A MODEL FOR A DOCUMENT MAINTENANCE SYSTEM†

Matthew Kaplan

Department of Computer Science *Brown University* *Providence, Rhode Island*

Abstract

We present the architecture of a system for editing formatted documents. Key among the requirements for this system are support for complex page formats and the ability to incorporate diagrams and other non-textual features. The system is interactive, using an incremental formatter to adjust the text after each change.

Our model allows for extending the set of supported document objects and permits multiple views of these objects. Documents are highly structured, and relationships between structures are maintained by constraints resident in each object.

KEYWORDS: interactive editor/formatters, incremental formatting, document model, constraint maintenance, page layout.

0. Motivation.

The system we describe here is intended to support creation of formatted documents. Important to us are the ability to incorporate many kinds of graphic entities (text, line drawings, tables, equations, musical notation, foreign language text, etc.) in a document as well as the ability to easily design a page in all its details (such as alignment of certain components with others, and relationships between dimensions of objects). The system is designed to make the extension of its capabilities feasible.

Our system differs from batch formatters in its interactive user interface. Our system is a 'WYSIWYG' formatter in that its user is given an accurate depiction of the printed page at all times. However, it differs from many existing editor/formatters by allowing many kinds of of graphical and textual objects to be incorporated in a document. The construction of complex documents is facilitated by the system's highly structured, object oriented design. Object orientation, by enforcing modularity, enhances extensibility. The structured approach benefits the user by making possible fast incremental reformatting, which is especially important in large documents. Such reformatting is automatically performed after editing operations, so that the user always sees an up-to-date version of what he is working on.

Many WYSIWYG document editor/formatters [MacWrite,Cham,MeyvD] have difficulty with large documents with complex layouts. One reason for this is the inadequacy of their internal representation and the associated processing model. The architecture of our system is specifically designed to address this problem so that, even for difficult documents, a fully formatted version can be presented to the user as he edits. In order to manage the richness of our domain, we provide multiple views of the document for the user to edit, each capturing a well-defined subset of the document's content. Thus, the user may edit text in one view, edit a picture in another, and see a fully formatted page, updated periodically (e.g., whenever an editing operation is complete) in a third. The correspondence between locations in these views is maintained, as they are merely filtered versions of the underlying internal representations. In this paper we present this architecture (a structured network of communicating specialists) and two components that display key features of our system (incremental reformatting and a flexible page layout facility).

These two components work together to create formatted pages. Specifying a page layout involves dividing a page into a pattern of rectangular regions that can hold columns of text, pictures, tables, etc. This division of the page into regions is user-driven. Reformatting, on the other hand, is an automatic process that fills columns defined in the layout with formatted text whenever necessary.

† This work has been supported, in part, by IBM Corp. and by the Online Computer Library Center, Inc. (OCLC)

1. Introduction.

Document maintenance, as used here, is a task that combines and extends the traditional notion of both *batch formatting* and *interactive editing*. Batch formatters (such as *troff* [KerLe], TEX [Knuth], and Scribe [Reid]) do not take advantage of the fact that documents are not created all at once, but, rather, are the object of frequent manipulation and adjustment. Formatting is done all at once, on the entire document or, in sophisticated systems, on substantial segments of the document. A *document maintenance system* (hereafter dms), like some interactive programming environments, makes use of the relatively slow and steady rate of change of a document to present the user with a complete and consistent picture of it at all times.

Some formatting systems (e.g., Scribe) have assumed that formatting details are not really the author's concern: that he does not care. This is often the case, but it is often not the case as well. In a dms, typography and layout do not take second place to the text, nor does text editing suffer from an overemphasis on the graphical components of the document. Its user can modify diagrams, page layouts or typographical characteristics as easily as words, and the document will properly adapt to all changes in an orderly, efficient way. Our system stands between the dual worlds of the graphical, layed-out document and logical, textually structured document, providing a channel of communication between them and allowing the user to manipulate both.

Unlike batch formatting, text and graphical editing are interactive but usually applicable only in domains where changes are local (i.e., are not propagated far). In a formatted document, as opposed to unformatted text or graphics editors, small changes can propagate over great distances; direct editing of formatted documents is not necessarily a local process. Unlike traditional interactive editors a dms lets its user edit interactively in the formatted document domain, where small changes can have global ramifications.

Our system is a *direct* editor of formatted documents. By "directly" editing a document we mean editing the layed-out version itself, as opposed to a linear command or specification file that would drive a batch formatter creating a layout. A document linearized into a command file is only indirectly related to the two-dimensional document that is being created. Eliminating the need to encode complicated two-dimensional relations in a one-dimensional command file is one of our fundamental goals. The capacity for direct editing is, to a great extent, what we refer to when we say that our system is "interactive".

Our dms could be classed as a WYSIWYG system designed for users creating publications, as opposed to letters or office memos. The nature of our goals (interactive editing of large and varied documents with sophisticated page layout needs) forces a different emphasis in this research than that in word processors and office systems (e.g., Etude [Good,HamIA]). Rather than concentrating on the user interface design, with secondary emphasis on the system design, we are primarily interested in how the the system's document representation can be used to give us a powerful, flexible, and extensible system. The representation, which is described in the section 2, can be put to great advantage in keeping the document in a presentable, formatted state. The control mechanism of the system, which oversees incremental formatting, is distributed throughout the components of the document, parallelling the document representation. Each component (for example, paragraph, diagram, column, sentence) has its own control center that oversees changes to sub-components, locally maintaining an appropriate format. Constraints are propagated between control centers as described in section 3. The format just mentioned is graphical in some components: for example, a column arranges its component rectangles so that they are stacked vertically, all separated by the same amount of space, and makes sure that they fit within the column width. In other components the "format" is not graphical, as in a sentence, which maintains an ordering of its words, independently of typographical characteristics. A component that is a composition of other, simpler, components, can exert some control over these, its children. The formats can be thought of as constraining the appearance or structure of document parts. Each kind of component is associated with its own mechanism for maintaining these constraints, as will be explained later.

2. Document organization.

A system's domain model has a profound influence on its strengths and weaknesses. A quarter-plane model of text is not well suited to a system that has to let its users examine the text at varying levels of detail. A hierarchical model is not well suited to systems that have to be able to move rectangular blocks of positioned text. The challenge for a hybrid editor/formatter, such as ours, is to reconcile the conflicting ways of decomposing the document. A user has to be able to refer to document elements both by appearance, as part of a particular column, for example, and textually, as part of a particular section or chapter.

Each useful organization can be regarded as a document *view*. Multiple views of a document are related to multiple viewpoints of programs as discussed in [Winog,BarSh], and used in the Pecan system [Reiss], among others. Two important views are the *abstract*, structured text view, and the *concrete* graphical text view, though multiple views can exist on a smaller scale, as in multiple formats for tables. In our domain it is important not to favor one view over another, as have batch formatters. For example, abstract formatters (such as Scribe) model documents hierarchically, by textual function, so users have little control over typography. Conversely, concrete formatters have inadequate facilities for changing text without regard to format [FurSS]. Because the abstract and the concrete document are only indirectly related we must accept multiple representations of the document, corresponding to multiple views, if all significant aspects are to be modelled.

Within any single view trees will be the dominant motif. Hierarchical representations are ubiquitous: many page layouts are patterns of non-overlapping nested boxes; unformatted text is hierarchically organized; line drawings can usually be decomposed hierarchically. Hierarchies express useful, natural groupings and we need them.

A true hierarchy cannot be used to express multiple uses of a single object. A word must be either a part of a sentence or a part of a text line, not both, in a tree representation. Our representation extends a strict hierarchy by allowing objects to be shared by multiple parents, each parent viewing the common child in a different way. We are still able to speak, for example, about a sentence as a group of words but our system can simultaneously support any other objects that contain words of the sentence in one way or another (formatted lines and index entries, for example) without favoring one usage of a word over another.

An important property of this representation is its tolerance of multiple structures built from one set of objects. However, in documents, it is also necessary to coordinate such multiple views; the relationships between different views are what bind the document objects into a whole. Our system is greatly concerned about how the textual and layout aspects of the document are intertwined. The internal representation of our dms is the mediator between conflicting views of a formatted document. As an example of the dependencies that must be represented, the effect of a textual change is felt throughout the column into which the text is set. Similarly, if a column's dimensions change, all the text within that column will be affected. Thus, the document model has to represent the relationships between textual document objects and the graphical containers within which they are arranged.

The dual functions of providing and coordinating multiple views are supplied, in our representation, with the use of links to multiple parents, and children, respectively. While words, for example, may split into multiple views by their parents, there are objects (such as chapters) that parent different kinds of views (see figure 1). Such objects are responsible

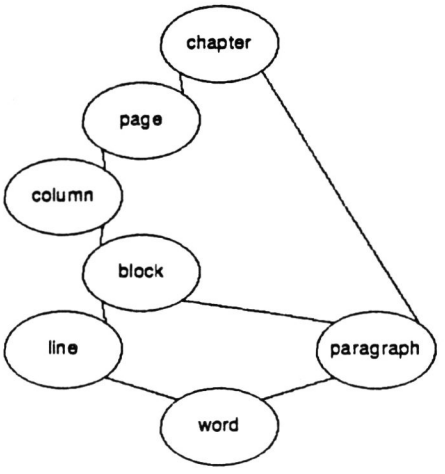

Figure 1. Document objects.

for communication between and coordination of objects. The next section examines how this coordination takes place by describing how pages are filled with text.

3. Constraint maintenance.

All actions in our system are initiated within a uniform constraint maintenance paradigm. Each object is responsible for properly maintaining itself according to its particular requirements. Such maintenance could involve, for example, keeping columns full in a magazine page format, preventing overfull pages or widows in a standard book, keeping sentences ordered in a paragraph, or characters aligned with respect to the baseline of a word. The tools available to an object to aid in this process are predefined manipulations of its internal representation, and application of functions made available to the object by its children. A child may not directly demand help of parents, so the object needs no knowledge of its parents to operate.

Our document world is dynamic: objects can change at any moment. When an object is altered, a parent, whose consistency is dependent on the characteristics of its children, might have to change. Thus whenever an object changes it wakes up each of its parents. A parent may decide that the change is insignificant. For example, if the arrangement of words in a paragraph changes but not the number of lines, a text column doesn't react to the change. If the change is significant, the parent must adapt and, because of its own change, propagate the change upward to its parents. Knowledge of changes spreads upward from child to parent; orders for adaptation are handed down from parents to children. Such structured constraint maintenance is the fundamental pattern of control flow in our system. All adaptation follows it. Similar constraint propagation schemes, in which the conditions necessitating adaptation are separated from the means of adapting, have been used in other systems (e.g., [SteSu,Borni,Gosli]). The paths that propagation of changes can follow are unrestricted in most such systems, while we limit propagation to follow links between parents and children as described, integrating constraint maintenance with the document representation.

4. Incremental text formatting.

When part of the text is changed the effects of this change are often propagated to the end of the document. But it will be noticed that often the most dramatic format changes occur closest to the source of the disturbance, and the effects very far away are minimal. If, for example, the document in question is a book, a change in the text will undoubtedly require resetting the edited paragraph. Blocks of formatted text after that paragraph, and in the same chapter, will have to be repositioned within their page, and possibly split over two or more pages. Individual lines, however, will not have to be reset and, in most cases, the paragraphs can be shifted whole. Past the chapter boundary, page numbers and references, at worst, will have to be changed. To take advantage of this we use an adaptive formatter, that knows that formatting needs are nonuniform, and that takes short cuts when possible. Our system, when reformatting a document after a change, notices where work needs to be done and where work that was done at a former time can be salvaged. We refer to this parsimonious formatting strategy as incremental reformatting. In this section the way that formatting objects and their associated constraints perform incremental formatting is described.

Similarities in various aspects of the formatting process can be observed: successive words are fit into lines of text, successive lines into blocks of text representing paragraphs, blocks into columns, and columns into pages. Similarly, when a word cannot fit entirely in a line, it is hyphenated and split across two, while if a paragraph can't fit completely within a column, it is split across successive columns. We abstract from the many formatting objects appearing on many levels (e.g., words, text lines, paragraph blocks, columns, pages) a single characteristic shared by all, that is the basis of our formatting method: in our system the objects engaging in formatting activities are *containers*, whose primary function is to be *filled* by a list of content objects. Containers may have many properties only loosely related to their fillability. For example, columns have rectangle dimensions and positions on their containing page, geometric properties that are sensitive to page layout constraints, and that affect how, within a column, the contents (paragraph blocks) are geometrically

arranged. But such properties vary from one type of container to another. The fundamental operation of filling is uniform across all containers, and is independent of whether a container is a box or circle, whether it fills left to right or top to bottom, etc.

Changes to document objects (made explicitly by users, or otherwise) trigger refilling the containers holding those objects. For example, when a paragraph's text is changed, its containing block knows that it is obsolete, and must refill itself with the new text. After this, the block, in turn, reports its change to its own parent, a column. If resetting hasn't changed the formatted paragraph's length, the constraints maintained by the column are still satisfied, so the column can end the propagation. (This is an example of a constraint that is column-specific, and not shared by other kinds of containers). If, however, the paragraph has grown or shrunk, the column can rearrange its other paragraphs, and possibly spread the change to its containing page. But the column doesn't have to tell any of its other paragraphs to reformat themselves. Thus, only one paragraph is fully reformatted, after which reformatting continues on progressively higher levels (i.e., with coarser granularity) as propagation spreads away from the initial change.

The key to incremental reformatting is making intelligent decisions about when to change formatting granularity. This decision can be made within the context of the uniform formatting method just introduced. Before describing the general condition allowing the formatter to coarsen its granularity, we give three concrete examples. When reformatting after changing a paragraph's text, granule size is first increased from the lowest level after the first paragraph has been reformatted, as described above. If a column-width block (a diagram, for example) is inserted between paragraphs we can immediately treat blocks of formatted text as basic units, and start repositioning text within columns using already formatted paragraphs as a fundamental unit. And if a column sized block is inserted, formatting can be reduced to column motion after the blocks of the disrupted column are repositioned (as in figure 2(b)).

The examples above differ greatly in details: in the first case the paragraph being reformatted can grow or shrink depending on its new contents, while columns are fixed in length, and it is unlikely that propagation could ever be reduced to column motion when formatting is set off by changing a paragraph's text. However, in all three examples, granularity is increased at analogous times. In our reformatting method there are three objects of interest uniformly, across all kinds of containers: the *container* of the changed objects, its *contents before* modification, and the *replacement contents*. What the three examples above have in common is that formatting granularity can be coarsened when, in the reformatting process, the last element of both the replacement contents stream is inserted into a container that previously ended with the last element of the old, replaced, contents. At such time we say that the new and old contents are *synchronized* with respect to their container. The significance of synchronization is that it happens at those times when the formatter can elevate its concerns to a level of lesser detail.

Some types of synchronization are more common than others. Because the length of a block is determined by its the amount of text it contains (i.e., the block can grow and shrink), synchronization will always occur after reformatting a paragraph (figure 2(a)). Synchronization with respect to a column (figure 2(b)), however, will be much less common because boundaries of columns are fixed independently of their contents. In this case synchronization

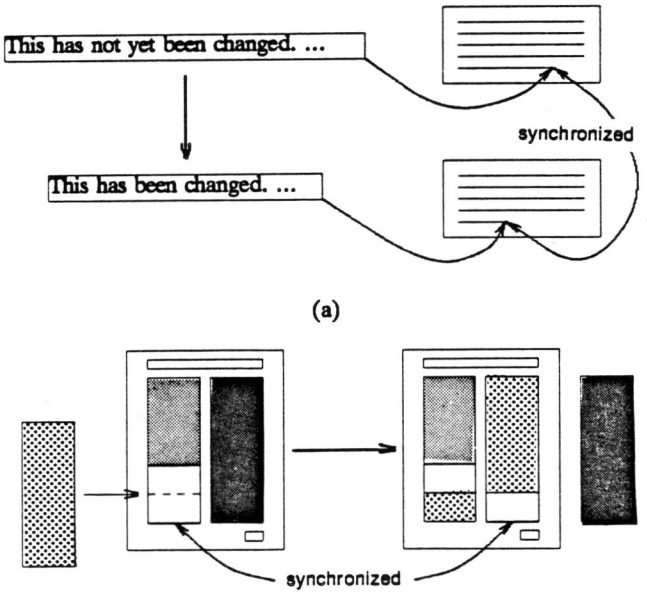

Figure 2. Synchronization points.

will occur only under unusual circumstances, such as the column insertion example above, or at the end of a chapter, when text is filled out with blank space to column-length.

5. Page layout.

TEX's glue is a well known substance used to separate the elements of patterns of boxes. Our system incorporates an analogous device for positioning boxes in patterns, the *spring*. Springs are often hooked up in *chains* that can either stretch or squeeze, depending on whether the sum of the *relaxed lengths* of the links is less or greater than the distance between the endpoints of the chain. If the chain stretches or squeezes, so do all its component springs. Within a chain, the amount individuals stretch or shrink is inversely proportional to their *strength*, except when the flexibility is limited by *minimum length* or *maximum length* constraints.

Springs appear only inside boxes. Conversely, nearly all boxes appearing in a document are bound up with springs. We refer to these, the underlying objects in layout, as *springboxes* and they can be informally regarded as boxes whose dimensions are determined by springs within them, in conjunction with "forces" exerted by neighboring springboxes. Thus, rather than using two kinds of materials, one for containing objects and one for separating them, we have only one.

Configurations of such boxes are not made by first creating individual springs and then arranging them around each other. Rather, patterns are made, starting from a box initially whole, by subdivision. This is one major difference between our model and the boxes separated by glue. We believe that dividing a page into regions is more natural for page layout than constructing a page layout from component pieces. A springbox can be recursively split, horizontally or vertically, into several sub-boxes. The positions of the partitions are governed by the spring properties of the component boxes, which are specified by the user when the containing box is split. Springs in adjacent split boxes form a chain, stretched across the containing box. When the width (or height) of the container is somehow *fixed* (how this is done is often independent of the springbox mechanism) component boxes will stretch or shrink as described above. Often, as a result of this adjustment, the component boxes' sizes are fixed and their own components' sizes can be fixed in the same way. The operation of spring chains is local to their containing box, and based on a simple physical model. Thus, users can manipulate springboxes with relative ease to create layout patterns.

Once a pattern of boxes is created, each member can be stuffed with one of a variety of fillings (such as text, artwork, or equations) to construct a page that can adapt to its environment through the stretching and shrinking of its component rectangles. As are all other components of our dms, layout objects are implemented using the model of section 2. They are integrated into the document formatting framework described in the last section by assigning multiple roles to boxes (for example, springbox and column). So, if the user changes a page format the page contents will immediately adapt: page design and formatting are not separate domains that cannot communicate.

Though the above scheme has some applications, such as dividing windows into regions, the class of patterns it can generate is too restrictive for full page layout. The inadequacy is rooted in its inability to align more than two positions with one another. The only alignment in springboxes is of common sides of adjacent rectangles with the joints of individual springs, and of sides of split boxes with those of their containing box. To springboxes, then, we add *alignments* between (equality of) horizontal or vertical positions. This mechanism permits the specification of a wider class of layouts (for example, systems of boxes that overlap can be defined).

Figure 3 gives three examples of useful and non-trivial page constructs that can be produced with our mechanism. The first of these is a page with a footnote, in which the footnote and page regions each occupy one partition of the text-region box. The two boxes grow as the need arises, the text downwards and the footnote upwards. There is one footnote partition for each footnote in the text, created by the page when a footnoted item is inserted. In 3(b) a margin note is associated with a word on the page. The arc indicates an alignment, thus the note follows its reference if it moves. 3(c) shows a page with a cutout. The size and placement of the cutout can be changed, and the text regions reshape and align themselves around it. These changes will trigger reformatting if necessary.

The layout facilities discussed above operate at a fairly low level. While we think they are flexible enough to be used by authors, springboxes and alignment may be more useful as the basis of less general, but more intelligent, page views. For example, multi-column formats can be defined in which regions can be cut out from a page, with automatic reshaping of page boxes to avoid overlapping with the cutout. In this way, users could change page layouts without being involved in mechanical translations from useful high-level page constructs to geometric page decompositions.

6. Summary

We have described an interactive system that allows the user to directly edit formatted documents. Key among the mechanisms used to achieve a responsive interface is a constraint oriented processing model, in which document objects automatically adjust themselves in response to changes in closely related objects. The constraint model, together with an extended hierarchical document structure, provides good communication between different parts of a document, an important property for a system that promises a fully consistent view of a complex document at all times. Furthermore, our model of independent document objects coupled by constraints supports the inclusion of an unspecified number of object types (such as different kinds of graphics and tables) in documents. Also important is incremental reformatting, which minimizes the amount of work needed to propagate typographical changes far from their source. Finally, a simple page layout scheme was presented that is able to specify pages more complex than those generally obtainable from current systems.

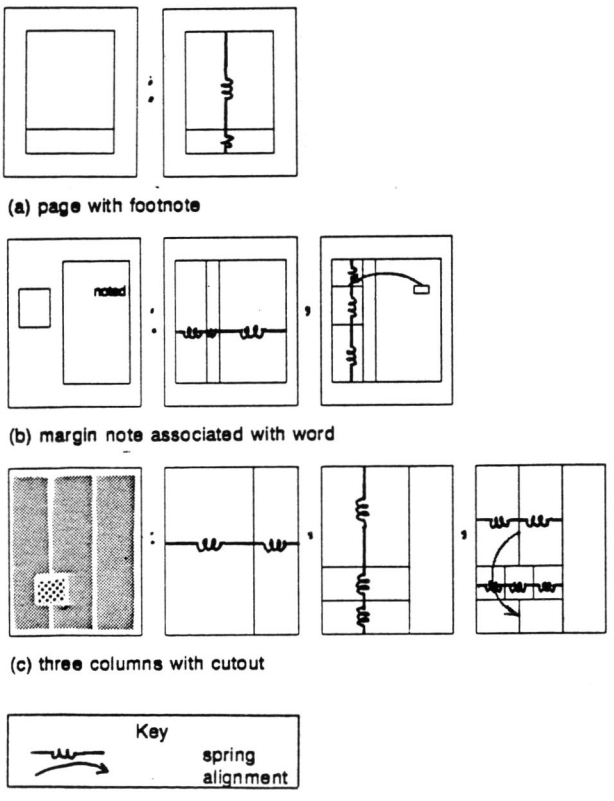

Figure 3. Building page layouts from springboxes.

References

[Chamb] Chamberlin, D.D., O.P. Bertrand, M.J. Goodfellow, J.C. King, D.R.Slutz, S.J.P. Todd, B.W.Wade, "JANUS: An interactive document formatter based on declarative tags", IBM Syst. J., 21, 3 (1982)

[MeyvD] Meyrowitz, N., A. van Dam, "Interactive editing systems: Part II", ACM Computing Surveys, 14,3 (Sept. 1982)

[MacWrite] MacWrite reference manual, Apple computer (1984)

[KerLe] Kernighan, B.W., M.E. Lesk, "UNIX document preparation", in Document Preparation Systems, Nievergelt et. al. (eds.), North Holland (1982).

[Knuth] Knuth, D.E., The TEXbook, Addison-Wesley (1984)

[Reid] Reid, B.K., "A high-level approach to computer document formatting", in Proc. 7th Annual ACM Symp. on Prin. of Prog. Lang. (1980)

[HamIA] Hammer, M., R. Ilson, T. Anderson, E. Gilbert, M. Good, B. Niamir, L. Rosenstein, S. Schoichet, "The implementation of etude, an integrated and interactive document production system", Proc. ACM Conf. Text Manipulation (1981)

[Good] Good, M., "Etude and the folklore of user interface design", Proc. ACM Conf. Text Manipulation (1981)

[Winog] Winograd, T., "Beyond programming languages", Comm. ACM 22,7 (July, 1979)

[BarSh] Barstow, D.R., H.E. Shrobe, "From interactive to intelligent programming environments", in D.R. Barstow, H.E. Shrobe, E. Sandewall (eds.), Interactive Programming Environments, McGraw-Hill (1984)

[Reiss] Reiss, S.P., "Graphical program development with PECAN program development systems", Proc. ACM Softw. Eng. Symp. on Practical Softw. Dev. Envir. (1984)

[FurSS] Furuta, R, J. Scofield, A. Shaw, "Document formatting systems: survey, concepts, and issues", ACM Computing Surveys, 14,3 (Sept. 1982)

[SteSu] Steele, G.L., G.J. Sussman, "Constraints", MIT AI memo 502, (Nov., 1978)

[Borni] Borning, A., "Thinglab - a constraint oriented simulation laboratory", Stanford Ph.D. dissertation (July 1979)

[Gosli] Gosling, J., "Algebraic constraints", CMU Ph.D. dissertation (May, 1983)