Numbering Document Components*

Michael A. Harrison Ethan V. Munson

June 30, 1990

Abstract

Numbering document components such as sections, subsections, figures and equations gives each component a unique identifier and helps the user locate the component when it is cross-referenced. This report discusses ways in which such numbering can be described and proposes a simple paradigm for declarative specification of how components should be numbered. The class of algorithms for incremental update of component numbers is studied and the "best" such algorithm is developed in detail.

1 Introduction

It is common, particularly in technical documents, to assign ordinal numbers to certain document components, such as sections, figures, and equations. The purpose of numbering is to facilitate cross-referencing of distant components and to help the user locate distant components in a long document.

Numbering serves these purposes in two ways. First, it gives each component a unique identifier within its type, so that Figure 2 is the only figure given the number 2. Unlike such alternate identifiers as section titles and figure captions, numbers are short, can be generated automatically, and are certain to be distinct. Also, there is an inexhaustible supply of them. However, numbers have none of the mnemonic qualities of these textual alternatives. Secondly, a component number gives the reader a strong hint about the component's location in the document. Figure 2 is guaranteed to appear after Figure 1 and before Figure 3. Page numbers might specify the location more precisely, but they may not specify a unique component (since a page may contain multiple figures or equations). Moreover, in certain computerized document systems, the text is assumed to be an infinite scroll, i.e. there are no page numbers. Thus, while component numbering is neither the best unique identifier nor the best locator, it is probably the best tool for achieving both.

*Sponsored by the Defense Advanced Research Projects Agency (DARPA), monitored by Space and Naval Warfare Systems Command under Contract N00039-88-C-0292.

Support for automated component numbering has traditionally been one of the features that distinguish "document-processing" systems from "word-processing" systems. Document-processing systems free the user from the tedious task of updating component numbers and cross-references by hand. However, for those document-processing systems that support interactive editing, the maintenance and incremental update of component numbers and their related cross-references is a difficult problem.

This paper discusses two related topics. The first is a paradigm for specifying how document component numbers should be computed. This paradigm improves on existing approaches by being fully (rather than partially) declarative. Next, possible algorithms for the incremental update of component numbers in an interactive structured document editor are explored with an emphasis on one particular algorithm, the last/previous algorithm. This algorithm keeps component numbers up to date at all times, is reasonably efficient, and requires very little overhead for editing operations on unnumbered components.

The next section of this paper discusses conventions for component numbering and provides some background on the domain of structured documents. Section 3 discusses previous work on component numbering. Sections 4 and 5 present our declarative specification paradigm and the incremental update algorithms, respectively. The final section presents our conclusions.

2 Background

We have been led to investigate the problem of numbering document components by our work on the design of Ensemble, an interactive editing system for structured multimedia documents. The needs of Ensemble underly our interest in novel specification methods and fast incremental formatting algorithms. Our work on numbering is motivated by a set of assumptions about

- the ways that documents are numbered,
- the nature of structured documents,
- and the design principles required by interactive editing systems.

This section provides some background on these three topics.

2.1 Numbering Conventions

Numbering conventions for documents vary considerably between certain classes of documents. These conventions are not rigidly adhered to, but there is enough consistency to allow some general assertions.

In discussing these conventions it is useful to make a distinction between two classes of document components. *Structural* components, such as chapters, sections, subsections, etc., are used by an author to give his work a hierarchical structure. *Non-structural* components, such as figures, footnotes, tables, and equations, are simply distinguished elements of the document and may be placed at almost any depth in the work's structural hierarchy. The distinction may seem fairly arbitrary, but the strictly hierarchical relationship between the structural components makes it easier to specify and implement automatic numbering for them.

Literary works often have some structural components in the form of parts, chapters, and sections. While such works may have illustrations or maps, these are seldom numbered and rarely cross-referenced in the text. Cross-references are particularly rare because they violate the convention that the reader should not be reminded that he is merely reading a book, rather than directly observing events. Literary works typically number parts and chapters independently. Part II of a novel may contain Chapters 9 through 15 because the first eight chapters were found in Part I. However, sections within the chapters usually start their numbering anew with each new chapter. In such documents, numbering simply serves to help the reader understand the flow of the document.

Technical documents and some scholarly works make much greater use of component numbering. In general, their structural components are numbered strictly hierarchically. That is, each type of component is numbered *with respect to* its enclosing structure. For example, subsection numbers start over from one with each new section. Non-structural components are generally numbered with respect to a structural component of medium size. In a journal article, they are numbered with respect to the entire article. In a book, their numbers are likely to start over with each new chapter. In a multi-volume technical manual, the numbers may be based on an even smaller unit, like a subsection.

Scholarly works in the humanities and social sciences tend to fall in between these two extremes. They usually number structural components in the manner of literary works but include a substantial number of non-structural components, especially footnotes or endnotes, which are typically numbered with respect to a chapter.

In all these cases, document components are numbered in ascending order as they are encountered in a linear pass through the document. It is the conventions about when component numbering should start (i.e. be reset to one) that make these cases differ. From these observations, we make the following assumptions:

- No matter what kind of document is being produced, the key problem for a document processing system is identifying when the numbers for a particular component type should start over.
- Component numbers always start over at the beginning of an enclosing structural component.
- The structural components of a document are always arranged in a strict hierarchy (i.e. a chapter is never divided between two parts of a book).

- Thus, the specification of document component numbering reduces to the problem of specifying a particular subtree of the document's structural hierarchy.

2.2 Structured Documents[1]

All document processing systems are based, implicitly or explicitly, on a *document model.* Perhaps the simplest document model is that of a linear sequence of printable symbols interspersed with control symbols indicating changes of formatter state (e.g. position on the page or character font). This model forms the basis of most direct-manipulation systems (e.g. MacWrite [4] and the Interleaf publishing software [8]) and of some batch-oriented document processing systems (e.g. *troff* [12] and Plain TEX [9]). The user of such a system becomes both author and typographer, a position which can be both empowering and burdensome.

Structured document systems are based on a higher-level document model which emphasizes structure, rather than appearance. The canonical example of a structured document is a book composed of a sequence of chapters, which are themselves composed of sections, which are, in turn, composed of subsections. The hierarchical organization of this example is typical of many kinds of documents. As a result, structured documents are usually conceived of as trees. Structured document systems typically support a variety of document types, each having different legal arrangements of components. Common document types are book, article, letter, and memo.

The author provides the structured document system with the text of the document and specifies how the text is broken into components. The appearance of the document is determined by a separate choice of *style* or *presentation.* In most cases, this style was designed by a *style designer*, who fills the traditional role of typographer. Thus, by separating the specification of content from the specification of appearance and by providing predefined styles, structured document systems relieve their users of the typographer's duties. In addition, they encourage the reuse of documents and their components because little effort is required to make changes of style.

In practice, there are two kinds of structured document systems, batch and interactive. The most widely used batch systems are Scribe [15] and LATEX [10]. While they present the user with a tree-structured document model, the underlying implementation does not actually build a tree, because their single-pass formatting implementation does not require it. Also, the separation of structure and presentation is not reflected in the way styles are defined. For example, LATEX style files not only define the appearance of a class of documents, but also the legal configurations of members of the class.

In contrast, the implementations of interactive structured document systems,

[1]This discussion of structured documents is based on the complete presentation in [7].

like the **pedtnt** [7] and Grif [14], closely reflect the structured document model. In both the **pedtnt** and Grif, the system's central data structure is a k-ary tree, the *document tree*. These systems must explicitly maintain the document tree in order to support the structure-oriented editing operations that are naturally required in an interactive structured document system. In both systems, document types are defined using a grammar-based specification language. Grif calls this specification a *structural schema* and uses a separate *presentation schema* to specify the appearance of the document.

2.3 Design Principles for Interactive Editor

One of the most common editing operations in an interactive document editor is text entry. If entered by a fast typist, new characters may arrive at rates as fast as 10 per second [5]. Users of an editor will quickly become dissatisfied if on-screen feedback for this text entry does not keep up with their typing. These same users may be much more tolerant of brief delays when performing less frequent operations that appear complex on their face. Clearly, the implementors of interactive document editors should focus their optimization efforts on those operations that are performed frequently and appear simple to the user.

However, the designer of an interactive document editor is faced with a more difficult decision when choosing algorithms. The standard wisdom is to choose algorithms and data structures that minimize the maximum complexity of the editor's operations. For example, it is generally considered best to choose a data structure for which all operations are run in log time even though it means sacrificing constant time complexity on some individual operations.

We believe that this wisdom does not hold for interactive editors. It is important that most operations (e.g. insertion of unnumbered document components) resulting from normal text entry run in constant time , even if it increases the complexity of less frequent operations. If this is not the case, the designer runs the risk of creating a system that is only useful for "toy" documents.

3 Previous Work

3.1 Numbering Mechanisms

Component numbering actually involves two separate tasks, component number generation and cross-reference resolution. It nicely illustrates some of the differences between batch and interactive systems because batch systems can easily generate component numbers but have trouble with cross-reference resolution, while the situation is reversed for interactive systems.

The original batch-oriented formatting systems [9, 12] did not support numbering directly. However, they did provide registers which could be used for various purposes, including component number generation. The user allocates a register for each component type being numbered and then directly manages updates to the value stored there. Thus, as the formatter scans the document, each register holds the current number for its corresponding component type. This approach to number generation is used by all batch formatters, though the details are usually hidden from the user.

Cross-reference resolution is made difficult by the possible presence of *forward references*, which are cross-references to components appearing later in the document. Resolution of forward references requires two passes over the document source. In addition, the second pass must be a full formatting pass because the width of the cross-reference characters cannot be determined until after resolution. There are two basic approaches to cross-reference resolution for batch formatters.

Aho and Sethi [2] show how a simple preprocessor, using the UNIX utilities `grep`, `awk`, and `sed`, can be used to resolve cross-references. Using this approach, document formatting requires three passes over the source, but formatting is only done on the last step.

In the second approach, introduced by Scribe and also used by LaTeX, the batch formatter is run twice so that it acts as its own preprocessor [11]. On the first run, the formatter saves cross-reference information in an auxiliary file. The second run of the formatter reads the auxiliary file and uses the information there to resolve the cross-references. While only two passes are made over the document, they are both relatively expensive passes.

The register approach to number generation is not adequate for interactive WYSIWIG systems. In such systems, the user may move to random locations in the document and expects any component numbers found at those locations to be up to date. If the system is to respond quickly to the user's movement commands it must be able to "jump" to the new location without having to inspect all of the intervening material in order to update its numbering registers.

A solution is to abandon the register approach in favor of number slots in the components themselves. The use of number slots largely eliminates the cross-referencing problem because, as long as the slots are kept up to date, a cross-reference need only point to the component it references. Unfortunately, keeping the number slots up to date is not so straightforward. Whenever a numbered component is inserted or deleted, the number slots of all subsequent components must be made up to date. Without some secondary data structures, this could require a full pass over the document. In addition, whenever multiple components are inserted or deleted by a single editor operation they must be scanned for the presence of numbered components.

3.2 Specification

As mentioned above, the early batch formatters provided registers which could be used for component number generation. Direct allocation and update of such

```
COUNTERS
   Chapter_Count : RANK OF Chapter;
BOXES
   Chapter_Number :
      Content : (VALUE (Chapter_Count, UpperRoman));
RULES
   Chapter :
      Create (Chapter_Number);
   Ref_Chapter :
      Copy (Chapter_Number);
```

Figure 1: Example showing the use of a counter in Grif's presentation schemas.

registers was used to support hierarchical section numbering in the *-me* macros for *troff* [3].

The general trend in structured document systems has been to use a more declarative specification method, but one still based on the register model. Scribe introduced a system of counters whose definition was essentially declarative [16]. This approach was brought to interactive systems by Grif.

Grif documents are described by two schema types, structure and presentation. The structure schema defines the logical structure of the document, which can be thought of as a tree. The presentation schema defines how each of the document components defined in the structure schema is displayed on the page or screen. To support component numbering, Grif's presentation schema language, P, provides special registers called *counters*, which only support a restricted class of updates [13].

An abridged sample of a presentation schema showing the definition and use of a counter for chapters can be seen in Figure 1. This example defines a counter called `Chapter_Count`. The formatted version of the counter is produced by the definition of the `Chapter_Number` box. This box of formatted text is placed at the beginning of the chapter by the `Create` command and cross-references to it are generated using the `Copy` command.

The primary type of counter specification, shown in Figure 1 is based on a component's *rank*, which is the ordinal rank of a component relative to its siblings in the document tree. This approach works well for the hierarchical numbering of structural components, like chapters, because they are guaranteed to be siblings. However, it does not work for non-structural components which either appear at varying depths in the tree or are numbered with respect to an ancestor which is not their parent. Grif handles these cases with a specification that looks like a return to the register model:

```
CountEquation: Set 0 on Article  Add 1 on Numbered_equation;
```

A counter can only be set by one type of component, can only be added to by one other type of component, and its initial and added values must be positive integer constants. So, the arbitrary updates that are possible with registers in *troff* or TEX are not possible in Grif.

Grif's counters are separate entities from the document components they count. The connection between the counter and the counted component is established by how the document designer uses the counter to construct the document's presentation. This lack of connection results in complex semantics for the `Copy` command [13].

4 Declarative Specification

Having examined this earlier work, we have designed a completely declarative method for the specification of component numbering. It improves on the approach of Grif by generalizing the notion of rank and by binding the component number to the definition of the component. Our approach is based on the following assumptions:

1. The document is represented as a tree.
2. The order of the document's components is equivalent to their positions in a preorder traversal of the (k-ary) tree [1, p. 54].
3. All components of a particular type are numbered with respect to a particular ancestor, specified by its type.

These assumptions point the way to a generalization of the notion of rank, as introduced by Grif. Since the order of components can be found by a preorder traversal of the document tree, the definition of rank can be extended from that of "rank among siblings" to that of "preorder rank within a subtree". The subtree within which to perform the traversal is specified by naming the type of its root. An application of this approach to the equation numbering example we examined for Grif (using modified Grif syntax) is shown in Figure 2. This example says that each `Equation` is numbered "with respect to" (`wrt`) its `Chapter` ancestor. The formatted version is created by accessing the `Number` attribute of the `Equation`.

This new specification method has several favorable qualities. It makes no commitment, implicit or explicit, about the way component numbers are computed. It unifies in one construct both the "rank among siblings" and "set-add" paradigms used in Grif. Finally, the new method makes the number an attribute of the component, rather than an independent data structure.

4.1 Recursively-Specified Structure

While it is not common in existing document processing systems, it is possible to create document specifications whose structural components are defined

```
BOXES
   Equation_Number :
      Content : (VALUE (Number(Equation), UpperRoman));
RULES
   Equation :
      Number: wrt Chapter;
      Create(Equation_Number);
   Ref_Equation :
      Copy (Equation_Number);
```

Figure 2: Example showing the specification of equation numbers using the preorder rank approach.

```
   Section = BEGIN
             Section_heading = TEXT;
             ? Section_Preamble = LIST OF (Paragraph);
             ? Section_sequence = LIST OF (Section);
             END;
```

Figure 3: Sample definition of recursive sections.

recursively. An example would be the definition of sections shown in Figure 3 where each section is composed of a heading, an optional list of paragraphs, and an optional list of sections. For the sake of conciseness, we use the adjective *recursive* for such recursively-specified document components. Note that a *recursive section* does not contain a list of copies of itself, just other sections with arbitrary content.

Our specification method is not adequate when documents can have recursive structure. Consider an article with recursive sections. As one descends the document tree from the root toward some leaf, one encounters a sequence of sections. The first section encountered corresponds to the traditional notion of section, the second to a subsection, and so on. So, specifying a subtree within which to number a particular component type requires being able to count sections on the downward path from the root to the component. Our construct for doing this is

```
   Equation:
      Number: wrt second Section
```

where `second` is a predefined keyword.

This approach is quite sufficient for recursive sections but does not handle all structures which could be defined recursively. Recursive figures are an example

```
Figure =  BEGIN
          Figure_content = CASE OF
                             Frame = GRAPHIC_OBJECT;
                             Sub_figures = LIST OF (Figure);
                             END;
          Figure_caption = Contents;
          END;
```

Figure 4: Structural specification for recursive figures.

of such a structure. A specification for such a structure appears in Figure 4. Typically, a figure is numbered with respect to its enclosing section or chapter. However, subfigures are usually numbered with respect to their enclosing figure. As an example, Figure 10 (tenth figure in its chapter) might contain two separate charts, which might be lettered (a) and (b) (corresponding to numbers 1 and 2). Accommodating both ways of numbering figures requires the use of some sort of conditional expression in the numbering specification. Such a scheme could be described using parser generation tools, such as attribute grammars, but support for such mechanisms has yet to provided by any existing document processing system.

5 Incremental Update Algorithms

For interactive structured document editors, the most difficult aspect of component numbering is the maintenance of correct values in the number slots of the components. Nearly every insertion or deletion operation involving numbered components must update some component numbers. This section describes the task more formally and presents a series of algorithms for the incremental update of these numbers.

In order to make our presentation more concrete, we will discuss the incremental update problem in terms of a running example: a book containing equations which are numbered with respect to their enclosing chapter. A specification corresponding to this numbering scheme appeared in Figure 2. The use of this example allows the update problem and the algorithms to be presented in terms of "equations" and "chapters", rather than "numbered components" and "components of the type they are numbered within."

5.1 The Problem

In a structured document editor, all modifications of the document can be considered to be built from three primitive operations *set-attribute*, *insert-subtree*, and *delete-subtree*. As described here, these operations immediately update any invalid component numbers. This is a necessary feature if the system is to ensure that all component numbers and cross-references that are on screen are correct.

Set-attribute(N, A, v) assigns the value v to attribute A of node N. Except when used to change the value of a node's equation number slot, the *set-attribute* operation has no effect on equation numbers.

The *insert-subtree(S, N, i)* operation inserts the subtree S as the i^{th} child of the node N. In some editor implementations, the N and i parameters may not be necessary because the editor supports a notion of *current insertion point*. The insertion operation involves four steps:

1. Attach S to the document tree as the i^{th} child of N.

2. Determine whether S contains any equations. If not, the insertion operation is complete. Otherwise, continue.

3. Find $E_{pre}.number$, the number of the equation preceding S in a preorder traversal of the chapter subtree.

4. Assign new values to the number slots of the equations in S and all equations following S in the chapter subtree, beginning with the value $E_{pre}.number + 1$.

The *delete-subtree(S)* operation removes S from the document tree. Like insertion, deletion involves four steps:

1. Determine whether S contains any equations. If not, skip to step 4.

2. Find $E_{old}.number$, the number of the first equation in S. Alternatively, find $E_{pre}.number$, which is always one less than $E_{old}.number$.

3. Assign new values to the number slots of the equations following S in the chapter subtree, beginning with the value $E_{old}.number$.

4. Remove S from the document tree.

These two operations have several common subtasks. They both must determine whether S contains any equations, find a starting equation number, and update all equations after the point of change in the document. Because their underlying subtasks are so similar, they generally have similar complexity.

5.2 The Algorithms

Given our specification method based on preorder rank, the naive approach to the problem is to traverse the tree to locate the relevant equations. Using this approach, the *insert-subtree* operation would:

- Attach S to the document tree as the i^{th} child of N.
- Traverse S until an equation is found. If no equation is found in S, insertion is complete. Otherwise, call this equation E_{new} and continue.
- Traverse the chapter subtree backwards from S to find the immediately preceding equation, E_{pre}, and its number, $E_{pre}.number$. If there are no preceding equations, consider $E_{pre}.number$ to be zero.
- Set $E_{new}.number$ equal to $E_{pre}.number + 1$. Traverse the chapter subtree forward (in preorder) from E_{new}, setting the numbers of all subsequent equations.

The naive *delete-subtree* operation is:

- Traverse S until an equation is found. If no equation is found in S, deletion is complete. Otherwise, call this equation E_{old} and continue.
- Traverse the chapter subtree forward (in preorder) from S, updating the numbers of all subsequent equations, starting with the value $E_{old}.number$.

Both operations can, in the worst case, require a complete traversal of the chapter subtree. If equations appear at random locations in the subtree, the final step of the algorithm will require traversing, on average, half of the chapter subtree. In addition, both operations must completely traverse S to ascertain that S contains no equations. A subtree traversal requires crossing each edge in the subtree twice. Since the number of edges in a tree is $n - 1$, where n is the number of nodes in the subtree, we consider this to be an $O(n)$ algorithm.

The naive algorithm does not require any additional storage. Every other algorithm described here attempts to improve on the naive approach by trading space for time.

5.2.1 The Traditional Approach

In this algorithm, the additional space is required by a new data structure, a single binary search tree containing all nodes in the chapter subtree. The nodes are ordered in this search tree according to their position in a preorder traversal of the chapter subtree. In addition, each node has a slot holding a count of the number of equations below it in the search tree and the equations themselves are chained together in a "thread", the head of which is also stored in the chapter node. This new data structure allows the computation of $E_{pre}.number$ by ascending the search tree from S to the root and summing the equation counts of the left siblings (if any) of S and its ancestors. If the search tree is balanced, this is an $O(\log n)$ operation.

The *insert-subtree* operation first attaches S to the document tree. S must have an attached search tree and equation thread, even if S is not a full chapter. Once S is attached to the document tree, it is straightforward to locate its predecessor, N_{pred}. If S is a singleton tree, then S's search tree can be inserted directly into the search tree for the chapter. Otherwise, a more complicated sequence of operations is required. The search tree is split around the new predecessor of S, leaving a left subtree L, the predecessor N_{pred}, and a right subtree R. The search tree is rebuilt by two successive joins, as in

$$join(join(L, N_{pred}, S), D, R)$$

where D is a dummy node that is immediately deleted.

The *delete-subtree* operation is similar. If S is a singleton tree, normal deletion is used. Otherwise, the chapter's search tree is split at the predecessor of S and at the successor of S. Again using a dummy node, the resulting left and right subtrees are re-joined and the dummy node is deleted. The middle subtree, which contains the nodes of S, is attached to S in case S is later re-inserted, as is the middle portion of the equation thread.

All search tree operations must be enhanced to update the equation count slots and to correctly split and merge the equation thread. In addition, when either of these operations actually inserts or deletes equations, those equations following the point of insertion or deletion must have their number slots updated. If the search tree operations are implemented using self-adjusting binary search trees [17], the search tree manipulations of both operations will run in log time (amortized). The cost of updating the equation numbers will be linear in the number of equations in the chapter, because the presence of the equation thread obviates the need to perform a tree traversal. We consider this algorithm to have $O(\log n + k)$ running time when S contains k equations.

The problem with this approach is that *all* insertions and deletions become log time operations. This conflicts with the design principles we endorsed in Section 2.3.

5.2.2 Our Approach

We set out to construct an incremental update algorithm which would provide constant time operations on equation-less subtrees and sub-linear operations on subtrees containing equations. Every algorithm we considered makes use of the idea of an *equation thread.* An equation thread is a list containing every equation in the chapter subtree. Each equation's position in the list corresponds to its position in a traversal of the chapter subtree and, therefore, to its equation number. When a subtree is inserted (deleted), a list of the equations in the subtree is inserted into (cut out of) the chapter's equation thread. The numbers of any equations following the point of the editing operation are updated by stepping through the elements of the thread. Use of an equation thread reduces the time required for this final step of the insertion and deletion operations from $O(n)$ to $O(k)$, where k is the number of equations in the chapter subtree. Thus, the algorithms differ only in the running time for determining whether S

contains any equations, the running time to find the number of E_{pre}, and the amount of additional storage required.

Before settling on the algorithm described in detail below, we examined and rejected five algorithms:

Equation Thread Only. This algorithm simply added an equation thread to the chapter subtree. Subtrees being inserted are assumed to have an equation thread (possibly empty) attached. Using this algorithm, the central operation is the comparison of the position of S to that of the equations in the chapter's equation thread. This comparison operation is used to determine the position at which S's equation thread should be inserted into or deleted from the main thread. The complexity of a single comparison has an upper bound of $(2h + b)$ where h is the height of the chapter subtree and b is its branching factor. This comparison is necessary when determining whether S contains equations (when deleting S) and when finding the previous equation's number. The number of comparisons performed can be minimized by representing the equation thread as a balanced binary search tree, rather than as a list. However, since all subtrees being deleted must be checked for the presence of equations, this algorithm does not allow deletion of equation-less subtrees in constant time.

Equation Count 1. This algorithm adds a slot to every node which records the number of equations at or below that node. A subtree contains equations whenever the equation count at its root is non-zero, which can be tested in constant time. The previous equation number is computed by summing the equation counts of the left siblings of the ancestors of S (inclusive of S). Thus, this algorithm performs both operations in constant time with equation-less subtrees and in $O(k + bh)$ time when S contains equations.

Equation Count 2. The equation count slot could be used only to determine whether S contains equations. This approach achieves constant running time for equation-less subtrees while the running time becomes $O(k + (\log k) \cdot (2h + b))$ when S does contain equations. In the special case of section numbering, where the k numbered components are siblings, the running time is $O((\log k) \cdot k)$.

Cumulative Equation Count. Alternatively, each node can hold a slot which records the number of equations at or below its left siblings. The number of equations at or below the node can be computed by subtracting the node's value from that of its right sibling[2]. This cumulative equation

[2]Nodes which have no right sibling must subtract their slot value from the number of equations at or below their parent. For nodes along the right edge of the tree, this recursive process is halted when reaching the chapter node which records the total number of equations it holds.

count slot makes it possible to compute the previous equation number by inspecting only the ancestors of S. Unfortunately, the slots of the right siblings of those same ancestors must be kept up to date. As a result, this algorithm does not improve on its predecessor.

Lazy Cumulative Equation Count. The update of the cumulative equation count slots can be done "lazily" if each internal node also stores a pointer to its *rightmost-up-to-date-child*. While this allows the algorithm to avoid pointless update operations, it adds complexity by requiring comparisons of the position of each ancestor of S to the pointer stored in its parent. Since this approach merely exchanges two tasks of equivalent complexity and requires more storage, it was rejected.

The best incremental update algorithm is the last/previous algorithm. It requires two additional slots per tree node which, together, permit fast computation of equation numbers. For a particular node N, they are

- $N.leq$ (last equation) points to the last equation in the subtree rooted at N. If N is an equation itself, then $N.leq = N$. If the subtree rooted at N has no equations, then $N.leq = nil$.
- $N.peq$ (previous equation) is a pointer to the leq slot of the closest (i.e. rightmost) left sibling of N which has a non-nil leq slot. If there is no such "previous equation", then $N.peq = nil$.

There are two operations which affect equation numbers, *insert-subtree* and *delete-subtree*. An inserted or deleted subtree contains zero or more equations and is assumed to have correct values for the leq and peq slots of its nodes[3]. Subtrees being inserted must also have an attached equation thread which contains all equations in the subtree in traversal order. The deletion operation will create such a thread. We consider a single equation, E, to be a special case of a subtree, with $E.leq = E$, $E.peq = nil$, and with an attached list containing only itself.

The *insert-subtree(S, N, i)* operation inserts the subtree S as the i^{th} child of the node N. The insertion operation involves two phases. The first phase attaches S to the document tree and determines the value of $S.peq$, as follows:

1. Attach S to the document tree as the i^{th} child of N.
2. Let $L \leftarrow$ *left-sibling*(S).
   - If $L = nil$, then set $S.peq \leftarrow nil$.
   - If $L.leq \neq nil$, set $S.peq \leftarrow$ *address-of*$(L.leq)$.
   - Otherwise, set $S.peq \leftarrow L.peq$.

[3] In the case of an inserted subtree, the correct value of the peq slot is always nil.

If S does not contain any equations, the insertion operation is complete. However, when S does contain equations, they must be numbered correctly, which requires finding E_{pre}, the equation immediately before S in a traversal of the chapter subtree. Also, it may be necessary to update the *leq* slots of the ancestors of S and the *peq* of the right siblings of the ancestors of S. In this case, the insertion operation involves the following additional steps:

1. Scan the right-siblings of S, setting their *peq* slots to point to *S.leq*. Stop when there are no more right-siblings or after setting the *peq* field of a sibling with a non-*nil leq* slot.
2. Let $C \leftarrow S$ and $P \leftarrow N$. C will always be a child of P. Let $E_{pre} = nil$.
3. Let *CouldBeLast* $\leftarrow$ *true*. *CouldBeLast* will be set to *false* when it has been proven that S is not the last equation in the chapter.
4. While $CouldBeLast = true$ and C is not the chapter node:
   (a) If $E_{pre} = nil$, then let E_{pre} be the equation pointed to by *C.peq*. If $C.peq = nil$, then $E_{pre} \leftarrow nil$.
   (b) If $P.leq = nil$, then P had no equation descendants prior to the insertion of S. Set $P.leq \leftarrow S.leq$ and update the *peq* slots of the right-siblings of P.
   (c) Otherwise, if *P.leq* points to E_{pre} then S now contains the last equation in the subtree rooted at P. Set $P.leq \leftarrow S.leq$.
   (d) Otherwise, there are equations following S in the subtree rooted at P. Set $CouldBeLast \leftarrow false$.
   (e) Let $C \leftarrow P$ and $P \leftarrow parent(P)$.
5. Get the list of equations attached to the chapter node. Insert the list of equations attached to S immediately after E_{pre}. Renumber all equations that follow E_{pre} in the list, starting with the value $E_{pre}+1$. If $E_{pre} = nil$, insert the new equations at the front of the list and start numbering with an initial value of 1.

The *delete-subtree(S)* operation removes S from the document tree and is almost precisely the reverse of the insertion operation. The major difference is that if S does not contain any equations, no updating work is necessary. If S does contain equations, the deletion operation must:

- Update the *peq* slots of the right siblings of S.
- Update the *leq* slots of any ancestors of S whose last equation was *S.leq*. If an ancestor's subtree will no longer contain any equations after the deletion of S, the right siblings of the ancestor must have their *peq* slots updated. In the process of updating the ancestors of S, the algorithm will locate E_{pre}, the immediately preceding equation.

- Get the list of equations attached to the chapter node. Remove the sublist containing the equations between E_{pre} (exclusive) and $S.leq$ (inclusive) and attach this sublist to S. It is the list of all equations at or below S.
- Update the number slots of any equations which followed $S.leq$ in the equation list attached to the chapter node.

The worst case running time of both operations is $O(k + bh)$, where b is the branching factor of the chapter subtree, d is the depth of S in the subtree, and k is the total number of equations in the subtree[4]. The k term is derived from the need to update all subsequent equations. The bd term results primarily from updating the *peq* slots of the right siblings of S and its ancestors. The actual cost of this task is limited as much as possible by using double indirection for the *peq* slots. Because the *peq* slot is actually a pointer to some other node's *leq* slot, changes to the *leq* slot automatically update all *peq* slots that point to it. It is also important to note that in the special case where the k numbered components are children of a single node, both operations run in $O(k)$ time.

5.3 Discussion

Each of the algorithms described in Section 5.2 has been implemented to test its correctness. However, we have not attempted to analyze the performance of all of the algorithms in detail.

Every algorithm we considered represented a clear improvement over the naive approach based on tree traversal. Of these algorithms the two best are the "last/previous" algorithm and the second "equation count" algorithm (which uses the equation count only to determine if a subtree contains equations). The order statistics for the worst-case running times of these algorithms are $O(k+bh)$ and $O(k + (\log k) \cdot (2h + b))$, respectively.

Comparison of the performance of these algorithms is not easy because, even ignoring degenerate cases like trees of height $n-1$, there are no guarantees about the nature of structured document trees. The relationship between branching factor, height, and the number of equations in a chapter subtree cannot be determined *a priori*. Experience with structured documents suggests that height grows logarithmically with overall subtree size and that branching factor can be fairly large but is bounded. In addition, it is very rare to see component numbers with even three decimal digits, so there seems to be a practical upper limit to this parameter of running time.

In an attempt to compare the algorithms more quantitatively, we measured the values of k, b, and h for several technical documents, including this one. Then, for each of these examples, we computed upper bounds on the number of

[4] We assume that, given a tree node, it is a constant time operation to find its right sibling. We also assume that immediately after inserting or deleting S, it is a constant time operation to access its left sibling. These assumptions are sufficiently weak to allow the use of singly-linked lists as the representation for children of a node in the document tree.

medium level operations (e.g. assignment to a slot or getting the next item in a list) required to insert a subtree containing numbered components. This analysis indicated that the last/previous algorithm is generally 33% to 50% faster than the equation count algorithm, except when k is small ($k < 3$). When k is small, the equation count algorithm is 15–25% faster than the last/previous algorithm. The last/previous algorithm is always better for the special case of section numbering, running at least twice as fast as the equation count algorithm.

Other arguments in favor of the last/previous algorithm are:

- It appears to minimize redundant operations since the update of the *leq* and *peq* slots is halted at the earliest possible moment and the use of double pointers in the *peq* slots eliminates the need to update them in many cases.
- It does not require the implementation of balanced binary trees, which are needed in the equation count method.
- When multiple types of numbered components are present in the inserted or deleted subtree, update of their numbers can be performed in the same pass up the tree. In contrast, the equation count algorithm must update each type of component number independently.

Extending each of these algorithms to the case where multiple *types* of component numbers must be updated is straightforward. In general, each numbered component type must have its own thread and set of slots.[5] The algorithms must be modified to take into account the presence of component types that are numbered within different subtree types.

6 Conclusions

We have presented a simple method for declaratively specifying how the components of a traditional linear document are numbered. The specification method appears to be sufficient for all traditional numbering schemes.

One class of document that the method does not support is the class of non-linear documents, of which hypertext is the obvious example. The problem with hypertext documents is that they do not conform to a tree structure. This makes the notion of preorder rank, on which the new numbering method is based, invalid for hypertext. However, ordinal numbering in any form does not make sense for hypertext, since there is no natural unique ordering for a hypertext document. A hypertext document does need some system of unique

[5] The only exception is the "traditional" algorithm. It does require a separate thread and equation count slot for each numbered component type. However, multiple numbered component types that are numbered within the same type of subtree can share that subtree's search tree.

identifiers so that textual cross-references (as opposed to hypertext links) can be stated in a sensible manner. These identifiers could well be numbers, but there is no general method for assigning such numbers on the basis of the hypertext document's structure. However, any subportion of the hypertext document which is linear could make use of our numbering mechanisms.

We have also presented a number of algorithms for the incremental update of component numbers, focusing primarily on the last/previous algorithm. This algorithm improves considerably on the performance of the naive alternative while maintaining constant time insertion and deletion of non-numbered components.

We plan to use both the specification method and last/previous algorithm in Ensemble, an interactive editing system for structured multi-media documents, including programs, currently being designed here at Berkeley [6]. An interactive system like Ensemble requires the use of efficient incremental algorithms like the one described in Section 5. Also, Ensemble will use the model of separate specifications for document structure and presentation previously seen in Grif. The numbering specification presented in this paper will be part of the presentation language of Ensemble.

Acknowledgements

We would like to thank Raimund Seidel and John Hauser for useful discussions on incremental update algorithms.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1974.

[2] Alfred V. Aho and Ravi Sethi. Maintaining cross references in manuscripts. *Software—Practice & Experience*, 18(7):1001–1012, July 1988.

[3] Eric P. Allman. *-ME Reference Manual.* Electronic Research Laboratory, University of California, Berkeley, California. Available in bsd UNIX User's Manual.

[4] Apple Computers, Inc., Cupertino, California. *MacWrite Manual*, 1984.

[5] Kenneth P. Brooks. A two-view document editor with user-definable document structure. Technical Report 33, Digital Systems Research Center, Palo Alto, California, November 1988.

[6] Wayne Christopher. The architecture of Ensemble. Ensemble Internal Document, November 1989.

[7] Richard Furuta, Vincent Quint, and Jacques André. Interactively editing structured documents. *Electronic Publishing—Origination, Dissemination and Design*, 1(1):20–44, April 1988.

[8] Interleaf, Inc., Cambridge, Massachusetts. *Interleaf Publishing Systems Reference Manual, Release 2.0, Vol. 1: Editing and Vol. 2: Management*, June 1985.

[9] Donald E. Knuth. *The TEX Book*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1984. Reprinted as Vol. A of *Computers & Typesetting*, 1986.

[10] Leslie Lamport. *LATEX: A Document Preparation System. User's Guide and Reference Manual.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[11] Bruce Leverett. One-pass text formatting. Technical report, Scribe Systems, Inc., Pittsburgh, PA, 1988.

[12] Joseph F. Ossanna. Nroff/troff user's manual. Computer Science Technical Report No. 54, AT&T Bell Laboratories, Murray Hill, New Jersey, October 1976. Also available in UNIX User's Manual.

[13] Vincent Quint. *Les langages de Grif*, December 1988. Distributed with the program.

[14] Vincent Quint and Irène Vatton. Grif: An interactive system for structured document manipulation. In J. C. van Vliet, editor, *Text processing and document manipulation*, pages 200–213. Cambridge University Press, April 1986.

[15] Brian K. Reid. *Scribe: A document specification language and its compiler*. PhD thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, October 1980. Available as technical report CMU-CS-81-100.

[16] Brian K. Reid, Michael I. Shamos, and Janet H. Walker. *SCRIBE Database Administrator's Guide*. Unilogic, Ltd., Pittsburgh, PA, first edition, July 1981. Preliminary Draft, Order No. AA-L507A-TK.

[17] Robert Endre Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM, Philadelphia, PA, 1983.