ELECTRONIC PUBLISHING, VOL. 6(4), 515–526 (DECEMBER 1993)

Separate compilation of structured documents

MICHAEL J. GROVES AND DAVID F. BRAILSFORD

Department of Computer Science
University of Nottingham
University Park
Nottingham, NG7 2RD
UK

SUMMARY

This paper draws a parallel between document preparation and the traditional processes of compilation and link editing for computer programs. A block-based document model is described which allows for separate compilation of various portions of a document. These portions are brought together and merged by a linker program, called *dlink*, whose pilot implementation is based on *ditroff* and on its underlying intermediate code. In the light of experiences with *dlink* the requirements for a universal 'object-module language' for documents are discussed. These requirements often resemble the characteristics of the intermediate codes used by programming-language compilers but with interesting extra constraints which arise from the way documents are 'executed'.

KEY WORDS Link editing Separate compilation Structured documents Formatting Troff PDF

1 INTRODUCTION

The printed page is the most common physical manifestation of a document but electronic publishing, in all its forms, has demonstrated the value of regarding the computer as being more than just an aid to producing printed output via 'new technology'. Documents can usefully be regarded as computer programs of a new and different kind which possess some fascinating properties when compared to traditional computer software. One obvious parallel is that documents tend to undergo constant development and incremental change, just as computer programs do. Consequently, one might expect that a document could be built from pre-processed modules in much the same sort of way that software development systems support separate compilation and link-editing of object modules. However, it turns out that there are some subtle difficulties in carrying over this analogy completely.

This paper examines how far the processing of documents can be made to resemble the compilation and linking of computer programs. It then examines what is required for a generalised link-editing language for documents.

2 STRUCTURED DOCUMENTS

Documents may be analysed at a number of different levels. For example, a document may be seen as a series of alphanumeric characters, interspersed with punctuation and laid out on physical pages. In a sense, this corresponds to the final executable machine-code

0894–3982/93/040315–12$11.00
Received 15 August 1993
Revised 1 December 1993

of a computer program. The difference is that *humans* can directly 'execute' a document by reading it, usually in a serial fashion but with occasional 'subroutine calls' when references are encountered to footnotes, figures, bibliography citations and so on. The various languages of the world, and their character sets, can be compared to different instruction sets for various types of computer hardware. But whereas all computers share the common ground of representing their programs, ultimately, as unadorned ones and zeroes, the analogy, in the world of documents, would be the adoption of a fixed-pitch single-typeface representation for all the world's character shapes. Happily for us documents do not exhibit such tedious orthodoxy. We scan a document not just for its meaning but also for the elegance of its type size and typefaces, the quality of its hyphenation and justification, the appropriate use of bold-face type for headings, the absence of widows and orphans and so on. All of this adds an extra layer of interest and complexity to the process of understanding or 'executing' a document.

At a higher level, a document may be described according to the logical structure of its components — chapters, sections, paragraphs, etc. — with physical attributes being ignored. Here, it is not difficult to see the connection with the data structures, classes and procedures of a high-level computer language where, again, the final 'physical' attributes of the eventual machine code are not relevant. And just as procedure libraries are set up so that frequently-needed code can be used many times, so also may parts of a document be easily reused, or the whole document may be reformatted in a different way, without the logical structure and content having to be changed. This method of document specification using *generic markup* was first popularised by the Scribe document formatter [1] and subsequently by LaTeX [2]. It has also been incorporated into SGML [3] and ODA [4]. Documents represented in this way may be termed *structured documents* [5]. Shaw [6] associates three distinct representations of a document with a document processing system;

- The *document model representation* specifies the *logical structure* of the document.
- The *output model representation* describes the physical appearance of the document, for example, the fonts, character positions, point sizes, line breaks etc.
- The *display model representation* is the form usually displayed on the output device (usually the printed page or displayed on screen).

The document model representation is mapped to the output model representation by *formatting* which is mapped to the display model representation by *viewing*. The main areas of interest for us are first, the structured document representation itself, and second, the way in which the mapping between the document model and output model is carried out. There is a clear parallel here between the formatting of documents and the compilation of computer programs.

3 COMPILERS AND LINKERS

A translator is a program that takes as input a program written in one programming language (the source language), and produces as output, a program in another language (the object or target language). If the translator is translating a high-level language into a low-level language, such as assembly language or machine code, then the translator is called a *compiler* [7].

When writing a large software system it is convenient to break the program up into separate modules and to compile each module, in turn, into object code. A *linker* is then used

to link together all the modules to form an executable program. On modifying a program, only the modules that have changed need to be recompiled. This separate compilation facility has greatly decreased program development time, especially when used with a program maintenance utility such as the UNIX *make* [8] program.

For most high-level languages, translating source programs into executable code involves two steps: compiling and linking. Compilers translate source code directly to object code or to assembly language that is then compiled into object code. The role of the linker [9] is;

1. to combine object modules and run-time libraries into a stand-alone executable program with a uniform linear address space.
2. to resolve all cross-references between modules, and,
3. to set the run-time addresses of all symbols.

Linking is usually carried out after compilation but before loading. This last-named process is where an executable program, produced by the linker, is placed in memory by a loader [10]. Once the loaded program begins to execute, its linear address space may be mapped onto 'pages' within the computer. These pages are imposed by the hardware, and organised by operating system software, so as to improve memory performance and utilisation. The applications programmer need not be concerned with them.

The input to the linker consists of one or more subprograms in *binary symbolic* form which are called *object modules*. Addresses in an object module are usually offset from zero. When linked these addresses must have a relocation constant added. A relocation dictionary contains a list of those addresses that need to be relocated.

Linking usually requires two passes. During the first pass the linker reads all object modules and builds a table of module names and lengths, and a global symbol table containing all external references. During the second pass the object modules are read, relocated, and linked — a module at a time.

It is instructive to relate all of these processes to the activities involved in document creation. A document can be split into parts such as chapters, sections etc. These are often kept in separate files so that each part can be worked upon and processed separately from the rest of the document. When the whole document is to be processed, all the parts are formatted together to resolve page numbers, section numbers, cross-references etc. But here lies the important difference between the separate compilation of programs and document processing. To combine parts of a document, and to resolve cross-references, the vast majority of text processing systems will require the whole document to be re-processed. With some formatters such as Scribe and LaTeX the formatting process can take two or even three runs to resolve all references. It might be thought that the formatter is fulfilling the role of compiler *and* linker for the document, but the analogy would be misleading. In fact, the formatter re-compiles the entire document and the various portions of it are re-combined and re-formatted from a document's source code rather than from 'semi-compiled' object modules.

It is now clear that the preparation of large and complex documents would be greatly helped if document preparation systems offered genuine link-editing facilities, but the fact remains that the overwhelming majority of systems choose to recompile the entire document. This shortcoming is due, in part, to the fact that link-editing of document components, and checking their consistency, can be much harder than the process of

combining object modules for a program. The first requirement for successful document link-editing is to analyse the requirements for a suitable intermediate language in which the ‘document object modules’ can be expressed.

4 DATA ABSTRACTION LEVEL FOR A DOCUMENT LINKER

In the previous section we drew a parallel between the pages of a printed document and the pages of a computer program when it executes on paged and segmented computer hardware. The key difference is that the precise layout of machine code within a hardware page is seldom the subject of aesthetic scrutiny. Moreover, when a call is made to a procedure most modern hardware will implement a fixed-length instruction for the call itself and a fixed-length address structure to contain the destination address of the procedure. If by any chance the precise length of an instruction, or a data structure, is hard to work out then there are various ways to get round the problem, and any infelicity is ultimately concealed within a binary program which is not directly visible to human eyes. Only with the determined use of a debugging tool can one discover the ‘padding’ of a data structure with extra zeroes or the insertion of benign ‘no-op’ instructions. Contrast this with the situation where a forward reference (c.f. ‘procedure call’) within a document will certainly cause a small but significant change in page layout if it says “see section 2.1.1” as opposed to “see section 2.1” and in pathological cases it may alter entirely the hyphenation and pagination of the document from that point on. In other words, if we add new sections to a document they may affect the ‘address length’ of the ‘procedure call’ at the place where the section is cited.

This is not to imply that hardware pagination of computer programs is entirely a free-for-all. Occasionally there may be particular data structures which must reside in some particular page (e.g Page 0) or which must not straddle a page boundary, but for the most part the uniform address space of the linked program can be mapped onto hardware pages without having to worry about the machine-code equivalents of footnotes, page headers, ‘floating’ material, widows, orphans and the like.

In the case of documents the precise mapping of modules, or ‘blocks’, of pre-formatted code onto pages has to take into account all of the features just mentioned. It follows, therefore, that the mapping of printed material onto pages requires an intermediate language, at the link-editor stage, which has easy access to low-level details of typefaces, point size, leading, size of tables, current position on page and so on. But if we did have such a language there still remains the question of the size and nature of the document components that should be separately compiled and stored as ‘object modules’.

In other words, if a document were to be link edited then at what level of granularity should link editing take place? A linker usually works at the procedure level and it resolves external references and relocates addresses. A document linker must also resolve cross-references, page numbers, and it must ‘relocate’ component numbers such as chapter and section numbers. An additional problem for a document linker, which we have already noted, is that changes in the formatted width of a reference will affect the formatting and justification of the page itself.

The document model representation describes the logical structure of the document. The typical example is the tree structure of a book with chapters, sections, paragraphs etc. A problem in the structured document model representation is the level of granularity of

the most primitive objects. Granularity, or where the document *content* takes over from the document *structure* [11], is identified in our model with features such as paragraphs, tables, figures, footnotes etc.

5 THE BLOCK MODEL

The block model describes a method for the mapping from the document model representation to the output model representation as defined in section 2.

A block is taken to be some 'rectangular' part of a document: for example, a paragraph, table, figure etc. It may be of arbitrary size. Blocks can have two main properties — they can float and/or they can be broken. This results in four general block types;

1. A breakable block (i.e. unable to float).
2. An unbreakable block.
3. A breakable float.
4. An unbreakable float.

Blocks also have several other properties. They have a *need* which is the minimum amount of space required for the block on the page. This could be the space required to prevent a widow from being formed if the block is split, or it could be the height of an unbreakable block. The height of the block is, naturally, the amount of vertical space taken up by the block on the page. The block may also be referenced by other blocks. The most common example of this is that of a footnote being referenced within a text block.

Blocks are separated on the page by *glue*. This is analogous to the glue in TEX [12] and each block has a value for the glue separating the block from the previous block and the amount by which it may be stretched or shrunk to justify the page.

An important point to note with the block model is that it separates the process of line breaking which is part of the formatting/compilation phase from page breaking which is now part of the linking phase. Text is formatted into blocks relative to some local zero of page coordinates. The linker then relocates these blocks onto pages in the linking process and makes their positions absolute (this is analogous to readjusting the run-time addresses of all symbols, in a program link-editor, once the address spaces have been merged and linearised).

Some previous work by Kernighan and van Wyk for the *pm* macro package and post-processor [13] helped us greatly in the design phase of our document linker. Two other pieces of work have also implemented block models in a form which has been useful to us. ParTexT [14], a parallel processing system for documents, implemented block-based page layout: text is split up into blocks which are then formatted separately in parallel and a post-processor creates the page layout using the formatted blocks. The Quill document editor [15] has implemented a more sophisticated version of a block-based formatting system [16]. In Quill blocks may be nested inside each other, forming a hierarchy in which the root block is the document itself.

The block model described here is based upon the model used in *pm* and ParTexT where a document is viewed as a linked-list of blocks. Blocks may not be nested but may float forwards or backwards or be broken, depending on the page layout.

6 IMPLEMENTATION

We wanted to base our link-editing experiments on an existing native code or intermediate language of some sort but it was not easy to find one that offered high-level structural information without losing the low-level details vital to the correct placement, floating and splitting of blocks.

In the end we adopted as our experimental link-editing language the intermediate code from *ditroff* [17], which is the device-independent version of the UNIX *troff* [18] batch text formatter. The document linker — imaginatively called *dlink* — processes the output from *ditroff* (hereafter referred to as *ditroff* intermediate code or DIC) to create a document by merging and linking the pre-formatted code from the various input blocks. Extra semantic information is passed to the linker by means of comments inserted into the DIC by a special set of macros.

6.1 *dlink*

Document components to be link edited are written using the *mlink* set of *troff* macros which are compatible with the well-known *ms* macros [19]. The *mlink* macros impose the block model upon documents by inserting start- and end-block markers into the text which are passed through *ditroff* as comments using the **\!** construct. From the markup used in the document the block type can be determined. For example, a simple new paragraph **.LP** or **.PP** command is converted into a 'start breakable block' command. A **.KS** is indicative of an unbreakable float.

Link editing then takes place in two stages. Pass one reads the DIC, picking up the block delimiters inserted by the macro package. From these delimiters it writes an auxiliary file containing a descriptive header for each block. This includes the block number, block type, the height of the block, its 'need', a value for the 'glue' between blocks, links to any other blocks dependent on it (e.g. footnotes) and the byte offset of the start of the block within the intermediate file.

Each document has an overall project or document name as well as an individual filename. For example a document project name may be 'thesis' which is made up of a number of chapters, e.g. **chap1.doc**, **chap2.doc**, etc. All files are processed by the first pass of *dlink* and their block header information is stored in the project auxiliary file.

The DIC output from *ditroff* contains absolute vertical and horizontal motions for the placement of text on the page. However, one of the factors behind the choice of DIC is that it also allows for relative motions which greatly eases the task of relocating and splitting document blocks. Thus, during the first pass of *dlink*, all absolute vertical and horizontal motions are converted to relative motions, based on an origin at the top lefthand corner of the block. The setting of text in DIC is represented as a series of relative horizontal escapements, which is well-suited to our task. Converting to relative coordinates allows the positioning of the block on the page by *dlink* with just one vertical motion command which is issued just before the block is output.

The second pass of *dlink* reads the block header information from the project auxiliary file and uses this information to lay out the blocks on the page. The auxiliary file provides *dlink* with the type, height and need of each block. The page layout algorithm uses this information to lay out the page. Blocks that do not fit on the page are split, or if they can not be split they are either floated or a new page is begun. Pages are justified vertically

using the values stored in the block header for inter-block glue. The space between blocks is stretched or shrunk accordingly to justify the page.

If a block is to be split, the DIC for the block is read into a buffer and all break points for the block are calculated. The page layout algorithm then decides where to split the block taking into consideration the 'need' of the block and the desirability of avoiding widows and orphans.

Finally the page is output as DIC. Positioning of blocks on the page is straightforward by virtue of the relative vertical motions. The position of each block is calculated during vertical justification of the page and each value is output, just before its corresponding block, as a DIC vertical motion command.

6.2 Separate compilation

A facility for separate compilation exists with *dlink*. Consider an example where a document is split into four separate files. After the initial processing of all the files, the second file is modified and so has to be reprocessed. File two is first passed through *ditroff* and piped to the first pass of *dlink*, where the file is reprocessed and the block information is recalculated and stored in the project auxiliary file. Pass two of *dlink* may now be invoked to format the pages of the document. Since file one is unchanged page layout begins on the page where file one ends and file two will begin.

6.3 Component numbering

We have already hinted at some of the difficulties which arise when section numbering changes as a result of revising a document. We now look in more detail at what needs to be done.

Numbered components of a document can be divided into two categories, *structural* components include chapters, sections, subsections etc., and *non-structural* components, for example footnotes, tables and figures. A full discussion of component numbering may be found in [20]. Both categories of numbered components are treated in a similar way with each component having a 'number slot'. This stores the level of the number (for example this subsection has a level two number of 6.3). The physical position of the numbers is marked in the text by a *template*. The template acts as a marker for each number. The template has the same width as the number, so when the block is processed by *ditroff* the block is correctly formatted. During linking, component numbers are resolved and the templates are replaced with the correct number.

The use of a template reduces the number of times the block has to be formatted. In most cases the block is only formatted once but if a number has been formatted with a single-figure template and the number, when resolved at the linking stage, becomes a two-figure number then the block will require reformatting with a larger template — to take into account the increase in width of the number.

Cross-references are handled in a similar way. In contrast to batch formatting systems such as LaTeX and Scribe which require two full passes over the document to resolve cross-references, *dlink* requires only one pass. This is the result of undertaking page layout in memory using solely the information from the block headers in the project auxiliary file. Forward references may be resolved after page layout but before the blocks are actually output as DIC.

It will be clear that the processing of cross-references in documents strongly echoes that in the link-editing of programs. Once again, though, there are some interesting new twists. A call of a subroutine in a high-level language might appear in the source text as **CALL FRED**. When compiled link-edited and loaded it loses all trace of the logical tag **FRED** and would appear, in the machine code, as something like 'JSR 1028' to denote a jump to a subroutine at location 1028 in memory. In a document, on the other hand, our 'subroutine calls' can be left in logical form, ('see section on widgets') in numbered logical form ('see section 2.1.1'), in absolute positional form ('see second paragraph on page 128') or in some combination of these ('see section 2.1.1 about widgets on page 128').

7 DISCUSSION AND CONCLUSIONS

At the purely physical level of document appearance *dlink* provides improved layout compared to the standard *ms* macros of *ditroff*. Pages are vertically justified and widows and orphans are avoided. We have already mentioned Kernighan and van Wyk's *pm* which does a good job in vertically justifying text and removing widows and orphans, especially when producing one and two-column page layouts. In a similar way to *dlink*, the *pm* macro package inserts special comments in the DIC which are used by the postprocessor to lay out each page. It then splits the document up into groups, called streams and floats, with each group made up of a series of *slugs* or lines of text. But although it may seem superficially similar to *dlink*'s blocks, *pm* does not implement a true block model. It still uses a line-based approach to page layout and, as a result, it is unable to handle the splitting of footnotes and generation of footnote rules correctly. It also does not offer the separate compilation facilities of *dlink*.

The block model simplifies the task of formatting by taking away a large part of the formatting process from the formatter. In this example, *ditroff* is used purely as an assembler of the marked up document into blocks of DIC with the page layout being carried out by *dlink*. This approach has several advantages. All the formatted blocks are available for manipulation by the page layout algorithm and manipulating lists of objects is much easier than incorporating a complex page-layout algorithm into a line-based formatter.

One of the main advantages of separate compilation of programs is that of decreasing the time spent compiling and linking. This benefit is also seen with *dlink*. First time processing of a document with *dlink* is slower than processing the document in one file with *ditroff* because of the time needed to create the auxiliary file. However, after initial processing reasonable speedups are achieved. For example, a 30,000 word document marked up with the *ms* macros and split into four separate files of roughly equal size takes approximately 1.2 times longer than plain *ditroff* for initial processing with *dlink*. Reprocessing and linking one file is approximately three times faster than formatting the whole document with *ditroff*. Greater speedups may be obtained by splitting the document up into smaller files. However, the restricting factor is that each file has to be processed by *ditroff* before linking. In the example given, over three quarters of the time is taken by *ditroff* in the reprocessing and linking of one file. One level of complexity is removed by using the *mlink* macro package. Because page makeup is undertaken by *dlink* the *mlink* macros are able to avoid complex *ditroff* commands such as diversions and environments.

The block-based formatting model also has advantages for the writer. A block represents a logical part of the document and can be easily understood. The use of the block model provides *dlink* with a certain amount of knowledge about the document's structure. This

information allows *dlink* to lay out the page in a more intelligent manner than plain *ditroff* with the *ms* macros.

7.1 A revised document model

The block model and *dlink* can be seen to follow the document model described in section 2. However, the mapping between the document model representation and the output model representation has been split into two parts by *dlink*. The mapping, or 'formatting', first involves a 'low-level' assembling of text into blocks. The second part of the output model representation is the page makeup. From this we have split the output model representation into two distinct representations. First, the *block model representation* and second, the *page model representation.* These correspond almost exactly to the compilation and link-editing phases of a computer program.

Thus, with *dlink* we can now associate four distinct representations of a document;

- the document model representation,
- the block model representation,
- the page model representation,
- the display model representation.

The document model representation is mapped to the block model representation by formatting the document model representation into blocks. Each block is a separate entity, described in an intermediate language with vertical coordinates relative to a particular origin. The block model representation is mapped to the page model representation by the page layout process. The blocks are laid out on the page. Also cross-references, component numbers and page numbers are resolved. By a suitable extension of the structured comments inserted into the DIC it would be possible to reify the various cross-references as hypertextual links to other objects in the same document, or within other documents.

7.2 Why use ditroff and DIC?

Why did we use *ditroff* and DIC instead of the more recent and technically superior TEX [21]? This question is answered, in full, in a recent paper by Srouji and Berry [22], but an outline of the reasons is as follows. First, *troff* and it's plethora of pre- and postprocessors provides a modular approach to formatting. New functionality may be added to *troff* by writing an appropriate preprocessor. This ability of *troff* to adapt so as to meet ever more sophisticated demands is a large part of the success of the UNIX typesetting system.

TEX takes the opposite approach and provides a unified system with many of the features provided by *troff*'s preprocessors incorporated into the main program. However, it is difficult to add extra features with pre- and postprocessors.

A second, and more serious problem with TEX is the lack of information generated by TEX in its output. The DIC output from *ditroff* provides end-of-word and end-of-line markers. This feature is absent in TEX's device independent output code (DVI) [23]. Knowing where line breaks occur within the text is crucial to the block model method of formatting.

Just as a linker will link object modules produced from more than one compiler each using a different source language, so *dlink* should be able to link *ditroff* documents with documents produced by other text processing systems. To illustrate this point a DVI to DIC converter, called *dvi2dic* was written enabling the linking of parts of a document written

with LaTeX to be linked with *ditroff* parts. Although DVI does not provide end of line markers it is possible, for relatively simple documents, to infer where line endings occur provided the point size and vertical line spacing are known. To achieve this and to enable linking by *dlink* a subset of the LaTeX article style has been modified to provide the extra information required by *dlink*. Information is passed through LaTeX in a similar way to that used by *ditroff* by the using of comments with the **`\special`** command. This facility enables LaTeX articles to be linked with *ditroff* documents whilst preserving component numbers, page numbers etc. LaTeX documents processed by *dvi2dic* must use PostScript fonts that are already installed for *ditroff*. All horizontal and vertical motions within the DVI file are converted to relative motions and output at the resolution of the *ditroff* output device as specified in the *ditroff* **`DESC`** file.

7.3 The way forward

As described in the previous section it is possible to convert DVI to DIC and it may also be possible to convert other formatter output languages to DIC, but it is clear that DIC is not an ideal intermediate language for the linking of documents. Its advantages are that it uses 7-bit ASCII, is well designed, supports relative positioning and can have comments inserted into it to convey some semantic information about the text. Its big drawback is that it is highly resolution dependent. All horizontal and vertical movements are based upon the resolution of the output device. Therefore it is impossible to link two files produced for two different output devices. This creates problems when linking documents from different text processing systems.

We need an intermediate language for document link editors which possesses the following attributes:

- Independent of application, hardware and operating system
- Structured and therefore capable of containing document structural information
- Full retention of low-level information (fonts, point size, page length etc.)
- Portable (e.g. 7-bit ASCII)
- Easily converted to a form suitable for previewing and printing
- Easy to correlate display-model ‘objects’ with corresponding source-code objects
- Support for explicit intra- and inter-document hypertext links
- Easy to revise and with full support for indirection and relative addressing when accessing document objects — in particular all objects should be accessible by *relative* byte offsets within the file and by *relative* page displacements when being viewed or typeset

Many languages, ranging from Office Document Interchange Format (ODIF) through to PostScript, via Computer Graphics Metafile (CGM) and Rich Text Format (RTF) can address some, but not all, of these needs. However, the recent announcement from Adobe Systems Inc. of their Acrobat software, and its underlying Portable Document Format (PDF) [24], is particularly exciting because it can potentially address all of the above issues.

Adobe’s PDF is based on Level II PostScript [25] for its imaging model and this enables it to describe documents containing any combination of text, graphics and images in a device- and resolution-independent format. Moreover, a PDF file describes not only the visual aspects of a document, but also additional elements such as annotations, outlines,

hypertext links and thumbnail views of each page. Internal tables enable objects to be addressed indirectly and it is possible that future releases of the software will support an extension of these tables to allow blocks of low-level PDF, which image the page on screen or on a printer, to be cross-correlated with the embedded, tagged source code that produced them. Research is currently under way at Nottingham to evaluate PDF as a candidate for a universal document link-editor language.

REFERENCES

1. B. K. Reid, *Scribe: A Document Specification Language and its Compiler*, 1980.
2. Leslie Lamport, LaTeX*: A Document Preparation System*, Addison-Wesley, 1986.
3. *GCA standard 101-1983, Document Markup Metalanguage: GENCODE and the Standard Generalized Markup Language (SGML)*, ed., Charles F. Goldfarb, Graphic Communications Association, 1983.
4. ISO/DIS 8613 Information processing, *Office Document Architecture (ODA)*, 1986.
5. *Structured Documents*, eds., J. André, R. Furuta, and V. Quint, Cambridge University Press, 1989.
6. R. Furuta, J. Scofield, and A. Shaw, 'Document formatting systems: survey, concepts, and issues', *Computing Surveys*, **14**(3), 417–472, (1982).
7. Alfred V. Aho and Jeffrey D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Massachusetts, 1977.
8. S. I. Feldman, *Make — A Program for Maintaining Computer Programs*, AT&T Bell Laboratories, Murray Hill, New Jersey 07974, August 1978.
9. Leon Presser and John R. White, 'Linkers and loaders', *Computing Surveys*, **4**(3), 149–167, (September 1972).
10. D. W. Barron, *Assemblers and Loaders*, Macdonald and Jane's, Macdonald & Co. (Publishers) Ltd., 1972.
11. Frans C. Heeman, 'Granularity in structured documents', *Electronic Publishing — Origination, Dissemination and Design*, **5**(3), 143–155, (September 1992).
12. Donald E. Knuth and Michael F. Plass, 'Breaking paragraphs into lines', *Software — Practice and Experience*, **11**, 1119–1184, (1981).
13. Brian W. Kernighan and Christopher J. Van Wyk, 'Page makeup by postprocessing text formatter output', *Computing Systems*, **2**(1), 103–132, (Spring 1989).
14. David R. Evans, *An Investigation of Parallelism in Document Processing*, Ph.D. dissertation, University of Nottingham, Nottingham, NG7 2RD, November 1990.
15. Donald D. Chamberlin, Helmut F. Hasselmeier, A. W. Luniewski, Dieter P. Paris, B. W. Wade, and M. L. Zolliker, 'Quill: An extensible system for editing documents of mixed type', in *Proceedings of the 21st Hawaii International Conference on System Sciences*, Washington DC, (1987). IEEE Computer Society Press.
16. Allen W. Luniewski, 'Intent-based page modelling using blocks in the Quill document editor', in *Proceedings of the EP90 Conference*, pp. 205–221. Cambridge University Press, (September 1990).
17. Brian W. Kernighan, 'A typesetter-independent TROFF', *Computing Science Report No. 97*, (March 1982).
18. Joseph F. Ossanna, 'NROFF / TROFF user's manual', *Computing Science Technical Report No. 54*, (October 1976).
19. M. E. Lesk, *Typing Documents on the* UNIX *System: Using the -ms Macros with Troff and Nroff*, Bell Laboratories, Murray Hill, New Jersey 07974, 1978.
20. Michael A. Harrison and Ethan V. Munson, 'Numbering document components', *Electronic Publishing — Origination, Dissemination and Design*, **4**(1), 1–19, (January 1991).
21. D. E. Knuth, *The TeXbook*, Addison-Wesley, Reading MA, 1984.
22. Johny Srouji and Daniel Berry, 'Arabic formatting with *ditroff/ffortid*', *Electronic Publishing — Origination, Dissemination and Design*, **5**(4), 163–208, (December 1992).
23. D. E. Knuth, 'Device-independent file format', *TUGboat*, **3**(2), 14–19, (1982).

24. Adobe Systems Inc., *Portable Document Format Reference Manual* , Addison-Wesley, Reading, Massachusetts, June 1993.
25. Adobe Systems Inc., *PostScript Language Reference Manual* , Addison-Wesley, Reading, Massachusetts, second edition, December 1990.