay)mold stream="body">
      <(lay)mold stream="body">
    </(lay)body_area>
    <(lay)footnote_area>
      <(lay)mold stream="footnote">
      <(lay)mold stream="footnote">
    </(lay)footnote_area>
  </(lay)page>
</(lay)root>
```

occur in this order, where 1) `<(lay)root>`, and `</(lay)root>`, are a start tag and an end tag for the logical root, and so forth, and 2) `<(lay)mold stream="body">` is a mold tag representing *M1* (and *M2*), and `<(lay)mold stream="footnote">` is a mold tag representing *MF1* (and *MF2*). Layout nodes have attributes such as ‘dimensions’ and ‘position’, but we have omitted them from layout start tags for the space limitation.)

Condition 4 Consider any mold tag *m* in the token sequence. The mold node represented by *m* has a prime (layout) node. This prime node is under some embedding node *F* such that the logical subtree referenced by *E* references *F*. (In the break-out case other embedding nodes may intervene between *F* and the prime node.) Condition 4 requires that the place holder temporarily representing *F* precede *m* in the token sequence. (In a fragment representation of *E0*, the place holders `?E1?` and `?E2?` precede the mold tags (`<(lay)mold stream="body">`) temporarily representing *M1* and *M2*, respectively, because the prime nodes of *M1* and *M2* are subordinate to *E1* and *E2*, respectively. The place holders `?E1?` and `?E2?` precede the mold tags (`<(lay)mold stream="footnote">`) temporarily representing *MF1* and *MF2*, respectively, because the prime node of *MF1* is subordinate to *E1* through *EF1* and the prime node of *MF2* is subordinate to *E2* through *EF2*.)

Condition 1 provides representations of correspondences. Conditions 2 and 3 provide representations of logical subtrees and layout subtrees, respectively. (Observe that paragraphs with subordinate logical structures can readily be represented.) Conditions 2 and 3 further ensure that no leaf descriptor is duplicated although a leaf node belongs to both structures. Condition 4 makes the depth-first traversal of layout structures tractable (see Section 5). Appendix A shows a proof that token sequences satisfying these conditions always exist. (For *E0*, one such sequence is shown below:

```
<embedding stream="" subordinate_streams="body footnote">
  <(log)root>
    ?E1?
    ?E2?
  </(log)root>
  <(lay)root>
    <(lay)page>
      <(lay)body_area>
        <(lay)mold stream="body">
        <(lay)mold stream="body">
      </(lay)body_area>
      <(lay)footnote_area>
        <(lay)mold stream="footnote">
        <(lay)mold stream="footnote">
      </(lay)footnote_area>
    </(lay)page>
  </(lay)root>
</embedding>.)
```

If a fragment representation is provided for each embedding node, we can create a data stream representing the entire document. We begin with the fragment representation of the top-level embedding node. We then recursively replace place holders with fragment representations: for each embedding node *E*, the place holder temporarily representing *E* is replaced with the fragment representation of *E*.

As an example, we show a data stream representing the document shown in Figure 7. To save space, we omit the output of text formatting; that is, no tags for line nodes and word nodes occur. (We will more precisely introduce such omission in Section 6.) A sequence of leaf descriptors is denoted by `leaf ... leaf`.

Tokens shown in lines 10∼16 and those shown in lines 26∼32 originally occurred in fragment representations of *EF1* and *EF2*, respectively. Those shown in lines 06∼08 and 17∼20 and those shown in lines 22∼25 and 33∼36 originally occurred in fragment representations of *E1* and *E2*, respectively. Those shown in lines 01∼05, 21, and 37∼46 originally occurred in a fragment representation of *E0*.

```
01 <embedding stream="" subordinate_streams="body  footnote">
02   <(log)root>
03   <(lay)root>
04     <(lay)page>
05       <(lay)body_area>
06         <ebedding stream="body" subordinate_streams="">
07           <(log)para>
```

```
          <(lay)text_box>
            leaf ... leaf
            <embedding stream="footnote"
                                subordinate_streams="">
              <(log)footnote>
              <(lay)foonote_text_box>
                leaf ... leaf
              </(lay)footnote_text_box>
              </(log)footnote>
            </embedding>
            leaf ... leaf
          </(lay)text_box>
          </(log)para>
        </embedding>
        <(lay)mold  stream="body">
        <embedding stream="body" subordinate_streams="">
          <(log)para>
          <(lay)text_box>
            leaf ... leaf
            <embedding stream="footnote"
                                subordinate_streams="">
              <(log)footnote>
              <(lay)foonote_text_box>
                leaf ... leaf
              </(lay)foonote_text_box>
              </(log)footnote>
            </embedding>
            leaf ... leaf
          </(lay)text_box>
          </(log)para>
        </embedding>
        <(lay)mold  stream="body">
      </(lay)body_area>
      <(lay)footnote_area>
        <(lay)mold  stream="footnote">
        <(lay)mold  stream="footnote">
      </(lay)footnote_area>
    </(lay)page>
  </(lay)root>
  </(log)root>
</embedding>
```

5 ACCESS ALGORITHMS

In this section we sketch three algorithms to access documents stored in files. These algorithms are for traversing logical structures, traversing layout structures, and storing documents in files, respectively.

5.1 Traversing logical structures

As an example of traversing logical structures, we show a procedure for the conversion from our file format to an SGML-like logical-structure-only file format. This procedure does not make a copy of the entire logical structure in the main memory.

Basically, we only have to sequentially read tokens from a file, and write them in the output file if they are either logical start tags, logical end tags, or leaf descriptors. Other tokens are simply discarded. For symmetry between this procedure and the next one, however, we introduce a recursive procedure. This procedure sequentially reads tokens from a file, invokes itself when an embedding start tag is encountered, and returns when an embedding end tag is encountered. Logical start tags, logical end tags, and leaf descriptors are stored in the output file. Mold tags, layout start tags, and layout end tags are discarded. Condition 2 in Section 4 ensures that the tokens stored in the output file collectively represent the logical structure.

5.2 Traversing layout structures

We also show a procedure for the conversion from our format to an SGML-like layout-structure-only file format. Again, this procedure does not make a copy of the entire layout structure in the main memory.

In an augmented document, the layout structure is fragmented by mold nodes. Each fragment is a layout subtree, in which mold nodes occur as leaf nodes. The prime (layout) node of each mold node is the root of another layout subtree. To reconstruct the original layout structure, we have to replace each mold node with its prime node. Consequently, our procedure has to replace each mold tag m with a sequence of tokens, which represents the layout subtree rooted by the prime node of M, where M is the mold node represented by m. Hereafter we call this sequence a 'layout subtree'.

We use queues to temporarily hold 'layout subtrees'. A queue is introduced for each stream. Mold tags are replaced with 'layout subtrees' held in the queues. (The suitability of this approach is shown in Appendix B.) We use a stack to manage queues. Elements stored in this stack are tuples of the form $< stream\_name, pointer\_to\_queue >$. When an embedding start tag is encountered, queues for subordinate streams are created and tuples are pushed into the stack. When an embedding end tag is encountered, queues are destroyed and tuples are popped from the stack.

Now, we are ready to introduce our procedure. Like the previous one, this procedure is recursive. It sequentially reads tokens from a file, invokes itself when an embedding start tag is encountered, and returns when an embedding end tag is encountered.

- When invoked for an embedding start tag, this procedure performs two initialization actions:
    - *Finding an output queue to store 'layout subtrees'*. The procedure obtains the stream name by examining the 'stream' attribute of the current embedding start tag. Then, from the stack top, the procedure searches for a tuple whose first element is equal to the obtained stream name. Note that a tuple can be found even in the break out case. Having found a tuple, the procedure examines its second element and identifies the queue it references as an output queue.

— *Creating queues and pushing tuples into the stack.* The procedure obtains the names of the subordinate streams by examining the ‘subordinate_streams’ attribute of the current embedding start tag. Then, for each of the subordinate stream names, the procedure creates a queue and pushes a tuple into the stack. The first element of this tuple is the stream name and the second element is a pointer to the created queue.

- Having performed the initialization actions, this procedure sequentially reads tokens from a file. Layout start tags, layout end tags, and leaf descriptors are stored in the output queue. Condition 3 in Section 4 ensures that these tokens collectively represent layout subtrees. Logical start tags, logical end tags, and leaf descriptors are discarded.
- When a mold tag is encountered, the procedure first obtains the stream name by examining the ‘stream’ attribute of this mold tag. Then, from the stack top, the procedure searches for a tuple whose first element is equal to the obtained stream name. Having found a tuple, the procedure examines its second element and identifies the queue it references. Then, the procedure moves a ‘layout subtree’ from this identified queue to the output queue. In other words, the procedure repeatedly reads a token from the identified queue and stores this token in the output queue until one layout subtree is moved.
- When an embedding start tag is encountered, the procedure recursively invokes itself. (The assumption that no footnote nodes exist under other footnote nodes is used here. It ensures that the recursively invoked procedure does not interfere with ‘layout subtrees’ being constructed.)
- When an embedding end tag is encountered, the procedure returns. Before doing so, the procedure disposes of the queues and tuples created during the initialization. That is, the queues are destroyed and the tuples are popped from the stack.

We need a main procedure, which invokes this recursive procedure. This main procedure creates a stack, creates a queue for the anonymous top-level stream, pushes a tuple into the stack, and invokes the recursive procedure. After it finishes constructing the entire ‘layout tree’ in the queue, the main procedure copies tokens from the queue to the output file.

5.3 Storing documents in files

To store a document in a file, we have to create a data stream representing this document and then store this data stream. Section 4 shows how to construct data streams from fragment representations. Now, we only have to consider how to construct fragment representations.

A fragment representation of an embedding node E is constructed by the following steps.

- First, we create a sequence of logical start tags, logical end tags, leaf descriptors, and place holders, which represents the logical subtree referenced by E. This is done by traversing the logical subtree in the depth-first manner. When we encounter subordinate embedding nodes during this traversal, we skip their subordinates.
- Second, we create a sequence of layout start tags, layout end tags, leaf descriptors, and mold tags, which represents the layout subtrees referenced by E. This is

done by traversing each of the layout subtrees in the depth-first manner, and then concatenating the results.

- Third, we merge the two sequences and eliminate duplication of leaf descriptors so that Condition 4 is satisfied.
- Last, we create an embedding start and end tag pair for the embedding node. Then, we insert, in the merged sequence, the embedding start tag as the first element and the embedding end tag as the last element.

6 OMISSION OF TEXT FORMATTING OUTPUT

The data stream shown in Section 4 contains a layout start and end tag pair for each of the layout nodes. If a page has 50 lines, the data stream contains a minimum of 100 tags per page. Furthermore, if text formatting is elaborated, a layout node is introduced for each word and this node has a position in the line node. Layout tags for these layout nodes take an unacceptably large amount of space.

However, these layout nodes are often unnecessary. For example, queries based on both the logical and layout structures do not usually impose any conditions on line nodes and word nodes. Conditions on layout nodes concern only the layout nodes above text boxes (e.g., pages and columns). If we omit line nodes and word nodes while retaining the other layout nodes, the data stream becomes much more compact but remains useful for information retrieval systems, for example.

The tags for line nodes and word nodes is omitted by modifying Condition 3 as below:

Condition 3' The layout start tags, layout end tags, and place holders in the token sequence collectively represent the layout subtrees referenced by the embedding node; however, the layout nodes under text boxes, namely line nodes, word nodes, and character leaf nodes, are not represented. Note that character leaf descriptors still occur in fragment representations because of Condition 2. (Assume that the text of *G1* is 'This is an example of omission' and that the text box corresponding to *G1* has two line nodes, each of which in turn has three word nodes. The original Condition 3 requires tokens `<(lay)text_box> <(lay)line> <(lay)word> This </(lay)word> <(lay)word> is </(lay)word> <(lay)word> an </(lay)word> </(lay)line> <(lay)line> <(lay)word> example </(lay)word> <(lay)word> of </(lay)word> <(lay)word> omission </(lay)word> </(lay)line> </(lay)text_box>`. However, under the modified Condition 3, we can eliminate all of the `<(lay)line>`, `</(lay)line>`, `<(lay)word>`, and `</(lay)word>`.)

Omitted layout nodes can be reconstructed by reformatting text; it is possible to format the text of *one* logical node rather than the entire document (*reduced reformatting* [9]). Such reduced reformatting of text requires a much smaller program, less time, and less memory than does reformatting of an entire document. Text boxes, which are not omitted from the data stream, provide information (such as dimensions) required for text reformatting. In the case of captions, we only have to reformat each caption with respect to the line length specified by the text box corresponding to that caption.

7 DISCUSSIONS

Two standards, namely CONCUR of SGML [4] and FPDA of ODA [8], provide file formats for formatted-processable documents. Neither fulfills our requirements, however.

CONCUR allows the mixture of logical tags and layout tags in one document entity. Data contents (sequences of leaf descriptors) are shared by these two sets of tags. By considering only logical tags and data contents, we have the logical structure of the document. By considering only layout tags and data contents, we have the layout structure. It is of course possible to have both structures by considering both logical tags and layout tags. However, CONCUR can not handle reordering and break-out. The reason is that the order in which data contents occur in the logical structure and the order in which data contents occur in the layout structure are both forced to be identical to the textual occurrence order. Correspondences are only vaguely represented by those tags which are both logical and layout. The omission of text formatting output is not possible.

Applied to documents without reordering, break-out, and/or elaborated text formatting, CONCUR and our file format provide very similar results. In this sense, our format can be considered as an extension of CONCUR, where 1) the introduction of mold tags, embedding start tags, and embedding end tags allows reordering and break-out, and 2) the omission of text formatting output becomes possible.

In the FPDA format, all logical nodes except leaf nodes occur first, all layout nodes except leaf nodes occur next, and all content portions (sequences of leaf nodes) occur last. The physical ordering of nodes and content portions does *not* represent their hierarchical relationship. That is, even if a node *G1* is a child of a node *G2*, *G1* does not have to follow *G2* in the file; *G1* may appear first and *G2* may follow. The hierarchical relationship is represented by node identifiers in (a minor variation of) the Dewey decimal notation. For example, if *G1* has an identifier '3 0 0 ' and *G2* has an identifier '3 0 0 0', *G2* is the first child of *G1*.

FPDA can handle reordering and break-out. It is also possible to omit the result of text formatting by storing processable contents rather than formattted processsable contents. However, since content portions and their superior logical (layout) nodes are separated, the depth-first traversal of the logical (respectively, layout) structure is possible only after copying the entire logical (respectively, layout) structure into the main memory. As a result, to do a simple task by the depth-first traversal of the logical or layout structure, the programmer is forced to use a tree management library called an ODA toolkit. The reference manual of such an ODA toolkit is lengthy and somewhat daunting. Furthermore, FPDA provides no information about correspondences.

SGML and ODA significantly differ on their approaches to semantics. ODA standardizes the semantics [4] as well as the syntax. Meanwhile, SGML provides a machanism (DTD) for defining the syntax, and does not concern the semantics. HyTime, a standard based on SGML, introduces *architectural forms* and standardizes their semantics. A DTD in HyTime references to such architectural forms, thereby borrowing their standardized smenatics. Any of the ODA, SGML, and HyTime approaches to the semantics can be combined with our file format. That is, 1) we can standardize the syntax and semantics, 2) we can provide a mechanism for defining the syntax only, or 3) we can introduce architectural forms as well as a syntax-definition mechanism.

The TEI (Text Encoding Initiatives) project [3] is an attempt to provide guidelines

[4] To be precise, ODA standardizes the semantics regarding formatting and imaging only.

for representing textual documents in electronic form. Technically, the TEI guidelines are given as DTD's of SGML. The representation of multiple hierarchies, such as both the logical and layout structures, has been an important issue in TEI, and a chapter of the TEI guidelines is devoted to it. TEI uses CONCUR and also introduces *milestone elements, fragmentation of an item, virtual joins,* and *redundant encoding*. Nevertheless, as the editor of the TEI guidelines writes [12], "the TEI was never able to devise a fully satisfactory general solution". After an early version of this paper was presented at SGML'94, the TEI project started to consider incorporating ideas of this work.

We have to extend our file format to handle duplication (e.g., running headers), removal (e.g., omitted annotations), and addition (e.g., page numbers), which we mentioned in Section 2. Duplication can be handled by introducing a special stream comprising only one logical node, which is repeatedly laid out. Removal can be handled by introducing another special stream, the logical nodes of which are not laid out. Addition can be handled by introducing yet another special 'stream', which does not exist in the logical structure [5].

Bibliographies, indexes, and glossaries require further extension. The main problem here is that the ordering of items (such as indexes) is not controlled by the order in which they appear in the logical structure, but is rather controlled by alphabetical sorting of items, for example. One solution is to introduce a 'stream' for each item. Such 'streams' uniquely identify items, and thus can control any reordering. Though such 'streams' deviate from the original scope of streams, our file format and algorithms have no problems in handling such 'streams'.

Last, we discuss the expressive power of our file format. Syntactcially, it can handle coexisting multiple hierarchies such that 1) most of the leaf nodes are shared by these hierarchies, and 2) the difference of the leaf node ordering in different hierarchies can be handled by the stream mechanism. To the best of the present author's knowledge, any WYSISYG document can be represented by such multiple hierarchies, and is thus representable in our file format. Furthermore, the multiple *logical* structures addressed in TEI might also be representable in our file format. Examining this possibility by investigating the requirements for multiple logical structures is an interesting direction for this research to take.

8 CONCLUSION

We have introduced a file format for documents containing both logical structures and layout structures. It is possible to consider this file format as an extension of CONCUR.

To store a document in a file, we create a data stream representing this document as follows: First, we augment the document with embedding nodes and mold nodes. By doing so, we can temporarily dissolve reordering and break-out; the document becomes a hierarchy of embedding nodes. We then construct a fragment representation (a sequence of tokens) for each embedding node. In this step, we can compact fragment representations by eliminating tokens for line nodes and word nodes, if we like. Finally, we create a data stream by recursively replacing each place holder with a fragment representation.

To conclude, we summarize advantages of this file format as below:

Less spatial overhead. No leaf descriptors are duplicated, although leaf nodes belong to both the logical and layout structures. Additional tags for simultaneously representing

[5] We also have to introduce a mechanism for representing leaf nodes that belong to layout structures only.

both structures, namely embedding start tags, embedding end tags, and mold tags, do not take much space since mold nodes and embedding nodes are introduced for only some of the logical and layout nodes.

Omission of text formatting output. The text formatting output can be omitted from the data stream. The omitted information, when needed, can be reconstructed by text reformatting. Such text reformatting requires a smaller program, less time, and less memory than does reformatting of an entire document.

Ease of handling logical structures. The depth-first traversal of logical structures is highly important, as a variety of processing is possible with such a traversal. Our traversal algorithm introduced in 5.1 is efficient and easy to implement. The file is sequentially read only once, and the logical structure is not duplicated in the main memory.

Ease of handling layout structures. The depth-first traversal of layout structures is important for the same reason. Our traversal algorithm introduced in 5.2 is also efficient and quite easy to implement. Likewise, the file is only sequentially read once and the entire layout structure is not duplicated in the main memory.

Explicit representation of correspondences. An embedding start and end tag pair explicitly represents the correspondence between a logical node and a sequence of layout nodes. Thus, application programs can exploit the correspondences. For example, a query engine can use the correspondences between table logical and layout nodes to to retrieve a table occurring in the second page of section 3.

Handling of reordering. Since reordering is temporarily dissolved by embedding nodes and mold nodes, documents having reordering can be represented. In particular, documents containing footnotes or marginal nodes, and synchronized bilingual documents can be represented.

Handling of break-out. Again, since break-out is temporarily dissolved by embedding nodes and mold nodes, documents having break-out can be represented. In particular, documents containing footnotes within paragraphs can be represented.

APPENDIX A: FRAGMENT REPRESENTATION EXISTENCE

We claimed that a fragment representation is a token sequence satisfying four conditions. We have to prove that such a sequence exists for any embedding node E.

If we choose to omit the output of text formatting, we have to use the modified Condition 3. However, if a token sequence satisfying the original condition exists, we can derive a token sequence satisfying the modified condition by deleting layout tags for line nodes and word nodes. Thus, we only have to consider the original Condition 3.

Condition 1 claims that an embedding start tag and an embedding end tag, which collectively represent E, occur as the first token and the last token, respectively, and that there are no other embedding start or end tags. This condition can be easily satisfied. Thus, we consider token sequences without embedding start or end tags; that is, we prove that a token sequence satisfying Conditions 2, 3, and 4 exists.

Consider a sequence of logical start tags, logical end tags, leaf descriptors, and place holders, which represents the logical subtree referenced by E. Step 1 of the fragment construction procedure provides this sequence. Let A be the set of tokens in this sequence. Let R be the order on A in which tokens occur. In other words, $x R y$ if and only if either token x precedes token y in the sequence or x is equal to y. Obviously, R is a total order.

Likewise, consider a sequence of layout start tags, layout end tags, leaf descriptors, and

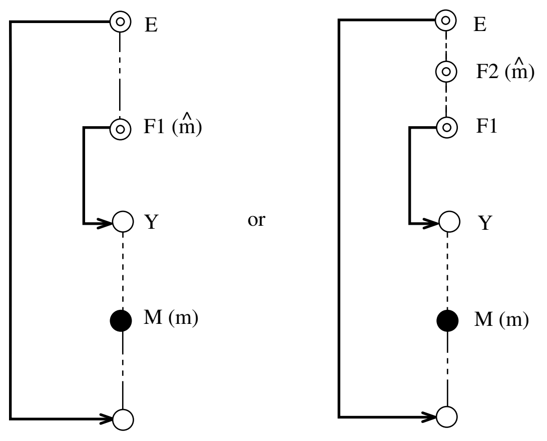

Figure 8. A mold tag m and a place holder denoted by $\hat{m}$

mold tags, which represents the layout subtrees referenced by E. Step 2 of the fragment construction procedure provides this sequence. Let B be the set of tokens in this sequence and let S be the order on B in which tokens occur. Relation S is also a total order.

Now, we are ready to prove the existence of fragment representations. The key idea is to find a partial order on $A \cup B$ such that Conditions 2, 3, and 4 are satisfied. If such a partial order exists, a total order satisfying these conditions also exists (topological sorting). This total order on $A \cup B$ provides a token sequence that satisfies the conditions.

We consider two cases. In Case 1, A does not contain leaf descriptors. In Case 2, A does.

Case 1: *A* does not contain leaf descriptors.

First we prove that A and B are disjoint. Since A contains no leaf descriptors, A contains logical start tags, logical end tags, and place holders only. Since B does not contain any of them, A and B are disjoint.

Let m be an arbitrary mold tag in B. The mold node represented by m, say M, occurs in a layout subtree referenced by E. Let Y be the prime node of M. Layout node Y is referenced by another embedding node *F1*, which is subordinate to E. Furthermore, the logical node referenced by *F1* pertains to a stream below the one referenced by E (Figure 8).

We introduce a notation $\hat{m}$, which denotes a place holder in A. Assume that no embedding nodes occur between E and *F1*. In this case $\hat{m}$ denotes the place holder temporarily representing *F1*. Assume not. Then, the logical node referenced by the embedding nodes between E and *F1* break out. Since footnotes in footnotes are not permitted, only one embedding node, say *F2*, occurs between E and *F1*. In this case $\hat{m}$ denotes the place holder temporarily representing *F2*.

We now define a binary relation T on $A \cup B$ as

$$x\,T\,y \iff \begin{array}{l} \text{i) } x, y \in A \text{ and } x\,R\,y; \\ \text{ii) } x, y \in B \text{ and } x\,S\,y; \text{ or} \\ \text{iii) } x \in A, y \in B, x\,R\,\hat{n}, \text{ and } n\,S\,y \text{ for some mold tag } n. \end{array}$$

It is easy to show that T satisfies Conditions 2, 3, and 4. Conditions 2 and 3 are ensured by i) and ii), respectively. To show that Condition 4 is satisfied, we have to prove $\hat{m}\, T\, m$ for any mold tag m. Since R and S are total orders, $\hat{m}\, R\, \hat{m}$ and $m\, S\, m$. By iii), we have $\hat{m}\, T\, m$.

It remains to show that T is a partial order; that is, we prove the reflexivity, transitivity, and antisymmetry of T.

Reflexivity ($x\, T\, x$). Assume that $x \in A$. Since R is a total order on A, we have $x\, R\, x$. By i), $x\, T\, x$ holds. The $x \in B$ case is similarly proved.

Transitivity (if $x\, T\, y$ and $y\, T\, z$, then $x\, T\, z$). Since the definition of T has 3 cases, we have to consider 9 (3×3) sub-cases. We prove only 3 sub-cases, but the other sub-cases can be proved similarly. Sub-case 1: i) $x, y \in A$ and $x\, R\, y$ and i) $y, z \in A$ and $y\, R\, z$. Since R is a total order, $x\, R\, z$ holds. By i), we have $x\, T\, z$. Sub-case 2: i) $x, y \in A$ and $x\, R\, y$ and ii) $y, z \in B$ and $y\, S\, z$. Since A and B are disjoint, this sub-case does not occur. Sub-case 3: i) $x, y \in A$ and $x\, R\, y$ and iii) $y \in A$, $z \in B$, $y\, R\, \hat{n}$, and $n\, S\, z$ for some mold tag n. Since R is a total order, $x\, R\, \hat{n}$. By iii), we have $x\, T\, z$.

Antisymmetry (if $x\, T\, y$ and $y\, T\, x$, then $x = y$). We again have to consider 9 sub-cases, but we prove only 2 sub-cases. Sub-case 1: i) $x, y \in A$ and $x\, R\, y$ and i) $y, x \in A$ and $y\, R\, x$. Since R is a total order, $x\, R\, z$ holds. By i), we have $x\, T\, z$. Sub-case 2: i) $x, y \in A$ and $x\, R\, y$ and ii) $y, x \in B$ and $y\, S\, x$. This sub-case does not occur, since A and B are disjoint.

Case 2: A contains leaf descriptors.

First we prove that B does not contain any mold tags by contradiction. Assume that B has a mold tag m. Let M, Y, and $F1$ be the mold node represented by m, the prime node of M, and the embedding node referencing Y, respectively (Figure 8). The logical node referenced by $F1$ pertains to a stream below the one referenced by E. Thus, the correspondence between the logical node and layout nodes referenced by E is not at the lowest-level. Therefore, leaf nodes can be subordinate to this logical node only through logical nodes having counterparts. However, since A contains leaf descriptors, we have a contradiction.

We assume that the occurrence order of leaf descriptors in A is the same as that in B. In other words,

$$\text{if } x \text{ and } y \text{ are leaf descriptors, then } x\, R\, y \Leftrightarrow x\, S\, y. \tag{1}$$

This assumption is natural, because the text appears out of order, otherwise.

We now define a binary relation T on $A \cup B$ as

$$x\, T\, y \quad \Leftrightarrow \quad \begin{array}{l} \text{i) } x, y \in A \text{ and } x\, R\, y; \\ \text{ii) } x, y \in B \text{ and } x\, S\, y; \\ \text{iii) } x \in A - B, y \in B - A, x\, R\, c, \text{ and } c\, S\, y \text{ for some leaf descriptor } c; \text{ or} \\ \text{iv) } x \in B - A, y \in A - B, x\, S\, c, \text{ and } c\, R\, y \text{ for some leaf descriptor } c. \end{array}$$

It is easy to show that T satisfies Conditions 2, 3, and 4. Conditions 2 and 3 are ensured by i) and ii), respectively. Condition 4 does not apply, because B contains no mold tags.

It remains to show that T is a partial order.

Reflexivity ($x\, T\, x$). As in case 1.

Transitivity (if $x\,T\,y$ and $y\,T\,z$, then $x\,T\,z$). Since the definition of T has 4 cases, we have to consider 16 (4×4) sub-cases. We prove 4 sub-cases only. Sub-case 1: i) $x, y \in A$ and $x\,R\,y$ and i) $y, z \in A$ and $y\,R\,z$. Since R is a total order, $x\,R\,z$ holds. By i), we have $x\,T\,z$. Sub-case 2: i) $x, y \in A$ and $x\,R\,y$ and ii) $y, z \in B$ and $y\,S\,z$. Since $y \in A \cap B$, y is a leaf descriptor. Assume that $x \in A - B$ and $y \in B - A$. Then, by iii), we have $x\,T\,z$. Assume not. Then, either $x \in A \cap B$ or $y \in A \cap B$. If $x \in A \cap B$, x is a leaf descriptor. By (1) and $x\,R\,y$, we have $x\,S\,y$. By $y\,S\,z$ and i), $x\,T\,z$ holds. The case that $y \in A \cap B$ is similarly proved. Sub-case 3: i) $x, y \in A$ and $x\,R\,y$ and iii) $y \in A - B$, $z \in B - A$, $y\,R\,c$, and $c\,S\,z$ for some leaf descriptor c. By $x\,R\,y$ and $y\,R\,c$, we have $x\,R\,c$. Assume that $x \in A - B$. Then, by (iii), we have $x\,T\,z$. Assume not. Then, $x \in A \cap B$ and x is thus a leaf descriptor. By (1) and $x\,R\,c$, we have $x\,S\,c$. Since $c\,S\,z$, we have $x\,S\,z$. By (ii), $x\,T\,z$ holds. Sub-case 4: i) $x, y \in A$ and $x\,R\,y$ and iv) $y \in B - A$, $z \in A - B$, $y\,S\,c$, and $c\,R\,z$ for some leaf descriptor c. Obviously, this sub-case does not occur.

Antisymmetry (if $x\,T\,y$ and $y\,T\,x$, then $x = y$). We again have to consider 16 sub-cases, but we prove 3 sub-cases only. Sub-case 1: i) $x, y \in A$ and $x\,R\,y$ and i) $y, x \in A$ and $y\,R\,x$. Since R is a total order, $x = y$ holds. Sub-case 2: i) $x, y \in A$ and $x\,R\,y$ and ii) $y, x \in B$ and $y\,S\,x$. Since $x, y \in A \cap B$, x and y are leaf descriptors. By (1), we have $x\,S\,y$. Since S is a total order, $x = y$ holds. Sub-case 3: i) $x, y \in A$ and $x\,R\,y$ and iii) $y \in A - B$, $x \in B - A$, $y\,R\,c$, and $c\,S\,x$ for some leaf descriptor c. Obviously, this sub-case does not occur.

APPENDIX B: SUITABILITY OF USING QUEUES

The procedure for the conversion from our format to an SGML-like layout-structure-only file format introduces a queue for each stream. Each queue holds "layout subtrees", which will later replace mold tags. In this appendix we show the suitability of this approach.

We show this suitability by proving two lemmas. The first lemma ensures that the "layout subtree" for an encountered mold tag has already been read from the file. The second lemma ensures that "layout subtrees" read first are used first (i.e. they are used in the same order in which they occur in the file) as long as they correspond to logical nodes of the same stream.

Lemma 1 *Let* M *be a mold node and let* Y *be its prime node. Then, the tokens representing the layout subtree rooted by* Y *precede the mold tag representing* M.

Lemma 2 *Let* M1 *and* M2 *be mold nodes such that (1) an embedding node* E0 *references both the layout subtree containing* M1 *and that containing* M2, *(2)* M1 *and* M2 *specify the same value for the attribute "stream", and (3)* M1 *precedes* M2 *in the layout structure.*

Then, the tokens representing the layout subtree rooted by Y1 *precede those representing the layout subtree rooted by* Y2, *where* Y1 *and* Y2 *are the prime nodes of* M1 *and* M2, *respectively.*

Proof of Lemma 1

Let m be the mold tag representing M, let E be the embedding node that references the layout subtree containing M, and let $F1$ be the embedding node referencing Y (Figure 8).

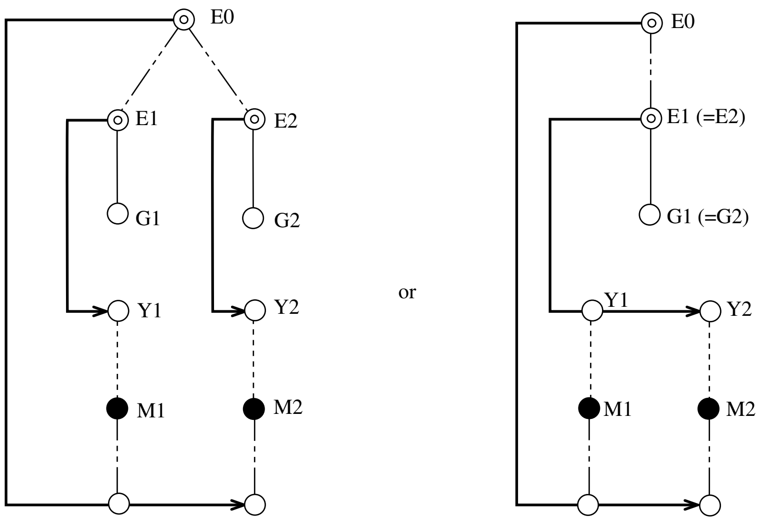

Figure 9. *Node relationship in Lemma 2*

As we have observed in Appendix A, either no embedding nodes occur between *E* and *F1*, or only one embedding node *F2* occurs between *E* and *F1*.

Assume that no embedding nodes occur between *E* and *F1*. Then, $\hat{m}$ denotes the place holder temporarily representing *F1*. By Condition 4, $\hat{m}$ precedes *m* in the fragment representation of *E*. By Condition 3, the tokens representing the layout subtree rooted by *Y* occur in the fragment representation of *F1*. Since $\hat{m}$ is replaced with the fragment representation of *F1*, these tokens precede *m*.

Assume that only one embedding node *F2* occurs between *E* and *F1*. Then, $\hat{m}$ denotes the place holder temporarily representing *F2*. Again, by Condition 4, $\hat{m}$ precedes *m* in the fragment representation of *E*. Place holder $\hat{m}$ is replaced by the fragment representation of *F2*. This fragment representation contains a place holder, which is in turn replaced by the fragment representation of *F1*. Since, by Condition 3, the tokens representing the layout subtree rooted by *Y* occur in the fragment representation of *F1*, these tokens precede *m*. Thus, we have completed our proof.

Proof of Lemma 2

Let *E1* and *E2* be the embedding nodes referencing *Y1* and *Y2*, respectively. Further, let *G1* and *G2* be the logical nodes corresponding to *Y1* and *Y2*, respectively.

Since *M1* and *M2* occur in the layout subtrees referenced by *E0* and the values of the attribute "stream" of *M1* and *M2* are equal, *G1* and *G2* pertain to the same stream under *E0*. Since *M1* precedes *M2* in the layout structure, either *G1* precedes *G2* in the logical structure, or *G1* is equal to *G2* (Figure 9).

Assume that *G1* precedes *G2* in the logical structure. Since *G1* and *G2* are the only children of *E1* and *E2*, respectively, *E1* precedes *E2* in the logical structure. It is straightforward to show that the tokens occurring in the fragment representation of E1 precede those occurring in the fragment representation of E2 in the final data stream. By Condition 3, the fragment representation of *E1* contains the tokens representing the layout subtree rooted by *Y1*, and the fragment representation of *E2* contains those representing the layout subtree rooted by *Y2*. Therefore, we have the conclusion.

Assume that *G1* and *G2* are equal. Logical node *G1* (= *G2*) corresponds to both *Y1* and *Y2*. Since *M1* precedes *M2* in the layout structure, *Y1* precedes *Y2*. By Condition 3, in the fragment representation of *E1* (= *E2*), the tokens representing the layout subtree rooted by *Y1* precede those representing the layout subtree rooted by *Y2*. This ordering is preserved even after all place holders are replaced with fragment representations. Thus, we have completed our proof.

ACKNOWLEDGEMENTS

I am grateful to Tom Bauer, who has greatly contributed to the readability of this paper. Moriene Personte prepared figures for me.

REFERENCES

1. C. L. A. Clarke, G. V. Cormack, and F. J. Burkowski. An algebra for structured text search and a framework for its implementation, August 1994. URL: `ftp://cs-archive.uwaterloo.ca/cs-archive/CS-94-30/structxt.dvi`.
2. D. Rus and J. Allan, 'Does navigation require more than one compass?', in *AAAI-95 Fall Symposium on AI Applications in Knowledge Navigation and Retrieval*, pp. 116–122, (November 1995).
3. C. M. Sperberg-McQueen and L. Burnard. TEI guidelines for electonic text encoding and interchange(P3). ACH/ACL/ALLC Text Encoding Initiative, April 1994. URL: `http://etext.virginia.edu/TEI.html`.
4. International Organization for Standardization, *Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML)*, 1986.
5. R. Furuta, 'Concepts and models for structured documents', in *Structured Documents*, eds., J. André, R. Furuta, and V. Quint, Cambridge University Press, (1989).
6. V. Joloboff and T. Schleich, 'Introduction to Interscript, ISO/IEC JTC1/SC18/WG3/N439R', Technical report, International Standard Organization, (1985).
7. International Organization for Standardization, *Information Technology – Text and Office Systems – Document Style Semantics and Specification Language (DSSSL)*, 1994.
8. International Standard Organization, *Information Processing – Text and Office Systems – Office Document Architecture (ODA) and Interchange Format, ISO/IS 8613*, 1987.
9. M. Murata and K. Hayashi, 'Formatter hierarchy for structured documents', in *EP92*, eds., C. Vanoirbeek and G. Coray, Cambridge University Press, (1992).
10. A. Brown, T. Wakayama, and H. Blair, 'A reconstruction of context-dependant document processing in SGML', in *EP92*, eds., C. Vanoirbeek and G. Coray, Cambridge University Press, (1992).
11. P. Pedersen, 'Streams and the layout process for formatted-processable documents, ISO/IEC JTC1/SC18/WG3/N1408', Technical report, International Standard Organization, (1989).
12. C. M. Sperberg-McQueen. Trip report: SGML '94 and SGML/Open technical meeting, November 1994. Available by sending an e-mail containing only the line `GET TEI-L.LOG9411` to `LISTSERV@UICVM.UIC.EDU`.