A Multiple-Representation Paradigm for Document Development

Pehong Chen

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, CA 94720

July 1988

A dissertation
submitted in partial satisfaction of
the requirements for the degree of
Doctor of Philosophy in Computer Science in
the Graduate Division of
the University of California, Berkeley.

A Multiple-Representation Paradigm for Document Development

Pehong Chen

Computer Science Division
University of California
Berkeley, CA 94720

July 5, 1988

Abstract

Powerful personal workstations with high-resolution displays, pointing devices, and windowing environments have created many new possibilities in presenting information, accessing data, and efficient computing in general. In the context of document preparation, this workstation-based technology has made it possible for the user to directly manipulate a document in its final form. The central idea is that a document is immediately reprocessed as it is edited; no syntactic constructs are explicitly used to express the desired operations. This so-called *direct manipulation* approach differs substantially from the traditional *source language model*, in which document semantics (structures and appearances) are specified with interspersed markup commands. In the source language model, a document is first prepared with a text editor, its formatting and other related processors are then executed, usually in batch mode, and the result is obtained.

A complete document development process involves a number of subtasks ranging from authoring, reading, filing, to printing. There are certain aspects of document development that are best-suited to a source-language approach while others are easier to deal with using direct-manipulation techniques. A hybrid paradigm combining the best of both approaches seems most desirable. In such a hybrid system, a document has at least two representations: a *source* representation with embedded commands that yields flexible high-level abstractions, and a *target* representation displaying an object's final appearance that gives precise placement and orientation in response to direct manipulation.

Simultaneously maintaining more than one user-manipulable representation of the same document is not an easy task. In particular, the historically batch-oriented processors that correspond to source-to-target transformations would have to be made *incremental.* Furthermore, there must be a systematic way of mapping changes from the target representation back to the source representation. Finally, an effective intermediate representation needs to be derived in order to make transformations in both directions possible.

Another interesting issue concerns the integration of system components. Because a complete document-development environment involves many tools and processors, it is important to make the system "seamless". A coherent set of user interfaces is also imperative so that context switches between different subtasks can be reduced to a minimum.

In this dissertation, the concept of multiple representations is first examined. A complete document development environment's task domain is then identified and several aspects of such an environment under both source-language and direct-manipulation paradigms are compared and analyzed. A simple but robust framework is introduced to model multiple-representation systems in general. Based upon this framework, a top-down design methodology is derived. As a case study of this methodology, the design of VORTEX (Visually-ORiented TEX), a multiple-representation environment for document development, is described. Focuses are on design options and decisions to solving the problems mentioned above.

Specifically, the design and implementation of VORTEX's underlying representation transformation mechanisms in both the forward and backward directions (i.e., the incremental formatter in the forward direction and the reverse mapping engine in the backward direction) and the integration techniques used are discussed in detail. A prototype of the VORTEX system has been implemented and it works. Finally, this multiple representation paradigm for document development is evaluated and the underlying principles with implications to other application domains are discussed. Some research directions are pointed out at the end.

Michael A Harrison
5 July 1988

To my wife Adele,
my sons Albert and Nelson,
who went through all this with me.

Acknowledgements

I am most indebted to my research advisor, Michael A. Harrison, for being a source of guidance, advice, support, and encouragement. To me, Mike is a teacher and a friend.

I would like to thank Richard Fateman and Michael D. Cooper for serving on my qualifying and thesis committees, and Susan L. Graham and Paul N. Hilfinger for serving on my qualifying committee. Many discussions with them have generated enlightening ideas and have kept my work on track.

I gratefully acknowledge crucial contributions to this project from my colleagues in the VORTEX group. They are: John L. Coker, Jeffrey W. McCarrell, Ikuo Minakata, Ethan V. Munson, Steven J. Procter, and Peter Vukovich. Without their ideas, experience, and diligent work, the project would not have progressed so smoothly. Special thanks must go to Ikuo, who worked closely with me in the past several months to implement the incremental formatter. I would also like to thank Pat Monardo for devoting his time to CommonTEX, a faithful translation of TEX from Pascal to C, upon which the VORTEX incremental formatter is based.

My wife Adele deserves my warmest thanks and gratitude for her love, patience, and support during the years of my graduate study.

I would like to express my gratitude to the various institutions that have supported my research through generous graduate fellowship grants. From 1983 to 1984, I was supported, in part, by a grant from the Chinese American Education Foundation in Chicago. From 1985 to 1986, I was supported by the state of California MICRO (Microelectronics Innovation and Computer Research Opportunities) Fellowship. During the academic years 1986 to 1988, I was supported by an IBM Graduate Fellowship. The VORTEX research has been sponsored by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4871, monitored by Space and Naval Warfare Systems Command, under Contract No. N00039-84-C-0089.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The history of applying computers to document preparation and processing can be traced back to the 1950's. Most of the early systems were simple-minded compared to today's standard. At that time, processor cycles were expensive, memory was a scarce resource, and monofont line printers were the only commonly available output devices. As technology advanced, there has been a significant amount of work done in text processing and document preparation systems in the past 10 to 15 years. A landmark in this development was the 1981 ACM symposium on text manipulation, which summarized the state-of-the-art research of the late 1970's. A later survey done by Richard K. Furuta et al. [56] provided detailed classifications on document formatting systems known at approximately the same time frame. This dissertation research has been inspired by a number of these systems plus several newer ones.

Two different trends can be observed in the development of document preparation and processing systems. In one, which can be called the *source-language model*, documents are prepared with interspersed formatting commands. The system is then run, usually in batch mode, and the results observed. Typical processors of this type include *nroff/troff* [106] and the associated UNIX subsystems and preprocessors [85]. The high point of this model in terms of output quality has been Donald E. Knuth's $\TeX$ [87] and affiliated processors. The basic versions of *troff* and $\TeX$ are *procedural*, in which the user possesses fine control over how the formatting ought to take place. In contrast to this approach, another important scheme in the source-language model is typified by Scribe [112], which allows documents to be specified *declaratively* as logical entities. The declarative approach also includes the set of so-called generic markup languages like GML [60], which was first developed at IBM, and has evolved to SGML [61] as an international standard endorsed by the American National Standard Institute (ANSI) and the International Standard Organization (ISO).

With the advent of powerful personal workstations featuring high-resolution displays, pointing devices, and windowing environments, new possibilities were created in information presentation and manipulation, data accessing, and efficient computing in general. In the context of document preparation, this workstation-based technology has made it possible for the user to directly manipulate a document in its final form. The central idea behind this so-called *direct manipulation model* [75,123], which is the second trend of development being considered here, is that the user manipulates a document's final appearance directly by invoking built-in operators available in the form of menus, palettes, buttons, and so on; no syntactic constructs are explicitly used to express the desired operations. These systems are highly interactive; the result of invoking an operation is instantaneously observed, thereby creating an illusion that the user is "directly" manipulating the underlying object.

Derived from the Bravo editor [92] at XEROX Palo Alto Research Center (PARC), subsequent systems such as Etude [69,76], Pen [5], Mint [71], Tioga (in Cedar) [130], Andra/Lara [67,66], etc. have been developed at various research institutions. Also, with personal computers and workstations becoming commonplace, a number of commercial products like Star [140], MacWrite [10], MicroSoft Word [99], PageMaker [4], Ventura Publisher [135], Interleaf Publishing System [78], FrameMaker [52] and more have already penetrated many homes and offices.

One common feature of direct-manipulation document preparation systems is that editing and formatting are strongly integrated. Furthermore, document semantics, such as page layout attributes, are specified by a declarative language, which can be usually encapsulated as form-based property sheets. These property sheets correspond to certain textual markup tags, which may be imported to or exported from the direct-manipulation system for document interchange purposes. While a source representation is maintained explicitly in the source-language model, the notion of a document semantics specification language is somewhat implicit in direct-manipulation systems.

Each of these trains of development has important advantages and disadvantages.

By and large, the output quality produced by source-language based systems is higher than that by direct-manipulation editors. This is because most source-language based systems are rooted upon a batch-oriented document compiler. Being batch-oriented gives them the unique advantage of exercising global considerations in formatting strategies, which, in terms of output quality, yields a better optimized end result. Direct-manipulation systems are limited in this respect, by certain performance requirements in terms of response time. Quality is frequently compromised as a result.

Another important advantage of the source-language model is the "expressiveness" provided by symbolic languages. Suppose a document processing system is a set of operations defined on a collection of objects. With respect to simple operations there may be little difference between the two models. The major difference becomes noticeable in cases where higher level abstractions such as macros, mathematics, and conditional structures, are required. These are normally available as first-class citizens in a document formatting language. In most direct-manipulation editors, handling complicated cases like these are either impossible or very cumbersome.

On the other hand, the most common criticisms against source-based systems are (1) the unnecessary overhead they always pay in reprocessing a document with minimal changes, and (2) the low degree of interaction and poor interface they have with the user. It is in these areas that the direct-manipulation approach seems preferable. As highly interactive systems, direct-manipulation editors are incremental in nature; they only perform the minimal work required to reprocess a document. They must also provide a good user interface so that the user can directly and efficiently manipulate different objects.

Superficially, the differences between the source-language model and the direct-manipulation model seem to stem from the degree of interactiveness and the level of integration. The former is the canonical interpreter versus compiler issue; there is nothing prohibiting a document composition language from being incremental or interpreted to gain better interactive behavior. Integration is not the dominating issue either; with proper editor support, it has been shown that an effective integrated environment based on batch processors can be created [42].

A major distinguishing factor, therefore, is the explicit user manipulation of the document target appearance versus the manipulation of a programmable language representation. Advocates of the direct-manipulation model claim that such systems bridge the gap between the user's perception and the actual task domain, and that such systems relieve the user from any concern with detail and are very easy to use. On the other hand, critics argue that the friendly user interface comes at the expense of generality and flexibility, or in another word, "power". If none of the two models is perfect, is there another alternative?

1.2 Research Goals

The analysis above strongly suggests a different direction, which calls for a hybrid paradigm that employs multiple representations. A multiple-representation system can take advantage of the strengths of each model and complement individual weaknesses. There are certain aspects of document development that are best suited to a source-language representation, while others are easier to deal with using direct-manipulation techniques. Direct-manipulation user interfaces can be viewed as simply another class of specification languages. By blending a variety of languages, including that of direct manipulation (encapsulated as palettes, menus, etc.) in an integrated environment, the user interface becomes a matter of choice, dictated by convenience or preference. In such a hybrid system, a document has at least two representations: a *source* representation with embedded commands that yields flexible high-level abstractions, and a *target* representation displaying an object's final appearance that gives precise placement and orientation in response to direct manipulation.

Simultaneously maintaining more than one user-manipulable representation of the same document is not an easy task. In particular, the historically batch-oriented processors that correspond to source-to-target transformations would have to be made *incremental*. Furthermore, there must be a systematic way of mapping changes from the target representation back to the source representation. Finally, an effective intermediate representation needs to be derived in order to make transformations in both directions possible. Furthermore, document development is not confined to editing and formatting *per se*. A complete document development process involves a number of subtasks that range from authoring, reading, filing, to printing. An ideal paradigm should be able to create an extensible environment that would allow the bulk of these subtasks to be handled coherently.

It was with these principles in mind that the the VORTEX (Visually-ORiented TEX) project was begun. The mission is to study the multiple-representation approach to document development. Its research goals were stimulated by the observation of a number of conflicts. As if they were inherent in the tasks involved, polarized approaches to document development can be found along four orthogonal dimensions:

1. Document specification: *source language* versus *direct manipulation*, which translates into maintaining program constructs and high-level abstractions versus possessing precise object placement and orientation. As pointed out above, the source language approach is more expressive, while the direct manipulation approach is more reactive.

2. Document evaluation: *quality* or *immediate response*? The former focuses on global considerations and optimal results, which often turns out to be slow in response time. The latter emphasizes immediate evaluation and quick response, and, unfortunately, may have to settle for local considerations and mediocre quality.

3. Degree of detail: *how* versus *what*, or *procedurality* versus *declarativeness*, which is related to the issues of flexibility and fine control on the one hand, and those of logical entities and document styles on the other hand.

4. The glue: *strong integration* versus *weak integration*, which combines everything into a monolithic system in one extreme, while exploiting divide-and-conquer with some simple sharing in the other.

A fundamental question concerns whether a polarized situation is absolutely inevitable. The most desirable paradigm may be a compromise, which takes advantage of the strengths of the rival approaches. The primary objective of the VORTEX research is to investigate this fundamental question, to resolve these inherent conflicts, and to come up with compromises that reflect sensible solutions to the problems posed above.

1.3 Road Map

The next chapter introduces some basic concepts that underly document development. It defines the notion of multiple representations, compares procedural and declarative approaches, and explains why TEX is chosen as the base language. Chapter 2 also covers a few important issues encountered in program development environments that are related to document development. In Chapter 3, a complete document development environment's task domain is identified, and several aspects of such an environment under both source-language and direct-manipulation paradigms are compared and analyzed. It indicates which aspects of document preparation are more conveniently handled under which model, and discusses several approaches to the hybrid paradigm that exploits multiple representations. A simple but robust framework is presented in Chapter 4 to model multiple representation systems in general. Based upon this framework, a top-down design methodology is derived.

As a case study of this methodology, Chapter 5 describes the functional specification of VORTEX, a multiple-representation document development environment. Chapter 6 describes the design and implementation of a VORTEX prototype system; the focuses are on internal structure and reverse mapping. Chapter 7 covers the principles of incremental processing, and specifically, the strategies and their tradeoffs used in VORTEX. Next, integration mechanisms and the techniques employed by VORTEX are analyzed in Chapter 8.

The VORTEX project is not the first to recognize the importance of maintaining multiple representations in a system. Earlier work on multiple-representation systems spans a wide spectrum, among which Janus [30] is a document composition system, Sam [134] is a VLSI layout editor, and Juno [104] is an interactive graphics editor. As others have come to realize this new trend [23,26], systems like Tweedle [11], Lilac [27], and Quill [32] have been developed independently and concurrently with the VORTEX project. There are very interesting commonalities and differences in the approaches taken by these systems in tackling the multiple representation and other document development problems. References to these systems can be found throughout this dissertation.

Chapter 9 comprises three parts: in the first part, VORTEX's multiple representation paradigm and the prototype is evaluated in terms of design and implementation tradeoffs, as well as what may be improved. In the second part, a number of related systems are briefly described in terms of their design and functionality. The the last part, an analysis is given that compares VORTEX with a number of systems, including those mentioned above. In conclusion, Chapter 10 highlights the contributions of this research and points out some underlying principles that may be of interest to other application domains. It also indicates possible extensions to the system and raises potential research issues that can be pursued further.

VORTEX is a rather complex system. Its implementation involved extensive group efforts. This dissertation describes the work for which I have been responsible including the underlying multiple-representation paradigm, the conceptualization of the system and its overall design, and a prototype implementation of the incremental formatter and the pre- and post-processing facilities. Several other people have also contributed to VORTEX. John Coker is responsible for the VORTEX source editor and its Lisp interpreter. Steve

Procter is the primary person behind the VORTEX target editor, to which Peter Vukovich did some extensions. Jeff McCarrell worked on an early version of the target editor. Ikuo Minakata helped in the implementation of the incremental formatter. Ethan Munson has been extending one aspect of the pre- and post-processing facilities, namely bibliography processing.

Several papers and technical reports have been published during the course of this dissertation research. Materials presented in Chapter 3 through Chapter 5 have been published in Reference [41], which deal with the multiple-representation paradigm. An earlier design specification of VORTEX appears in Reference [38]. The design of the pre- and post-processing facilities is described in Reference [42] and a more complete report is published as Reference [39]. References [35,36] are user manuals of the facilities. As part of these pre- and post-processing facilities, the design of an indexing system is reported in Reference [40].

Chapter 2

Basic Concepts

2.1 Multiple Representation Systems

Many large-scale software systems are multiple representation systems in the sense that they often deal with different representations of the same objects. Each representation usually corresponds to a different level of abstraction in terms of the computation it describes. For instance, the purpose of a compiler is to translate a computation described by a high-level language to a sequence of machine instructions. As a special case, a document compiler transforms an object (the document) from its original source form to a formatted target form, with the printer description language being the set of machine instructions.

The *multiple representation problem* [109] has to do with how to systematically carry out the mappings among the different representations of the same object. Depending on the nature of a system, some of the mappings are easier than others. A good example is the traditional asymmetric translation of programming languages. Here, information incorporated in a high-level language program is decomposed into low-level machine code. Although the computation represented by the program itself is preserved algorithmically, the syntax and semantics of its high-level construct are often difficult to recover. Besides, the target representation in this case is generally not human-readable; manipulating it requires a great deal of wizardry. On the other hand, there are cases where the mappings are more symmetric. Document formatting, for one, is a transformation that deals largely with text and appearance specification commands. Modifying the target appearance has a natural correspondence with the source commands used to describe the same effects.

Aside from the issue of symmetry, in early days we were confined by the lack of an effective scheme for manipulating more than one representation at once. However, new technologies have emerged to balance the situation. In the world of document development, for example, it has become possible to preview or even modify a formatted document on the screen the same time its unformatted source is being edited. The ability to maintain and present both representations at the same time opens many doors to new research in text editing, document processing, and user interfaces.

This research derives a model of document preparation, in which both source and target representations of a document are explicitly presented to the user via a set of cooperating editors. An incremental formatter provides the necessary transformation from source to target. A reverse mapping facility propagates changes made to the target appearance back to the source. Moreover, the editing of non-textual objects such as tables, graphics, and raster images is an integral part of the environment.

2.2 Procedurality versus Declarativeness

Document processing systems can also be classified according to their "degree of procedurality", which refers to the granularity of control the user is allowed to possess over a specific task. Equivalently, one can think of this as the amount of information a system must know a priori in terms of the document's style and structure. Consider formatting as an example. At one end of the spectrum, there is what may be called the pure *procedural* scheme, which requires the user to specify exactly *how* the formatting ought to be carried out at the *physical* layout level. At the other end is the pure *declarative* scheme, in which the user specifies just *what* a document should be at the *logical* level; formatting details associated with various document styles are hidden from the user.

There are pros and cons for both schemes. The first consideration is the issue of device independence. Descriptive systems like SGML [61] achieve device independence by associating different formatting functions for different devices to the same document style. Device independence here means the same source document, when processed with the right formatting procedure, can be printed on a wide range of devices from line printers to high resolution typesetters. Rather than maintaining it explicitly at the source level, a procedural system realizes device independence by generating its output in some generic format (e.g., the DVI format [53] in TEX or the *ditroff* [84] format in *troff*), which can then be translated to a variety of device (printer or screen) languages. The premise here is that the capabilities of these devices are comparable and the fonts used by the formatter are available on these devices. Device independence, in this respect, guarantees that exactly the same output can be produced on comparable devices with resolution being the only difference.

The merits of the declarative scheme are that it provides a logical structure for complex documents and it relieves the user from dealing with details of formatting. The system is normally more compact and thus easier to implement. The major limitation, however, is with its rigidness. To manipulate document styles the user must master a set of functions different from what is used for document composition. Hence, it is generally difficult for a casual user to perform fine tuning if the formatted result is unsatisfactory. A direct manipulation approach is more appealing in this respect. Instead of programming in a meta-language for defining the style, all declarative attributes can be encapsulated in property sheets with an obvious form-filling user interface. Then the question becomes whether or not every bit of detail in controlling the formatting information can be parameterized declaratively.

In contrast, it is in the issue of fine control that a procedural system demonstrates its strengths. Another unique advantage of a procedural system like TEX is its extensibility: macros can be used to define high-level structures or even emulate declarative properties (e.g., LATEX [91]). On the other hand, emulating procedural properties in a declarative system like SGML is very difficult. The tradeoff here is "power" versus "ease of use". The two schemes seem to complement each other in many respects. In any event, the degree of procedurality serves as a basis for evaluating different approaches of the hybrid model in document processing.

2.3 The Choice of TEX

TEX is chosen to be the base language of VORTEX for a number of reasons, which center around its unique strengths[1]. Unfortunately, TEX has weaknesses as well, primarily because it is based on a batch-oriented software technology. This section highlights some of the important advantages and disadvantages of TEX that underline the motivation for VORTEX.

[1]Besides the practical reasons that TEX is in the public domain and is in wide-spread use.

2.3.1 Strengths of TEX

TEX has outstanding algorithms for dealing with the basic problems of computerized typesetting. In particular, the line breaking algorithm [90] is excellent and the hyphenation algorithm [93] gives impressive results for relatively small table sizes.

An important contribution of TEX is its simple and effective model of text formatting. Traditional typesetting technology has the notions of *slug*, *box*, *gap*, *galley*, and *page*. A slug is the piece of text that forms a single line. A box is a collection of slugs that represents a textual unit such as a paragraph, a heading, or a caption. Gaps are inserted between slugs to adjust the inter-line spacing. Finally a galley is a long tray of boxes, which represents the formatted document before it is broken into pages. This model is generalized by TEX so that it is better suited for computerized document production. TEX makes up pages by pasting together *boxes* with *glue*. A page box is composed of paragraph boxes, each of which comprises boxes of lines, each of which is made of word boxes, each of which is a collection of individual character boxes. All these *boxes* are pasted together using pieces of *glue*. By cleverly parameterizing these two types of abstract objects,[2] TEX makes it possible for its formatting algorithms to yield outstanding results. This model has clearly influenced some later systems like Etude, Pen, Quill, and Lilac. VORTEX is committed to getting the highest possible quality and is 100% compatible with TEX.

TEX's greatest strength is its handling of mathematics. It is in processing mathematics that the advantages of a source-language based system like TEX become very noticeable. In the early stage of this project, an experiment was conducted to compare the work involved to typeset a page of complicated mathematics from a textbook. It was not difficult to accomplish the job using TEX. On the other hand, with XEROX Dandelions available, the same page was typeset using its Star [140] text processing facilities. It took much longer, and the results were very disappointing in terms of appearance. Direct-manipulation equation editing has improved significantly since Star. Systems like the Expressionist on the Apple Macintosh allows mathematical equations to be entered by *plagiarizing* (i.e., by copying template formulas from what is available in system database). It also supports

[2]A box is a rectangular object with three parameters: *height*, *width*, and *depth*. These are referenced with respect to a *base line* and a *reference point* at the left of the base line. Glue has three attributes: *natural space*, *stretchability*, and *shrinkability*. When making large boxes out of smaller ones, TEX will stretch or shrink each piece of glue involved in proportion to its associated stretchability or shrinkability. See Reference [87] for detailed definitions and explanations.

some internal structure for editing purposes (not necessarily for manipulation purposes). For instance, when the upper bound of an integration is expanded, the integral sign enlarges automatically. In an environment like the Macintosh where inter-application data exchange is done via a global clipboard, the output of an equation editor is normally copied into a document editor as an opaque box. The user is normally not allowed to touch this equation as a text image in the document editor, although rescaling may be permitted. Such arms-length interrelations make it hard to keep equations, equation-numbering, and text integrated. This is not a problem in TEX because equations and document text are processed by the same processor. VORTEX inherits the full equation handling capability from TEX in the source representation, although under the current prototype an equation is also treated as an opaque box in the target representation.

Some additional features of TEX are worth mentioning, one of which is the concept of the device independent (DVI) file format [53] that gives the same results on different output devices, and the only limitation is the resolution of the device. VORTEX maintains an internal representation that incorporates more information than a DVI file does. For example, a physical document structure in terms of page, paragraph, word, and character is supported by the VORTEX internal target representation, but not by the DVI representation. Each node in the VORTEX internal representation is typed for prompt interactive processing. Again, this information is missing in the DVI representation. However, a DVI generator can be supported in VORTEX to transforms the complete or partial internal representation into DVI files[3]. Another important feature of TEX is its macro facility, which, among other things, allows one to simulate declarative properties discussed in Section 2.2. A major portion of VORTEX's semantic routines is devoted to handling macro expansions.

2.3.2 Weaknesses of TEX

TEX is oriented toward a batch environment. It integrates only loosely with text editors and does not integrate at all with a number of supporting programs such as bibliography preprocessors, the spelling checkers, and various device drivers including any screen previewers. In a preliminary study [42], it has been demonstrated that the TEX document preparation environment can be greatly improved in terms of its throughput by integrating

[3]The current version of VORTEX formatter is incremental on a per-page basis. A DVI file is generated for each page automatically.

the formatters (i.e., `tex`, `latex`, `slitex` and any TEX dialects) and auxiliary processors with the display editor GNU Emacs [126]. The improved environment has delivered an important message to VORTEX in that "integration" makes life easier. VORTEX entails several editors all integrated coherently with a central incremental formatting module.

The human interface to TEX's macro system is poor; it requires a high degree of expertise to use it. While source-based systems are superior in the quality of their output for difficult typesetting jobs like mathematics, the situation in the user interface is reversed with the WYSIWYG editors. In addition to a regular text editor, in which source files can be manipulated, VORTEX supports another editor with a WYSIWYG-style human interface tailored specifically to the manipulation of the document's formatted appearance. A number of action routines are embedded in this editor, which automatically transforms editing operations done to the target representation back to the source in terms of their corresponding TEX commands. These action routines, based on heuristics derived from expert TEX users' experience, can provide effective assistance to inexperienced users.

TEX is weak in handling non-textual objects. It has primitive facilities for setting tables. Constructing tables in TEX is an enormously difficult and time-consuming chore. The situation is somewhat ameliorated with LATEX, which makes producing tables almost as easy as using *tbl* and *troff*. VORTEX can provide a direct manipulation editor to make table constructions even easier. Another major criticism against TEX is that it has no graphics facility whatsoever, although a "hook" is available (`\special`) and many proposals have been made over the years in public forums such as the computer bulletin board called `tex-hax`, to standardize its use. Although the first version of VORTEX does not support graphics rendering explicitly, it does take this hook into account and has `\special` as a built-in data type.

2.4 Interactive Editing

Interactive editing is an essential part of the VORTEX system, therefore prior developments on interactive editors deserve some attention. An in-depth survey and classifications of interactive editors can be found in Reference [98]. For our purposes, it is sufficient to divide modern display-oriented text editors into two groups: *stream-based* and *syntax-directed*. The stream-based model is the traditional way of manipulating documents and programs, in which text is handled on an individual character basis; no information on either the document or the program's structure or syntax is assumed a priori. The most advanced system of this type is probably Emacs [125], which supports multiple buffers, multiple windows, and a customizable user interface.

A syntax-directed editor recognizes a pre-defined language syntax and supports structure-oriented input and browsing capabilities. One input mechanism in syntax-directed editing is through templates. A template is a syntactic structure that can be displayed upon request. In addition to syntactic keywords, a structure also includes certain place holders into which the user can enter text. Templates can be nested; each place holder may, in turn, contain other templates depending on the syntactic validity. Most syntax-directed editors also support stream-based editing for text manipulations in the place holders. Syntax-directed editors are often used as the front-end to the incremental compiler of program development environments such as Aloe (in Gandalf) [97], the Cornell Program Synthesizer [129], Pan [15], etc.

To contrast the stream-based model and the syntax-directed model, it is necessary to understand the role of underlying source representation. There are three levels of abstractions to consider:

I. the actual images displayed on the screen in terms of characters, words, lines, etc.,

II. the structural representation of the source based on predefined language syntax, and

III. the internal representation of the source which is maintained by the editor.

A designer must decide which abstractions should be made explicit to the user.

Most stream-based editors support abstraction I explicitly. Mappings between I and III are inevitable, although in most cases they are straightforward (e.g., text on the screen to/from the text array internally). Some customizable text editors can be extended to support some functionalities belonging to abstraction II. Emacs, for example, has been extended to work with document preparation or programming environments such as the Document Editor [136], the Lisp Machine [65], Ease [46], etc. The advantage of the stream-based approach is that there is a natural correspondence between the screen image and the internal representation of the text, which makes editing commands more appealing to the user's intuition and the necessary supporting routines easier to implement.

Ideally, a template-based editor should provide abstraction II, instead of III, explicitly because manipulating internal structures may be counterintuitive. This is true for

Pan and the current version of Aloe, both of which require extensive user interface support for the mappings between I and III and between II and III. There are systems that do the opposite. For instance, the early Aloe [50] bypasses II and requires users to work directly with Gandalf's internal syntax tree. Editing commands are implemented in terms of tree traversals, and simple expressions such as $a + b$ must be entered in prefix notation (i.e., $+ab$). This is unnatural from the user's standpoint for what is displayed on the screen is in fact an infix expression.

A syntax-directed system must maintain an abstract syntax tree for the underlying source representation, which is significantly more complex than its counterpart in a stream-based editor. To keep the size of the syntax tree under control, the normal approach is to eliminate all redundant information (syntactic sugar) from being represented in the tree, and have a set of functions that maps the tree representation to the actual screen images. These functions are often referred to as formatting, pretty-printing, or display routines in most systems, or as *unparse schemes* in Gandalf. The advantage of this relatively declarative approach is that a number of *elision* filters can be implemented to provide different *views* of the same source under different constraints. Furthermore, maintaining the syntax tree allows the parser and the expression evaluator to function incrementally. This is another major advantage of syntax-directed editing.

2.5 Incremental Compilation

The key elements of an incremental compiler in program development environments are the incremental parser and the static and dynamic semantic analyzers. These are always closely coupled with some sort of structure-oriented editors in programming environments. The job of the incremental parser is to isolate and reparse only the portion of the abstract syntax tree that has been modified. Static semantics refer to the type of semantics that can be checked by examining the program's lexical context, such as type checking, scope rule checking, etc. Finally the dynamic semantics analyzer provides the necessary run-time semantic support for program execution.

Most programming environments support parsing and static semantics incrementally. A number of incremental parsing algorithms have been proposed by researchers Ghezzi and Mandrioli [58,57], Wegman and Alberga [137,138], Morris and Schwartz [101,121], Kaiser and Kant [82], etc. These algorithms require either augmenting traditional LR or LL parsing algorithms or just syntax-tree transformations. It must be pointed out that these algorithms are concerned with the recognition of arithmetic or logical expressions and assume the support of structure-oriented editors such as the ones described earlier.

There are a variety of strategies to arrive at incremental static semantics checking. Exemplified by the early Aloe system, one approach associates an *action routine* with each production in the grammar requiring semantics. The routine is invoked whenever its corresponding production is used by changes to the abstract syntax tree. Typical operations performed by an action routine include tree modifications, symbol-table updates, user action aborts, etc. A more involved approach uses attribute grammars [86], which attach a set of *semantic equations* to each production in the original context-free grammars. Each semantic equation defines an attribute as the value of a function applied to other attributes in the production. The result is an attribute tree closely-coupled with the abstract syntax tree. As the syntax tree is modified, incremental analysis is performed by updating attribute values throughout the attribute tree in response to modifications. This idea has been realized in systems like the Cornell Program Synthesizer and a detailed algorithm is given in [114]. Yet a third alternative is Pan's logic-programming model [14]. During a Pan editing session *facts* are incrementally updated in a system database and semantic analysis is performed by evaluating *clauses* associated with productions against database contents. The power of this approach is that general logic programming-style queries are supported, which enable the system to gather more information than can be managed by the first two schemes. Once again, all these approaches rely on incremental parsing in the first place, which assumes the existence of a structure-oriented editor.

2.6 Document Development versus Program Development

From VORTEX's point of view, neither pure stream-based editing nor pure syntax-directed editing is adequate. Supporting incremental processing is one of the primary goals of VORTEX; pure stream-based model without any knowledge of document structure would not suffice. The notion of elision is very desirable in the case of browsing large documents, which is simply impossible to achieve without knowing the structure of the source document. On the other hand, documents in VORTEX are not pure "programs". Most syntax-directed editors work on programs that are fragments of *statements* and *expressions*. The template-based input mechanism is impractical in systems like VORTEX for two reasons. First, the bulk of a document is continuous text instead of discrete statements or expressions in the programming language sense. Second, the premise for syntax-directed editing is that the language syntax be predefined. This is difficult in cases like TEX, in which the syntactic extension facility allows macros to be defined anywhere in the document. The user interface must be flexible enough to accommodate potentially unlimited new macros with variable syntax for their argument lists.

The VORTEX approach to text editing is a hybrid of the two models. It is closer to the Cornell Program Synthesizer and Pan than to some other syntax-directed editors in that the text stream and the underlying hierarchical structure are integrated. Unlike the Synthesizer and Pan, which attempt to be multilingual, VORTEX is a system specifically designed to processing TEX documents. In VORTEX, the influence of Emacs can be found in its abstractions on files, buffers, windows, regions, etc., the modeless straight text input mechanism, and most importantly, its Lisp programming subsystem. At the same time, VORTEX provides some syntax-directed capabilities as well, which are restricted to operations involving modifications, such as *delete*, *replace*, *cut*, *paste*, etc.

A document specification language like TEX, which is based solely on syntactic extensions for abstractions, poses very interesting problems with respect to incremental processing. Maintaining the correspondence between a document's source representation on the screen and its internal syntax representation is difficult for macro-oriented languages. A macro may go through recursive expansions before reaching the primitive form that eventually appears in the internal syntactic structure. Changing the definition of a macro means all instances of its invocation must be reevaluated. Under such circumstances, structure-oriented operations and incremental parsing are more involved. One of the major research issues of VORTEX is to perform efficient incremental processing without being penalized by TEX's macro-oriented abstraction mechanism. Chapters 6 and 7 describe the VORTEX system structure and incremental strategies that are designed to solve these problems.

Source-language based document processing is often described in terms of program compilation. The analogy is that a system (e.g., TEX, *troff*, or Scribe) takes a source-level representation as input, and "compiles" object code (e.g., the DVI file in TEX, the `.mss` file in Scribe, and the *ditroff* file in *troff*), which is then "executed" by a device driver. If this analogy were correct, one might expect the techniques of incremental compiling to be helpful. In fact, the analogy is rather superficial. In document-processing systems, parsing is much less involved than in program compilation while code generation is the most time-consuming task.

In particular, VORTEX is concerned with incremental code generation because the execution of a document's target code is effectively the process of rendering its output image. In other words, the sensation of directness is dictated by efficient redisplay (execution) of a document's target view, which, under a multiple-representation paradigm, depends on how efficiently the target representation can be derived from the source representation. It is interesting to observe the differences between a document's target representation and program's object code. In document processing, code generation involves extensive computation for such tasks as hyphenation, line breaking, pagination, and so on. Also, because a document is a sequential list of pages, a local modification may cause ripple effect on the target representation in both its pre- and post-contexts. Code generation in program development is somewhat different. A program is usually modularized and its object code is relocatable; local changes do not cause as much chaos as they would in document processing. Chapter 7 discusses incremental code generation strategies applied under VORTEX.

Chapter 3

Document Development

3.1 Task Domain

There are quite a few tasks that an effective document development environment must be able to accomplish. Understanding them is important because evaluating the souirce-language model and the direct-manipulation model relies on the underlying task being clearly identified. These tasks can be divided into two major categories: *writing* and *reading*. The following is a list of several classes of tasks that are considered essential: classes 1 through 7 belong to the writing category, class 8 covers both, and the last one concerns primarily issues in reading. It is by no means an exhaustive list, however.

1. *Editing*. Tasks here include the editing of text and graphics in general and various classes of special objects like tables, mathematical or chemical formulas, data-driven statistical charts, fonts, raster images, musical scores, animation scenes, digitized audio signals, and so forth. Many of these objects can be intermixed, yielding what may be called a *compound document*. One important consideration in handling compound documents is whether the editing tasks are integrated or disjoint. In the integrated case, objects are displayed on a single "view port" and a context-sensitive set of menus is presented — depending on the type of objects being manipulated. In the disjoint case, selecting an object of a particular type invokes its corresponding editor in a separate window. The various editing tasks, therefore, proceed in distinct contexts until the user explicitly requests that modified objects be reconciled with the system's top-level.

2. *Formatting.* The primary issue in formatting is document appearance, or equivalently, the layout of specific pages. At a global perspective, certain types of documents must obey certain styles. Consequently, some default styles must be provided to cover a wide range of commonly used documents. On the other hand, it is also desirable to support customization so that uncommon styles can be defined by the user. At a finer granularity, either the system or the user must be able to control the placement of objects within a page. This control may be as trivial as setting a piece of text in a certain font, or as complicated as floating text around an arbitrarily shaped object.

3. *Style Specification.* The set of "style definitions" map a document from its logical structure to physical layout. In some systems the creation and editing of style definitions may be a separate task. The most common approach to style specification is to represent document attributes in a declarative language, which can be manipulated using a form-based user interface.

4. *Preprocessing.* This class of tasks refers to operations that must be performed prior to formatting. Typical examples include spelling checking, writing style verification, bibliographical citations, etc. It may also include graphics, table, mathematics, or any other processing filters not integrated with their main formatting engine.

5. *Postprocessing.* These are tasks that cannot be carried out until the main document body has been formatted. Cross references and indexes depend on certain object permutations (e.g., page, section, or figure numberings); they cannot take place unless such numbering has been resolved by the formatting process.

6. *Imaging.* Another important class of tasks is imaging the formatted result onto either the display or the printer. These tasks typically involve interpreting the document's certain intermediate representation (its internal data structure or output file format) and rendering the bits onto the workstation display or translating it into a specific printer language.

7. *Filing/Document Interchange.* This class concerns the filing of documents. There are two important issues. The first has to do with how to effectively save the internal state so that future invocations can be carried out incrementally. The second issue focuses on information interchange and system dependence, namely whether or not a

filed document can be transmitted across different machines and be recognized and processed by heterogeneous document preparation systems.

8. *Annotations/Narrations.* Annotations and narrations can be embedded in a document to convey more information than what is available in the main document body. This additional information can be represented in the form of text, graphics, voice, etc. in an electronic environment. More than one author can be involved in creating such information and the reader can be granted appropriate access. For instance, in an instructional environment, a different set of narrations can be presented according to the level of a particular student. In a publication process, an author can be working with a paper annotated with comments from different referees while annotations intended for the editor are hidden from him or her. Providing these features involves protection and general distributed systems issues.

9. *Dynamic Reading.* From the reader's point of view, a hard-copy document generated by a print medium is fixed and static. Electronic media such as workstation-based environments provide an alternative that does not have to be reminiscent of its static counterpart. A good deal of "dynamics" can be exploited in an integrated document development system. For example, instead of thumbing through the pages for a reference as one would do when reading a printed document, in an integrated environment it is possible to display and examine the target of a reference in a separate window when the source of the reference is being read. This kind of context-sensitive browsing, along with a number of other features not available in the print medium, require complex system support and deserve further investigation.

3.2 Pros and Cons — A Comparison

Most of the tasks listed in the preceding section can be carried out under the source-language model or using direct-manipulation techniques. In some cases, one approach may be more appropriate than the other, while in other cases a combined approach may make more sense. An analysis based on each task in the domain is given in the remainder of this section.

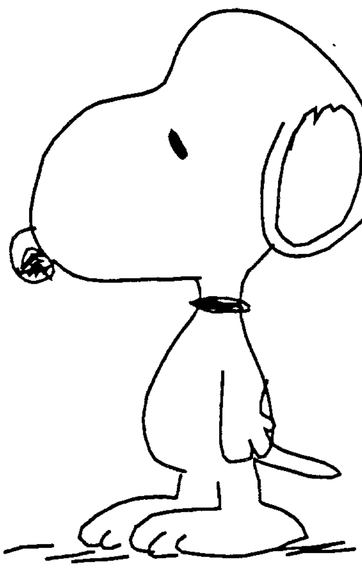


Figure 3.1: *Snoopy*. The picture of Snoopy created using a direct-manipulation graphics editor.

3.2.1 Text Editing

Display-oriented editors can be regarded as direct-manipulation systems when the underlying task is restricted to text editing, such as *vi* [81] and GNU Emacs [126]. They are superior to the old-fashioned line-oriented text editors because a full screenful of text can be directly manipulated.

Emacs also supports a source representation; a Lisp programming subsystem is embedded underneath direct manipulation. Each simple operation corresponds to a Lisp primitive, upon which more complex operations can be coded, which can then be bound to user level commands in terms of a few keystrokes or mouse clicks. This Lisp programming subsystem makes Emacs customizable and extensible and thus a very powerful text editing tool [125].

3.2.2 Graphics Specification

The repertoire of techniques for specifying and generating graphics is very rich. Some of the techniques are source language oriented, others exploit direct-manipulation user interfaces, while a few have employed a hybrid model. There are relative strengths and weaknesses for each approach, as described below.

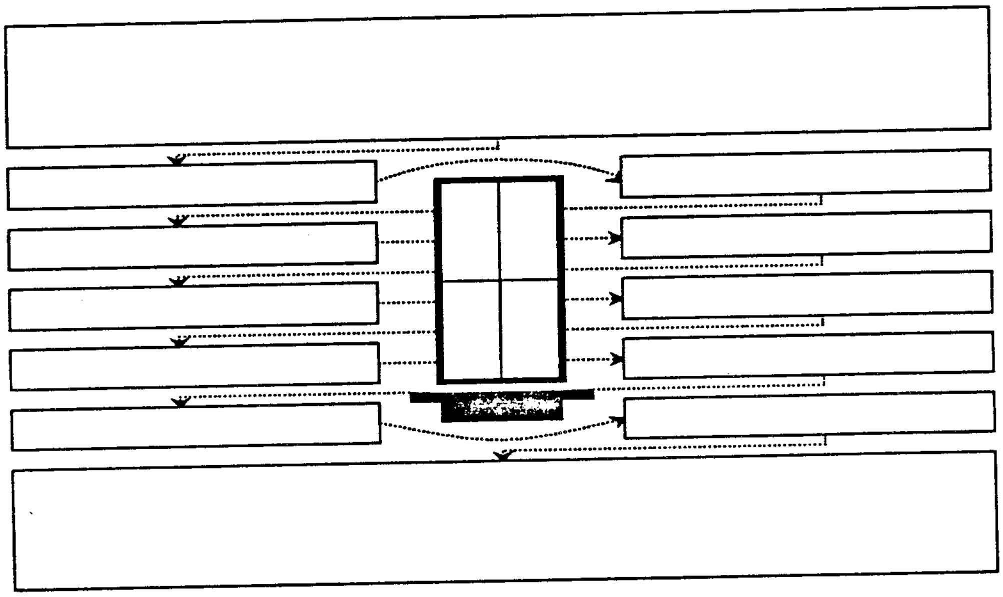
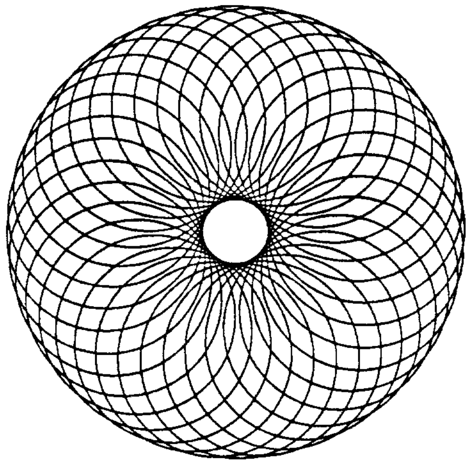


Figure 3.2: *Layout of windowed text.* A technical diagram created by a direct-manipulation graphics editor. This diagram is intended for explaining an exotic page layout, or windowed text, in document formatting. The window in the center is a piece of graphics. The two large boxes on top and bottom contain a paragraph's *lintel* (leading text) and its *sill* (trailing text), which may each contain multiple lines. Each narrow box in the two sides holds a single line of text. Finally, the dotted arrows represent the flow of text.

Direct Manipulation Graphics Editing

In general, it is easier to specify freehand drawings such as the Snoopy shown in Figure 3.1 using a direct-manipulation graphics editor like MacPaint [9] for the Apple Macintosh than directly programming it with a graphics language like *pic* [83] or *ideal* [139]. Also, it is more convenient to create technical drawings such as the one shown in Figure 3.2 with a direct-manipulation editor like MacDraw [8] than with a noninteractive graphics programming language when visual feedback concerning operations such as object placement and orientation is essential.

MacPaint is a typical example of graphics editors specifically designed for creating artistic drawings. Once specified, the notion of object disappears in these editors; what remains is just a raster image. Drawings of this kind are obviously confined by the device resolution, with which they are created. By contrast, MacDraw represents another class of editors more suitable for creating technical diagrams. In MacDraw, objects and their structure are maintained throughout the editing session. When the drawing is done, it is



```
36 { 60 0 45 0 360 arc stroke 10 rotate } repeat
```

Figure 3.3: *Rotated circles.* A set of rotated circles and the corresponding PostScript code used to generate it. PostScript has a postfix syntax, so the one line code here means iterate a procedure 36 times. In each iteration first a circle centered at $(60, 0)$ with a radius of 45 is drawn (in points), then the coordinate system is rotated by 10 degrees.

the description of the objects and their structure, rather than the image, that gets saved. The advantage is that the image can be reproduced on a variety of devices with different resolutions.

A significant intermediate system is the Adobe Illustrator [2], which accepts a raster image, but allows the user to recover the underlying mathematical description by tracing the image. From that point on, the drawing can be manipulated in terms of its component geometric objects, thereby making it possible for the user to fine tune images of any kind.

There is no question, however, that drawings created by direct-manipulation editors can also be described by graphics programming languages or some meaningful textual representations. In fact, drawings created with most of these editors are eventually translated into a source language representation or some textual format for filing and document interchange purposes. The issue here is that in order to create such pictures efficiently, direct responses from the drawing apparatus in terms of its underlying objects' *placement* and *orientation* are crucial. direct-manipulation interfaces, in this respect, act as an interactive agent between the user and the task domain and are more effective than attempting to do the programming at the source level.

Graphics Programming

Direct-manipulation editing breaks down when a great deal of *regularity*, a finer degree of *control*, any sort of *abstraction*, or other basic building blocks of programming languages are required. Figure 3.3 demonstrates the superiority of a graphics programming language in expressing a highly regular design. This picture is a circle repeated many times while rotating the axis. The PostScript source code needed to describe it is a "one liner" shown at the bottom of the figure. One can imagine how cumbersome it would be to create this picture by direct manipulation.

Another good example, which shows the advantage of working with programs, is the *pico* picture editor [73]. It is an interactive editor for digitized graphic images. Instead of directly manipulating the pixels involved as many bitmap editors do, *pico* treats a raster image as a two-dimensional array of pixels. Typical operations such as changing contrast, masking or enhancing pixels, and merging, fading, or transforming the image are defined in terms of a C-style expression language. The user edits images by entering programs expressed in this language with references to the pixel array. This programming approach is able to produce startling effects on images very difficult to arrive at with direct-manipulation bitmap editors.

As another example, suppose the job is to create a page that contains Snoopy and recursively the same page nested within itself like Figure 3.4. No direct-manipulation graphics editor known to us is able to realize such an illustration by any obvious means. With a programming language like PostScript, however, such a page can be defined as a procedure that recursively invokes itself to a specified depth. As was argued previously, it is easier to create pictures like Snoopy with a direct-manipulation editor and that generating a textual representation for the drawings is not difficult. The ideal approach here is to draw Snoopy by direct manipulation first, generate the corresponding code next, and finally perform some adjustments to realize the recursive invocations.

This approach is one flavor of the hybrid model that utilizes direct manipulation as the frontend interface and a meaningful textual representation for off-line filing. This representation has the advantages of being more compact and device independent over directly filing bitmap images. CricketDraw [45], for example, can generate a filing representation either in PICT, Macintosh's standard graphics description format, or in PostScript. Some adjustments may be done on this textual representation before the drawing is filed or sent

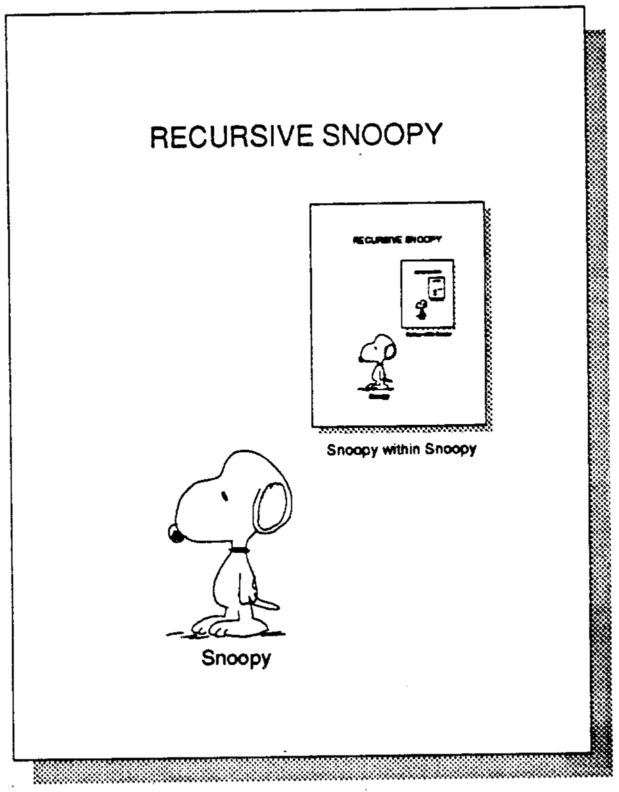


Figure 3.4: *Recursive Snoopy*. The picture of Snoopy recursively appears within itself. This picture is created by drawing Snoopy first with a graphics editor that generates PostScript output and then adjusting the resulting PostScript code. The idea is borrowed from an example given in reference [113].

off to a printer.

Achieving Accuracy

An important issue arises in graphics specification when certain geometric properties of graphical objects must be satisfied or when their precise placement is required. In direct-manipulation editors, the most naive solution to precise placement is to echo the current cursor coordinates on demand. A more commonly used technique is to provide the user with a rectangular grid. Sometimes a gravity facility that automatically attracts the cursor to fixed positions in the grid may be useful. Yet a more powerful paradigm based on the ruler and compass metaphor [21], as what is available in Gargoyle [110], also increases the desirable accuracy.

Another possibility is to apply a class of techniques known as the *constraint-based* *approach* [25], which requires the user to specify a set of parameters that satisfies certain algebraic or logical equations. Given the constraints, the system tries to solve the equations simultaneously and returns the corresponding graphical objects. There may be more than one solution to the same set of constraints, hence a mechanism for selecting the desired solution must be provided.

This approach is often counter-intuitive, which makes it difficult for inexperienced users to add new kinds of constraints to the system. This situation is actually due to the multiple representation issue inherent in this type of system; no matter what the user interface appears on the surface, there is an underlying source program that realizes the constraints. Switching back and forth from a highly-encapsulated graphical interface to a more primitive textual one is difficult for casual users. Some recent developments have focused on graphical specification of constraints with the goal of closing the gap between the task domains [24].

The constraint-based approach has been incorporated in graphics programming languages like *ideal* as well as in graphics editors like Juno [104]. Juno, in particular, is interesting because in addition to a direct-manipulation user interface, the underlying constraint definition language is also made explicit to the user — both representations are editable, and changes will propagate automatically — although the language capability is somewhat restricted from the programming language point of view.

The Hybrid Approach

Clearly the ideal approach is one that exploits the prompt visual feedback available in direct manipulation as well as the programming capability provided by the source language model. This approach is exemplified by the Tweedle graphics editor [11]. In Tweedle, the user is allowed to edit objects in the direct manipulation manner; also supported is a text editor for editing the underlying procedural language description. Each object in the graphical representation corresponds to a piece of code in the textual representation. Changes made to either representation will be mapped to the other automatically.

This approach differs from the off-line hybrid model mentioned earlier in that the textual representation (program) is manipulated interactively. Any modification to the source program is immediately re-evaluated, which then updates the graphical representation, and vice versa. A great number of language design and user interface issues are involved in creating such a system. Typical problems include variable naming and binding, object sharing and linking, and most importantly, the internal state to be maintained in order to incrementally reevaluate the objects.

The programming side of the hybrid approach can be realized using a visual interface, in which program constructs like variables, conditionals, procedures or macros, iterations, recursions, etc. are encapsulated as menu items in the standard direct manipulation fashion. The user is still required to switch back and forth between editing programs (graphical rather than textual in this case) and editing the actual drawings (results of program evaluation), which is the essential the difference between *visual programming* and *program visualization* [103]. These graphical programs are equivalent to the textual ones in every respect. A multiple representation system's emphasis is not so much on having something textual per se, but on explicitly maintaining a representation that is programmable. This is a very important point and will be reiterated throughout this discussion.

3.2.3 Formatting and Layout

Traditional document development systems like the *troff* family of processors [85] under UNIX, Scribe [112], TEX [87], and SGML [61] are largely noninteractive language compilers. A document described in such a language has a textual source representation that contains its content as well as formatting commands. A target representation can be created by passing the source through the formatter. Normally, the task of editing (or simply browsing) either representation is separate from the task of formatting.

By contrast, in direct-manipulation systems like MicroSoft Word [99] or Interleaf Publishing System [78], formatting is an integral part of document editing. Here, the document is reformatted as it is edited. The distinction between source and target representations is blurry or even nonexistent. No textual commands or tags would explicitly appear in the document during an editing/formatting session. Rather, information like document structure and page layout is known a priori; their attributes can be modified by the user via built-in menus or property sheets.

A special class of direct-manipulation systems is the *layout-driven* type, which includes desktop publishing programs like PageMaker [4] and Ready-Set-Go! [95] for the Apple Macintosh and Ventura Publisher [135] for the IBM PC. These systems usually have a strong "import facility" that accepts a variety of file formats generated by ordinary text processors, which have no or limited formatting capabilities. Thus, most layout-driven publishing systems import raw documents and perform detailed formatting as a postprocessing task. What is special about these systems is that they allow page layout to be specified under direct manipulation. Normally, one specifies the layout of a text block or graphics region by "rubber-banding" a box on the screen with a pointing device. The order of these blocks or regions can also be determined by the user. Once the layout is set, text and graphics are then filled in according to the specified flow..

Each of these trains of development has important advantages and disadvantages. By and large, the output quality produced by source language oriented systems is higher than that by direct-manipulation editors. This is because most such source language compilers are batch-oriented, which means their formatting strategies can be better optimized. direct-manipulation systems are limited, in this respect, by certain performance requirements in response time. In order to achieve better quality, some direct-manipulation systems like Cedar's Tioga editor [128], Andrew's [100] base editor EZ [107], and the BBN Diamond editor [47] provide an option that performs some off-line formatting optimizations before the final hard-copy is generated. This formatted version can be previewed on the screen, but not edited. Thus, these systems are essentially direct-manipulation *galley* editors. They assure only "what you see is an approximation to what you get" when higher quality formatting is taken into account.

An important advantage of the source language model is the "expressiveness" or "programmability" provided by symbolic languages. Suppose a document processing system is a set of operations defined on a collection of objects. With respect to simple operations, there may be little difference between the two models. The major difference becomes noticeable when higher level abstractions such as macros and conditionals are desired. These are normally available as first-class citizens in a document formatting language. But in most direct-manipulation systems, manipulating complicated cases like these are either impossible or very cumbersome.

On the other hand, the most common criticisms against source language oriented systems center around (1) the unnecessary overhead they always pay in reprocessing the whole document with only few changes, and (2) the low degree of interaction and poor interface they present to the user. In these areas, the direct-manipulation model seems to prevail. As highly interactive systems, direct-manipulation editors are incremental in nature; they only perform the minimal work required to reformat a document and display the

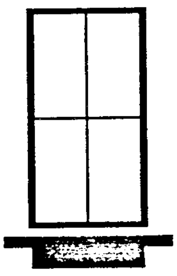
```
\beginwindow\lintel 2\lines \side 2.5in \window 6\lines
   Creating an exotic page layout like this is quite
   difficult, if not impossible, in some rigid source
   language oriented systems like ...
   ......
\endwindow
```

Figure 3.5: *Producing windowed text in TEX*. This piece of text shows how one could create a windowed paragraph with a *lintel* (text above the window) of 2 lines tall, a 2.5-inch wide block at each side, and a window of 6 lines tall (whose width is the width of the paragraph minus 2 times the width of the side block). Naturally, the remaining text forms the *sill*. The principal macros involved are `\beginwindow`, which takes these settings as parameters, and `\endwindow`, which outputs the formatted text. The actual code corresponding to these two macros is much more complex and is very difficult for an average TEX user to generate. The example here is based on the work of Alan Hoenig [72].

new image immediately. This immediate response is crucial to certain operations requiring visual feedback.

For instance, consider the task of laying out a page of windowed text. In a layout-driven direct-manipulation system one would do this simply by dragging the mouse, specifying the text blocks involved, and defining their links for the flow of text, as illustrated in Figure 3.2. The interface to the task domain is direct and straightforward.

Creating an exotic layout like this is quite difficult, if not impossible, in some rigid source language oriented systems like Scribe and SGML. In more flexible systems such as *troff* and TEX, however, such things are possible, but nontrivial. Defining macros that handle general windowed text requires a high degree of wizardry. In TEX, for instance, output routines can be redefined to layout pages differently, but the code required to produce general windowed text is quite involved. But once it has been figured out, generating a paragraph such as the present one is relatively straightforward. Based on the macros defined in Reference [72], producing the present paragraph is reduced to calling a pair of window opening and closing macros at the beginning and end of the paragraph (see Figure 3.5).



The standard direct-manipulation approach to this kind of irregular page layout and the way it is handled in TEX are somewhat different. In the direct-manipulation approach, the abstractions of text blocks and their links are general enough to cover a variety of situations. For example, creating a circular window would be based on the same interface used to create a rectangular one as shown in Figure 3.2. A layout whose text first fills up every line on the left side and then the right, instead of running across the window for each line would be done similarly. In TEX, producing a circular window requires modifying `\beginwindow` to accept a much more complicated parameter passing scheme. As for the other case, a new set of macros must be defined for that purpose specifically. The real issue, however, is a visual approximation to the ultimate page layout. It is clear that direct manipulation is more appealing in this respect.

Finally, it is important to consider the concept of WYSIWYG (*what-you-see-is-what-you-get*) systems, which has been one of the primary features of many electronic publishing systems. The term WYSIWYG should not be considered a synonym of direct manipulation, however. WYSIWYG refers to the correspondence between what is presented on the display and what can be generated in some other devices. In the context of electronic publishing, WYSIWYG means that there is a very close relationship in terms of document appearance between the screen representation and the final hard-copy. Direct manipulation is a more general concept that models user interfaces. A direct-manipulation document preparation system may not have a WYSIWYG relationship between its display representation and the final hard-copy. Galley-oriented document editors like Tioga, Andrew's text editor, and Diamond belong to this type. Conversely, a batch-oriented, source language based formatter may be coupled with a previewer that is WYSIWYG.

3.2.4 Pre- and Post-Processing

There are a number of tasks that must be performed either before or after formatting, which are collectively called pre- and post-processing facilities. A common nature of these facilities is that they often require a stand-alone processor to accomplish the intended task. For instance, a spelling checker, a bibliography processor, and an index processor are needed for checking spelling, resolving citations, and permuting index entries, respectively. The traditional approach is, of course, source language based and batch-oriented: an intermediate file is generated as a derivative of the main document; this file is then passed to a designated processor, and the result is incorporated back into the main document. This approach applies not only to standard noninteractive document compilers like *troff* and TEX, but to a number of direct manipulation environments as well. For example, index processing in FrameMaker, MicroSoft Word, and Ventura Publisher are all handled by a noninteractive off-line program.

Direct manipulation, from the processing point of view, requires too much overhead for a relatively minimal payoff. For instance, in compliance with the direct-manipulation paradigm, an index entry, whenever entered into the document body, must immediately appear in the index section (with its page number) in alphabetical order with respect to other entries already there. This capability requires extensive support in the internal data structure, but does not contribute toward any significant improvement in generating the final index. The same criticism applies to the processing of similar objects such as bibliography, glossary, table of contents, cross references, etc.

In spite of the need for stand-alone processors, which seem inevitably batch-oriented, there are other aspects of these pre- and post-processing facilities that require more interactive support. These include, among others, correcting misspelled words in the manuscript, browsing the bibliography database to make citations, and placing index commands into the document body in a systematic fashion. All of them require a close integration with the document editor. If these objects are not maintained in the internal representation, a good manuscript level pattern matching mechanism (e.q. regular expression search, query replace, query insert, etc.) is imperative.

Notice that in a direct-manipulation system, the tags for marking citations, cross references, and index entries cannot appear directly in the document under manipulation because their original forms may not correspond to any physical appearance. A common solution is to put them in "shadow pages" instead of the direct-manipulation representation. Thus, a shadow document is the original document plus these tags whose markers can be displayed upon request for editing purposes. As a result, users operating under this extended direct manipulation model are actually dealing with dual representations of the document, although the variation between the "source" and "target" is not as significant as that in a true source language oriented system.

3.2.5 Imaging, Filing, and Interchange

The on-line imaging mechanism of most direct-manipulation systems is based on immediate interpretation of their internal representation. Their off-line filing representation is some textual description of the internal structure, but not necessarily in any real programming language. This textual description can in turn be passed to a device-specific printer driver for a hard-copy. Similarly, most source language oriented systems generate their output in some generic representation; device drivers are needed for either screen previewing or hard-copies. Typical examples include TEX's DVI format and the *ditroff* format. A common feature of this type of device independent "virtual machine" is that its imaging model is extremely simple-minded; its basic construct resembles low-level assembly code more than a high-level programming language.

Recently, a new breed of programming languages known as *page description languages* (PDLs) has emerged as the preferred representation for imaging as well as filing. There are currently three major players in the arena: Interpress [20], PostScript, and DDL [77]. Advantages of PDLs include high-level program constructs, arbitrary transformations at the imaging level, uniform treatment of graphics and text (fonts), device independence, etc.

Many systems use PDLs as the off-line filing representation because PDLs are supported by an increasing number of printers. The on-line imaging mechanism in most direct-manipulation systems, nevertheless, is still based on lower-level descriptions. The result is a discrepancy between the on-line imaging and the potentially more versatile one given by the off-line PDL representation. This problem can be resolved by providing a PDL server on-line and representing the internal structure as the PDL. In reality, a PDL server (interpreter) can be realized as, in ascending degree of generality, a client level application for graphics specification (e.g., PostScript in VorTEX, see Section 6.1), the underlying imager of a window system (e.g., Interpress in Cedar), or even the window system server itself (e.g., PostScript in the NeWS window system).

Another aspect of off-line filing concerns saving a snapshot of the internal state so that future invocations can take place incrementally. This aspect is analogous to saving object files for a source program. It can also be viewed as a checkpointing mechanism that provides backups as well as a means to support undo operations. The issue here is the standard space/time tradeoff. The simplest approach is to save the entire "state" and reload it when a rollback or reinvocation is requested. The penality, of course, is tremendous storage overhead. A more "source-based" approach would take more time abstracting only the essential parts and saving them structurally in a textual format. Again, it takes time to recover the state when a rollback or reinvocation happens, but the filing representation would be much more compact.

An area of emerging importance in document development concerns interchange formats. The goal is to exchange documents electronically among geographically distributed sites that are heterogeneous in hardware and software. To avoid developing n^2 translators among n different types of systems, the idea is to devise a common intermediate format so that only $2n$ translators are needed. Three possible exchange formats have been proposed: office document architecture (ODA) [6,74], standard generalized markup language (SGML) [61], and Interscript [12]. A comparison of the three can be found in Reference [80].

3.2.6 Annotations/Narrations and Dynamic Reading

So far the focus has been on document composition, or the *writing* side of document preparation as a whole. The other half of the story, which has too often been ignored, concerns effective *reading* of a document. Reading is a rather "direct" process; when references are involved a reader relies on a somewhat indirect approach. For instance, when a bibliographical reference is of interest, the reader needs to go to the bibliography section and look up the cross reference information available there.

This static notion of documents still dominates our way of reading even in the era of electronic media. On one's favorite document preparation systems, users are still artificially creating bibliographies and indexes for their papers, manuals, books, etc. Part of the reason is being able to generate hard-copies consistent with the tradition. But if hard-copy compatibility as an issue is relaxed, then one should think seriously about what exactly are the purposes of references like the bibliography and index. Their foremost function is to allow the reader to access relevant information efficiently. Creating separate bibliography and index sections is the best one can do with the static print medium.

In an integrated electronic document environment, much of this linking information can be stored internally. Therefore, instead of requiring the reader to actually "read" the section that contains the references (indirect accessing), it is possible to access references in a direct and context-sensitive way. For instance, when a citation is of interest, a menu of options would allow the reader to (1) inspect the content of the bibliography entry in a separate window so that the reading of the main document is not hindered, or (2) visit the actual document referenced by the citation (this can occur recursively). If an object of a different nature is selected, its context gets reflected in the menu immediately.

Operations like these can go beyond "what you see is what you get". With the "shadow document" approach mentioned earlier, for example, annotations can be associated with key concepts and embedded in the document invisibly. Thus the information a document is able to convey is much more than meets the eye. A system that supports this type of non-linear reading is referred to as a *hypertext* system [44]. The key here is the ability to create links among objects within the same document and, in many cases, across different documents. More elaborate hypertext operations are possible [141]. Some candidates include local features like *filtering* (restricted reading) and *fisheye viewing* (focused reading) [54], or more global issues like document navigation and dissemination, and so on.

Another important property of an electronic document is that its presentation need not be confined to a single, static medium. It can comprise dynamic pictures (animation) with voice narrations, for example. Such hypermedia documents require extensive internal support, and its user interface must be based on a clever blending of the two models. Apple's HyperCard [63] is an excellent example that blends the two approaches successfully.

Chapter 4

Design Methodology

We have raised a number of requirements in Chapter 3; some of the issues are orthogonal, while others are somewhat contradictory. The most essential questions concern the relationships among the two models, the task domain, the various representations, their transformations, and the notion of procedurality. This section tries to answer these questions and establishes a general framework for analyzing and designing multiple-representation systems. The framework differs from some formal models like Sandewall's *Theory of IMS* [118] or Goguen and Meseguer's order-sorted algebra approach [59] in that this framework (1) is less complex and therefore much easier to follow, and (2) addresses more properly the multiple-representation aspect of document preparation, with possible extensions to similar software environments.

The basic structure of multiple representation document development systems, or the *representation domain*, is illustrated in Figure 4.1. As shown in the figure, it includes four generic representations:

1. S: a source representation supporting high-level programming constructs such as abstraction mechanisms (e.g., macros, procedures, or variables), control structures (e.g., conditionals, iterations, or recursions), etc. A document in TEX or *troff* is a representation of this type.

2. O: a structural view of the basic objects involved in the system. This representation may be one with built-in declarative logical components such as a document in SGML, or one with object-based input/output such as a drawing under MacDraw, or one with a hierarchical structure like the internal representation of VORTEX.

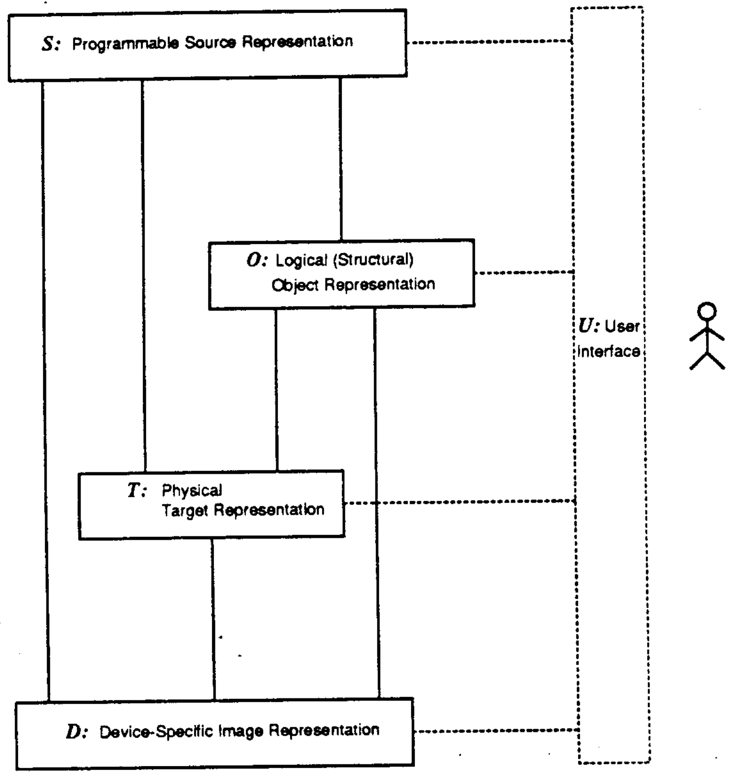

S: Programmable Source Representation
O: Logical (Structural) Object Representation
T: Physical Target Representation
D: Device-Specific Image Representation
U: User Interface

Figure 4.1: *Fundamental structure of multiple representation systems.* The legend is the following: the four *solid boxes* are the various representations in question, *solid lines* connecting these boxes each refers to a transformation between the two representations, the *dashed box* on the right stands for the user interface abstraction, and finally, the *dashed lines* each indicates a link between the various representations and the user interface. The figure shows that all four are connected among themselves; they are also connected with the user interface. The real situation, however, is that the connections are directional and for certain systems some of the nodes and edges in the graph may be absent.

3. T: a representation corresponding to the objects' physical structure after processing. This representation is usually device-independent.

4. D: the actual device-dependent image representation.

This basic structure may be augmented to include derivatives of the four generic representations as required by a particular task. It must be pointed out that S is not the exclusive representation of the source-language model mentioned earlier. For instance, SGML, a source-language oriented system, is classified as having a primary representation of O rather than S. The distinction here stems from the availability of program constructs.

The basic structure also has an abstraction for various types of user interfaces (U);

possible distinctions are keyboard/command-based versus mouse/menu-driven versus their combination, textual versus graphical, etc. Figure 4.1 shows U as a single entity, but it may be refined to reflect these distinctions or be specified according to more sophisticated guidelines.

There are several important aspects of this structure that underscore and unify all the issues in question:

- Whether or not a representation (solid box) exists.
- Whether or not an existing representation is made explicit to the user (i.e. if there is a dashed line connecting the solid box and the dashed box); if so, whether or not the relation is bidirectional.
- Whether or not a transformation (solid line) exists between two existing representations; if so, whether or not such a transformation is bidirectional.

More precisely, an instance of the fundamental structure (call it Ω), or the *representation instantiation*, is described by a quadtuple

$$\Omega = (\Pi, \Theta, \Gamma, \Delta)$$

where

$\Pi = \Pi_S \cup \Pi_O \cup \Pi_T \cup \Pi_D$, set of one or more representations,
$\Theta = \{ U_1, U_2, \cdots \}$, set of user interface abstractions,
$\Gamma = \{ \pi_1 \rightarrow \pi_2 \mid \pi_1, \pi_2 \in \Pi \}$, set of interrepresentational transformations,
$\Delta = \{ \pi \rightarrow \theta \text{ or } \theta \rightarrow \pi \mid \pi \in \Pi, \theta \in \Theta \}$, set of user interface relations.

and

$\Pi_S = \{ S_1, S_2, \cdots \}$, one or more programmable source representations,
$\Pi_O = \{ O_1, O_2, \cdots \}$, one or more structural object representations,
$\Pi_T = \{ T_1, T_2, \cdots \}$, one or more physical target representations,
$\Pi_D = \{ D_1, D_2, \cdots \}$, one or more device image representations,

Intuitively, $\pi_1 \rightarrow \pi_2$ ($\pi_1, \pi_2 \in \Pi$) means representation π_1 can be directly transformed to representation π_2. This transformation can be illustrated graphically by an arrowhead at the end of the solid line connecting π_1 and π_2. Similarly, $\pi \rightarrow \theta$ means representation $\pi \in \Pi$ can be explicitly viewed by the user with interface $\theta \in \Theta$, and $\theta \rightarrow \pi$

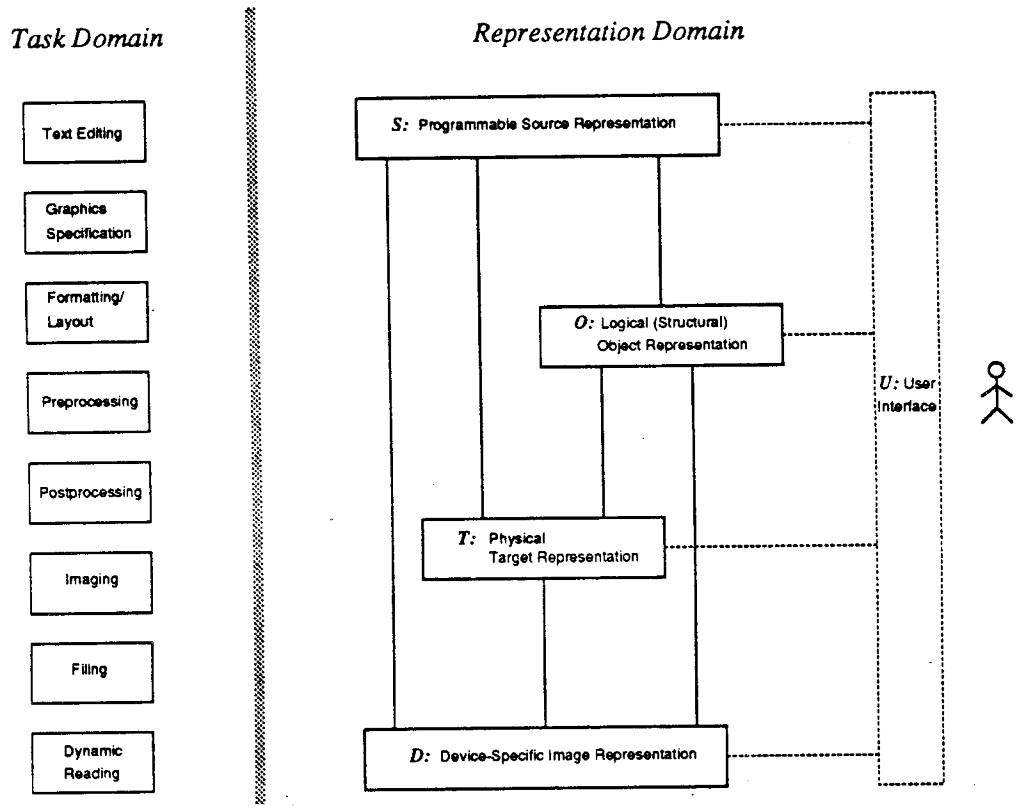


Figure 4.2: *Task and representation domains.* The gray vertical bar represents the boundary between the task domain on the left and the representation domain on the right. A multiple representation system is a mapping from the task domain to the representation domain.

means representation $\pi \in \Pi$ can be accessed or manipulated by the user through interface $\theta \in \Theta$. For convenience, $\pi_1 \rightarrow \pi_2$, $\pi_2 \rightarrow \pi_1$ can be abbreviated as $\pi_1 \leftrightarrow \pi_2$, and $\pi_1 \rightarrow \pi_2$, $\pi_2 \rightarrow \pi_3$ as $\pi_1 \rightarrow \pi_2 \rightarrow \pi_3$.[1]

The design of a multiple representation system, therefore, is to derive a representation instantiation (Ω) *for each member of the task domain* as shown in Figure 4.2. Defining an Ω requires identifying (1) the representations to be maintained (Π), (2) the specification of a user interface abstraction (Θ), (3) the set of inter-representational transformations among members of Π, and finally (4) the set of user interface relations from Π to Θ and vice versa.

Given this framework, it becomes natural to analyze systems belonging to either source-language or direct-manipulation camp, or to discriminate procedural systems from

[1]These are solely for the convenience of notational abbreviations. No transitivity is implied in $\pi_1 \rightarrow \pi_2 \rightarrow \pi_3$ (i.e. it does not imply $\pi_1 \rightarrow \pi_3$).

declarative ones. For instance, a source-language based batch system implies the existence of a representation S or O and some unidirectional relations from this representation to T or D.[2] A direct-manipulation system, on the other hand, will be based on an instance of Ω having $\pi \leftrightarrow \theta$ in Γ ($\pi \in \Pi$ and $\theta \in \Theta$), with the added criterion that feedback from the system be immediate in order to create the sensation of directness. Furthermore, the property of procedurality usually means that either $S \rightarrow T$, $S \rightarrow D$, or $T \rightarrow D$ is in Γ, and that either $U \rightarrow S$ or $U \rightarrow T$ is in Δ (i.e. either S or T is user manipulable). Finally in a declarative system, Π_S will normally be empty and Π_O will be the only set of manipulable representations.

Based upon these observations, it is interesting to compare direct-manipulation graphics editors, such as MacPaint, MacDraw, and Illustrator. MacPaint can be described by

$$O \rightarrow D,\ U \rightarrow O,\ D \leftrightarrow U$$

because the user creates drawings with some object-level menus ($U \rightarrow O$). But once specified, objects are transformed into a device-dependent image ($O \rightarrow D$), which can be viewed and manipulated by the user ($D \leftrightarrow U$). By contrast, MacDraw corresponds to

$$O \leftrightarrow T,\ T \leftrightarrow D,\ O \leftrightarrow U.$$

There are two major differences here: (1) in MacDraw the user views and manipulates drawings at the object level, and (2) drawings in MacDraw are device-independent due to the presence of a target representation. Finally, Illustrator is close to

$$O \leftrightarrow T,\ T \leftrightarrow D,\ D \rightarrow O,\ O \leftrightarrow U,\ U \rightarrow D.$$

The crucial difference between Illustrator and MacDraw is $U \rightarrow D \rightarrow O$, which underscores Illustrator's capability for the user to unravel geometric objects from bitmaps by tracing the image.

More importantly, this framework facilitates the analysis of existing multiple representation systems and the design of new ones. The basic criterion here is that at least two members of Π must be manipulable. Emacs, for example, can be viewed as a system

[2]The notational conventions are obvious here; it is assumed that $S \in \Pi_S$, $O \in \Pi_O$, $T \in \Pi_T$, $D \in \Pi_D$, $U \in \Theta$, and similarly with subscripted ones used later.

having Ω_{Emacs} for text editing, with the majority of remaining members in the task domain mapped to the empty set, where Ω_{Emacs} is defined as follows:

$$\begin{aligned}
\Pi_{Emacs} &= \Pi_S \cup \Pi_O \cup \Pi_T \cup \Pi_D = \{S\} \cup \{O\} \cup \{T_1, T_2\} \cup \{D\}, \\
\Theta_{Emacs} &= \{\, U \,\}, \\
\Gamma_{Emacs} &= \{\, S \rightarrow S,\, O \rightarrow S,\, S \leftrightarrow T_1,\, T_1 \leftrightarrow T_2,\, T_2 \leftrightarrow D,\, \}, \\
\Delta_{Emacs} &= \{\, U \leftrightarrow S,\, U \rightarrow O,\, U \rightarrow T_1,\, U \leftrightarrow T_2,\, \}.
\end{aligned}$$

To interpret this specification, one can think of S as the Lisp code under which Emacs operates, O as the collection of user-level objects such as characters, words, lines, regions, etc., T_1 as the one-dimensional text stream (where linefeed is just an ordinary ASCII character), and T_2 as the corresponding two-dimensional text array (where linefeed causes a line break). The combination of T_1 and T_2 forms the overall target representation T.

Thus Ω_{Emacs} says that the user can view both source and two-dimensional target representations ($S \rightarrow U$ and $T_2 \rightarrow U$) and manipulate either the one-dimensional text stream ($U \rightarrow T_1$), the two-dimensional text array ($U \rightarrow T_2$), the objects ($U \rightarrow O$), or the program ($U \rightarrow S$). Operations on objects get transformed into code ($O \rightarrow S$) which is then evaluated and represented in the one-dimensional text stream ($S \rightarrow T_1$). Both O are T_1 are implicit since they are not explicitly exposed to the user (i.e. no $O \rightarrow U$ or $T_1 \rightarrow U$).

T is split into T_1 and T_2 because the user operates on the one-dimensional text stream as well as the two-dimensional text array, although the actual screen appearance is two-dimensional. Normally, `next-line`, `previous-line`, and a host of common operations are two-dimensional. But things like `forward-char` and `backward-char` are one-dimensional; at the boundary of current line they move to the adjacent boundary of the next or previous line linearly. Furthermore, the actual internal representation is one-dimensional because a character is addressed by an offset relative to the beginning of buffer instead of by a two-dimensional coordinate.

One interesting point is that a recursion can be observed in Ω_{Emacs} — although its is not explicitly shown — the editing of S is essentially an instance of Ω_{Emacs}. In other words, S can be expanded to a secondary Π_{Emacs}, $S \rightarrow S$ is equivalent to the composite of the rest of Γ_{Emacs}, and $S \leftrightarrow U$ is, in effect, the composite of the rest of Δ_{Emacs}.

An Ω may be specified graphically. Figure 4.3 shows Ω_{Emacs}'s corresponding graphical specification. Figure 4.4 is an instance of a design reflecting some major features

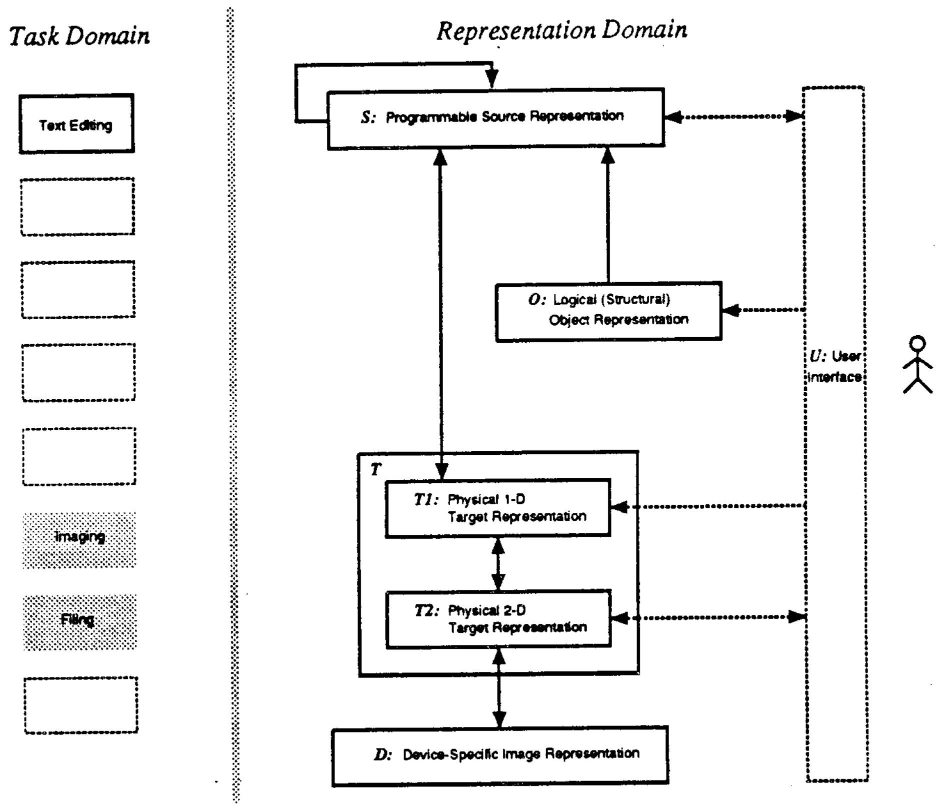

Task Domain
Representation Domain
Text Editing
Imaging
Filing
S: Programmable Source Representation
O: Logical (Structural) Object Representation
U: User Interface
T
T1: Physical 1-D Target Representation
T2: Physical 2-D Target Representation
D: Device-Specific Image Representation

Figure 4.3: *Graphical representation instantiation of Emacs.* The task of text editing has been singled out by the darkened box on the left. The shaded boxes represent other tasks involved in the overall system that are not the focus of current instantiation. Empty boxes are those tasks that play no roles in the system.

of Tweedle mentioned earlier in Section 3.2.2. Because Tweedle supports both a textual form of procedural language as well as an object level graphical representation, the task domain includes primarily text editing and graphics specification. The text editing side of the story is identical to that of Emacs discussed above. Figure 4.4 illustrates an Ω corresponding to its graphics specification task only.

The representations and transformations involved are self-explanatory except that there is no transformation from O to T because no object level evaluation is available in Tweedle — graphical objects always get transformed into code, which is then evaluated. One can normally expect low level primitives in terms of registering cursor positions or mouse clicks be provided by the underlying window manager. Note that the editing of S is another instance of Ω_{Emacs}. This example shows, as an integration mechanism, how one Ω may be plugged in as a component of another Ω.

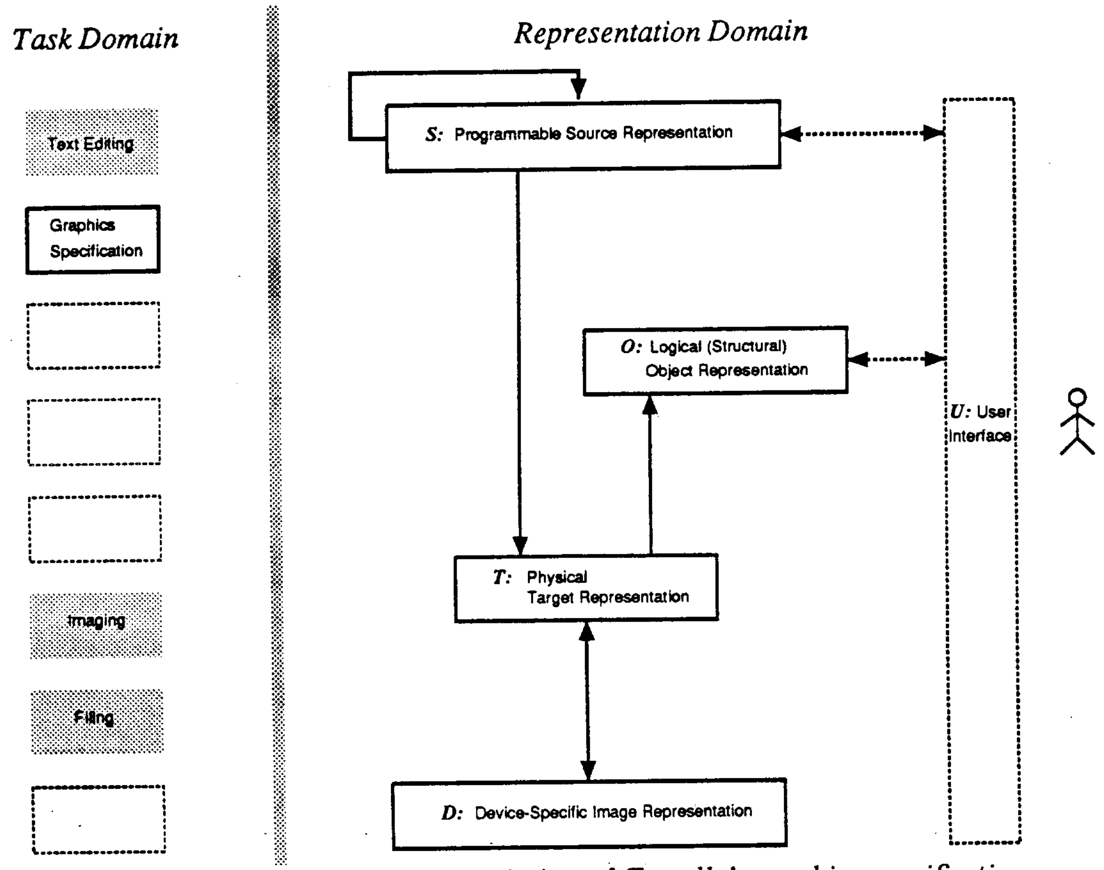

Task Domain
Text Editing
Graphics Specification
Imaging
Filing
Representation Domain
S: Programmable Source Representation
O: Logical (Structural) Object Representation
T: Physical Target Representation
D: Device-Specific Image Representation
U: User Interface

Figure 4.4: *Representation instantiation of Tweedle's graphics specification.*

This framework is by no means complete or precise, but it does establishes a good approximation of what is to be accomplished by a multiple representation system. We envision a top-down methodology based on this framework can be of use to designers. The design process starts with identifying the task domain. For each element in the task domain, the representation domain is instantiated with the specification of an Ω. Within each Ω, finer issues are then sorted out, and that may go down as deep as stepwise refinement requires.

Chapter 5

Functional Specification

This chapter discusses the principal properties of VORTEX as a case study of the methodology introduced previously. An implementation of a prototype system is given in Chapter 6. Some key ideas in VORTEX are compared to systems like the `tnt` editor/formatter [55] and Quill [32], which focus on the same set of issues as VORTEX does. This chapter concentrates on properties insofar as specifying Ω is concerned; issues of finer granularity are postponed until later.

Based on a top-down methodology described in Chapter 4, it is appropriate to start identifying the task domain as containing everything listed previously plus a few derivatives (to be described later). We then have to define an Ω for each member of the task domain. Since graphics in TEX is virtually undefined, POSTSCRIPT has been chosen as the graphics specification language for its rich graphics capability and powerful imaging model. Both of these tasks are maintained in multiple representations.

5.1 Text Editing

Text editing in VORTEX is Emacs-based. Despite some differences in the fine points, Ω_{Emacs} given in the last section would suffice in describing VORTEX's multiple representational view of text editing. As in Emacs, language-specific modes are available for editing code in TEX or POSTSCRIPT. The underlying Lisp subsystem is not confined to the the task of editing; it also serves as the basis for system integration and a host of computational jobs, as it will become clear later.

5.2 Graphics Specification

As in Tweedle [11], a program representation as well as a graphical view of the objects are implicitly maintained by the system and explicitly manipulated by the user. Therefore, the Ω defined in Figure 4.4 also describes VORTEX's graphics subsystem. In detail, however, the actual representations are distinct due to the differences between PostScript and Tweedle's underlying procedural language.

A PostScript interpreter has been implemented by the VORTEX group[1] as a standalone PostScript language previewer. The same program can also be used as a server that interacts with VORTEX's main document display module. Under the current design, when a picture is encountered in the target representation, the corresponding code is transmitted to the PostScript server. It, in turn, hands back the graphics as a raster image, which is then incorporated into the document's device representation. As described in Chapter 6, although the current VORTEX cannot render PostScript graphics, support for interacting with the PostScript server is already in place.

5.3 Formatting and Layout

For the task of specifying a document's textual content in general and its formatting and layout information in particular, VORTEX provides a source level program (in TEX) as well as a target level view to the user. Operations performed on one representation are propagated to the other automatically. The idea is to take advantage of the "expressiveness" of a source programming language and also the immediate visual response given by a direct manipulation user interface to the target representation.

Figure 5.1 shows the representation instantiation (Ω) of VORTEX's formatting and layout. It says that the document's source representation (S) is transformed into an internal object structure (O), which then becomes the physical layout (T) of the document after formatting. This target representation can be interpreted by a displayer on-line, or translated off-line into a file format such as DVI or a program in certain printer language like PostScript. Both S and T may be manipulated by the user. The bidirectional transformation between S and U is an extended version of Ω_{Emacs}. Changes to S are reflected

[1]Primarily due to John Coker and Steve Procter.

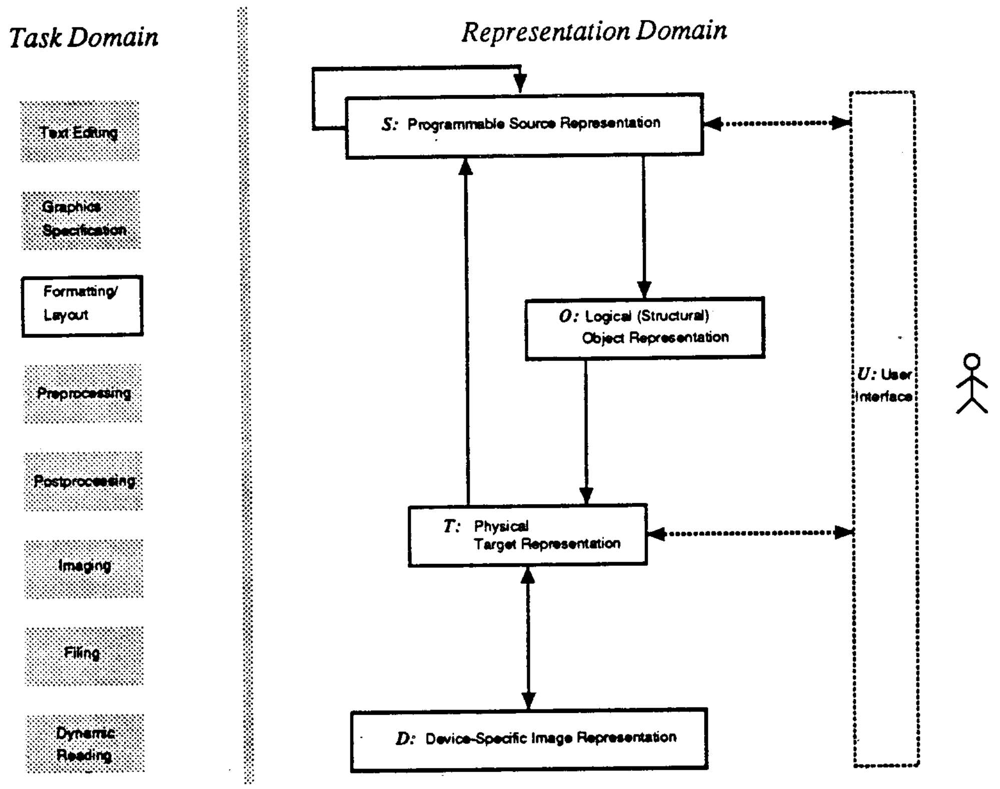


Figure 5.1: *Representation instantiation of VorTeX's formatting and layout*

in itself directly and are propagated to T through O. Changes to T, however, are first propagated to S and finally go through the $S \rightarrow O \rightarrow T$ cycle to be reflected back to itself.

Propagating changes from source to target is straightforward in concept because that is exactly what TeX does. The subtlety here is that instead of a batch-oriented implementation, VorTeX needs to be incremental, which generates a number of interesting issues not encountered in the batch version. The fact that TeX is macro-based complicates this problem even more.

In VorTeX, a close relationship is maintained between the source representation (S) and the two internal representations (O and T). Incremental formatting is based on marking and sweeping dirty nodes in S and O and on comparing newly generated code with what is stored in T.

```
(defun create-windowed-par (begin end lh sw wh)
   (goto-char (target-to-source begin))
   (insert "\\beginwindow"
           "\\lintel " lh "\\lines "
           "\\side " sw "in"
           "\\window " wh "\\lines\n")
   (goto-char (target-to-char end))
   (insert "\\endwindow\n"))
```

Figure 5.2: *Reverse mapping of page layout.* Above is the Lisp function to be triggered when a paragraph like Figure 3.2 is laid out in the target editor. This function operates in the source representation. The first two arguments, given in target positions, must be translated into source positions via internal representation accessing before used. The next three arguments represent, respectively, lintel height, side block width, and window height, as required by the windowed text environment shown in Figure 3.5.

5.4 Reverse Mapping

The next major issue concerns identifying the set of direct manipulation operations that must be encapsulated in T and realizing the reverse mapping mechanism that propagates side effects back to S. We believe *page layout*, *object placement*, *attribute update*, and similar operations would benefit most from prompt visual feedback and are therefore reasonable candidates to be incorporated in the direct manipulation interface to T. For instance, a page layout specified at the target level in direct manipulation like Figure 3.2 would correspond to the TEX source code of Figure 3.5 by the reverse mapping facility.

The question is how to carry out reverse mappings systematically. In VORTEX the reverse mapping mechanism is realized by associating each target level operation having any side effects with a Lisp function at the source editor. Whenever such an operation is executed in the target editor, the corresponding Lisp function gets invoked and evaluated by the source editor. The user interface is quite flexible in that it can be either command-driven (with the standard Emacs keyboard binding scheme), menu-driven (with mouse as the primary input mechanism), or a combination. It is also extensible; new instances of reverse mapping can be added to the system by the user, which will be consistent with the overall interface structure.

All of these are made possible with the support of a Lisp programming subsystem within the environment. Reverse mapping is programmed on top of the system's editing primitives for source level pattern matching and some extended functionality for internal representation accessing. Thus, to lay out something as exotic as Figure 3.2 in the middle of a page, the corresponding Lisp function may look like what is shown in Figure 5.2. In the code, `begin` and `end` represent the beginning and end, in target positions, of the paragraph to which a window is to be opened. The function `goto-char` positions the cursor to the point given as argument in the source editor, where the positions are translated by `target-to-source` from target to source via internal data structure accessing. Finally, inserting the text for opening and closing the windowed paragraph is straightforward.

The reverse mapping of page layout is relatively trivial compared to things related to *macro unraveling*. A macro and its arguments in the source representation (S) may not have a one-to-one correspondence with the expanded text that ultimately appears in the target representation (T). Typically there are three cases in a macro expansion:

1. text as arguments of a macro in S gets *copied* over to T,
2. text in S is consumed by the expansion and therefore is *deleted* in T,
3. new text originally not in S is *inserted* in T by the expansion.

When the expanded text is selected in T, what are the semantics of target-level operations using the selected text as an operand?

As a premise, the selection mechanism must be able to tell if the text is part of an expanded macro. Since internal representation accessing primitives are available to the Lisp subsystem and since the object and target representations (O and T) are tightly coupled, one can easily identify if any selected text is in the proper scope of a macro. To handle the semantics, case 2 can be eliminated to begin with, because deleted text cannot be selected in T. Depending on the user's intention, there are three possibilities:

1. The interest is in plain text only regardless of how the macro is expanded. Thus, including the text introduced by a macro, all "characters" seen by the user can be used as the operand, but no other attributes (e.g., typeface, size, etc.) will be associated with it. Operations of this type must be non-destructive with respect to the selected text itself. A plausible operation belonging to this group is *copy*.
2. If the selected text is copied over from S, destructive operations such as *insert*, *delete*, *move*, and so on are legitimate. Side effects are first reflected in S (the cursor will be

"warped" to the source editor window) and eventually get reflected in T through the $S \rightarrow O \rightarrow T$ cycle.

3. If the text is inserted, destructive operations will be disabled with some warning messages. One step beyond this approach is a query asking the user if the intention is to modify the definition of the macro in question. If so, the macro-unraveling Lisp code can scroll to the most recent spot in context where the macro is defined and let the user do the modification at the source level. A more elaborate approach incorporates certain rules that correlate encapsulated operators and operands in T with the underlying TEX code to be inserted to the macro definition in S. Multiple levels of macros may be involved in a macro expansion. An effective selection mechanism must be able to distinguish macros of different levels. For example, a first-order selection (e.g., single mouse click) always highlights and selects the inner-most macro enclosing the pointed text. A second-order selection (e.g., two consecutive mouse clicks) highlights and selects the enclosing macro of a level higher, and so on.

Reverse mapping on the basis of per-target-level operation is somewhat special to VORTEX. By contrast, in Quill, the underlying source language is the fully declarative SGML. There are two levels of internal representations (O and T) maintained in Quill as in VORTEX. Unlike VORTEX, however, Quill's external source representation is hidden during editing (i.e. no connections between S and U). The only role SGML plays is off-line filing and document interchange. In other words, reverse mapping becomes unnecessary in Quill. Its logical object representation (O) is a mirror of an SGML document; each node in O corresponds to an SGML markup tag. Thus, when the document is to be filed, all that is needed is to traverse O and the corresponding file in SGML can be generated.

As was argued in Section 2.2, the tradeoff boils down to complexity versus flexibility. Compared to Quill, VORTEX's overall architecture is more complex due to TEX's low degree of declarativeness and its macro-based abstraction mechanism. On the other hand, VORTEX is more flexible; to create a direct manipulation type page layout like Figure 3.2 and be able to map it back to the source is simply beyond Quill's model. Imposing logical document structure is also possible in VORTEX. Although O does not carry any logical meaning in VORTEX, document structure and style like those defined in LATEX can be realized by the reverse mapping facility, which operates at the source level. Since the user interface is customizable, one can effectively hide the procedural aspects of TEX in VORTEX.

5.5 Pre- and Post-Processing

The pre- and post-processing facilities largely follow three steps of (1) placing task-specific markup tags (commands) in the document body, (2) processing an auxiliary file containing information related to these tags, and (3) incorporating the results back to the main document. In many cases, these tags do not appear in the target representation; instead, they create links between different objects. These links frequently destroy the strict top-down hierarchy of the document's internal logical structure (O).

In VORTEX, all three steps are again built on top of the Lisp programming subsystem. Since a source representation is explicitly maintained, there is no need to hide these tags in the "shadow". The advantage of operating at the source level is that the internal representation does not have to increase its structural complexity. Tags such as citations retrieved from a bibliography database are directly inserted into the document source. The programming layer also has control over external processors. Thus, when the off-line processing is finished, the result can be interactively incorporated back in the source representation by the top-level of a Lisp program that initiated the processing.

5.6 Imaging and Filing

VORTEX's on-line imaging mechanism is based on direct interpretation of the target representation. Both its source in TEX and a translation of T (e.g., in DVI or POSTSCRIPT) can be filed as the off-line representation. It is also possible to base the on-line displayer completely on a PDL like POSTSCRIPT because such a server is already available for rendering graphics. The Ω for on-line imaging has been covered above; the one for off-line filing and imaging is a straightforward batch approach.

5.7 Dynamic Reading

Given a complete PDL as the graphics image server (POSTSCRIPT in this case), VORTEX is able to present pictures dynamically. This dynamic behavior may happen in one of two modes: *playback* and *synthetic*. In playback mode, the document displayer constantly gets notifications from the POSTSCRIPT server with new raster images of the same picture. In processing each notification, the old picture is erased and replaced by a new image. If this redisplay happens frequently enough, it becomes, in effect, animation. In synthetic mode, the displayer simply executes a PostScript program that takes care of itself in terms of any dynamics involved. The premise, however, is that the document displayer be the PostScript server itself. From the multiple representation's viewpoint, the playback mode is closer to direct manipulation because scenes as raster images are the basic manipulable objects, while the synthetic mode is more like source language based due to its programming aspects.

Furthermore, given the Lisp programming subsystem, the ability to access internal document structure through some lower level primitives, and an extensible user interface, VorTeX is capable of providing the user with some dynamic viewing functionality. The Lisp subsystem is also tightly coupled with the pre- and post-processing facilities mentioned earlier. For instance, one can select a reference and have the content of the reference displayed in a separate window. This type of context-sensitive browsing applies to objects like citations, cross references, indexes, and the like. What is special here is that no hard links are built into the internal representations for browsing purposes. Each operation is realized as a user-level function that performs primarily pattern matching in the source manuscript with the aid of internal representation accessing primitives.

5.8 Integration

The complete VorTeX system is integrated by means of sharing certain representations $(\Pi \cup \Theta)$ or transformations $(\Gamma \cup \Delta)$. For instance, Ω_{Emacs} is essentially $S \rightarrow S \leftrightarrow U$ in the Ω of both graphics specification and formatting/layout; the Ω corresponding to dynamic reading just mentioned constitutes part of $T \rightarrow U$ in formatting/layout. Also, the internal representations of formatting/layout are shared by tasks such as reverse mapping, pre- and post-processing, and so on.

In particular, text and graphics integration in VorTeX employs a "cut-and-paste" model. The manipulation of text and graphics each operates under a distinct context. The integration is based on the PostScript imaging server. From the document formatter and displayer's point of view, graphics is just a piece of raster image. Therefore, text within graphics will not be formatted the way regular text is; it all depends on how the graphics imager treats text and fonts.

Quill represents a fundamentally different model in which arbitrary nesting of text and graphics is permitted and their processing is uniform. The uniformity is achieved by sharing a common object representation (O) between graphics specification and formatting. Like text, graphics nodes in O will eventually be mapped to their SGML counterparts [31]. These nodes can be arbitrarily nested, a context-sensitive menu will be displayed when a node of a particular type is selected. The integration mechanism here is based on representation sharing rather than a series of transformations as is in VORTEX.

VORTEX's Lisp programming subsystem provides the essential glue for integrating the bulk of tasks together in a coherent manner. These tasks include reverse mapping, the many phases of pre- and post-processing, dynamic reading, and so forth. Most importantly, it also serves as the backbone behind job control and user interface customization. For a complex environment like VORTEX, a user-level programmable source representation like the Lisp substrate reduces the complexity of the system's internal representations as well as its overall integration mechanism, which may otherwise be ubiquitous and difficult to manage.

Chapter 6

A VORTEX Prototype

A document in VORTEX is represented and presented in both source and target forms; mappings between the representations must be carried out efficiently and systematically. The task domain covers a vast diversity of jobs; an integrated environment must be provided so that their corresponding vertical tasks can be handled coherently. It must also support compound objects horizontally, so that documents are not restricted to the textual form.

This chapter discusses a prototype implementation of VORTEX. In this version, the system supports documents in multiple representations. An incremental formatter is in place that transforms a document from the source representation to the target representation. A reverse mapping facility is responsible for the transformation in the opposite direction. Vertical tasks are integrated by the system's gluing mechanism, yielding an environment that allows the user to access dictionaries, databases, and any external processors conveniently for non-formatting jobs. Finally, hooks are built into the system to incorporate graphics and other non-textual objects. Although the prototype does not support the full functionality specified in the previous chapter, it does demonstrate that the design is viable and the system is extensible. The remainder of this chapter explains how the prototype has been realized.

The prototype system is implemented in C and Lisp. Its kernel, which includes a source editor, a target editor, and an incremental formatter, are all written in C. Higher-level facilities for such tasks as reverse mapping and pre- and postprocessing are programmed in Lisp, which is supported by the source editor. C was chosen partly because it interfaces well with VORTEX's underlying distributed UNIX environment and partly because the VORTEX

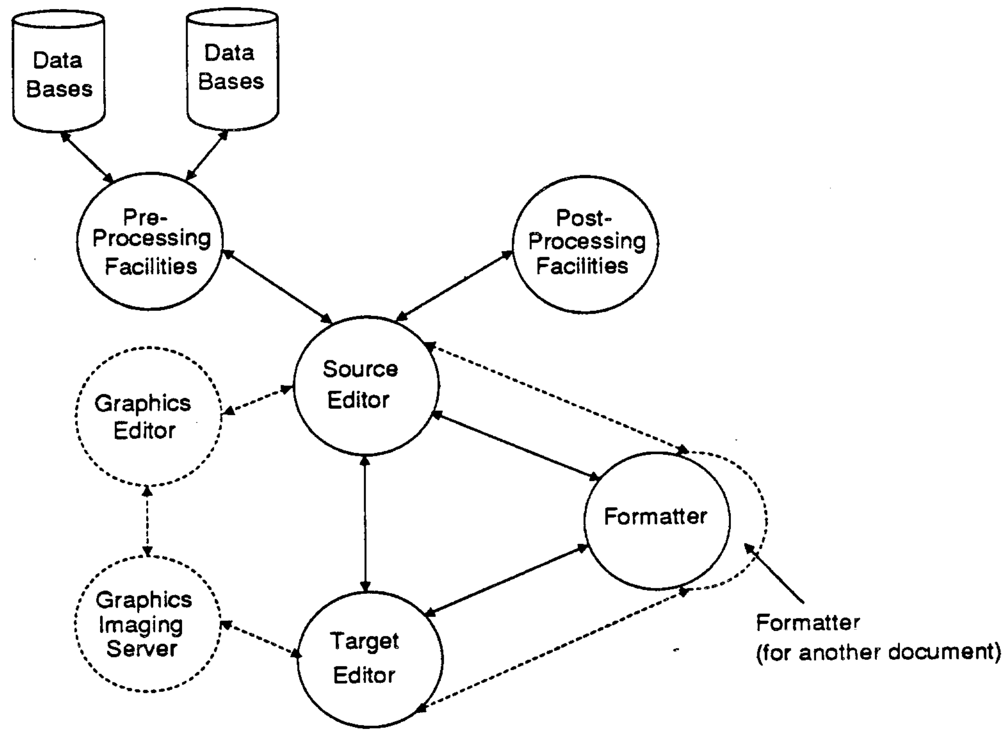


Figure 6.1: *VORTEX system structure.* The circles represent major system components, while arrows denote their inter-relationships. Dotted circles and arrows are not implemented in the current prototype, but hooks are available for them to be plugged in when necessary.

incremental formatter was adapted from a C version of TEX (i.e., Common TEX). Lisp was chosen as the higher-level gluing language because it is interpreted. It interacts well with the source editor, and is extensible. It would be awkward to deal with the highly interactive activities associated with pre- and post-processing tasks in C. Conversely, using Lisp alone would mean rewriting TEX from scratch, which we deliberately tried to avoid.

6.1 System Structure

The principal components of VORTEX are the incremental formatter and the pair of source and target editors with which the user interacts. Additional external programs used in pre- and post-processing are integrated with the overall system through the Lisp-based gluing mechanism. To the formatter, any special object (e.g., a figure) is treated as a box whose dimension is used to reserve the necessary space. It is the target editor's responsibility to render these special objects. Special-purpose editors can be brought in to manipulate these objects.

Figure 6.1 illustrates the basic system structure of VorTeX. The principal trio, *formatter*, *source editor*, and *target editor*, are tightly coupled during the processing session. Preprocessing tasks which may access external dictionaries and databases are built on top of the source editor. Postprocessing tasks are integrated by the same mechanism. Each instance of the incremental formatter maintains a single document whose component files originate from the source editor. If another document is to be processed, another instance of the formatter is connected with the two base editors. The formatter produces the target representation of the document it maintains and sends it to the target editor for display. For the purpose of synchronization, the two base editors are interconnected as well. They communicate with each other so that changes can be mapped back and forth between source and target representations.

Presumably a graphics editor can be connected with the source editor to exchange graphical objects in a source language like PostScript. The graphics editor is supported by an imaging server, which evaluates each graphical object (as a PostScript program, if PostScript is used as the source representation) and returns the corresponding screen image to the client editor. The target editor may also take advantage of this imaging server. Whenever a graphical object is identified in the target representation, its corresponding procedural description (e.g., the PostScript code) is transmitted to the server, and in return a screen image can be rendered in the user's viewport. Other special objects can be handled in a similar fashion.

A typical VorTeX session starts by invoking the source editor, in which component TeX files of a document can be visited and edited. The source editor is the driver of the entire system that spawns the formatter, the target editor, and any special editors. Multiple documents are simultaneously maintained in the source editor. Whenever a new document needs formatting, an instance of the incremental formatter is initiated which remains connected with the two base editors for incremental reformatting until it is explicitly exited. A formatter can conceivably handle more than one document at a time, but supporting a single document makes the structure logically cleaner.

6.2 The Principal Trio

One possible way of realizing the system is to implement it as a monolithic program in which the source editor serves as system top level, while other components are invoked as subroutines under a single thread of control. Since all components share the same address space, only one copy of the internal representation is needed to support multiple representations and their mappings. Systems like Diamond [47] and Quill [32] are good examples of the monolithic approach.

This approach has a number of practical limitations. First, the internal representation is a huge data structure. When it is coupled with the buffers and state information claimed by the two base editors, the required run-time heap memory may be quite significant and thereby limit the system capacity. Second, single-threaded control reduces the chance of exploiting fast computation resources available in a networked environment. Although it is possible for a monolithic system to prioritize events so that some computation-intensive tasks like formatting can take place in the background, it is not possible to take advantage of the computation power of some remote machines under single-threaded control.

An alternative approach distributes the system into multiple processes across disjoint address spaces. Under this model, the principal trio, as well as various special-purpose editors, all operate under their own address space,[1] so the constraint of dynamic memory usage is less of an issue. Parallelism can be exploited under distributed circumstances where multi-thread control is supported. While the user is browsing the document, background formatting can continue without intervention or delay. Another unique advantage that the distributed approach has over the monolithic one is that it is possible to off-load the back-end incremental formatter to a fast remote machine for increased performance. A network-based window system like Andrew [100], X [119], or NeWS [127] would even allow the two front-end base editors to be off-loaded to remote machines for similar reasons. As a practical consideration, delegating the system into three separate programs that are integrated via simple communication protocols makes it possible for our implementation team to work independently. During the course of the prototype implementation, each group was able to proceed without depending on the other groups until the final phase of integration.

[1]Systems with multiple processes running under disjoint address spaces are typified by certain versions of UNIX, such as 4.3 BSD, where shared memory is not supported. Under some single language environments, such as Cedar [128], multiple processes share the same address space.

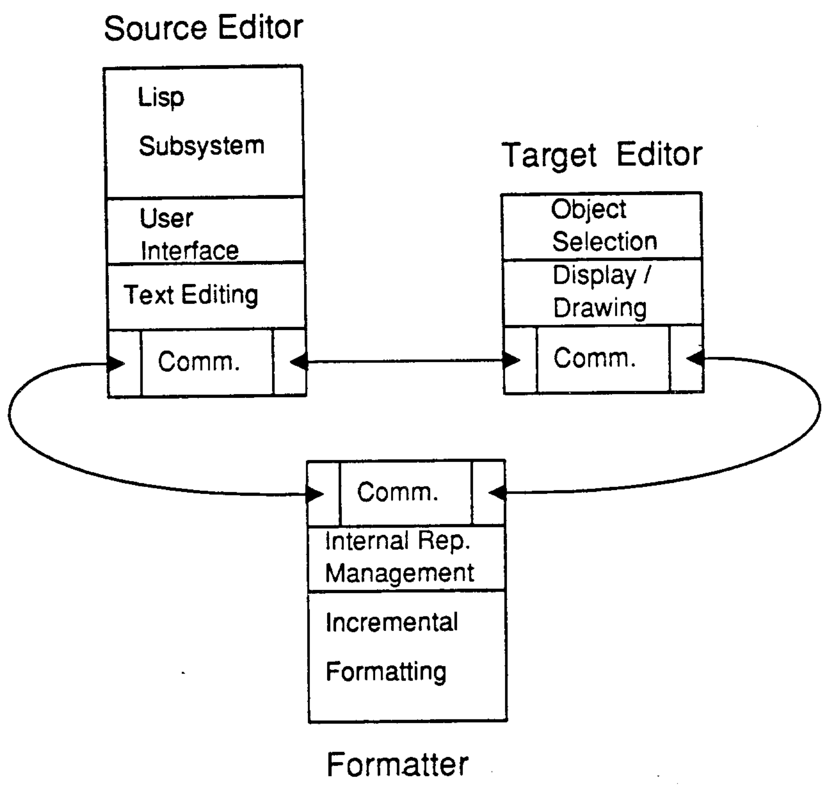


Figure 6.2: *Interconnection and functional responsibilities of the VORTEX trio.* The figure shows how the three principal components of VORTEX, the formatter and the two base editors (source and target), are connected and what their respective responsibilities are.

The VORTEX prototype discussed here is based on a distributed framework. Figure 6.2 shows the principal trio being connected by a socket between each pair of components. Figure 6.2 also depicts each component's primary responsibilities. All three components have a small subsystem that handles its communication protocols. The source editor supports a Lisp subsystem that is the basis of integration and reverse mapping mechanisms. Since it drives the entire system, the source editor also manages the user interface of the target editor as well as its own interface. The source editor also takes care of text editing, which is intertwined with its Lisp subsystem to some extent.

The target editor is responsible for rendering the formatted document. It also handles scrolling, selection, and menu-driven command invocations. The formatter is in charge of incremental formatting. It maintains the internal representation for a document and provides low-level primitives for accessing the internal representation in response to queries from the two base editors. These queries may be directly requested by the user

command op-code
primary operand
length of additional arguments (below)
...

Table 6.1: *Packet header of VORTEX's inter-process communication protocols.* Packets in the VORTEX inter-process communication protocols are variable-length datagrams with a fixed-length header as shown here.

through the target editor, or as commands decomposed from higher-level operations invoked in either of the two based editors.

6.3 Communication Protocols

The structure of the prototype is one that employs multiple processes under a distributed environment, so it is necessary to devise protocols for inter-process communication (IPC) among participating parties. There is one protocol for each pair of system components. Packets are variable-length datagrams with a fixed-length header. As shown in Table 6.1, the packet header has three fields: command op-code, primary operand, and an integer marking the length of additional arguments, if any, which comprise the remainder of the packet. The primary operand is the identifier of a low-level internal representation accessing primitive in some cases. It can also be a file identifier, a physical page number, or a window identifier under different circumstances.

The protocols defined among the principal trio are not too complicated. Tables 6.2 and 6.3 are each one half of the IPC protocol between the source editor and the formatter. Detailed protocol specification can be found in Appendix A. The prefix indicates the direction the command is going: `SF_XXX` goes from the source editor to the formatter; `FS_XXX` goes the other way around. The source editor uses `SF_Format` to start the initial formatting or any incremental run of reformatting. Since only one document is handled by the formatter, specifying document identifier is unnecessary In TEX, a document may contain multiple files, but the name of the root file is considered the document name. Therefore, in the initial run, `SF_Format` passes to the formatter the document root file name. For any incremental run, this information can be omitted.

Because the system is distributed and originates from the source editor's file space,

command op-code	*primary operand*	*additional arguments*
SF_Format	—	document (root file) name
SF_OpenFile	file identifier	file
SF_CloseFile	file identifier	—
SF_Insert	file identifier	offset, data to be inserted
SF_Delete	file identifier	offset, number of chars to be deleted
SF_Execute	routine identifier	actual parameters
SF_Abort	—	—

Table 6.2: *Protocol for "source editor → formatter" communication.*

whenever a file is needed by the formatter, it must be obtained from the source editor. The formatter makes a request by sending a FS_InputFile packet containing the file name in question. The source editor responds to such a request using SF_OpenFile, which transmits the file to the formatter. The formatter would replace the old copy of the same file if it already exists. In the case of a missing or unreadable file, the source editor may include a bad file identifier in the header of SF_OpenFile so that the formatter can abandon the pending file reading. Conversely, SF_CloseFile asks the formatter to remove a file from its work space. This may happen when a file is explicitly killed in the source editor. The next time this file is encountered in the document, it will be read from the file system (disk).

When a source editor is tightly-coupled with a formatter under a distributed framework, a file must be replicated one way or another using either the source editor's file space or the formatter's file space as a basis (no shared network file system is assumed). If a distributed document-processing system is based on its formatter's file space, the replication process can be somewhat optimized; only those files or portions of them that the user explicitly visits must be replicated in its source editor. The current VORTEX prototype takes the opposite approach and thus cannot take advantage of the optimization opportunity. This is because every file included in a document must be replicated in the formatter's end for processing when files exist in the source editor's file space. The primary reason behind this design decision is that a system centered around its source editor's file space supports sharing of files among documents. Under the current design, VORTEX requires each document to be processed by an instance of the formatter as a separate process with its own file space. If the formatter's file space were to be used as the base file system, different documents sharing the same set of files under a single editor control would not be possible.

command op-code	*primary operand*	*additional arguments*
`FS_InputFile`	—	file name
`FS_OutputFile`	—	file name, file
`FS_Message`	—	actual message
`FS_Error`	file identifier	line number, error message
`FS_Return`	routine identifier	return code/value

Table 6.3: *Protocol for "formatter → source editor" communication.*

There are a number of instances where the formatter feeds information back to the source editor. For example, when formatter output is generated in a file (e.g., the log file or the final off-line representation of the formatted document), it must be returned to the source editor's file space, because, after all, all files originate from that file system. Sending a file back is done by an `FS_OutputFile` packet. During a processing session, the formatter may also generate progress messages or error messages. In either case, the messages must be reported back to the source editor. The commands `FS_Message` and `FS_Error` are intended for reporting progress and error messages, respectively. An `FS_Error` packet includes the file identifier and line number where the error occurs. The source editor uses this information to pop to the context in question, allowing the user to make immediate corrections.

Once a file is replicated in the formatter's work space, changes made in the source editor are spontaneously propagated to the formatter piece by piece. `SF_Insert` tells the formatter to insert some number of characters in a designated file. `SF_Delete` is the opposite; it asks the formatter to delete some number of characters from a designated file. Given these two primitive operations, it is sufficient to keep the replicated file up-to-date. The most fine-grain operation is one character per insert/delete packet. This would cause potential degradation in network bandwidth. A number of optimizations can be exploited by the source editor to propagate changes in larger packets. For instance, updates to a region of consecutive text can be bundled in a batch. Changes that are immediately undone need not be propagated at all. Furthermore, positioning the *action marker*, which designates where insert/delete should take place, must be efficient to reduce search overhead. The *offset* shown in Table 6.2 as an argument to `SF_Insert`/`SF_Delete` can be thought of as the relative traveling distance of the moving marker between two successive update events in the same file.

The source editor assigns a unique identifier to each character which must, among

command op-code	*primary operand*	*additional arguments*
TF_SendPage	page number	—
TF_Execute	routine identifier	—

Table 6.4: *Protocol for "target editor → formatter" communication.*

other things, encapsulate the identifier of the file it belongs to. Because multiple files may be included in a document, this information is essential for the source editor itself to know where a character is once it is specified by the formatter.

The integration mechanism built on top of the source editor's Lisp programming layer may need access to the formatter-maintained internal representation. Each low-level internal representation accessing primitive can be uniquely identified, and its formal parameters and return code/value are known a priori. High-level functions provided by the integration mechanism, which are evaluated in the source editor, decompose into these low-level primitives, which are resident in the formatter. The bridge between them is the pair of SF_Execute and FS_Return commands. SF_Execute identifies a specific low-level primitive and passes it the required parameters. The formatter, upon receiving this request, invokes the corresponding routine and responds with FS_Return, including the function's return code and value.

Routine invocations can be folded into the protocol directly, instead of indirectly addressed through SF_Execute. However, the indirect addressing scheme yields a more stable protocol; new primitives can be registered without modifying protocol definition. This basic protocol does not support composite function invocations. There is no easy way to specify executing several primitives, each taking arguments from the preceding function, in one packet. Composite invocations reduce IPC round-trip overhead, but are more complex in semantics. Finally, the command SF_Abort terminates the formatter explicitly.

The communication between formatter and target editor is not as complex. Tables 6.4 and 6.5 show a simple protocol currently in use. Its formal specification is detailed in Appendix B. Whenever the target editor is asked to perform an operation, it issues a TF_SendPage command. The formatter replies with either (1) FT_PageNotFound, which means the document does not contain such a page, (2) FT_PageOkay, in which case the page is already in the target editor's work space and is still valid, or (3) FT_PageInfo, which transmits the target representation of the designated page as a sequential byte stream to

command op-code	*primary operand*	*additional arguments*
FT_PageNotFound	page number	—
FT_PageOkay	page number	—
FT_PageInfo	page number	flattened page stream
FT_Return	routine identifier	return code/value

Table 6.5: *Protocol for "formatter → target editor" communication.*

the target editor (see Section 6.5). In case 3, either the target editor has not requested that particular page before, or the page in its work space is no longer valid.

FT_PageOkay is intended to reduce unnecessary network transmission of a page that is still valid. Although not implemented in the current version, simple heuristics can be embedded in the target editor that makes the TF_SendPage query redundant in some cases. For instance, after one FT_PageOkay is received, the target editor can disable such queries for any continuous, non-destructive target-level operations on the same page. Here, "continuous" means the command sequence does not contain source-level operations, which conceivably may invalidate the current target page.

The way the TF_Execute/FT_Return command pair works is identical to the sequence of SF_Execute and FS_Return commands discussed above. The target editor relies on a host of internal representation accessing primitives to perform effective selections. For instance, the selection mechanism must know whether or not a selected text region contains any macro expansion, thereby prompting appropriate messages to clarify the user's intention (see Section 5.4). The formatter is fully capable of providing such vital information.

The communication between the two base editors deals largely with *windowing* (create/destroy a window), *scrolling* (goto a certain page), and *positioning* (goto a certain point in a page). Another important aspect of the protocol is to support reverse mapping. As mentioned in Section 5.4, each target-level operation corresponds to a source-level Lisp function that implements its semantics. When a target-level operation is invoked, the corresponding Lisp function is executed in the source editor, whose side effect gets propagated to the formatter automatically. The document is then reevaluated, and the result redisplayed.

6.4 Synchronization

The protocols defined above comprise a mix of asynchronous and synchronous activities. Inter-process communication in the VORTEX prototype is not remote procedure call (RPC) [22] *per se*, but it does support some flavors of RPC, in cases like execute/return. The major difference between RPC and the general form of message passing is that in RPC a call is guaranteed to return synchronously with a value, much as in the case of a local procedural call. General message passing guarantees arrival of a call, but does not need to block on the return event. RPC and message passing are functionally equivalent, but application-specific tradeoffs often dictate which gets applied [37]. In VORTEX, both types of semantics are supported. There are a few critical cases where a call must block on return, while most other IPC operations are asynchronous.

Asynchrony is essential for parallelism considerations. For instance, when the source editor is down-loading a file to the formatter, the formatter may be down-loading a target page to the target editor. Synchrony is important in some critical situations. A good example is requesting a file in the formatter, and subsequently processing it when it has been down-loaded by the source editor. If this were asynchronous, the re-entry point to the file processing routine may be lost in the first place. Moreover, before the `SF_OpenFile` arrives, other events may have taken place, which could prevent the continuation of the pending file reading. An elaborate scheduling techniques like an event priority queue can solve this semantic nightmare. In the VORTEX prototype, this undesirable phenomenon is avoided by supporting simple RPC semantics.

Whatever the case, protocol-level synchronization is transparent to the user. At a higher level, however, synchronization between source and target representations becomes fully exposed. In the VORTEX prototype, these dual representations are kept synchronized under both asynchronous and synchronous modes. Changes made in the source editor are propagated to the formatter automatically, but reformatting must be explicitly triggered by the user. In other words, the one-way transformation $S \rightarrow O \rightarrow T$ is asynchronous. On the other hand, changes made in the target editor are first propagated to the source representation by the reverse mapping device, but in this case, those changes are immediately mapped back to the target representation. In short, the transformation loop $T \rightarrow S \rightarrow O \rightarrow T$ is synchronous.

These design decisions are dictated by reformatting granularity as well as user interface considerations. The prototype employs a page-based incremental strategy (see Chapter 7), in which word hyphenation and line breaking involve examining every letter in a paragraph. Keeping the target synchronized with the source for every source-level keystroke is not only expensive but unnecessary. Unlike pure direct-manipulation systems where *immediate execution* of every single keystroke is crucial to the user's next move, the presence of both source and target representations supports *delayed execution*, in which the target is brought up to date only when the user explicitly requests so. More discussion on the differences between the two execution modes can be found in Chapter 7.

On the other hand, as in pure direct-manipulation systems, changes made in the target editor must be immediately evaluated. Otherwise it is difficult, after one change in the target representation, to continue the next target-level operation. Therefore, in VORTEX, fine grain changes, such as character insert and delete, only happen in the source representation. By design, the system does not preclude direct insert/delete on the target representation. Since the immediate evaluation is unable to keep up with the normal typing pace, a decision was made to "warp" any such operations to the source editor, making the re-evaluation user-driven. Target-level operations are oriented toward the manipulation of document appearance, such as attribute queries and object placements.

6.5 Internal Representation

The preceding discussion has made it clear that the two base editors are each responsible for the presentation and manipulation of one document representation. The two representations meet in the formatter, which derives target from source and is able to keep them correlated. In order to maintain an effective correlation, an intermediate object representation is used as the go-between. The complete picture of VORTEX's internal representation, the IR, is that the formatter maintains a copy of the source representation IR_S, on top of which sits the intermediate object representation IR_O, and tightly coupled to the $IR_S \cup IR_O$ is the target representation IR_T. A detailed specification of the IR is given in Appendix C.

The IR_S is essentially a replica of the source document in the formatter's work space. Each node in the IR_S contains one character that appears in the original document. Each character has a *unique identifier*, which is used by the source editor and the formatter in exchanging information. Events such as `SF_Insert` and `SF_Delete` from the source editor

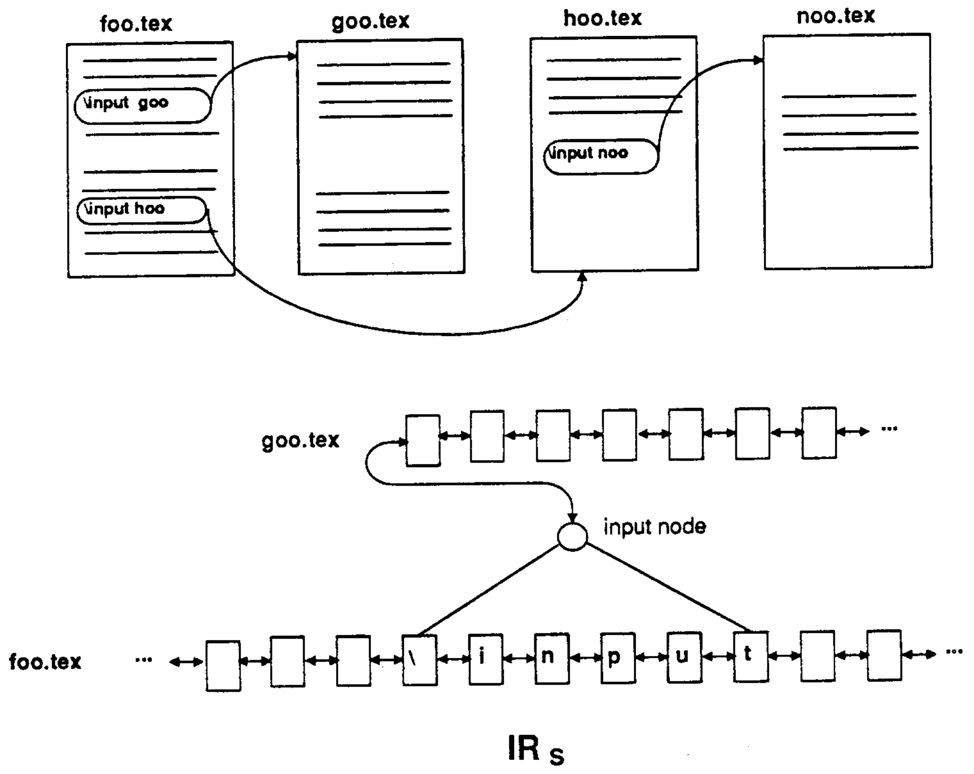


Figure 6.3: *VorTeX's internal source representation (IR_S).* The figure shows a document rooted at the file `foo.tex`, which includes files `goo.tex` and `hoo.tex`, which in turn includes the file `noo.tex`. The IR_S representation is a doubly-linked list. The spine is the sequence of character nodes from `foo.tex`, with branches corresponding to various external files.

update the IR_S asynchronously. If efficient update is the foremost concern, the IR_S can be organized as a doubly-linked list of character nodes. Figure 6.3 is an illustration of the IR_S in the current VorTeX prototype. Each node in the IR_S contains a character and pointers to its left and right siblings, its parent in the IR_O, and its corresponding target box, if any, in the IR_T. A document may comprise multiple files. Whenever the control sequence for file inclusion (`\input`) is identified, a new list of nodes is attached to the main IR_S spine.

The IR_O is intended as a hierarchy superimposed on top of the IR_S. This hierarchy reflects the structural view of the underlying document in terms of its syntactic objects. It does not support such logical entities as chapters, sections, figures, etc. According to the design outlined in the Chapter 5, those entities are emulated at a higher level. The IR_O nodes are generated as the IR_S is scanned and processed. It captures as much semantic information as possible for any syntactic token identified in the IR_S.

class	*type*

plain	`IRs_Char`
	`IRo_Ligature`
	`IRo_Word`
	`IRo_Paragraph`

control sequence	`IRo_DefFont`
	`IRo_DefMacro`
	`IRo_Font`
	`IRo_Rule`
	`IRo_Symbol`
	`IRo_Special`
	`IRo_Macro`

math	`IRo_Math` `IRo_DisplayMath`
file	`IRo_Input`
miscellaneous	`IRo_Group` `IRo_Space`

Table 6.6: *Classes and types of nodes in the IR_O.* The table lists the types of nodes currently supported in VORTEX's internal intermediate object representation (IR_O). Classes here are artificial; they are introduced for the purpose of classifying these types, and nothing more.

The most important piece of information for any IR_O node is its *type*, which indicates the nature of its children nodes in the IR_S. Table 6.6 lists possible node types in the $IR_S \cup IR_O$ by classes. There are five basic classes, each containing certain types of nodes. Classes have no special meaning here, other than to classify the types of nodes that are similar in nature. An `IRs_Char` node appears exclusively in the IR_S, while all `IRo_xxx` nodes appear only in the IR_O. In the *control sequence* class, the definition of a font (`IRo_DefFont`) is separated from the definition of a macro other than a font (`IRo_DefMacro`). Similarly, invoking a font (`IRo_Font`) is distinguished from invoking something other than a font. In particular, the invocation of a rule (`IRo_Rule`), a mathematical symbol (`IRo_Symbol`), or special object (`IRo_Special`), such as a piece of graphics, is separated from the invocation of a ordinary macro (`IRo_Macro`). Other types of nodes are self-explanatory.

The IR_O nodes can be nested, but the nesting is typically shallow. In most cases, going up from an IR_S node, it only takes a handful of links to the root, which is normally a `IRo_Paragraph` node. Figure 6.4 is a snapshot of a typical internal representation of the $IR_S \cup IR_O$. The way the IR_O is constructed is to follow the sequential flow of the

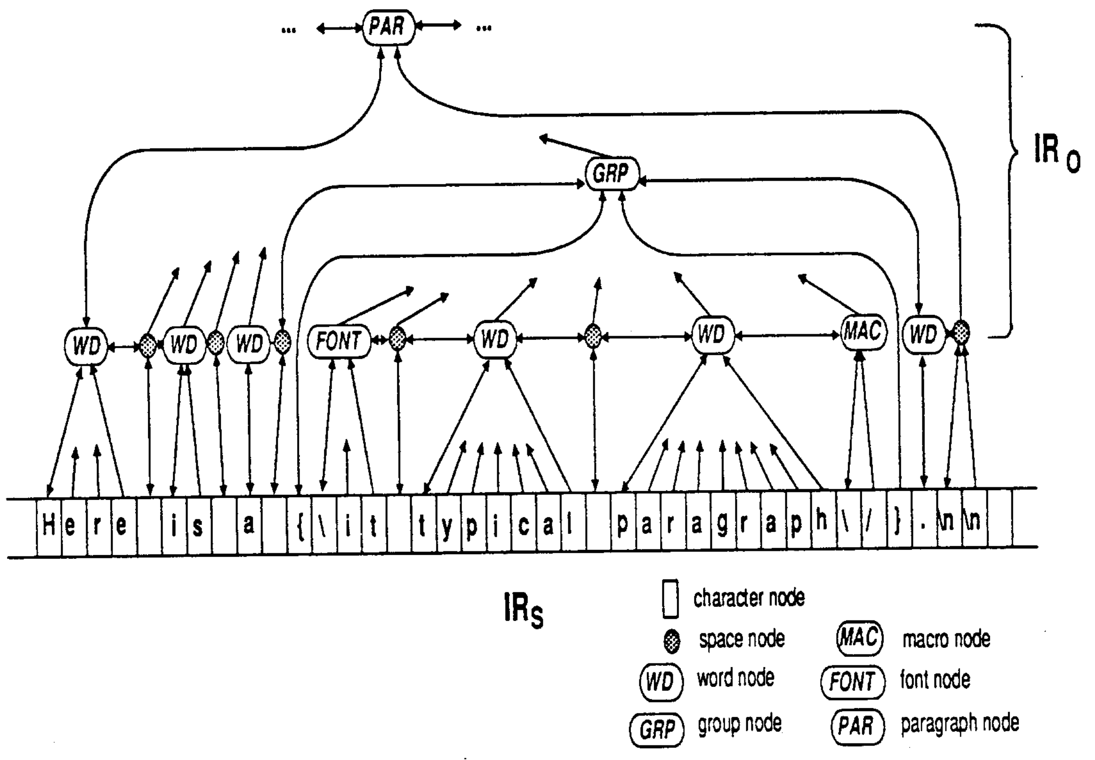


Figure 6.4: *VORTEX's internal source and object representation ($IR_S \cup IR_O$).* The figure is a snapshot of a sample $IR_S \cup IR_O$. An arrow indicates the direction of a link. All short arrows point to their parent. Most nodes in the $IR_S \cup IR_O$ have links to the IR_T, but they are not shown in the figure.

document. Most IR_O nodes are created during the parsing phase of the IR_S. Their types, in some cases, are not filled in until a later stage of the processing.

In the vertical direction, an IR_O node always points to its leftmost child, while every node except the root in the $IR_S \cup IR_O$ has a upward link to its parent in the IR_O. This is designed to support efficient dependence tracking. Recall that changes made in the source editor are spontaneously propagated to the formatter via *SF_Insert* or *SF_Delete* events. In response to these events, the formatter no only updates the content of the designated node in the IR_S, but also marks the node as *dirty*. When an IR_S node is dirty, it only takes a few links to mark its enclosing object as dirty. In Figure 6.4, to mark the paragraph as dirty, it take just 2 or 3 steps upward from an IR_S node. Traversing the IR_O horizontally yields the syntactic structure of a document. This can be done at several levels. For instance, in Figure 6.4, the node following the three leftmost `IRo_Word`

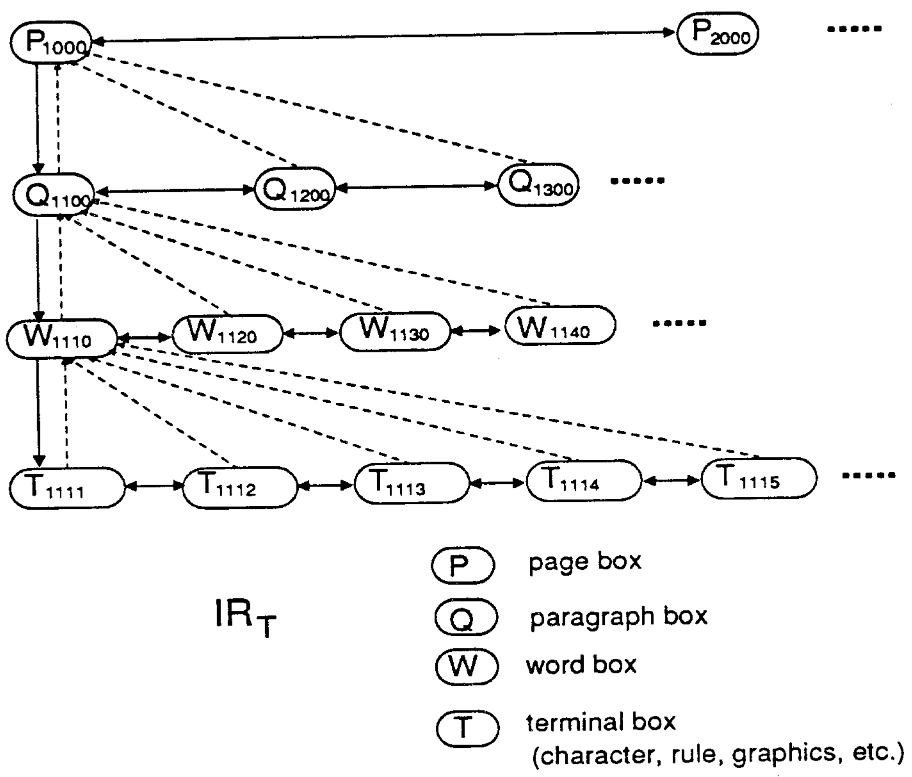


Figure 6.5: *VORTEX's internal target representation (IR_T).* The figure is a partial page of a sample IR_T. These boxes have cross reference links to the $IR_S \cup IR_O$, which are not shown in the figure. The subscript in each box is its identifier, which consists of four digits.

nodes ("`Here is a`") is a `IRo_Group` node, under which a `IRo_Font`, two `IRo_Word`, and a `IRo_Macro` are found to be horizontally linked. A similar structural relationship exists at the `IRo_Paragraph` level.

Certain nodes in the $IR_S \cup IR_O$ have cross reference links to related nodes (not shown in Figure 6.4). For instance, most non-space, non-delimiter nodes in the IR_S have a link to a counterpart in the IR_T, and vice versa. The more interesting case happens in font/macro definition and invocation. A node marking the invocation of a user-defined font (`IRo_Font`) or macro (`IRo_Macro`) has a link to the node where the font or macro is defined (ie., the corresponding `IRo_DefFont` or `IRo_DefMacro` node). This cross reference information provides the necessary low-level support for the macro unraveling facility (see Section 5.4). System-defined fonts and macros, which are preloaded into the system symbol table, do not have corresponding nodes in the IR_O. Therefore, an IR_O node marking the invocation of a system font or macro has no cross reference link to its definition. Under this

level/digit	*bit allocation*	*bit count*	*effective yield*	
1	31 — 22	10	1,024	pages per document
2	21 — 16	6	64	paragraphs per page
3	15 — 6	10	1,024	words per paragraph
4	5 — 0	6	64	letters per word

Table 6.7: *Bit allocation for the IR_T box identifiers.* Under the current VORTEX prototype, boxes in the IR_T have an identifier of 32 bits. This table shows how these bits are used. Under this scheme, the maximum number of pages for a document is 1,024. If this is considered too restrictive, a slight variation to this scheme could allocate 11 bits to the per-document page count (2,048 pages per document), leaving 5 bits to the per-page paragraph count (32 paragraphs per page).

principle, modifying preloaded fonts and macros is not allowed.

During code generation, the internal target representation (IR_T) is produced. As shown in Figure 6.5, the IR_T is a hierarchy with a fixed depth of four, consisting of three levels of non-terminal boxes (page, paragraph, and word), and one level of terminal boxes. In referring to the IR_T, the term *box* is deliberately used to distinguish it from a *node* in the $IR_S \cup IR_O$. Each non-terminal box is connected with its subordinate structure by a link to the leftmost child (down arrows). All except page boxes are linked to their parent (dashed up arrows). Every box, non-terminal or terminal, has links to its siblings (left-right arrows).

The IR_T hierarchy enables very simple *encoding* and *decomposition* schemes for the communication between formatter and target editor. The encoding scheme assigns an identifier to each box. The idea is, given an identifier, that the corresponding box can be located in the IR_T as quickly as possible. The current VORTEX prototype uses a 4-digit number to encode an identifier. From left to right, each digit encodes one hierarchy level in the top-down IR_T. The value of a digit reflects a box's absolute position in its sibling chain. For an arbitrary box at level i, $1 \leq i \leq 4$, the digit code is $D_{i1}D_{i2}D_{i3}D_{i4}$, where

$$D_{ij} = \begin{cases} 0 & \text{if } j > i \\ p & \text{if } j = i \\ D_{(i-1)j} & \text{if } j < i \end{cases}$$

where p is the absolute position of the box in its sibling chain and $1 \leq j \leq 4$,

A 4-digit identifier like this can easily be packed into a 32-bit word whose bit allocation is shown in Table 6.7. Based on this simple encoding scheme, finding an arbitrary box is quite efficient. On the average, the number of links that must be followed in locating a terminal box is

$$\frac{D_{41} + D_{42} + D_{43} + D_{44}}{2}$$

which, in the extreme case, is around 1,000. Practically, this number is an order of magnitude smaller, because both the number of pages per document and the number of words per paragraph are around 100, instead of 1,000, in the normal case. Since the identifier encodes explicitly the position of a box in the IR_T, no comparison is needed in following the path to the wanted box. Furthermore, if the *locality* of references is properly exploited (most references happen in the same page, or even the same paragraph), the search overhead can be reduced even further. Also, the way target box identifiers are encoded enables the target editor to perform effective selections. Simple comparisons on two identifiers can determine the ordering of their underlying boxes; the smaller identifier always correspond to a box in the precontext of the other box.

The role of the decomposition scheme is to flatten the IR_T into a sequential stream which is necessary when a page is to be transmitted from the formatter to the target editor over the network. The current scheme represents a page as the in-order traversal (root-down-right) of the IR_T. The sequential byte stream contains not only the information for every box in a page, but also commands like `DOWN`, `RIGHT`, and `UP`, which are used to mark the links among the boxes.

Table 6.8 shows various types of the IR_T boxes currently supported in the prototype. The host of terminal boxes provides necessary support for target-level object selections. The bottom line is that certain types certain operations are only allowed to be performed on certain types of terminal boxes. This "type" information facilitates the target editor making appropriate decisions. An `IRt_Char` box is just an ordinary character with a matching node in the IR_S. An `IRt_Ligature` box has a cross reference link to an `IRo_Ligature` node in the IR_O. An `IRt_Hyphen` box, with a cross reference link to a `IRo_Word` node in the IR_O, marks a hyphen generated by automatic word hyphenation. Massaging a character like this is illegal in the target editor. The next three types of boxes, `IRt_Rule`, `IRt_Symbol`, and `IRt_Special`, are similar to their the IR_O counterparts.

The last two types of terminal boxes, `IRt_Explicit` and `IRt_Implicit`, are crucial to the reverse mapping device. Both types of boxes contain a character that does not exist

class	*type*
terminal	`IRt_Char`
	`IRt_Ligature`
	`IRt_Hyphen`
	`IRt_Rule`
	`IRt_Symbol`
	`IRt_Special`
	`IRt_Explicit`
	`IRt_Implicit`
non-terminal	`IRt_Word`
	`IRt_Paragraph`
	`IRt_Page`

Table 6.8: *Classes and types of boxes in the IR_T.* The table lists the types of boxes currently supported in VORTEX's internal target representation (IR_T).

in the corresponding context of the IR_S, but instead is *introduced* by a macro expansion. The important difference is, in the case of `IRt_Explicit`, that the character comes from a macro invocation that explicitly appears in the source representation. In the other case, the character in a `IR_Implicit` box is produced by some implicit macro expansion, such as the header or footer of a page. Normally these implicit macros are part of the page construction routine, and are not explicitly invoked in the user document.

The correspondence between nodes in the $IR_S \cup IR_O$ and boxes in the IR_T is a relation, not a function. In particular, delimiter and space nodes in the $IR_S \cup IR_O$ have no counterparts in the IR_T. Conversely, text introduced by macro expansions in the IR_T cannot find exact matches in the $IR_S \cup IR_O$. For instance, all members of a ligature in the IR_S are associated with a single box in the IR_T. The most interesting problem arises when one object in the IR_O is split into two in the IR_T, as exemplified by a hyphenated word or a paragraph which extends accross a page boundary. If a one-to-one correspondence were appropriate, one approach would be to split the node in the IR_O into two. However, that would cause a rather significant ripple effect all the way to the root in the IR_O. The current prototype allows a *one-to-many* correspondence between an object in the IR_O and all its derivatives in the IR_T.

class	*routine (arguments)*	*return value*
position transformation	*irs_to_irt* (`IRs.id`)	`IRt.id/nil`
	irt_to_irs (`IRt.id`)	`IRs.id/nil`
type checking	*irs_type* (`IRs.id, &n`)	`IRo.type/nil`
	irt_type (`IRt.id`)	`IRt.type/nil`
bounded search	*irs_find_font* (`&IRs.bor, &IRs.eor, &name`)	`IRs_FontPackage/nil`
	irs_find_macro (`&IRs.bor, &IRs.eor, &name`)	`IRs_MacroPackage/nil`
	irs_find_math (`&IRs.bor, &IRs.eor`)	`IRs_MathPackage/nil`
	irs_find_group (`&IRs.bor, &IRs.eor`)	`IRs_GroupPackage/nil`

Table 6.9: *Primitives for accessing VORTEX's internal representation.* The table classifies the currently-supported primitives for accessing VORTEX's internal representation (IR). The prefixes IRs, IRo, and IRt represent a node/box in the IR, IR_S, IR_O, and IR_T, respectively. IRs.bor and IRs.eor represent the beginning and end of a region in the IR_S. An argument with the ampersand (&) prefix is optional.

6.6 Accessing Internal Representation

The formatter supports a number of low-level primitives that provide access to the IR. These primitives are used not only by the formatter itself in maintaining the IR, but more importantly, by the two base editors for a wide range of operations including selection, synchronization, and reverse mapping. Table 6.9 lists three classes of IR-accessing primitives currently supported in the prototype. The first class includes two *position transformation* primitives, which deal with transformations from source to target, and vice versa. Given the identifier of an IR_S node, *irs_to_irt ()* returns the identifier of its corresponding box in the IR_T, if any. The routine *irt_to_irs ()* does the opposite. These two primitives form the basis of VORTEX's *synchronized inter-representation scrolling* (intra-representation scrolling is carried out independently by each editor). In the class of *type checking*, primitives return the type of a node. In particular, *irs_type ()* takes an additional argument n, which specifies that the node to be checked is n levels above the given IR_S node.

Each primitive in the *bounded search* class takes a region specified by the identifiers of its two bounding IR_S nodes, sweeps from left to right searching for the corresponding node on the IR_O above the region, and returns all matches in a *package*, a list of one or more lists, each of which contains the scope information of a node found in a designated region. If a region is unspecified, the entire IR_S is assumed. There is a small variation in finding fonts or macros: both *irs_find_font ()* and *irs_find_macro ()* take an optional argument of

package	*content*
`IRs.FontPackage`	`[[ IRs.bon, IRs.eon, { IRs.def.bon, IRs.def.eon, IRs.def.bob, IRs.def.eob }] [ IRs.bon, IRs.eon, { IRs.def.bon, IRs.def.eon, IRs.def.bob, IRs.def.eob }] ...... ]`
`IRs.MacroPackage`	`[[ IRs.bon, IRs.eon, { IRs.bob, IRs.eob }, { IRs.def.bon, IRs.def.eon, IRs.def.bob, IRs.def.eob }] [ IRs.bon, IRs.eon, { IRs.bob, IRs.eob }, { IRs.def.bon, IRs.def.eon, IRs.def.bob, IRs.def.eob }] ...... ]`
`IRs.MathPackage`	`[[ IRs.bom, IRs.eom ] [ IRs.bom, IRs.eom ] ··· ]`
`IRs.GroupPackage`	`[[ IRs.bog, IRs.eog ] [ IRs.bog, IRs.eog ] ··· ]`

Table 6.10: *Return packages of bounded search.* The table specifies each package returned by bounded search. A package is a list of one or more lists, each of which contains the scope information of a node found in a designated region. Items enclosed in $\{\ldots\}$ are optional; they may be present in some cases, but absent in others. In the first two rows, `IRs.bon` and `IRs.eon` denote, respectively, the beginning and end of a font or macro's name. `IRs.bob` and `IRs.eob` denote the beginning and end of a macro's body or argument list. Similarly, `IRs.def.bon` denotes the beginning of the definition of a font or macro's name. The other mnemonic symbols follow the same convention. In particular, `bom` and `eom` stand for the beginning and end of a math mode (including display math mode); `bog` and `eog` represent the beginning and end of a group.

name, which, when bound to a string, returns only those instances that match such a name; if the name is unspecified, all instances of any font and macro, respectively, are returned.

Table 6.10 specifies each return package. Basically, a return package is a list of one or more lists, each of which contains the matched node's (in the IR_O) scope information (in terms of one or more IR_S node pairs). The reason for returning all matches in one package is to reduce network transmission overhead. In `IRs.FontPackage` and `IRs.MacroPackage`, not only the context of a font or macro is returned, but also its definition, if available, is included. These simple primitives are the backbone of a surprisingly powerful mechanism that realizes VORTEX's reverse mapping device.

6.7 Realizing Reverse Mapping

Instead of being a monolithic, stand-alone processor, VORTEX's reverse mapping facility is a layered device based on low-level support provided by the principal trio. Each class of target-level operations corresponds to a Lisp function in the source editor, which is executed with the necessary parameters supplied by the target editor when the target operation is invoked by the user. Under this model, the primary responsibilities of the principal trio are as follows.

- The target editor is responsible for determining the actual parameters and subsequently passing them to the source editor while triggering the corresponding mapping routines.
- The source editor's Lisp programming subsystem supports a full collection of editing functions as well as a host of *IR*-accessing primitives, which mirror what is available in the formatter.
- The formatter has two responsibilities. It responds to RPCs from the source editor in accessing the *IR*. Also, it reformats the document to reflect any side effect caused by a target-level operation. This provides the user with the sensation of directness.

Most target-level operations involve the *current selection*, which is normally a text region identified by its beginning and end characters and highlighted on the screen. Additional arguments are specified by invoking menu items or by typing at the editor prompt (dialogue box). The operator/operand relationship is realized by a mixture of prefix and postfix commands.

Consider the case of changing a text region to a new font. First the region is selected (two prefix operands, beginning and end of the region). Next a menu of all available target-level commands is invoked, from which the item *font change* (operator) is selected. A pull-aside menu immediately appears, which shows a number of options to which the font can be changed. Upon selection of a font (a postfix operand), the target-level operation is complete, and the corresponding source-level Lisp code is triggered.

Selecting a text region is non-trivial due to the macro unraveling problem. In principle, every character in the selected region must be examined before a decision can be made regarding the feasibility of the intended operation. At such a fine level of granularity, querying the formatter for this information is bound to be slow. The replicated the IR_T in the target editor actually has all the necessary information to make the judgement. It is for this reason that terminal boxes in the IR_T must include so many different types (see Table 6.8).

The premise of performing any target-level operation is that the target representation must be up-to-date before starting. When the target editor becomes active (mouse focus is moved into one of its windows), the target editor automatically issues a `TF_SendPage` command to bring the page being displayed up-to-date before anything takes place. If a page must be transmitted for that purpose, user operations are disabled before the transmission is completed. From the target editor's stand point, triggering the corresponding Lisp function for reverse mapping is an RPC. Upon return, another `TF_SendPage` is sent to the formatter so that any side effect of the operation can be shown synchronously. Additional RPCs occur in the source editor, where each Lisp-level IR-accessing primitive performs an RPC (with respect to the formatter) to obtain the result. Here, repetitive information for a region is iteratively obtained in the formatter side and is returned as a bundled package to reduce network communication overhead.

To get a flavor of VORTEX's reverse mapping capability, some basic mappings are described below:

1. *Synchronized scrolling*. The user selects a point in the target editor and demands that the source editor scroll to the corresponding position, and vice versa. Alternatively, a flag can be turned on so that the two representations would automatically scroll in synchrony. When the target editor is the driver, the identifier of a terminal box is sent to its corresponding mapping routine in the source editor, which invokes `irt-to-irs`, which decomposes into an RPC to formatter's *irt_to_irs ()*. Upon return, the source editor simply moves its cursor to the character using `goto-id`.

2. *Object placement: cut-and-paste*. The user selects a region (two target boxes) and a destination point (a third box). The corresponding mapping function takes these arguments, translates them into source positions, kills the region, and yanks it at the destination point.

3. *Text copy: copy-and-paste*. Two possibilities. The first is similar to *cut-and-paste* above, with region being copied as opposed to being killed. The other is related

```
(defun font-change (bor eor fnt)
  (let ((bor (irt-to-irs bor))
        (eor (irt-to-irs eor))
        (fnt (font-name fnt)))
    (message "Changing region font to %s..." fnt)
    (if (and bor eor)
      (let ((open (concat "{" fnt " "))
            (close (if (italic-correction-p fnt) "\\/}" "}")))
        (font-remove bor eor)
        (group-rearrange bor eor open close)
        (goto-id bor)
        (insert open)
        (goto-id eor)
        (inset close)
        (message "Changing region font to %s...done" fnt))
     (error "Changing region font to %s...failed (bad region)" fnt))))
```

Figure 6.6: *Changing font for a region.* This routine takes a region and a font of choice, in target representations, embraces the corresponding source region with an invocation to the font. The predicate `italic-correction-p` checks if a particular font requires some extra space at the end. The routine `goto-id` moves the insertion point in front of a specified source identifier, which may involve go to a file different from the one currently visited.

to macro unraveling, in that the selected region contains explicit or implicit text introduced by a macro expansion and that the user's interest is only to make a copy of the text, instead of the associate attributes. In this case, the target editor passes every single character in this region to the corresponding mapping routine which positions itself to the destination point, and inserts the string as plain text.

4. *Macro update.* If the user's intention is to modify the macro in the case where explicit macro-expanded text is selected, the target editor passes an arbitrary box in the region to its mapping routine, which calls upon `irs-find-macro` to find the position of its definition and thus scrolls there for the user to make any modifications.

As a more advanced example, Figure 6.6 is the Lisp-based reverse mapping function bound to the target-level *font change* operation described above. This function takes three arguments, `bor`, `eor`, and `fnt`, which represent, respectively, the beginning and end of a

```
(defun font-remove (bor eor)
  (let ((pack (irs-find-font bor eor))
        info bor* eor*)
    (while pack
      (setq info (car pack))
      (setq bor* (car info))
      (setq eor* (cadr info))
      (setq pack (cdr pack))
      (kill-id-region bor* eor*))))
```

Figure 6.7: *Removing all font information from a region.* This routine strips off all font invocations in a source region. It finds out where every font invocation is by calling `irs-find-font`. It removes the string using `kill-id-region`, which takes two identifiers and puts everything in between into the kill buffer. A region specified by two source character identifiers may cross over file boundaries.

region, and the font to change to. Both `bor` and `eor` are identifiers of IR_T boxes, so the first step is to convert them to their corresponding source positions by calling `irt-to-irs`, which decomposes into a remote procedure call to *irt_to_irs ()* in the formatter. The font information `fnt` is also specified in the target editor's representation, as a number, so it is also necessary for `font-change` to convert the font into a string. These are realized as the three local bindings in the outer `let` clause. If any of the converted positions is `nil`, an error is raised; otherwise the inner `let` clause is evaluated, which removes all font information from the region, rearranges the group structure within the region, and encloses the region as group, with the font string attached in front of it.

Stripping off any font information already placed in the region is an essential part of the semantics, because the new font string prepended to the region will be overridden by any font invocation inside. Removing every font invocation in the region guarantees effective influence would be exercised by the new font. The function `font-remove` shown in Figure 6.7 calls upon `irs-find-font` to identify every character included in font invocations. Recall that font information is returned as a package of one or more lists, each of which containing at least one pair of IR_S identifiers that point to the beginning and end of the font invocation string. Each iteration of `remove-font`'s main loop deletes one instance of the font string.

```
(defun group-rearrange (bor eor open close)
  (let ((pack (irs-find-group bor eor))
        info bor* eor*)
    (while pack
      (setq info (car pack))
      (setq bor* (car info))
      (setq eor* (cadr info))
      (setq pack (cdr pack))
      (if (or (left-p bor* eor) (left-p eor* bor))
        (error "This shouldn't happen, something wrong in IR.")
        (if (and (right-p bor* bor) (left-p eor* eor))
          nil        ;; group within bounds, which is okay
          (goto-id eor*)
          (insert close)
          (next-char 1)
          (insert open))))))
```

Figure 6.8: *Rearranging group structure in a region.* This routine looks up all group nodes in a source region and rearranges them by inserting the intended opening and closing delimiters. It receives information about group nodes by calling `irs-find-group`. Since all nodes are specified by unique identifiers, their relative positioning cannot be determined by simple comparison like $<$ or $>$. Rather, the predicates `left-p` and `right-p` are used to make the comparison.

In a trivial case where no group delimiters are found in the region, `font-change` is complete after font removal and the subsequent group and font attachment. The situation becomes more complicated when group delimiters exist within the scope of the region. The attachment work must take these groups into consideration so that proper scoping is not violated. As demonstrated by Figure 6.8, the routine `group-rearrange` utilizes the *IR*-accessing function `irs-find-group` to obtain every instance of grouping in the region. If both the beginning and end of a group fall inside the region, nothing special is necessary; if both are outside, an error is raised. Otherwise, an outer group's left or right branch must overlap with the region. In either case, some extra work is needed.

To illustrate the operational semantics of `font-change`, let "*italic*" be the font of choice, the opening and closing delimiters for enforcing font change on a region are "`{\it`" and "`\/}`", respectively. Figure 6.9 shows the stepwise development of enforcing *italic* font

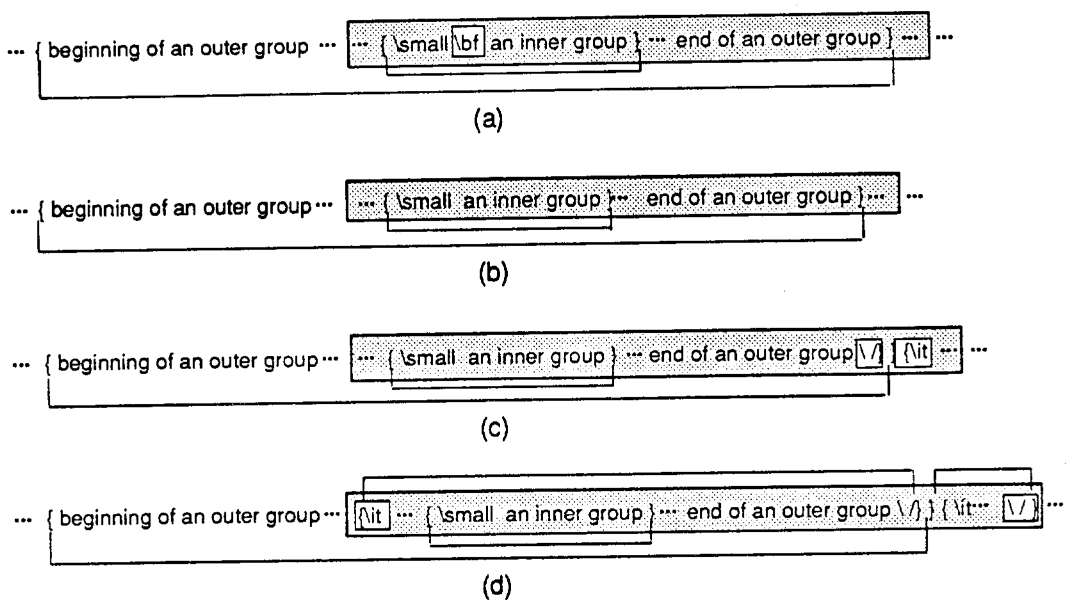

··· { beginning of an outer group ··· ··· { \small \bf an inner group } ··· end of an outer group } ··· ···
(a)
··· { beginning of an outer group ··· ··· { \small an inner group } ··· end of an outer group } ··· ···
(b)
··· { beginning of an outer group ··· ··· { \small an inner group } ··· end of an outer group \/} {\it ··· ···
(c)
··· { beginning of an outer group ··· {\it ··· { \small an inner group } ··· end of an outer group \/} } {\it ··· \/} ···
(d)

Figure 6.9: *Full treatment of region font change (case 1).* The figure shows the stepwise development of region font change. The shaded area denotes the region, in which unshaded areas are text to be deleted or just inserted. Lines connecting matching opening and closing delimiters of a group illustrates its scope. (a) An outer group's right branch overlaps with the region; an inner group inside the region contains a font invocation (`\bf`). (b) After `font-remove`, `\bf` is stripped off. (c) Extra delimiters are inserted by `group-rearrange`. (d) Finally, following `font-change`, the new font `\it` is embedded in the region under two separate groups.

on a region, which overlaps with the right branch of an outer group. The region also has a proper inner group, in which the boldface font `\bf` is explicitly invoked (Stage (a)). As indicated earlier, all instances of font invocations inside the region must first be stripped off to ensure the effectness of the font of choice (Stage (b)). Another complication stems from group overlapping, which is resolved by `group-rearrange` (Stage (c)). In this example where there is only one instance of group overlapping, the region is split into two subregions, each of which enclosed in `{\it` $\cdots$ `\/}` (Stage (d)).

Similarly, as shown in Figure 6.10, there is another case where the left branch of an outer group intersects with the region. An interesting aspect here is that the same routine in `group-rearrange` handles both cases uniformly.

The semantics of `font-change` may be extended even further. For instance, assume an outer group's left branch overlaps with a region. Suppose their intersection contains a font invocation, the current version of `font-remove` would simply strip it off. A more

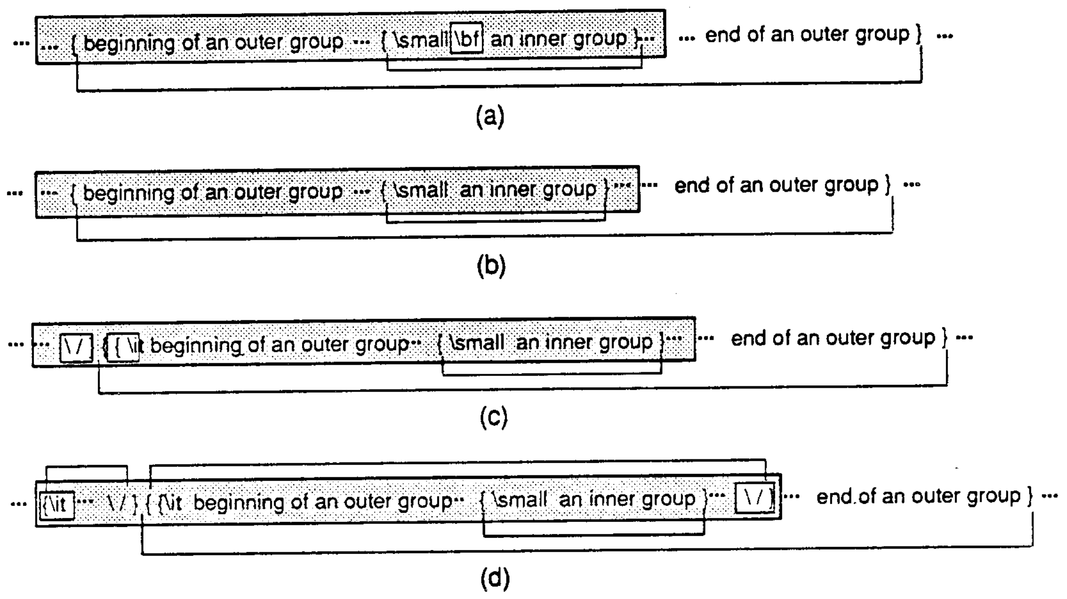

... ... { beginning of an outer group ... { \small \bf an inner group } ... ... end of an outer group } ...
(a)
... ... { beginning of an outer group ... { \small an inner group } ... ... end of an outer group } ...
(b)
... ... \/ } { { \it beginning of an outer group ... { \small an inner group } ... ... end of an outer group } ...
(c)
... { \it ... \/ } { { \it beginning of an outer group ... { \small an inner group } ... \/ } ... end of an outer group } ...
(d)

Figure 6.10: *Full treatment of region font change (case 2).* Similar to the previous figure, this figure shows the stepwise development of region font change. The shaded area denotes the region, in which unshaded areas are text to be deleted or just inserted. Lines connecting matching opening and closing delimiters of a group illustrates its scope. (a) An outer group's left branch overlaps with the region; an inner group inside the region contains a font invocation (`\bf`). (b) After `font-remove`, `\bf` is stripped off. (c) Extra delimiters are inserted by `group-rearrange`. (d) Finally, following `font-change`, the new font `\it` is embedded in the region under two separate groups.

elaborate approach may propagate this information out of the region so that its effect is preserved for the remainder of the group outside the region. There are certainly more cases along this line that may be taken into consideration. All that is needed is to extend or rewrite the functions involved.

What is special about VORTEX's reverse mapping facility is that routines responsible for carrying out the work are user-level functions. A certain group of routines in the source editor essentially defines the *operational semantics* of its corresponding class of target-level operations. As a consequence, the mechanism is flexible and extensible. For example, if preserving font attribute outside a region should be part of the semantics for font change, `font-remove` can be enhanced to handle relevant cases. Even better, more than one version of `font-change` can be supported simultaneously: one as what is given in Figures 6.6 through 6.8, a second as the elaborate scheme that preserves font attribute outside a region, and some more for other options.

Another advantage of building the reverse mapping device on top of source editor's programming subsystem is that undo operations need not be explicitly implemented for VORTEX's reverse mapping facility. Whatever the source editor supports in terms of recovering recent side effects is readily available to the reverse mapping facility. This, in effect, ensures a certain degree of reliability for reverse mapping. If the semantics of a target-level operation do not match the user's expectation, undoing any damage in the source editor is quite straightforward. If a mapping function stumbles upon odd cases, thereby introduces syntactic or semantic inconsistencies, the formatter, which is invoked synchronously to reevaluate recent updates, can easily pin-point the error. At this point, the user can either correct the error or undo the mapping. Since mapping routines are fully exposed user-level functions, the problem can also be located and repaired.

Chapter 7

Incremental Formatting

Traditional document formatting algorithms such as those for hyphenation [93], line breaking [1,90,117], and pagination [111] are non-incremental. They are designed for batch-oriented systems and do not take into account issues that are essential to interactive environments, such as response time, reprocessing granularity, etc. Since the user expects to wait for the result, a batch-oriented document processing system can use global optimization techniques to generate high quality output. Direct-manipulation systems depart from these non-incremental algorithms and focus on prompt visual response and fine grain reprocessing. Their goal is to achieve the sensation of directness as changes are being made to the document.

Shifting from a batch-oriented approach to an interactive paradigm has a number of technical ramifications. Most noticeable, quality usually deteriorates. For instance, hyphenation presents some difficulties because an attempt to hyphenate a word under construction or modification may produce nonsensical results, not to mention the semantic confusion of user-entered hyphens versus those introduced by automatic word hyphenation. Pagination is sometimes avoided or delayed due to similar concerns or to the need to achieve seamless scrolling. Line breaking is usually based on some obvious "first-fit algorithms", which do not perform as well as algorithms with look-ahead, such as TEX's line breaking algorithm [90]. As mentioned in Section 3.2.3, it is misleading to call these systems WYSIWYG because they are simply producing an approximation of the final hard-copy. A non-incremental formatter is often used as the postprocessor of the underlying documents if better quality is desired.

Quality and directness are not necessarily in conflict. Quality is a property of the formatted document, while directness is a sensation involved in the process of manipulating a document. One way to design incremental formatting strategies with increased directness without sacrificing quality is to focus on higher-level issues that are unique to the interactive situation. Some of the known non-incremental algorithms can be embedded in the innermost layer as subroutines. Many of the low-level document formatting problems have been formulated generically. Except for a few exceptions, some non-incremental solutions are still applicable in the incremental case.

This high-level approach is ideal for augmenting existing non-incremental programs. Compared to reinventing low-level incremental algorithms from scratch, this *augmentation approach* is also more flexible because a number of parameters can be adjusted to achieve the best solution for a particular situation. This chapter describes an instance of this augmentation approach — our experience of converting the non-incremental TEX to become the VORTEX incremental formatter. This methodology is the principal result reported here and appears to generalize beyond the VORTEX case.

There are quite a few systems that address incremental document processing issues. Examples include various approaches taken by systems like Etude [69], Pen [5], Janus [30], Lara [66], the **tnt** editor/formatter [55], Diamond [47], Quill [94], and so on. Work on program development environments is also related and research on incremental language processing strategies is even more extensive. References [49] and [115] give overviews and bibliographies on programming environments and incremental language processing strategies.

Despite the rich literature, treatment of incremental document formatting strategies has been somewhat ad hoc. This chapter attempts to give a more systematic analysis of the problem by introducing some basic principles of incremental processing under an interactive environment, in which editors are integrated with document-processing engines. These principles are not document processing specific; they can be applied to the design of most incremental strategies. The type of systems suitable for the high-level augmentation approach is identified. The algorithms applicable to the VORTEX paradigm are discussed in detail.

7.1 Principles of Incremental Processing

A typical integrated software environment can be identified by (1) provisions for the user to manipulate the underlying objects interactively, (2) the system's ability to reflect these modifications efficiently, and (3) the large amount of internal state that must be maintained throughout the session to facilitate the reevaluation of modified objects. These ingredients can be found in most integrated environments, whether the application area is program development, document preparation, CAD/CAM, or anything else.

One of the premises of an integrated software environment is to have the task of editing integrated with its main processing. Normally the main processor maintains the bulk of a system's internal state. A closely-coupled editor is used to register changes at a fine granularity. In the context of program development, for instance, this means editing must be closely linked to program evaluation and debugging. Similarly, in the context of document development, a document editor is intimately connected with the formatter and other processing engines. The strong integration of document editor and formatter can be found in most modern document processing systems, such as Tioga, EZ, Diamond, the tnt system [55], Quill [32], Lilac [27], VORTEX, etc.

A major distinction between an integrated environment and the traditional unintegrated approach is in the granularity of reprocessing. The unintegrated case often implies a batch-oriented strategy, in which processing always begins with a "cold start". The internal state is reconstructed in every pass and is discarded when the task is finished. Conversely, an integrated environment enables "warm start", or *incremental processing*, which reprocesses the system at a much finer granularity. Instead of starting from the very beginning, an incremental strategy detects the unchanged state and processes only the minimal necessary part, reducing computation overhead while yielding a higher degree of directness.

In designing an incremental strategy, one must consider the following important concepts: *dependence*, *pertinence*, *quiescence*, and *convergence*. Before defining these concepts, the underlying *execution model* must be clarified first. Two models are of interest here: under the *immediate-execution model*, the system is automatically re-evaluated whenever an update is registered; under the *delayed-execution model*, the system accumulates all the updates and is re-evaluated only when the user requests it. The two execution models differ in the number of update events registered. Under the immediate-execution model, there is only one update event to process; under the delayed-execution model, there are sev-

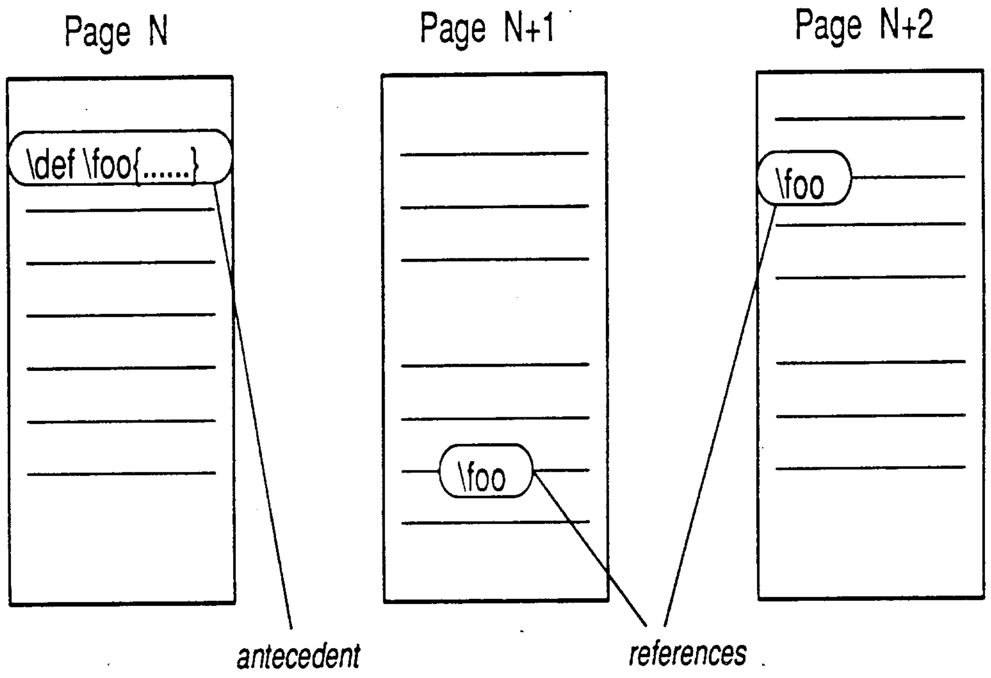


Figure 7.1: *Caller/callee name dependence.* This example shows the definition of a macro on page N and two separate invocations in the remaining context of the document. When the definition of the macro is changed, the references (invocations) become invalid and consequently their enclosing objects (e.g., pages $N+1$ and $N+2$) must be reprocessed.

eral. Clearly, the immediate-execution model is a simplified case of the delayed-execution model. The two models also differ in whether the user controls re-evaluation. Regardless of the execution model, however, the premise is that interactive editing is integrated with the main processing engine.

7.1.1 Dependence

When a system is modified, some data may be independent of the changes and hence need not be reprocessed. Detecting data dependence, therefore, establishes a condition for starting incremental processing and for skipping independent data during the processing. For example, if we assume that no global attribute management, such as cross references or table of contents, is involved, a change to page N normally has no effect on data between pages 1 and $N-1$. Hence reformatting can start from the state associated with page N, leaving those associated with previous pages intact. A finer grain algorithm can even detect data dependence at the paragraph level, resulting in even less reprocessing.

Objects directly affected by update events (i.e., those containing inserted or deleted data) are referred to as *directly-dependent data* with respect to changes made by the user. Those correlated with directly-dependent data are called *indirectly-dependent data*; there are two cases for the latter. In the first case they can be regarded as *ripples* originated by processing dependent data; the need to reprocess these indirectly-dependent data is a side effect of propagating changes. For instance, pluralizing a word changes some characters in that word. These characters are directly-dependent data. Since the word length changes, its enclosing line length changes accordingly, which potentially may change the height of its enclosing paragraph, which, in turn, may change the position of every paragraph in the rest of the document. We regard all these other changes as ripples.

In the second case, these indirectly-dependent data are essentially *references* to some *antecedents*, which are directly-dependent data or their ripples. There is a logical link between a reference and its antecedent. The enclosing object of a reference must always be reevaluated even if the ripples of reevaluating its antecedent do not go far enough to reach the reference.

There are two types of relationship between references and antecedents. The first is the *caller/callee* relationship based on *name dependence*, which is typified by the invocation and definition of a macro or procedure. Normally a reference (caller) here only appears in the post-context of an antecedent (callee) — *backward referencing* (see Figure 7.1). When the content of an antecedent is modified, all its references become invalid due to their dependence to it and their enclosing objects (e.g., the pages they appear on) must be reprocessed. The collection of all caller/callee dependencies forms a *call graph*, which has a similar counterpart in program compilation. In program development, a call graph is used for code optimization at compile time and the functions involved in such a call graph are not invoked until program run time. In document formatting, code generation requires these references be resolved dynamically and the call graph is actually used in the invocation of callers.

The second type of reference/antecedent relationship is *attribute dependence*. Here, it is the attributes associated with the antecedents involved, rather than the contents of these objects, that are correlated. A good example generally referred to as *cross referencing* can be found in document preparation, in which labeling an antecedent (e.g., a section or a figure) in a document and referencing it elsewhere is a common practice. References may occur either in the pre-context (forward referencing) or post-context (backward referencing)

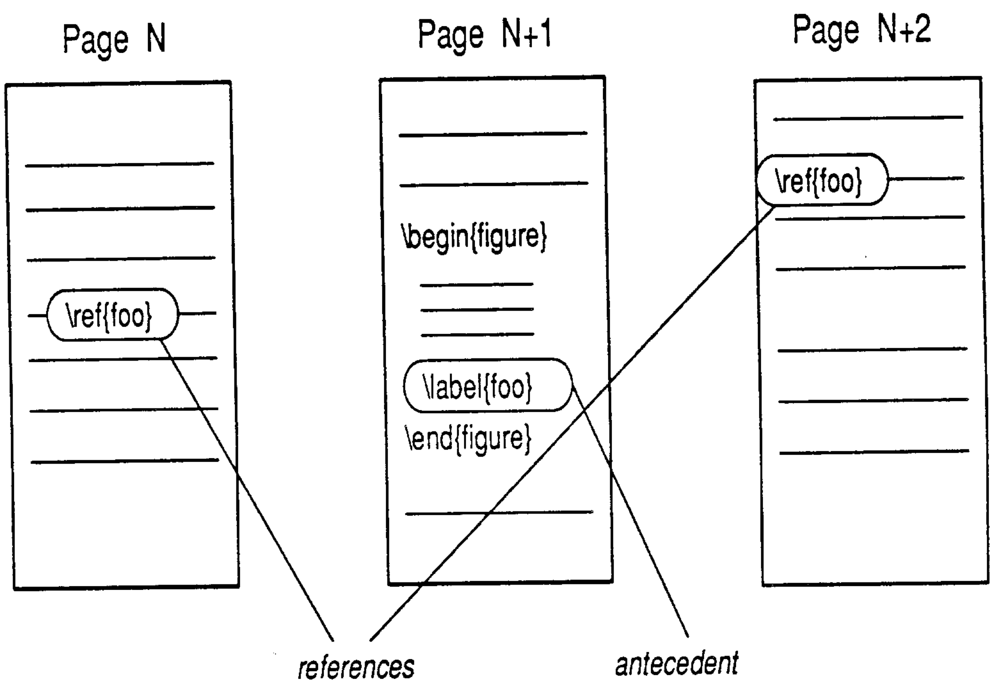


Figure 7.2: *Cross-referencing attribute dependence.* This is an example of cross referencing in a document. An antecedent is labeled somewhere and references can appear anywhere else. Usually it is not the content of an antecedent like this that matters. What matters is their ordering with respect to similar objects. When an antecedent changes its relative position in the document, multiple passes are needed to bring the references up to date.

of the antecedent (see Figure 7.2). When an object is added to or removed from a correlated group, the overall attribute relationship must be re-evaluated.

In the cross referencing example, the attribute dependence is built upon the antecedents' relative order of appearance in a document and sometimes the page numbers on which they appear. When the ordering changes, multiple processing passes are needed to resolve the references. This is because recomputing object ordering requires some processing and getting the antecedents' page numbers right also requires an additional pass of formatting (in fact, these can be merged into one pass). Substituting *actual references* for *symbolic references* must be done in a separate pass. Yet another pass of formatting is needed to generate the final result. The processing in each pass may be done incrementally in an integrated environment, but the passes can only take place sequentially. If nothing is changed after the last pass, a document is said to have converged (see Section 7.1.3 below).

If page numbers are involved in cross referencing, there may be pathological cases in which the last pass of formatting would alter some antecedents' relative positions (page numbers) produced by the previous formatting pass. In this case, this multipass effort to resolve cross references must start all over again, which goes on forever and never converges. Reference [29], which describes an algorithm for detecting document convergence, gives an interesting pathological scenario of an oscillating document.

There are specific techniques developed in artificial intelligence to solve data dependence problems [34]. Here, data dependence deals with logic assertions, which are used in reasoning. Normally no procedures are attached to these assertions. The structure of these logic assertions revolves around "if P then Q" predicates and are oriented toward consistency checking and truth maintenance rather than automatic updating and processing propagation. In other words, artificial intelligence techniques for solving data dependence problems, such as discriminate nets, are not directly applicable to this particular application domain.

7.1.2 Pertinence

An incremental system is able to do partial evaluation by processing only *pertinent data*. Non-pertinent data can be processed in the background or simply be delayed until the next processing cycle. The central issue here is to determine what information is pertinent and what is not. Suppose the processing in question is a sequence of events and there is a *focal point*, a place to which the user's attention is focused, then everything from the start of processing till the focal point is reached is pertinent.

In the document formatting example, data up to the page the user wants to examine are pertinent and everything beyond that page is non-pertinent[1]. Based on this simple heuristic, an incremental formatter can suspend its foreground processing when a desired page is encountered. The remainder of the document can be left unprocessed (dirty) until the next cycle. A more elaborate strategy would pass the processing into background for efficiency considerations.

A focal point is not fixed; it shifts back and forth throughout the whole session. There may even be multiple focal points active simultaneously in a windowing environment. The scope of a focal point may also enlarge or shrink as its viewing window resizes. For instance, different pages of a document can be displayed in different windows at the same

[1]This is not the whole story when attribute dependencies, such as table of contents or cross references, are involved. In such cases it may require additional passes before the document converges. See Section 7.1.3.

time. A window may reduces its size so that only a portion of a page is visible. For instance, systems like VORTEX allows different pages of a document to be displayed in different windows at the same time. Alternatively, an interleaf mode may be available that displays adjacent pages side by side in the same window. A good incremental strategy would take all these issues into account and intelligently decide what to process in foreground or background.

7.1.3 Quiescence and Convergence

The rippling process may ultimately subside when certain conditions are met. An incremental processing strategy may also reach *quiescence*, in which the system state is identical to that of the previous processing cycle. Subsequently, all independent data can be ignored and "real" processing does not have to resume until any dependent information is encountered again. If there are no dependent data in the remaining context and no cross reference resolution pending when quiescence is detected, the system is said to have reached *convergence*. At this point, no more processing, either in foreground or background, is necessary.

Under the immediate-execution model, the only piece of directly-dependent data is the most recent update made by the user. When there are no indirectly-dependent data other than ripples involved in a processing cycle, quiescence is synonymous with convergence. This is often the case for document editors like MacWrite and MicroSoft Word, which are direct-manipulation systems working under immediate-execution model. Whenever there is a change, formatting takes place immediately. There are no abstraction mechanisms available, thereby no caller/callee name dependence to maintain. They have no provisions for cross referencing, hence multipass reprocessing is not an issue either. It becomes more complicated when either of the two data dependencies is available. The immediate-execution model does not work properly when attribute dependence is involved.

Under the delayed-execution model, multiple update events may be registered before reprocessing. Quiescence and convergence are correlated but not necessarily synonymous. When multiple instances of dependent data are exist, many may have only local ripple effects. It is possible that multiple instances of quiescence may be exploited during the course of processing before final convergence is reached. The processing may be able to suspend itself, skip independent data, and resume upon encountering dependent data.

Incremental processing may reach the state of quiescence/convergence before crossing the boundary of pertinence/non-pertinence data and vice versa. In the document formatting example, a simple change to a paragraph may cause only local effects to the paragraph (e.g., inserting a word to a paragraph that does not increase the total number of lines in it). The remainder of the document should reach quiescence right after that particular paragraph is reformatted. In this case, pending pertinent pages are ignored and the focal point may advance to the designated page directly. On the other hand, if modifications are registered globally, it is very likely that processing would reach a focal point before quiescence is detected.

7.1.4 Checkpointing

How often should an interactive system checks for quiescence? In other words, when does it compare the newly generated data with that produced previously? Since *quiescence checkpointing* is a potentially expensive operation, the idea is to set checkpoints at appropriate locations so that the overhead of quiescence detection at a checkpoint is less than that of regular processing between checkpoints.

A related issue is *internal state checkpointing*, which saves a snapshot of the system state at each checkpoint. This is necessary for systems whose internal state is repeatedly destroyed by reprocessing. The checkpointed state information can be loaded in during some later cycle if the system, after suspending itself upon detecting quiescence, resumes processing from that point. Brute force checkpointing always saves the complete state, whereas incremental checkpointing only records the differences between previous checkpoints (called *deltas*). Some familiar space versus time tradeoffs are involved in the decision.

State checkpointing is absolutely mandatory in some cases and unnecessary in others. It all depends on how the internal state information is preserved during reprocessing. Document formatting or program code generation usually involves looking up global symbol tables. State checkpointing is necessary in these cases to ensure that the correct context can be inherited during incremental reprocessing. For example, in TEX (and in VORTEX) the same name can be used to define different macros in a document. To guarantee that reformatting starts with the correct context of name/body dictionary, it is necessary to periodically checkpoint the dictionary. Conversely, checkpointing is not required in the conventional *attribute grammar* approach for incremental program analysis [114], because

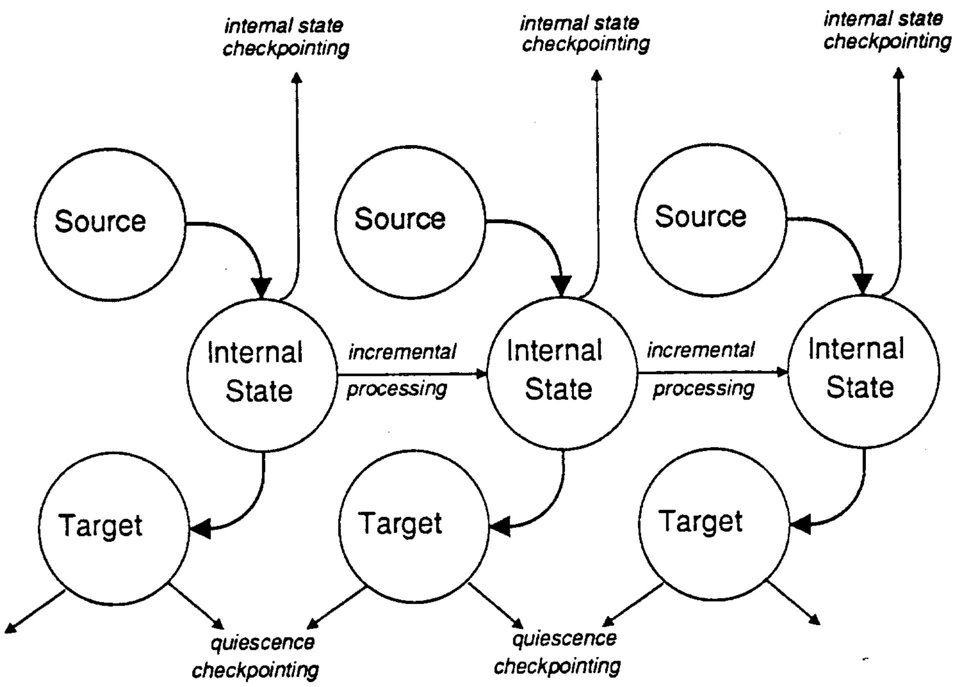


Figure 7.3: *Incremental processing and checkpointing.* This diagram illustrates the difference between quiescence checkpointing and internal state checkpointing. There are three checkpoints in the diagram. Quiescence checkpointing compares the newly generated data (target representation) with that produced previously. Internal state checkpointing saves a snapshot of the system state at each checkpoint.

attribute values are local to each production of the underlying context-free grammar; no global symbol tables are used in tracking dependencies.

Figure 7.3 illustrates the difference between the two types of checkpointing. Quiescence checkpointing involves comparing target representations produced by the present and previous computation cycles. Internal state checkpointing involves only the intermediate representation of the present cycle. In general, after every α state checkpoints, there is a quiescence checkpoint. Figure 7.3 shows an example of $\alpha = 1$.

Due to the nature of the processing, there may be constraints that prevent quiescence checkpoints from coinciding with state checkpoints (i.e., $\alpha > 1$). For instance, in a page-based incremental formatting strategy, the most sensible place to checkpoint the internal state is when the new target representation of a page is generated. Suppose the cost of quiescence checkpointing at this point is C_Q and that of processing a page and

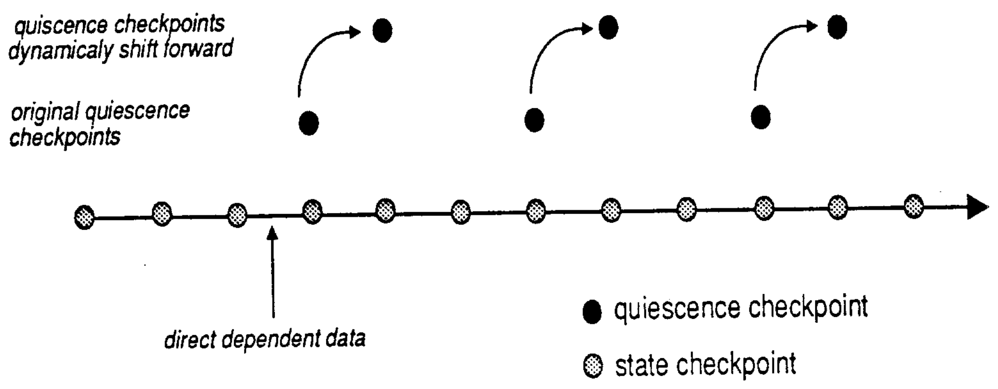

quiscence checkpoints dynamicaly shift forward
original quiescence checkpoints
direct dependent data
● quiescence checkpoint
◉ state checkpoint

Figure 7.4: *Dynamic shifting of quiescence checkpoints.* This diagram shows a relationship between state and quiescence checkpoints when $\alpha = 3$. State checkpoints are statically set. Quiescence checkpoints can be more dynamic. The original quiescence checkpoints are set at every fourth state checkpoints. When one such point is omitted, they all shift forward one place.

checkpointing the internal state are C_P and C_S, respectively. If

$$C_Q > (C_P + C_S),$$

then a per-page checkpoint is inappropriate for quiescence detection. The idea is to choose α, such that

$$C_Q < \alpha(C_P + C_S).$$

On the other hand, if

$$C_Q < (C_P + C_S)$$

to begin with, then the granularity of reprocessing may be refined.

Unlike state checkpoints, which are statically set, quiescence checkpoints may shift dynamically. There are simple heuristics to disable quiescence checkpointing at a potentially appropriate point. For example, there is no need to check for quiescence if there are any directly-dependent data between the current and preceding state checkpoints, in which case the current and succeeding potential quiescence checkpoints all shift forward one unit in the timeline of state checkpoints. This idea is illustrated in Figure 7.4.

Another overhead associated with incremental processing is the cost of loading the checkpointed state information. Evaluating this cost against that of regular processing determines whether or not suspending processing upon quiescence is worthwhile. Let C_L

be the cost of loading checkpointed state information, the requirement is

$$C_L < (C_P + C_S + C_Q/\alpha).$$

If this is not possible, then the necessary condition to suspend regular processing is that the next directly-dependent data is β units (state checkpoints) away, such that

$$C_L < \beta(C_P + C_S + C_Q/\alpha).$$

7.1.5 An Example

To further illustrate these ideas, consider a document processing scenario under a page-based incremental formatting strategy. Suppose this strategy operates under the delayed-execution model. For a six page document, assume the user first wants to view page 2 (initial focal point). Pertinent data include contents of pages 1 and 2. Initially, foreground formatting processes the first two pages, saving internal state at each page break. It reaches the focal point of page 2, displays it, and suspends itself. Background formatting, meanwhile, continues if there are no higher priority operations pending (in fact, background formatting is often the lowest priority operation). Suppose the focal point later shifts to page 5. If the page has already been processed by background formatting, all that is needed is to display the page; otherwise foreground formatting takes over and the cycle repeats itself.

Suppose later a global replacement happens that changes a word on page 2 and similarly on page 5 and that in both cases the total number of lines in their respective enclosing paragraphs is preserved. Data dependence checking shows that page 1 is independent of the replacements so formatting begins at page 2 by loading in the state information of page 1. Quiescence is detected after page 3 is produced due to the preservation of the paragraph's size integrity on page 2 and the absence of directly-dependent data on page 3. Processing resumes for page 5, which reaches the focal point and the page is redisplayed. Finally background formatting concludes convergence after page 6 (the last page) is generated.

Clearly this is a naive example. Real-world situations can be much more complicated. Detecting dependencies and quiescence are non-trivial tasks. There are also complications such as multiple focal points mentioned earlier or error handling and recovery problems that are not implicated in the examples above. The way VORTEX handles these problems is discussed in Section 7.3.

7.1.6 Augmentation

Incremental strategies can be exploited at many different levels. The so-called "processing" typically comprises several subtasks, each of which may be handled incrementally. Naturally, the ideal situation is one that fully exploits incremental strategies at all levels without being penalized by the extra overhead introduced from performing incremental processing. In practice, depending on the ultimate goal, the overall strategy is often a blending of incremental approaches at some levels and non-incremental ones at others.

For example, document formatting consists of parsing the source representation — if such a representation is present, establishing the semantic structure, and finally generating the target representation for display. Each of the three subtasks can be processed in an incremental fashion. An incremental formatting algorithm based on the high-level augmentation approach mentioned in the beginning of this chapter is incremental with respect to target representation generation (code generation). Between two adjacent state checkpoints, parsing and semantic analysis are non-incremental.

An augmentation approach like this devises incremental strategies based on non-incremental algorithms. The incremental behavior is achieved by putting high-level constructs on top of existing batch-oriented programs. An advantage of this work is that it injects the sensation of directness into the environment without having to create a totally a new system. The processing granularity of the underlying non-incremental strategy dictates that of the augmented incremental system. Since TEX generates code at every page break, a page-based incremental TEX formatter can exploit the technique of augmentation.

7.1.7 Summary

With the support of fine grain update events and persistent internal state information, an incremental strategy is one that exploits the ideas of dependence, pertinence, quiescence, and convergence under a specified execution model and carefully selected checkpoints. The short-term goal is always to reach focal points as efficiently as possible, while the ultimate goal is to arrive at convergence

Unfortunately, exploiting these ideas may sometimes introduce more overhead than simply performing regular processing. Compromise strategies would relinquish certain criteria and focus on the most cost-effective aspects. A more elaborate approach is to design adaptive algorithms that evaluate various costs in response to dynamic situations. The

```
top_level () {
    loop forever {
        get next event;
        process event;
    }
}
```

Figure 7.5: *A generic incremental formatter top level.*

dynamically shifting checkpoints discussed previously is just one of many possibilities.

In addition to devising incremental strategies from scratch, it is also possible to arrive at incremental behavior by augmenting existing non-incremental programs. This approach injects more directness into the system while retaining lower-level, high-quality algorithms. The remainder of this chapter is devoted to using this augmentation strategy on TeX.

7.2 Generic Issues

This section outlines some generic incremental formatting issues that are common to most integrated document development systems. Specifics of converting a non-incremental document formatter (TeX) to an incremental formatter — the VorTeX experience — are described in detail in the next section. A pidgin-C syntax is used to describe the algorithms.

7.2.1 Editor Events

The first problem an incremental document formatter must confront is interacting with editors. The separation of editors from the formatter is conceptual because, as mentioned before, in many direct-manipulation document preparation systems the two tasks are strongly integrated. For simplicity, assume update events from various editors are colored and automatically prioritized: colored because the formatter needs to know whether an event comes from the source editor or the target editor. Events must be prioritized because some are more urgent than others. A generic top level of an incremental formatter would be similar to Figure 7.5.

Events include *update* (insert/delete), *format* (or reformat), *display* (or redisplay), and miscellaneous requests such as communication protocol handling, internal state queries, and so on. In the immediate-execution mode, a pair of format and display events automatically follows each update event. In the delayed-execution mode, format and display requests are asynchronously generated by the user.

A preemption issue arises in the immediate-execution mode. Because the user is not in control of when to format, a newly arrived update event must be able to preempt current formatting. This makes sense from a user interface standpoint: when the user modifies the document, it is expected that the result will be immediately displayed. If current formatting cannot be preempted while processing the previous update event, the sensation of directness with respect to the new update is lost. An implication is that the granularity of reprocessing under the immediate-execution model should be relatively low, so that at the end of each unit, formatting can be preempted by a higher priority event.

Preemption is less of a problem in the delayed-execution mode because the user is in charge of initiating format requests. A user-driven asynchronous behavior like this has the advantage of having more tolerance for delay. Directness is not the paramount concern. Generally there will not be a second format request before the result of the current one is observed.

In addition to modifying directly-dependent data, it is also necessary for an update event to mark their enclosing objects as *dirty*. The scope of the enclosing object is determined by the granularity of reprocessing (i.e., state checkpoints). It is desirable to pack changes in consecutive contexts in one update event so that the marking overhead can be reduced.

If an update happens in the body of a macro/procedure, then the enclosing objects of all its callers must be marked dirty. If a macro/procedure definition is removed, the system symbol table must reflect that accordingly. If an update inserts or deletes a cross-referenced antecedent, a flag must be turned on notifying possible multipass processing. Whenever a reference is inserted or deleted, its antecedent's reference list must be updated correspondingly.

7.2.2 Incremental Formatter

The marking process facilitates dependence checking, which establishes a pre-

```
format () {
    do {
        if (suspend and current unit clean) {
            skip independent data;
            continue;
        } else {
            if (suspend) {
                suspend = false;
                load preceding context;
            }
            process current unit;
            mark current unit clean;
            state checkpointing;
        }

        if (quiescence checking needed) {
            check for quiescence;
            if (quiescence detected)
                suspend = true;
        }
    } (focal point reached);

    display focal point;
}
```

Figure 7.6: *A generic incremental formatting algorithm.* The flag *suspend* is used to mark quiescence. Its initial value is **true**.

condition of the general incremental formatting algorithm. Another pre-condition is that there is at least one user viewport with a designated focal point. Figure 7.6 describes a generic incremental formatting algorithm. With default value **true**, the global variable *suspend* is a flag that marks quiescence status during the processing. The same algorithm works under both cold and warm starts. In the case of cold start, every object is considered dirty. Objects already processed are marked clean by the formatter. As the editing session progresses, some objects are marked dirty again by update events, until they are cleared by the next cycle of formatting.

The main loop terminates when the focal point is reached, at which time the visible part of the document covered by the focal point is displayed on the user's viewport. It is simplified because only one focal point is assumed. The focal point is implicitly set at infinity in the case of background formatting. Thus the same algorithm applies to both foreground and background formatting. The granularity of reprocessing is unspecified, which means it can be page-based, paragraph-based, or based on some finer unit.

7.2.3 Tracking Convergence – The Multipass Problem

To keep track of document convergence, the dependence marking routine must take special care of any changes to an antecedent. One simple way of managing the caller/callee name dependence is to maintain a hash table for all the names of the callees, in which each entry points to a linked list of pointers to its references. Since a reference can only appear after the antecedent in this case, when a caller is identified as having been inserted or deleted, its callee should already exist in the symbol table. Therefore, when the antecedent is modified, it is straightforward to mark all its references dirty.

Managing attribute dependencies is more involved because multiple processing passes are usually required to resolve them. In the general case, as many as four passes appear to be needed to complete the job. Normally the first pass collects all the antecedents, which are specified symbolically in the document source, and puts them in some auxiliary file. The second pass processes these terms and establishes the actual attribute relationship. The third pass resolves the dependencies by substituting actual references for the original symbolic ones. Finally, the last pass reformats the entire document, in which actual references as well as any new material introduced by various processing tasks are incorporated. Sometimes the first and third passes can be accomplished as byproducts of formatting. The second pass, however, may be too complex to be embedded in the formatter and thus may be handled by some external stand-alone processors.

Attribute Dependencies

There are three common attribute dependencies in a document: *citations*, *cross references*, and *indexes*. Bibliographical citations are references to a certain combination of bibliographical entries listed in a special section of a document. The first pass collects all the citation terms specified symbolically in the document source. The second pass, which relies explicitly on a bibliography processor, extracts the corresponding entries of these terms from bibliography databases and sorts them according to a specific ordering. The third pass substitutes actual references for the symbolic ones. Finally, the last pass reformats the entire document that by now includes the newly placed actual references and the bibliography itself. Ideally, antecedents are generated by the bibliography processor and are placed in a special section (bibliography) appended at the end of a document. References (citations) in this case appear only in the pre-context of antecedents, which do not exist in the original document.

As mentioned above, the first and third passes of bibliography processing may be embedded as subtasks of formatting. In systems like Scribe and LaTeX, those two passes are handled precisely by the formatting engine. Establishing the desired attribute relationship, which in this case is arranging bibliography entries according to some predefined convention, does not depend on formatting per se. When an editor is tightly-coupled with a document processing environment, an alternative is to let the editor do much of the work. VORTEX has a preprocessing subsystem that takes this approach. The advantage of doing so is that bibliography processing becomes strictly a preprocessing task relative to formatting. The extra overhead of invoking the formatter in the first and third passes can then be avoided.

The second type of attribute dependence is cross referencing. Unlike citations, antecedents in this category are explicitly specified in the document source; references may appear in either the pre- or post-context of an antecedent. The attribute relationship to be established is the antecedents' physical order of appearance in a document. There may be several classes of antecedents of interest: for instance, one for all the figures, a second for all the tables, and a third for the main text in terms of chapters, sections, subsections, etc. Lists of figures/tables and the table of contents, which normally appear in the beginning of a document, may be regarded as artifacts of maintaining all those cross references (although they can still be produced without the presence of cross references).

Establishing the attribute relationship in cross referencing is rather straightforward (only simple counting is involved) so that an external stand-alone processor for pass 2 is usually unnecessary. Furthermore, as long as there are no page numbers involved in the actual reference string, the work can easily be accomplished by a programmable text editor like Emacs. What is needed is only sequential retrieval and counting of all antecedents. Resolving cross references, therefore, can also be realized as a pure preprocessing task if no page numbers are involved. When page numbers do get involved, however, cross references cannot be resolved without having the first pass being a formatting pass that computes the necessary page numbers.

Indexing is another common type of attribute dependence. An index is a collection of keywords that point to certain important concepts in a document. The links between a keyword and the concepts it points to are some numbers (usually page numbers), which inform the reader that related material can be found in the contexts those numbers designate (usually certain pages). Like bibliographical citations, index terms are placed in the document by the user, and like the bibliography, the index section is usually generated by some automatic processor. Placing index commands in a document is an important but tedious task. Reference [40] discusses a systematic approach that greatly facilitates this effort. Unlike a bibliography, which is a list of antecedents, an index is a list of references. The relationship between the set of references and that of antecedents is many-to-one in citationing and cross referencing, but is one-to-many in indexing.

To produce and format an index, the first pass must be handled by the formatter that collects each index term, together with a number. The object designated by the number, which is typically a page, contains the indexed concept. The second pass sorts and merges these raw index terms and produces an output index according to a specific style. The issues involved in such an index processor are quite complex, two different approaches can be found in References [18,40]. The next pass of actual/symbolic transformation is unnecessary because index commands simply disappear in the final formatted document. Finally, an additional pass of formatting is used to process the newly generated index section.

Inherent Conflicts

There are some inherent conflicts between resolving attribute dependencies and the incremental processing model. Resolving attribute dependencies requires multipass processing. The first issue is that updating antecedents, especially when page numbers are involved in the reference, may cause global consequences. For instance, whenever the same object appears on a different page, every reference pointing to this object must be updated accordingly. The ripple effect of this is quite significant because when an object appears on a different page every other object in its post-context may potentially get shifted. So every reference to any antecedent in those places must be updated unless quiescence is reached at some point. This is why it is a good idea to put off attribute dependence resolution, especially when indexes are involved, until the main document body is near completion.

The second issue concerns external stand-alone processors often required to handle the second pass of processing. It is possible to mush these special-purpose processors with the main formatting engine. One approach is to incorporate them on top of the formatter. Most systems avoid doing so because the tasks involved may be so complex that it is inconvenient to have them programmed in the formatting language. Blending them at a lower level does not make sense either, because not every document has attribute dependencies to resolve. Simple documents like memos, letters, short articles, etc., which people prepare most of the time, usually do not contain bibliographical citations and cross references, and definitely need no indexes. Asking all users to pay a price for something rarely used is a bad practice. This leads to the question of whether these external processors are themselves incremental, and if so, how does the formatter exchange state information with them. Making cooperating processes mutually incremental is nontrivial and deserves further research.

Another conflict is a user interface issue coupled with the semantics of these references. Suppose attribute dependencies have been resolved, what happens when the user attempts to massage the actual references. For instance, after resolving bibliographical citations, what end up in the final document are some actual reference strings in place of the original symbolic ones. If the use were allowed to modify these data, as some direct-manipulation systems would, the semantic correctness of this reference can be violated. This underscores the importance of the multiple representation paradigm for document development. In a system like VORTEX, symbolic references exist in the source representation while actual references appear only in the target representation. Massaging actual reference strings is forbidden to avoid the semantic confusions it may cause. Modifying the original symbolic references is still allowed, which triggers reformatting through normal channels.

Reaching Convergence

Since the multipass nature of resolving attribute dependencies does not promote a fine-grain incremental strategy, the best solution seems to be to still rely on external stand-alone processors for the tasks required in the second pass. These external processors should be integrated with the environment so that the input they need and the output they produce can be exchanged easily with the editor and the main document formatter. A tightly-coupled document editor can be used to share much of the preprocessing work if certain programmability is supported. It is also important to keep track of changes to any antecedents so that either the system knows additional passes of processing are needed and therefore automatically spawns the jobs, or the user is informed of the situation and jobs can be spawned asynchronously.

7.3 A Practical Application: VORTEX

This section describes various incremental strategies under the VORTEX model of document processing. Here, a page-based approach is assumed; issues on generalizing this scheme to finer granularity are discussed later. The algorithms discussed below do not cover convergence detection. A convergence detection algorithm can be found in Reference [29], which can be implemented as an extension to VORTEX in the future. The algorithm determines when to trigger additional passes of processing. Within each pass, the incremental algorithms described below are still applicable.

The VORTEX incremental formatter maintains full compatibility with TEX. Its implementation was derived from the original TEX source code [89]. We did not find it necessary to delete code from the non-incremental version of TEX, except that some TEX routines had to be split into a few smaller subroutines to comply with the VORTEX model of processing. Routines to maintain the VORTEX-specific internal representation (IR_S, IR_O, and IR_T) and those for realizing various incremental strategies are embedded as conditional statements. In the current C implementation of the incremental formatter, it looks like

```
#ifdef VORTEX
    ......
#endif
```

The sole purpose of doing so is to ensure TEX compatibility. In particular, the code responsible for maintaining the internal representation is buried literally everywhere in the innards of TEX. Incremental processing routines, on the other hand, are not so tightly knitted with the original TEX code. Nonetheless, some key TEX routines described in Reference [89] are invoked inside the incremental processing engine. Two TEX modules are of particular importance to the incremental strategies:

- *main_control ()*: TEX's chief executive that brings all the pieces together, including, among other things, reading tokens from source files (or IR_S in this case), expanding them, building the corresponding semantic structure, and generating the target code.

- *ship_out ()*: TEX's code generator, which is invoked by *main_control ()* when the number of lines reaches a certain threshold.

Furthermore, the following global flags and counters must be identified in order to explain the incremental strategies:

- *last_page* (default ∞): a non-negative integer indicator that is always ∞ unless the end of document has been processed, in which case it is assigned the last physical page number of the document.
- *total_pages* (default 0): a non-negative integer counter that always reflects the current physical count of formatted pages; it also indicates the most recently processed page.
- *starting_page* (default ∞): a non-negative integer indicator that always points to the page to be formatted.
- *viewing_page* (default 1): a non-negative integer indicator that indicates the current focal point designated through the target editor by the user to determine the foreground/background processing boundary.
- *format_suspended* (default **true**): a binary flag reflecting whether formatting is in quiescence; if so, independent data can be ignored.
- *page_shipped* (default **false**): a binary flag that is set **true** whenever the target representation of a page is generated; it is set **false** at the beginning of processing the following page.
- *page_done* (default **false**): a binary flag that is set **true** when the remaining context of a page in its source representation has been consumed after its code generation; it is set **false** at the beginning of processing the following page.

The flag *total_pages* is borrowed from TEX itself, whereas all other flags and counters are introduced to support incremental processing.

The TEX routine *main_control ()* is a long loop in which a master switch statement controls execution. In the batch version, this loop does not terminate until the end of document. The way TEX generates code makes it possible to escape gracefully from this long loop at the end of each page, which sets the stage for incremental formatting.

At the end of *ship_out ()*, which generates the target representation for each page, the counter *total_pages* is incremented by 1 and the flag *page_shipped* is turned on. This notifies *main_control ()*, which may still need to clean up some remaining syntactic tokens in that same page before it can really call it "a page", that the code for that page has been generated. Once the cleanup is finished, the flag *page_done* will be set, which releases the exit guard of the long loop and control returns to the top level of the incremental processing engine.

The following are some key routines that constitute VORTEX's incremental formatting engine.

- *top_level ()*: main event dispatcher; it is an infinite loop with a switch that delegates requests from both source and target editors to the corresponding routines.
- *send_page ()*: the routine that transmits page information to the target editor for display.
- *fg_format ()*: foreground formatting routine, which keeps on processing pertinent dependent data until *viewing_page* is encountered.
- *bg_format ()*: background formatting routine, which processes one page at a time.
- *save_state ()*: internal state checkpointing routine, which is invoked after a page is generated.
- *load_state ()*: the routine that restores a page's internal state before processing.
- *compare_page ()*: quiescence checkpointing routine, which returns **true** if the newly generated target page is identical to its predecessor.

7.3.1 Top Level

Figure 7.7 illustrates the top level of VORTEX's incremental formatting engine. Its body is an infinite loop that receives events from either the source or target editors. If no events have arrived before the receiver is timed out, the background formatting routine *bg_format ()* is invoked to process more data while waiting for an event. This assumes that foreground formatting has taken place already, in which case *starting_page* should have been

```
top_level ($\Theta$) {
      while (true) {
            select socket;
            if (timed out and (starting_page $\neq \infty$))
                  bg_format ();
            else {
                  process event;
                  if (event $\equiv \Theta$)
                        return;
            }
      }
}
```

Figure 7.7: *Top-level control loop of VORTEX's incremental formatter.*

assigned some value other than ∞. Otherwise the event is dispatched and the corresponding event handling routine is invoked. This top level serves dual purposes. On the one hand it drives the whole system, in which case the actual parameter for Θ is **nil** and the **while** loop goes on forever unless the event is an abort.

On the other hand, it can be used as a synchronous event processor that breaks out of the loop upon receiving the designated event Θ. This option is especially useful under VORTEX's processing model when certain synchronization between the formatter and either of the two editors is required (by default their communication is asynchronous). A good example is handling TEX's file inclusion. In VORTEX, the formatter and the two editors operate on disjoint address spaces and potentially distinct file spaces (due to its distributed nature). When the formatter realizes that an external file is included in the document being processed, *top_level ()* can be invoked immediately after the request `FS_InputFile` is sent to the source editor, which owns the external file. This second-order top level is identical to the "official" top level, except when the incoming event is `SF_OpenFile`, which is what Θ is bound to in this case, it returns to the point where the included file is supposed to be processed. Other synchronous operations may take advantage of this alternative as well.

7.3.2 Displaying

The target editor constantly sends `TF_SendPage` to the formatter requesting a cer-

```
send_page (N) {
    viewing_page = N;

    if (fg_format () returns an error) {
        if (the error indicates page not found)
            send (FT_PageNotFound);
        return (error);
    }

    if (viewing_page already sent)
        send (FT_PageOkay);
    else {
        flatten_page (N);
        send (FT_PageInfo, flattened page info);
    }
}
```

Figure 7.8: *Displaying a page.* This routine finds a designated page and sends it to the target editor for display.

tain page to be displayed. Figure 7.8 shows the formatter routine *send_page (N)* that takes a page number N supplied by the target editor and binds *viewing_page* to it. Foreground formatter *fg_format ()* (see below) is always invoked first to bring the viewing page up to date. If *fg_format ()* returns an error, which indicates that the focal page cannot be found, the target editor is notified by FT_PageNotFound. When the viewing page is clean, it may follow one of two cases. In one, the same page has already been sent to the target editor, which is notified by FT_PageOkay that the page it has is still good, saving significant transmission bandwidth. In the second case, either the target editor has not asked for this particular page before, or the target representation of the page has changed, hence the complete page information is flattened by *flatten_page (N)* (from a tree representation to a stream representation) and the target editor is notified by FT_PageInfo, together with the actual data.

7.3.3 Simple Formatting

The next key routine is the foreground formatter *fg_format ()*, as shown in Fig-

```
fg_format () {
    if  ($starting\_page \equiv \infty$) {
        /* cold start */
        pre_format ();
        starting_page = 1;
        save_state (0);
    } else if  (($starting\_page \leq viewing\_page$) and
            ($starting\_page \neq total\_pages + 1$)) {
        /* warm start */
        load_state ($starting\_page - 1$);
        last_page = $\infty$;
    } else if  ($starting\_page > viewing\_page$)
        return (no need to format);
    else if  ($last\_page \neq \infty$)
        return (nothing to format);

    while  (($total\_pages < viewing\_page$) and ($last\_page \equiv \infty$)) {
        if  (catch (error raised))
            return (error code);
        main_control ();
        save_state (total_pages);
        starting_page++;
    }

    if  ($total\_pages \equiv viewing\_page$)
        return(success);
    else
        return (nothing to format);
}
```

Figure 7.9: *VORTEX's foreground formatting routine, the simplified version.* This version does not support for quiescence checkpointing, so everything following the leftmost dependent data is presumed dirty.

ure 7.9. For the purpose of illustrating the ideas, this routine is simplified in that it does not support quiescence checkpointing. A more complete version that supports quiescence checkpointing is shown in Figure 7.11 and described in Section 7.3.4. The simplified algorithm of Figure 7.9 bypasses only the initial independent data and starts formatting from *starting_page*. Once formatting has started, there will be no suspension until the focal point (*viewing_page*) is reached.

Recall that *starting_page* is set spontaneously by every update event so that it always points to the first dirty page. Initially, it has the default value of ∞ and is immediately reset to 1 after a cold start. The routine *pre_format ()* performs the necessary initializations. For any incremental run, the pre-context of *starting_page* is loaded prior to the actual processing and *last_page* is reset to ∞ if the first dirty page is not located beyond the focal point (*starting_page* $\leq$ *viewing_page*) and the page to be processed is not the immediate successor of the most recently processed page (*starting_page* $\neq$ *total_pages* + 1), in which case the context in memory is the one to inherit. If the first dirty page is indeed located beyond the focal point (*starting_page* $>$ *viewing_page*), there is no need to format the document in the foreground (i.e., the viewing page is clean). There may also be nothing left to format, in which case *last_page* is assigned some value other than ∞.

The main loop invokes the TEX chief executive *main_control ()* to (1) format one page at a time, (2) perform state checkpointing afterwards by calling *save_state ()*, and (3) advance *starting_page*, until either the viewing page is reached (*total_pages* $\equiv$ *viewing_page*) or the end of document is encountered first (*last_page* is assigned some value other than ∞). There is a routine **catch** that handles errors and exceptions. It sets an environment pointer to which an error situation can return. In UNIX/C, the system call `setjmp` can be used to catch the error; `longjmp` can be used to throw the error. In Lisp, a pair of `catch` and `throw` is the obvious choice.

Background formatting is almost identical to foreground formatting, but the logic is less complex. Figure 7.10 shows a simplified version of *bg_format ()*, which, like its counterpart in the foreground, does not support quiescence checkpointing. Background formatting is precluded from happening when the document has reached its end while every page in the document is clean (*last_page* is assigned some value other than ∞). There is no such notion as the current focal point in background formatting; the system always formats the rest of the document in the background. Again, the pre-context of *starting_page* is loaded prior to the actual processing, provided the page to be processed is not the immediate

```
bg_format () {
        if (last_page ≠ ∞
                return (nothing to format);
        if (starting_page ≠ total_pages + 1)
                load_state (starting_page −1);
        last_page = ∞;

        if (catch (error raised))
                return (error code);
        main_control ();
        save_state (total_pages);
        starting_page++;
}
```

Figure 7.10: *VORTEX's background formatting routine, the simplified version.* This version is similar to foreground formatting, except that only one page is processed at a time and that reaching the focal point (*viewing_page*) is not a terminating condition here.

successor of the most recently processed page ($starting\_page \neq total\_pages + 1$), otherwise the necessary context can simply be inherited from the most recently processed page.

This simplified suite of foreground and background formatting routines has an important implication in that dependence marking and detection are rather straightforward. Since the granularity here is a page, whenever something is touched on page N, all pages from N to the end of document are presumed dirty. In fact, since there is no quiescence checkpointing, it is immaterial which pages are clean or dirty once the page containing the first directly-dependent data is determined. Tracking indirectly-dependent data is not an issue either due to the same simplicity. More elaborate approaches are discussed below.

7.3.4 Quiescence Considerations

An obvious problem with the simplified version of *fg_format ()* is the absence of quiescence checkpointing. For instance, suppose the focal point is on page N. A simple change to page M ($M < N$), which preserves its paragraph dimension integrity, would still make it necessary to process everything between pages M and N. When the $N - M$ gap is large, the overhead is tremendous. The enhanced version of the routine shown in Figure 7.11 is intended to remedy this.

```
fg_format () {
      if (starting_page ≡ ∞) {
            /* cold start */
            pre_format ();
            starting_page = 1;
            save_state (0);
      } else if ((starting_page ≤ viewing_page) and
                  (starting_page ≠ total_pages + 1)) {
            /* warm start */
            load_state (starting_page −1);
            last_page = ∞;
      } else if (starting_page > viewing_page)
            return (no need to format);
      else if (last_page ≠ ∞)
            return (nothing to format);

      while (true) {
            if (catch (error raised))
                  return (error code);
            main_control ();
            save_state (total_pages);
            starting_page++;
            if (total_pages ≡ viewing_page)
                  return (success);
            if (last_page ≠ ∞)
                  return (error: no such page);

            format_suspended = compare_page ();
            if (format_suspended) {
                  while (starting_page < viewing_page and starting_page is clean)
                        starting_page++;
                  if (starting_page is clean)
                        return (success);
                  if (starting_page ≠ total_pages + 1) {
                        load_state (starting_page −1);
                        format_suspended = false;
                  }
            }
      }
}
```

Figure 7.11: *VORTEX's foreground formatting routine, the enhanced version.* This version checks for quiescence, skips independent data upon reaching quiescence, and resumes processing upon encountering dependent data.

```
bg_format () {
    if (format_suspended)
        while (starting_page is clean)
            starting_page++;
    if (starting_page $\equiv$ last_page + 1)
        return (nothing to format);
    if (starting_page $\neq$ total_pages + 1) {
        load_state (starting_page −1);
    }

    if (catch (error raised))
        return (error code);
    main_control ();
    save_state (total_pages);
    starting_page++;

    format_suspended = compare_page ();
}
```

Figure 7.12: *VORTEX's background formatting routine, the enhanced version.* This version is a variation of it foreground counterpart in that it relies on *fg_format ()* to tell it if formatting is in suspension or not.

Figure 7.11 is essentially a merge of Figures 7.6 and 7.9. The logic to establish the initial condition for reformatting is identical in both versions. The infinite **while** loop in Figure 7.11 is a variant of the generic incremental formatter of Figure 7.6. Here, quiescence is reached when *compare_page ()* returns **true**, at which point formatting is suspended (*format_suspended* is **true**) and all independent data (clean pages) are ignored. Formatting resumes upon encountering the first instance of dependent data (dirty page).

The routine *compare_page ()* compares the newly generated target page with what exists in IR_T. It returns **true** if the two are identical, otherwise the old page is replaced by the new one and **false** is returned. It is relatively easy to determine non-quiescence; starting from the largest box in the page, the moment any property of a new box, which may be its content, attribute, dimension, etc., does not match its counterpart in the old page, *compare_page ()* can immediately return **false**. As in the simplified case, foreground formatting terminates when either the focal point (*viewing_page*) has been reached or the whole document has been processed (*last_page* is assigned some value other than ∞).

By the same token, background formatting can be enhanced to exploit quiescence checkpointing. One possible enhancement is illustrated in Figure 7.12. Basically, if formatting is suspended (*format_suspended* is **true**), all clean pages are ignored until either a dirty page is found, or the chain of target pages in IR_T has come to an end. Recall that a value other than ∞ is assigned to *last_page* only if the end of document has been processed. If the page to be processed (*starting_page*) is the immediate successor of *last_page*, no further processing is necessary. The rest of this enhanced *bg_format ()* is similar to that of the simplified version (see Figure 7.10), except at the end the enhanced version invokes *compare_page ()* to check for quiescence.

7.3.5 Cost Analysis

There is no dynamic analysis performed by the enhanced pair of foreground and background formatting routines discussed above, which means both strategies are non-adaptive. However, supporting dynamically shifting quiescence checkpoints like the example shown in Figure 7.4 is not difficult. What is needed, before the checkpointing routine *compare_page ()* is invoked, is to ensure that the page just processed is clean prior to processing (i.e., it contains no dependent data but ripples). Of course, more elaborate adaptive approaches can be considered.

In the strategies discussed above, the following inequalities hold:

$$\begin{aligned} C_Q &< (C_P + C_S), \\ C_L &< (C_P + C_S + C_Q), \end{aligned}$$

where C_P, C_Q, C_S, and C_L, are the costs of processing a page, quiescence checkpointing, state checkpointing, and state loading, respectively. Both C_S and C_L take constant time using crude state checkpointing, because the number of state variables that must be saved and restored is a constant. The current VORTEX implementation even optimizes it by putting all state variables in consecutive memory locations so that a save or restore operation can be accomplished in one memory write or read.

The cost of quiescence checkpointing, C_Q, is proportional to the number of character boxes on a page, which is roughly the same as the number of characters in the original document, plus an extra 30% of overhead for word, paragraph, and page boxes. In any case, its time complexity is a constant. The bottleneck in processing a page is in breaking paragraphs into lines, or the line breaking algorithm. The complexity is approximately proportional to the number of words in a paragraph (p) times the average number of words in a line (l), which is an $O(n^2)$ strategy, where n is the number of characters on a page [90].

The number of words in a paragraph is a fraction of the number of characters on a page, roughly $p = n/c_{word}$, where c_{word} is 5, the average English word length. This would be the worst case when there is only one paragraph on a page. In the best case, c_{word} is 5 times the maximum number of paragraphs on a page. Similarly, the average number of words in a line is a fraction of the number of words in a paragraph, $l = p/c_{line}$, where c_{line} is the average number of lines in a paragraph and may vary slightly depending on the font size used in a document. For instance, in this thesis where the font size is 11-point, c_{line} is 10. For convenience, both α and β (introduced in Section 7.1.4) are fixed at 1 in the algorithms discussed above, which means further refinements are possible.

7.3.6 Refinements

The algorithms of Figures 7.11 and 7.12 are oriented toward incremental code generation. It is natural that the granularity be a page because TEX generates code on a per-page basis. There are a number possible refinements to this page-based scheme.

The first refinement is to lower the granularity to a paragraph. Anything finer than a paragraph would be difficult if the original TEX algorithms are used, because line breaking involves every word in a paragraph (the reason is to achieve more even interword spacing within a paragraph). A paragraph-based strategy implies that code must be generated on a per-paragraph basis, which is rather straightforward even under our approach (see Section 7.1.6). In fact, the strategies illustrated in Figures 7.11 and 7.12 still apply; the only modification needed is to replace page considerations by paragraph considerations. The extra overhead introduced is in the storage required for state checkpointing if the brute force approach is used.

The next refinement is to perform incremental state checkpointing. This is not restricted to paragraph-based incremental formatting per se; the original page-based approach can also take advantage of this if external storage is a scarce resource. The idea is to save only the deltas. Both save and restore would take longer: *save_state ()* must know which state variables are touched and which are intact, thereby saving only the touched ones; *load_state ()* would have to traverse the delta tree to recover the complete state information before restoring it. Source code control systems like SCCS [116] and RCS [132] have devised important delta checkpointing techniques that can be applied here.

In the prototype VORTEX implementation where a brute force approach is used, the checkpointed state information for each page is approximately an order of magnitude larger than the page's off-line target representation (i.e., TEX's device independent representation, or DVI [53]). A compressed format cuts it down substantially to only a factor of 2, instead of the original 10. For a target page of size n, the checkpointed state information would occupy $2n$ storage. These files are temporary because when the VORTEX session is over, they are automatically removed from the file system. The longest target page size for an 11-point document is about 6,000 bytes. For a 100-page document, it would require about 6 megabytes of temporary storage for state information. A brute force approach is perhaps acceptable for a prototype implementation. For production software, however, an incremental checkpointing scheme is vital.

Chapter 8

System Integration

A number of issues regarding integration must be considered in a complex system like VORTEX. For example, because the system embodies multiple representations, the processors responsible for manipulating these representations are integrated by means of sharing a common intermediate representation or maintaining certain inter-representation mappings, or both (see Section 5.8). Within each representation, multiple objects must be integrated. Here, the notion of *compound document* and the concept of *document style* are especially important. Objects in a document are instantiations of some generic document data types, such as text, tables, figures, mathematical formulas, etc. The issue here is to arrive at a coherent treatment of different objects in terms of their manipulation and presentation. A document style defines a set of rules wherein objects are structured in a logically sensible way [79]. The question: how to support a document's logical structure for systems like VORTEX, in which no such notion is assumed *a priori*.

At a higher level, a complete document development system must be able to cover the multi-dimensional task domain mentioned in Section 3.1, or at least a substantial subset of it. Many direct-manipulation document preparation systems concentrate their efforts in integrating object editing with object evaluation (e.g. formatting and imaging), while pre- and post-processing tasks are treated as second-class citizens. Providing an integrated support for these tasks is crucial, because, for example, an effective mechanism for citations and cross references, from the standpoints of both link construction (writing) and link navigation (reading), is necessary for extension to more elaborate systems, such as hypertext or active documents. Relying on unintegrated off-line processors to do the job is unacceptable under interactive environments.

At a lower level, integration concerns the engineering aspects of a system, and, in particular, the underlying platform upon which the system runs. It touchs issues revolving operating systems, window systems, and programming languages. For instance, control flow being single-threaded or multi-threaded (i.e., based on local procedure calls under a single address space, or as separate processes in disjoint address spaces) is dictated by the process management paradigm of the underlying operating system. Similarly, whether the internal representations are shared in the real sense, or simply replicated across processes is influenced by the operating system's memory management facility. The choice of a window system is related. If a window system is network-based, it is possible to create an interactive environment based on a distributed framework, which introduces a whole range of problems not encountered in applications involving only a single site. Last, but not least, the gluing mechanism that ties all the pieces together is driven by the source languages used in carrying out such tasks as editing, formatting, graphics specification, etc. Programming language features, such as the abstraction mechanism (macro-based versus procedure-based) or the evaluation scheme (compiled versus interpreted), are directly correlated to a system's extensibility and versatility.

To summarize, at least four layers of integration must be realized by an interactive document development system:

1. system organization,
2. multiple representations and inter-representation transformations,
3. compound objects and document styles,
4. multi-dimensional task domain involving external processors and associated interactive activities.

VORTEX's system organization is based on the principal trio that runs under a distributed framework as multiple corporating processes. Integration of layer 1 has been covered in Section 6.2. In layer 2, multiple representations in VORTEX are tightly coupled through a shared internal representation (Section 6.5). Inter-representation transformations in both backward and forward directions (Sections 6.7 and 7.3) tightens the integration even more effectively. This chapter focuses on the principles behind the other two layers of integration in VORTEX.

8.1 Integration Mechanisms

Document-processing systems employ a wide spectrum of integration mechanisms — *strong integration* at one end and *weak integration* at the other. In a typical strongly integrated system, a single thread of control is employed, logical document structure is known *a priori*, compound objects are edited and presented on a single viewing surface, and there is a uniform, well-defined data transfer format for various processors to exchange information. In the extreme case of weak integration, the overall system consists of independent processors, each of which handles a specific class of objects. As a result, the editing and presentation of objects occur on multiple viewing surfaces. There may be multiple data transfer formats, or a single format known as a "vanilla" character stream whose semantics are subject to interpretation by individual processors.

8.1.1 Strong versus Weak Integration

Strong integration can be found in most "multimedia" document editors, exemplified by Andrew's [100] base editor EZ [107], the BBN Diamond editor [131], and the IBM Quill editor [32]. A system of this type is generally designed to incorporate compound objects in a single editing and viewing surface. It is usually a monolithic program with a top-level shell that dispatches events to subordinate routines in charge of a particular class of objects and coordinates image rendering requested by the subordinate routines on the display surface. Because the system is so tightly knitted, a user-level glue language is sometimes considered unnecessary. However, the absence of this extra level of programmability implies that every class of objects and every component in the task domain must be built into the system. This is rather restrictive and may undermine a system's extensibility.

The most famous weak integration mechanism is the *pipelining model*, a la UNIX. Standard UNIX document processing programs like *troff*, *tbl*, *eqn*, *pic*, et al collectively process a document `foo` by passing component data files from one to another through pipes, e.g.,

```
pic foo | eqn | tbl | troff -ms
```

where '`|`' establishes a pipe and `-ms` invokes a high-level macro package. Each program handles only the data it understands, passing any alien data through. The gluing mechanism consists of a host of tools, such as *sh* and *csh* scripts as the command language, *awk* [3] for pattern matching, *sed* [96] for non-interactive stream editing, *make* [51] for dependency control, and so on.

This model is ideal for rapid prototyping because it is very easy for a new processor to be integrated with existing ones. Over the years, the suite of standard UNIX document processing programs has grown substantially. New tools developed at AT&T Bell Laboratories and elsewhere, such as *chem* (chemical structure diagram processor) [17], *make.index* (index processor) [18], *vtbl* (interactive WYSIWYG table editor) [102], *grap* (graph typesetting program) [19], *gremlin* (interactive graphics editor) [105], *pico* (bitmap picture editor) [73], *drag* (graph drawing system) [133], and more, have been added to work coherently with the original establishment (*troff*, *tbl*, *eqn*, *pic*, etc.) Backward compatibility is guaranteed because supporting a new class of objects requires no or minimal modification to existing processors. The absolute adaptation is to make the newly generated code and its desired action known to the device driver[1]. A problem with this model is that the available glue languages are largely non-interactive processors. They are not event-driven, and therefore do not react to interactive activities that require complex conversational support.

8.1.2 Integration Under Broader Scopes

Many intermediate integration paradigms are possible. For example, document processing tools (or desktop publishing facilities, as they are often called) on the Apple Macintosh are integrated in a totally different manner. The bulk of Macintosh software conforms to a highly consistent user interface, because guidelines as to how applications should be presented are followed religiously by developers. This is no coincidence due to the way the current version of Macintosh's operating system and window system are structured. Like a large happy family, the Macintosh system and all its applications reside in the same address space under a single thread of control. Its window system is tightly coupled with the system. Basic window management and graphics primitives are grouped into the ROM-based QuickDraw package [7], which is highly optimized. There is a standard data transfer format called PICT [7] that is used by most applications in exchanging information. A centralized clipboard is used to import or export data across applications.

A highly consistent system like the Macintosh is a dream come true for naive users, but it is often a nightmare for programmers. In a sense, the Macintosh's friendly user in-

[1]Depending on the device's imaging capability, sometimes this can be a non-trivial undertaking.

terface is realized at the expense of reliability and extensibility. As a common problem of systems based on a single address space, any fatal error in one application would crash the entire system, including other law-abiding citizens (e.g., running into a system panic mode on the Macintosh when an application violates certain rules). The result of unexpected sudden termination like this may be frustrating, and sometimes devastating. Single-language systems like Smalltalk and Cedar prevent this from happening by enforcing some hygiene in their language constructs. The Macintosh is not a single-language system; it supports multiple languages for software development, hence it is not as easy to ensure reliability.

With a glue language, the coupling of applications can be expressed as *programs* and thereby becomes extensible. Unfortunately, this feature is not found in the Macintosh top level. The story is different for HyperCard [63], which is a Macintosh application program, but can be used as an alternative top level to drive other applications. In HyperCard, a glue language called HyperTalk [122] can be used to tie all the pieces together. In addition, every operation corresponds to an underlying HyperTalk routine. Objects created within HyperCard can be conveniently manipulated based on these HyperTalk routines.

Another familiar technique is the Emacs approach to integration. In Emacs, editing is expressed as Lisp routines which may also access external shell-level processors, including pipelines. Objects, such as text regions and Lisp lists, can be easily interchanged and can be interfaced with any external processors involved. Therefore, the Emacs approach is more powerful than the pipelining model. A significant advantage here is that as interpreted programs, a very high degree of extensibility is achieved in this integration mechanism. The Lisp-based user interface is able to react to events driven by the user or other programs. Furthermore, having Lisp as the glue language, the boundary between system libraries and user-defined software is blurred. Except for very low-level primitives, every single piece of the integration mechanism may be redefined, which yields an extremely flexible system that can be customized toward specific needs.

As the Emacs approach to integration revolves around an editor, the X window system [119] approach to integration centers around windowing services on a distributed platform. In the X paradigm, which exemplifies the *server/client model* of integration, a server resides in a local machine, while clients (applications) can be connected to it across the network. The server multiplexes client requests to the display and demultiplexes input events from various devices (e.g., keyboard, mouse, etc.) to clients. Information exchange is based on a predefined protocol, rather than the simple byte stream in the pipelining model.

No full-fledged programming language is explicitly used as the glue language.

Falling into the same vein of distributed server/client model is the NeWS window system [127]. Unlike X, in which a client-server protocol similar to an assembly language is devised, NeWS chooses PostScript (a high-level programming language) as the communication medium between clients and their server. The NeWS server is essentially a PostScript interpreter extended with the functionality of process and window management. Corresponding to each client, a light-weight PostScript process is down-loaded to the server to handle client-specific operations. Once again, the issue is extensibility. Each client can craft its server-resident process to take advantage of the PostScript interpreter, which is Turing equivalent in computation power and rich in windowing and imaging capabilities.

A system like VORTEX interacts heavily with the underlying window system. Can the VORTEX target editor take advantage of the PostScript server under NeWS? The answer is both yes and no. PostScript is a programming language with a rich set of graphics capabilities and a powerful imaging model. The VORTEX target editor can take advantage of the PostScript server supported by NeWS for its graphics rendering. However, a physical document structure like VORTEX's IR_T is necessary for effective selection on the target editor. A page completely described in PostScript without such a structure will not suffice. An ideal approach is to retain the IR_T structure, while replacing some low-level information by routines written in PostScript. Under this situation, exploiting PostScript's powerful imaging model becomes a challenge to the formatter's code generator.

8.1.3 Integration under VORTEX

The integration mechanisms and their representative systems discussed above are not targeted toward document processing *per se*. However, a certain mixture of those techniques can be crafted to suit a particular document development system. VORTEX's integration mechanism typifies this hybrid approach. At the lowest layer, components of the VORTEX's principle trio are integrated by a mechanism similar to the distributed, protocol-based communication of X, with a replicated internal representation as the medium of data interchange (the structure is linearized during network transmission). By design, special objects at higher levels such as graphics, are presented on the same viewing surface as the main document body, but are edited by a special editor on a separate viewing surface.

Other issues, such as document structure and the control over external processors, are closely related to document editing, so the Emacs approach to integration is an obvious choice. The remainder of this chapter details some of these problems.

8.2 Compound Objects

A premise in supporting compound documents is being able to identify an object's type when one is selected. Under the VORTEX paradigm, textual objects, such as plain text and mathematical formulas, are readily identifiable on its internal representation (see Table 6.6). Another class of textual objects, tables, can be identified with the joint support of pattern matching in the source editor and internal representation accessing within the formatter. In other words, when a better front-end to mathematical or tabular objects becomes available (e.g., a WYSIWYG mathematical formula editor or table editor), the facility to locate the desired object is already in place. The new tool can be integrated with the source editor based on a protocol to the one described in Section 6.3 so that the selected object can be passed back and forth between the source editor and the special editor.

The weakest point of TEX is in handling non-textual objects. The only primitives having to do graphics in TEX are horizontal and vertical rules. The situation is improved somewhat in LATEX, but its capability is still limited and rather restrictive. Fortunately, there is a hook, the primitive of `\special`, built into TEX as a remedy. The way `\special` works is that it takes two arguments, in which the first argument specifies the vertical dimension of the object, which is used by TEX to reserve the necessary space. The second argument describes the object, but is uninterpreted by TEX. Since VORTEX is compatible with TEX/LATEX in source form, alien non-textual objects and are encapsulated under `\special`, whose scope and content are clearly marked in the internal representation. As mentioned previously, although the current VORTEX prototype has no graphics rendering capability, the design supports graphics specified in PostScript to be incorporated with TEX documents. The integration of a graphics editor and the PostScript imaging server with the principal trio can be found in Section 6.1. The same mechanism can be applied to other media types, such as voice and video.

8.3 Document Structure

Several levels of document structure are relevant in VORTEX. As described in Section 6.5, the incremental formatter maintains the internal representation, in which a hierarchy of the document's syntactic structure ($IR_S \cup IR_O$) is constructed for the source representation, and a shallow tree of the corresponding physical layout (IR_T) is built for the target representation. The other levels of document structure, the document/file correspondence, the document type, and the style-specific logical document structure, are not part of the kernel. Instead, they are realized in the source editor through the Lisp gluing mechanism.

VORTEX makes the distinction between a *document* and a *file* by acknowledging that multiple files may be included in a TEX-based document, and that potentially multiple documents may be maintained by the source editor simultaneously. In the source editor, a document is viewed as a tree of files, with branches being the file inclusion commands `\input`. This document tree has a root called the *master file* (level 0), which may include external files (level 1), each of which may in turn include more files (level 2), and so forth. Operations involving the entire document, such as `SF_Format`, must be identified by the master file. An index table is maintained by the source editor to record each individual file's master files. When any document-wide operation is invoked in a component file, the user is queried to determine the master file for which the operation is intended. This is necessary because the file may be shared by more than one document.

VORTEX's source editor also maintains the notion of *document type*, which identifies a specific TEX dialect the document belongs to. Currently, four document types TEX, LATEX, SLITEX [91, Appendix A], and AMS-TEX [124], the four most popular TEX dialects, are supported. The type information is needed when the user invokes any type-specific operations, such as format/reformat a document, in which case an instance of the incremental formatter preloaded as `tex`, `latex`, `slitex`, or `amstex` is used. Another application of the type information is to identify a document filter, which preprocesses a TEX-based document by stripping off irrelevant information for the main task, such as spelling checking.

From the user's perspective, a document's type information is implicit, except perhaps for the first time an operation depending on document type is invoked before such information is available, in which case the user is asked to make a decision as to what type the document is. Once the document type is specified, it is converted to a line of comment in the document header, which can be accessed automatically by future invocations. The user can also specify a default document type in the VORTEX startup file, so that even the one time inquiry becomes unnecessary. From the user's perspective, operations in VORTEX are generic. For instance, an operation is known as *format* at all times instead of as `tex`, `latex`, `slitex`, or `amstex` under different situations. VORTEX uses operator overloading implicitly by consulting the type information.

A next level of structure concerns the logical view of a document, given a specific style. LaTeX supports a very comprehensive set of styles. A LaTeX document style defines a group of logical entities, which encapsulate the low-level formatting details. For instance, in the `article` style, a section is a logical entity. Many useful operations can be implemented, if a logical entity like a section can be identified. Although the generic VORTEX internal representation does not support any logical entity of this kind, the hooks are actually in place to do so. For example, passing `section` as the name argument to *irs_find_macro ()* (see Table 6.9) will identify every section in a region. Given this primitive, logical operations, such as go to a particular section, can be implemented in Lisp. Such a high-level operation can be unqualified in that it scrolls forward or backward (e.g., `next-section` and `previous-section`), or it can be qualified by either section number or section title (e.g., `goto-section-number` or `goto-section-title`).

One can program arbitrary logical operations in a similar fashion. One can even envision a set of meta-functions to be developed in such a way that, when associated with certain keywords found in a certain document style, would generate the corresponding high-level logical functions like `goto-section-number` and `goto-section-title` described above. This is a surprisingly powerful approach to emulating a declarative system on top of a procedural platform. This also allows systems like SGML to be built on top of VORTEX.

8.4 Interactive Activities

In addition to direct editing, a document editor with programming power equivalent to Emacs can support a framework that would automate the placement of certain objects and their corresponding processing tasks. These activities and their corresponding processors include, among others:

- spelling correction, a document filter, and a spelling checker,

- citation making/browsing and a bibliography processor,
- index placement/browsing and an index processor,
- cross reference labeling/linking/browsing and the document formatter itself.

In an extreme case, these activities degenerates into ordinary editing tasks under an unintegrated environment, in which an editor level programming subsystem is not available. Their corresponding processors are invoked by the user at operating system top-level. The result of such off-line processing is incorporated back to the document manually. For instance, in an unintegrated case, spelling correction is performed manually by passing the document through an external spelling checker, gathering the spelling errors, returning to the editor, searching for every occurrence of the errors — sometimes across several component files of a document, and finally, correcting them on the spots located. In an integrated environment, such as VORTEX, every step of these can be programmed in an editor-supported programming language (e.g., VORTEX Lisp in our case) and the whole process can be automated.

Under the VORTEX paradigm, the same integration technique is also applied to the creation of citations, indexes, and cross references as well. An extensive authoring environment based on TEX and GNU Emacs has been developed [39]. The facilities consist of two major editing modes: *TEX-mode* [36] and *BIBTEX-mode* [35], for manipulating TEX-based documents and the associated BIBTEX [108] bibliography database files. Although these facilities are currently implemented in Emacs Lisp, porting them to VORTEX is straightforward. To get a flavor of this approach, the facilities for citation making and indexing are described below (refer to References [35,42,39,36] for more details).

8.4.1 Bibliography Making

This is an area where the two major editing modes *TEX-mode* and *BIBTEX-mode* work hand in hand to provide a friendly on-line bibliography/citation facility. The user uses *BIBTEX-mode* to prepare BIBTEX database files, *TEX-mode* to make citations and to generate actual bibliography files, and the combined system to detect and correct any errors in the database or the citations. The same system not only works with LATEX, for which BIBTEX was originally created, but with plain TEX and AMS-TEX documents as well.

```
@INBOOK{,
============================= REQUIRED FIELDS ==========================
  -------------- Exclusive OR fields: specify exactly one -------------
        AUTHOR = {},
        EDITOR = {},
  -------------- Inclusive OR fields: specify one or both -------------
        CHAPTER = {},
        PAGES = {},
  ------------- Rest of required fields: specify every one ------------
        TITLE = {},
        PUBLISHER = {},
        YEAR = {},
============================= OPTIONAL FIELDS ==========================
        VOLUME = {},
        SERIES = {},
        ADDRESS = {},
        EDITION = {},
        MONTH = ,
        NOTE = {}
}
```

Figure 8.1: A skeleton bibliography entry of type INBOOK.

Bibliography Database Manipulation

A BibTeX database file is one that has a file name extension of .bib and contains one or more BibTeX entries. *BibTeX-mode* uses a template-based user interface for the preparation of these entries. It supports all standard BibTeX bibliography entry types as built-in functions so that to insert a new entry the user only has to specify a type which in this case corresponds to an Lisp function. A skeleton instance of the specified type will be generated automatically with the various predefined fields left empty for the user to fill in. A host of supporting functions such as scrolling, field copying, entry duplicating, ..., etc. is provided to facilitate this content-filling process.

Figure 8.1 is an instance of the entry type INBOOK. The user invokes an entry like this by typing

```
M-x @inbook RET
```

where M-x is the command to call on source editor's Meta-X prompt (i.e. holding down the meta or escape key and type x) and RET means carriage return. Function name completion is supported at this prompt; hence only the shortest distinguishable prefix needs to be typed before RET. In the skeleton entry, banner lines are displayed to give hints regarding the nature of specific fields. These banners, along with all unfilled optional fields, will be removed when a cleanup command is issued in the mode. The cleanup operation will also catch any mandatory but unfilled fields so that correcting errors of this kind does not have to wait till `bibtex` is executed.

The user can debug, preview, or create a hard-copy draft of any BibTeX data file. This is done by executing the BibTeX processor on a temporary file which contains citations to every entry of the target `.bib` file. If there are any errors, a correction mechanism will position the cursor to the spot, prompting for fixes. At the end of a successful session, a draft is created in the formatted form, which can then be previewed or printed.

Symbolic Citations

Each entry in the bibliography database must be named symbolically, such as `knuth:tex`. In the document source, this entry would be referenced by the command `\cite{knuth:tex}`. The user can certainly enter this command manually, but that requires him either to remember what actually appears in the database entry or to manually visit the data file, locate the entry, and look up the name. With the *TeX-mode* lookup facility, neither is necessary. All that is needed is to specify a data file name and a keyword related to the entry such as `Knuth`, `TeX`, the publisher's name, or any regular expression. A list of matching entries will be returned. The user can confirm an entry, in which case the `\cite` command plus the entry name is inserted at current cursor position. Alternatively, a request can be made to show the content of an entry, or to scroll and inspect the next/previous match, to create a new entry, or even to change the search key. A search path may be specified so that when a wildcard file name is given, every data file in the path is looked up.

Bibliography Processing

In the LaTeX/BibTeX combination, the input to the bibliography processor `bibtex` is a file generated by `latex` which contains all citations made in the document. Note that collecting citation entries has nothing to do with formatting *per se*. In *TeX-mode*, these entries are collected prior to formatting. The bibliography processor `bibtex` is then spawned to process these entries. Finally symbolic-to-actual substitutions are done by the editor at the source level. It also interpolates the actual bibliography file generated by `bibtex` at the appropriate place (usually before the end of document). Hidden from the user is a file of cross references to be used to recover symbolic references from actual numbers when new citations are added to the document and a new bibliography/reference file is needed.

There are two important issues with respect to this source-level bibliography making scheme. The first is related to the incremental growth of citations in a document. Recall that all citations are done initially in a symbolic form. A *TEX-mode* bibliography making session will replace all of them by their actual counterparts in the document source. As new citations are entered, the document will have mixed symbolic and actual references. VORTEX's approach is to keep the most recent symbolic/actual cross reference information in a file. Whenever a new bibliography is called for, the cross reference information is first consulted to recover symbolic citations from actual numbers. The cycle continues by collecting citation keys, spawning `bibtex`, performing symbolic-to-actual substitutions, and so forth.

The second problem has to do with the efficiency of the symbolic-to-actual substitution mechanism. In a text editor, replacing a target string by a source string requires a series of operations composed of (1) locating the target pattern, (2) erasing the target string, and (3) inserting the source string. Locating the target pattern is normally based on string matching. Since the same entry can be cited at multiple places, the search has to cover each and every file included in the document. This is an extremely expensive operation.

The pattern matching overhead is completely avoided in *TEX-mode*'s symbolic-to-actual substitution mechanism. The trick is that in the first citation collecting sweep, each entry's position[2] is recorded. Moving the cursor to a designated position is very fast. The substitution mechanism processes entries in each component file in reverse order so that current replacement will not destroy the recorded position of the next entry. That is, the last instance of a symbolic citation is visited and replaced by an actual reference first, then the previous entry, and so on.

[2]In the source editor, each character is assigned an offset relative to the beginning of buffer. This offset is what is meant by the position here.

Advantages

The integrated bibliography handling facility is superior to what is available in the unintegrated situation. Some noticeable advantages are the following:

- *BibTeX-mode* relieves the user from any concerns regarding the format of bibliography entries. The template-based user interface makes the preparation of bibliography database files an easy task.
- Based on the two cooperating modes, errors in both the database and document citations can be corrected interactively. The system will position the cursor to the spot in question and the user will have a menu of options to correct the mistake. When all errors have been corrected, the processing resumes incrementally.
- Due to the lookup facility, the user does not have to memorize or type in the exact entry names in order to make citations. The system prompts the user, in sequence, the matching entries found in the specified bibliography database. The selected entry will be interpolated into the source automatically.
- Multipass processing is normally required for bibliography/citation resolution. Due to the automatic invocation of `bibtex`, the error correcting facility, and the automatic substitution mechanism, only one or two passes of formatting are needed to produce the final output — depending on the presence of cross references.
- Once the bibliography/reference file has been created, there is an inspection facility which allows the user to examine the content of a citation entry interactively, yielding an effective link between a citation's context and its content. This is available for citation entries in both symbolic and actual forms.
- The same mechanism not only works for LaTeX documents for which BibTeX was originally designed, but for any TeX dialect as well.

More extensions to this bibliography/citation facility are being carried out by my colleague Ethan Munson. The new system [43] will provide a more elaborate lookup mechanism that supports queries specified by qualifying certain fields in a bibliography entry, in addition to the regular expression search of the original version. Currently, browsing citations and linking them to their actual content in the bibliography section works only for the document source representation. The new facility will also extend the browsing mechanism to the target view as well. This extension is straightforward under the VORTEX architecture. Since the source representation is tightly coupled with the target representation, extending a mechanism such as bibliography/citation browsing from source representation to target representation becomes a simple matter of synchronized scrolling (see Section 6.7.

8.4.2 Index Preparation

Two major subsystems for indexing have been developed for TEX-based documents. One subsystem provides a systematic approach to placing index commands in the document source; the other deals with transforming the index from raw entries (generated by the formatter) to the final result (sorted form). Some of the key features of the two subsystems below. Both subsystems are rather elaborate by design and very easy to use. Detailed information can be found in Reference [40].

Index Placing Subsystem

TEX-mode's index-placing facility is based on a very simple framework. All the author needs is to specify a *pattern* and a *key*. The editor then finds the pattern, issues a menu of options and inserts the index command, along with the key as its argument, upon the user's request. As a special case, when the pattern and key are identical, neighboring words of the current cursor position or the text in the current region can be inserted as the index argument in one *TEX-mode* command.

There are two query-insert modes to operate with: one based on single key-pattern pair and the other on multiple key-pattern pairs. In the former mode, the user specifies a pattern and a key, and for every instance of the pattern found, he decides whether to insert the index command with the specified key, or a variant of it. In the latter mode, each key-pattern pair in a global list is processed in a way identical to that of the former mode.

Placing index commands is a task that has been performed traditionally in an ad hoc fashion. It is often tedious and time-consuming. Our facility offers a systematic and efficient approach to this effort. It has been used in producing indexes for a book and a number of manuals and has proved very useful and effective.

Index Processor

The index processor MakeIndex transforms raw index entries generated by the formatter into the final index file. The tasks performed in this transformation include permutation (entries are sorted alphabetically), page number merging, multi-level indexing (three levels of subindexes are recognized), style handling (customizable input and output formats), and various special effects such as cross referencing (*see* and *see also*), setting page numbers in different fonts, etc. Due to the style handling facility, MakeIndex is largely independent of the typesetting system and independent of the format being used. There is an interface in *TEX-mode* to MakeIndex much the same as the one to BIBTEX.

8.4.3 Job Control

Under the VORTEX paradigm, every external processor needed in the entire course of document preparation is controlled by programs implemented in the source editor's Lisp subsystem. In *TEX-mode*, most generic operators take the current document, file (buffer), or region as an operand, but some apply to words. In *BIBTEX-mode*, objects include the current bibliography file, an entry, and fields in an entry. From the user's point of view, not only the typing overhead is greatly reduced, but the need to swap context both mentally and physically between the editor and operating system top-level is eliminated.

8.4.4 Generalization

Exploiting the rich programming capability and its interactive language interpreter of a text editor like Emacs is not a new concept. Under the VORTEX architecture, this extension mechanism is ideal for the integration of external programs with document processing. It also makes it possible for the various activities associated with these processors to be performed interactively. These activities include placing task-specific objects in the document source, retrieving them for processing, and incorporating the processed results back to the document. An integrated facility allows the user to interact with these activities, as demonstrated by the above examples of bibliography citation/correction and index placement.

Under the VORTEX paradigm, this level of integration interfaces nicely with the underlying system structure. As shown in Sections 6.6 and 6.7, the programming interface between the high-level integration routines and the low-level state accessing routines is seamless. Although what has been shown is a TEX-based environment, it seems to generalize to systems based on other document formatting languages as well.

Chapter 9

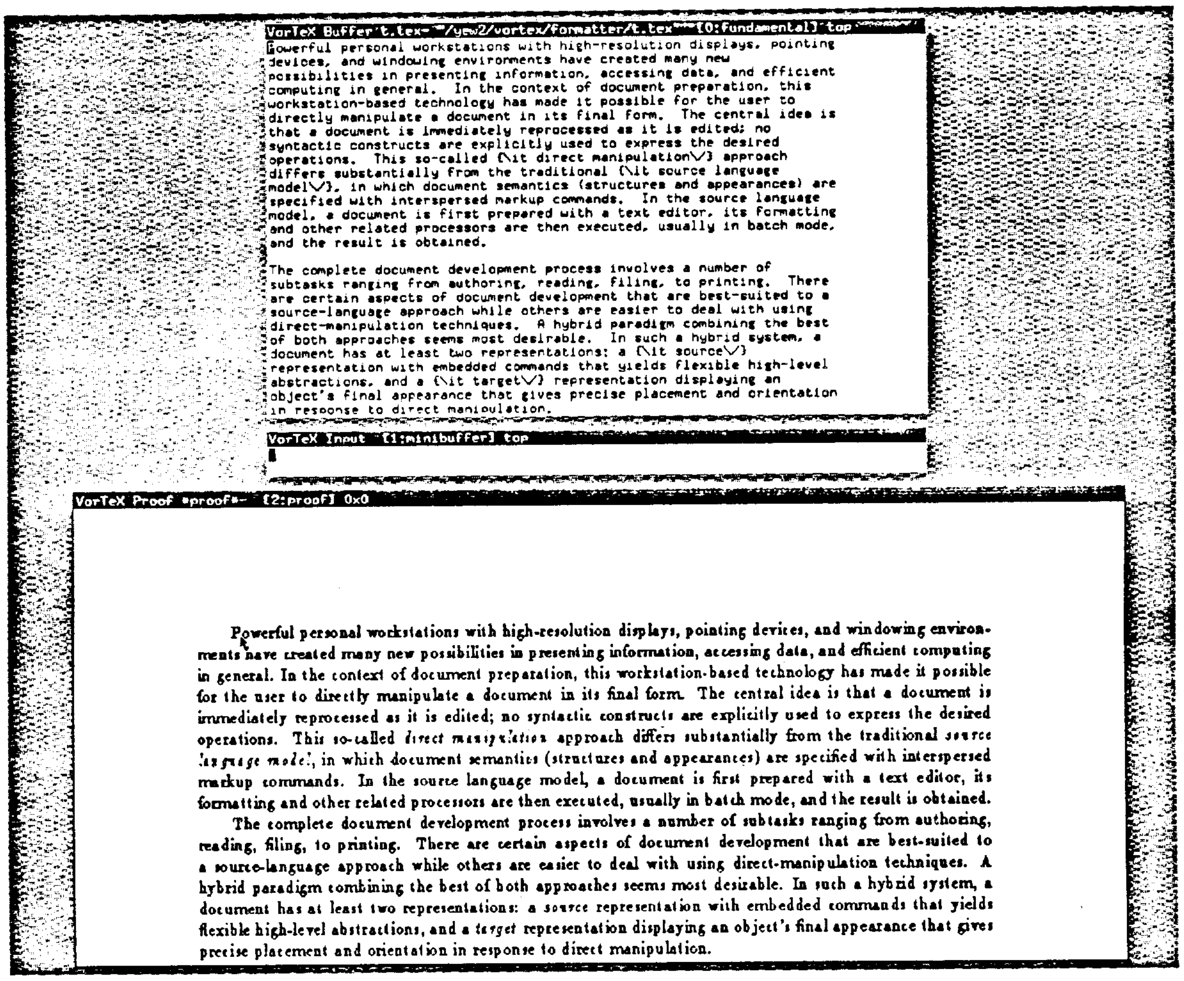
Evaluation

As pointed out in Section 1.2, a principal objective of the VORTEX research is to resolve the inherent conflicts found in various approaches to document development, and to arrive at some compromises that represent the best of each alternative. Having discussed an abstract paradigm (Chapters 3 and 4), as well as a concrete realization of the paradigm (Chapters 5, 6, 7, and 8), it is appropriate to put this alternative approach in perspective. Have the design goals been satisfied? How well does the prototype system work? How could it be improved? How does it compare with related work? This chapter consists of three parts: the first part is an evaluation of the VORTEX prototype, the second part discusses briefly several systems that are related to VORTEX, and the last part gives a comparison of VORTEX and these related systems.

9.1 Evaluating the VORTEX Prototype

The current VORTEX prototype has nearly 100,000 lines of high-level code in C and Lisp. The principal trio is implemented in C: the source editor is about 50,000 lines, the target editor is roughly 6,000 lines, and the incremental formatter has about 35,000 lines. About two thirds (23,000 lines) of the incremental formatter code comes from the original non-incremental version of TEX (CommonTEX). In addition to the principal trio, there are about 10,000 lines of Lisp code for the system's editing, reverse mapping, and pre- and post-processing facilities.

Figure 9.1 is a screen dump of the current VORTEX prototype running under the X window system. Three separate windows are present: the top window displaying the source



```
VorTeX Buffer't.tex-"/yew2/vortex/formatter/t.tex"""{0:Fundamental} top
Powerful personal workstations with high-resolution displays, pointing
devices, and windowing environments have created many new
possibilities in presenting information, accessing data, and efficient
computing in general.  In the context of document preparation, this
workstation-based technology has made it possible for the user to
directly manipulate a document in its final form.  The central idea is
that a document is immediately reprocessed as it is edited; no
syntactic constructs are explicitly used to express the desired
operations.  This so-called {\it direct manipulation\/} approach
differs substantially from the traditional {\it source language
model\/}, in which document semantics (structures and appearances) are
specified with interspersed markup commands.  In the source language
model, a document is first prepared with a text editor, its formatting
and other related processors are then executed, usually in batch mode,
and the result is obtained.

The complete document development process involves a number of
subtasks ranging from authoring, reading, filing, to printing.  There
are certain aspects of document development that are best-suited to a
source-language approach while others are easier to deal with using
direct-manipulation techniques.  A hybrid paradigm combining the best
of both approaches seems most desirable.  In such a hybrid system, a
document has at least two representations: a {\it source\/}
representation with embedded commands that yields flexible high-level
abstractions, and a {\it target\/} representation displaying an
object's final appearance that gives precise placement and orientation
in response to direct manipulation.
```

```
VorTeX Input "{1:minibuffer} top
```

```
VorTeX Proof *proof*- {2:proof} 0x0
```

Powerful personal workstations with high-resolution displays, pointing devices, and windowing environments have created many new possibilities in presenting information, accessing data, and efficient computing in general. In the context of document preparation, this workstation-based technology has made it possible for the user to directly manipulate a document in its final form. The central idea is that a document is immediately reprocessed as it is edited; no syntactic constructs are explicitly used to express the desired operations. This so-called *direct manipulation* approach differs substantially from the traditional *source language model*, in which document semantics (structures and appearances) are specified with interspersed markup commands. In the source language model, a document is first prepared with a text editor, its formatting and other related processors are then executed, usually in batch mode, and the result is obtained.

The complete document development process involves a number of subtasks ranging from authoring, reading, filing, to printing. There are certain aspects of document development that are best-suited to a source-language approach while others are easier to deal with using direct-manipulation techniques. A hybrid paradigm combining the best of both approaches seems most desirable. In such a hybrid system, a document has at least two representations: a *source* representation with embedded commands that yields flexible high-level abstractions, and a *target* representation displaying an object's final appearance that gives precise placement and orientation in response to direct manipulation.

Figure 9.1: *A snapshot of the current VORTEX prototype.*

editor, the middle window displaying an input/message mini buffer, and the bottom window displaying the target editor. The input/message window is shared by the two base editors. The incremental formatter is invoked as an background process and does not occupy any window explicitly.

9.1.1 Satisfying Design Decisions

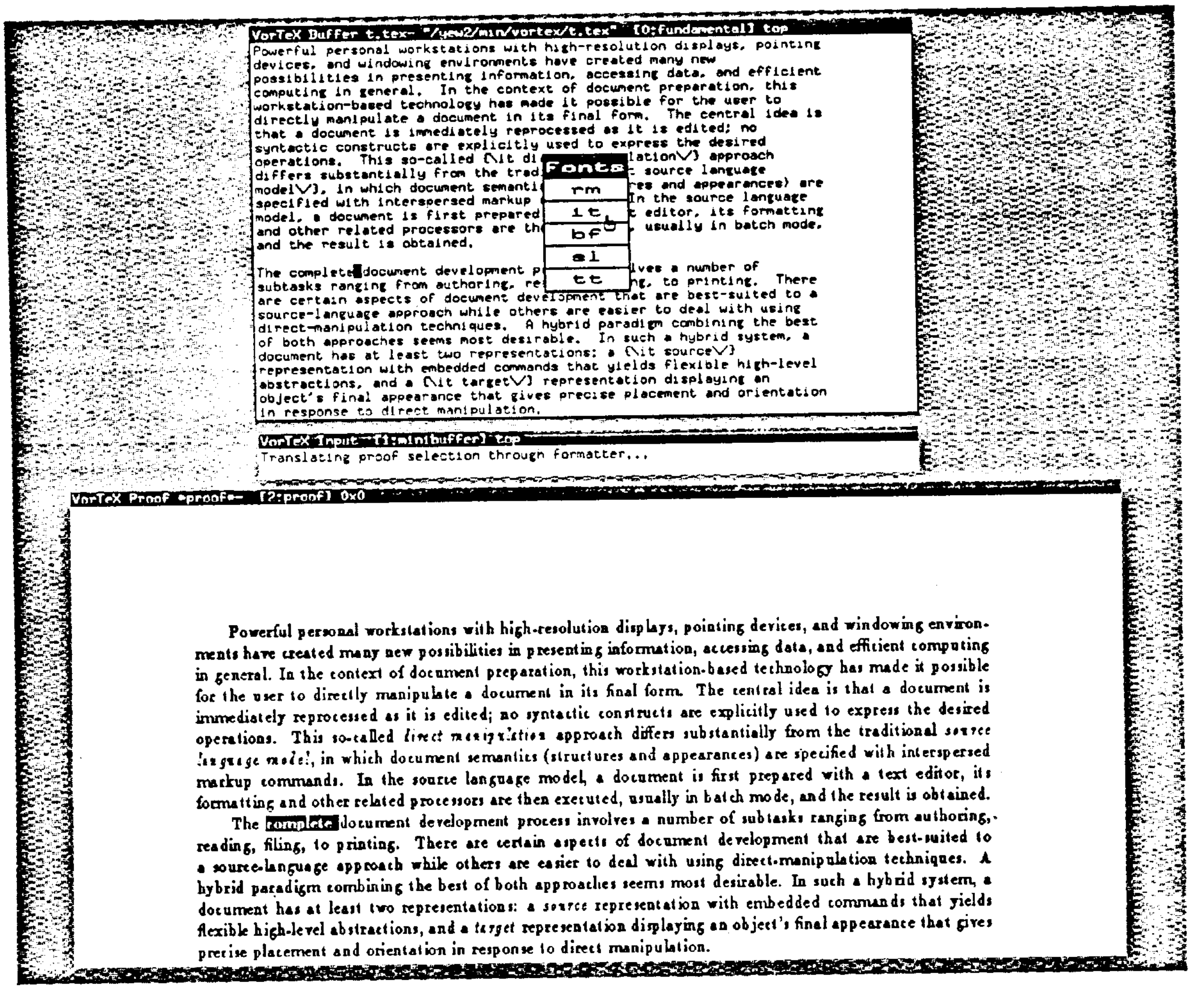
Section 1.2 identified four independent areas as potential conflicting spots in document development. The design decisions made in VORTEX have created an alternative that, to some extent, resolves these conflicts. A prototype implementation has proved that such an alternative is feasible. The remainder of this section reviews how these conflicting properties are resolved in VORTEX.

Document Specification: Source Language versus Direct Manipulation

VORTEX resolves the conflict between the source-language model and the direct-manipulation model by maintaining a document in two complementary views, *source* and *target*; both views can be manipulated by the user. The two document views are correlated by sharing a common internal representation. The transformation from source to target is carried out by an incremental formatter. Mapping in the reverse direction is realized by the source editor's Lisp programming subsystem, which implements the semantics of each target-level operation by embedding its corresponding source-level commands in a program that gets executed when the target operation is triggered. The resulting hybrid paradigm allows document development tasks to be delegated to either the source editor or the target editor based on convenience considerations. In VORTEX, depending on the user operation, the two document views can be synchronized in either the delayed mode or the immediate mode.

For instance, the user can perform Emacs-style text editing in the source editor. In particular, fine-grain insert and delete operations are performed on the source view. Direct modifications to the document's source view are evaluated under the delayed-execution model; they are mapped to the target view when the user explicitly invokes the *format* command in the source editor. In practice, this rarely happens. Instead, what happens most often is that the user can invoke any command in the target editor, and the reformatting will be triggered automatically. The target editor is used to incorporate direct manipulation operations. Two special steps are taken to ensure that each target-level operation is legitimate: (1) before a target-level operation can be invoked, reformatting is triggered automatically so that the target view is guaranteed to be valid, (2) when a target-level operation is actually invoked, the corresponding source-level Lisp program is executed immediately and the re-evaluation takes place automatically so that the sensation of directness can be achieved.

Figures 9.2 and 9.3 demonstrate a target-level font change operation in action. In Figure 9.2, a word is selected in the target editor (shown in the highlighted area) and the font change operation is invoked. Possible fonts are displayed in a menu. Suppose the italic font is selected, its corresponding source-level Lisp program (see Figure 6.6) is executed



```
VorTeX Buffer t.tex- "/yew2/min/vortex/t.tex"  [0:Fundamental] top
Powerful personal workstations with high-resolution displays, pointing
devices, and windowing environments have created many new
possibilities in presenting information, accessing data, and efficient
computing in general.  In the context of document preparation, this
workstation-based technology has made it possible for the user to
directly manipulate a document in its final form.  The central idea is
that a document is immediately reprocessed as it is edited; no
syntactic constructs are explicitly used to express the desired
operations.  This so-called {\it di           lation\/} approach
differs substantially from the trad           source language
model\/}, in which document semanti           res and appearances) are
specified with interspersed markup            In the source language
model, a document is first prepared           t editor, its formatting
and other related processors are th           , usually in batch mode,
and the result is obtained.

The complete document development p           lves a number of
subtasks ranging from authoring, re           ng, to printing.  There
are certain aspects of document development that are best-suited to a
source-language approach while others are easier to deal with using
direct-manipulation techniques.  A hybrid paradigm combining the best
of both approaches seems most desirable.  In such a hybrid system, a
document has at least two representations: a {\it source\/}
representation with embedded commands that yields flexible high-level
abstractions, and a {\it target\/} representation displaying an
object's final appearance that gives precise placement and orientation
in response to direct manipulation.

VorTeX Input  [1:minibuffer] top
Translating proof selection through formatter...

VorTeX Proof *proof*-  [2:proof] 0x0
```

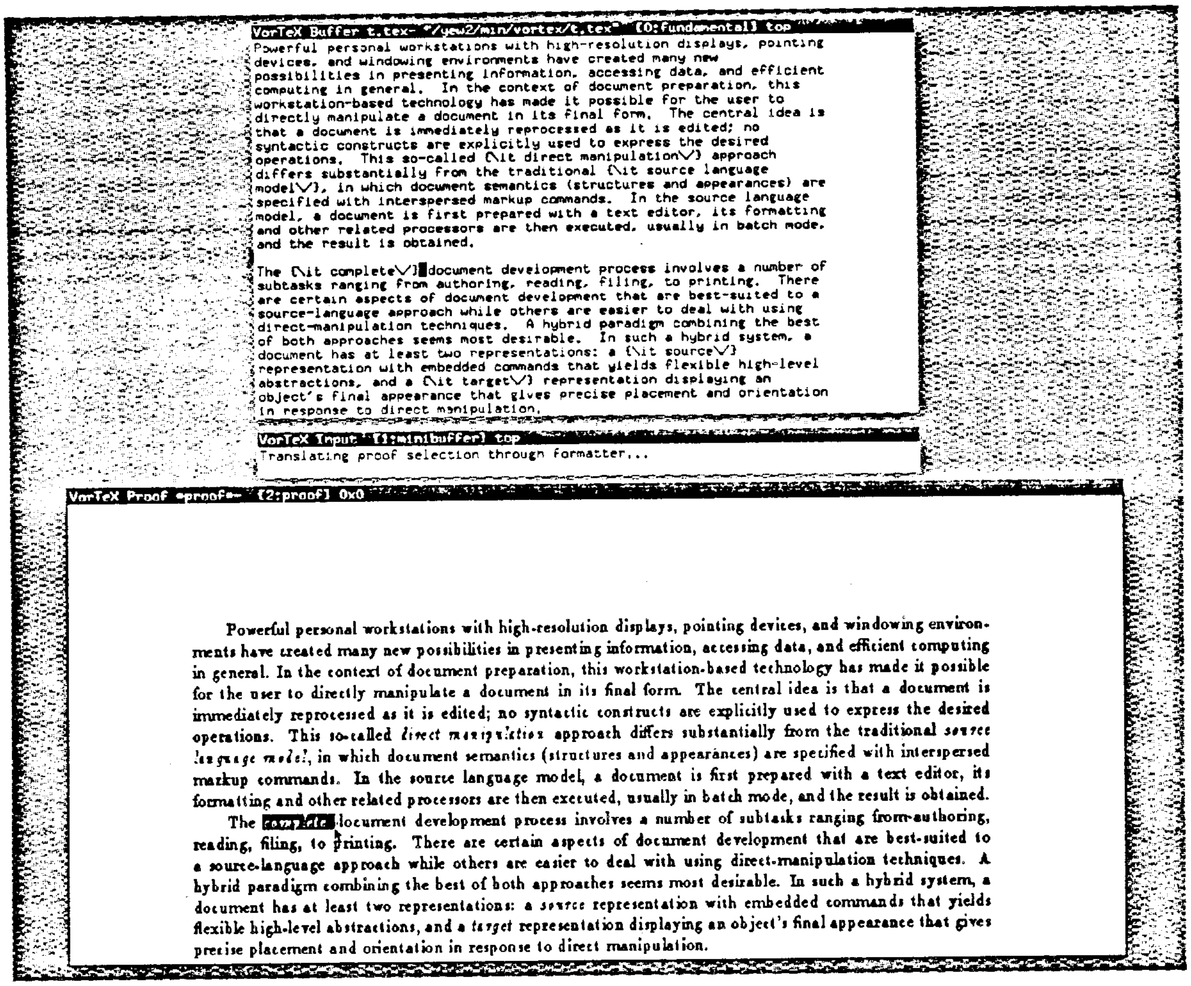
Powerful personal workstations with high-resolution displays, pointing devices, and windowing environments have created many new possibilities in presenting information, accessing data, and efficient computing in general. In the context of document preparation, this workstation-based technology has made it possible for the user to directly manipulate a document in its final form. The central idea is that a document is immediately reprocessed as it is edited; no syntactic constructs are explicitly used to express the desired operations. This so-called *direct manipulation* approach differs substantially from the traditional *source language model*, in which document semantics (structures and appearances) are specified with interspersed markup commands. In the source language model, a document is first prepared with a text editor, its formatting and other related processors are then executed, usually in batch mode, and the result is obtained.

The complete document development process involves a number of subtasks ranging from authoring, reading, filing, to printing. There are certain aspects of document development that are best-suited to a source-language approach while others are easier to deal with using direct-manipulation techniques. A hybrid paradigm combining the best of both approaches seems most desirable. In such a hybrid system, a document has at least two representations: a *source* representation with embedded commands that yields flexible high-level abstractions, and a *target* representation displaying an object's final appearance that gives precise placement and orientation in response to direct manipulation.

Figure 9.2: *Target-level font change operation in action (before).*

immediately. Afterwards, as shown in Figure 9.3, the corresponding TEX code is inserted in the source representation (`{\it` ⋯ `\/}`). Reformatting takes place automatically, and the italicized word is displayed and highlighted in the target representation.

The two document views and the dual execution models have created a flexible and extensible environment. Primitive target-level operations, such as scrolling and selection, are in place. The reverse mapping facility is fully exposed to the user; new operations can be extended rather easily.



```
VorTeX Buffer t.tex- "/yew2/min/vortex/t.tex"  {0:fundamental} top
Powerful personal workstations with high-resolution displays, pointing
devices, and windowing environments have created many new
possibilities in presenting information, accessing data, and efficient
computing in general.  In the context of document preparation, this
workstation-based technology has made it possible for the user to
directly manipulate a document in its final form.  The central idea is
that a document is immediately reprocessed as it is edited; no
syntactic constructs are explicitly used to express the desired
operations.  This so-called {\it direct manipulation\/} approach
differs substantially from the traditional {\it source language
model\/}, in which document semantics (structures and appearances) are
specified with interspersed markup commands.  In the source language
model, a document is first prepared with a text editor, its formatting
and other related processors are then executed, usually in batch mode,
and the result is obtained.

The {\it complete\/}█document development process involves a number of
subtasks ranging from authoring, reading, filing, to printing.  There
are certain aspects of document development that are best-suited to a
source-language approach while others are easier to deal with using
direct-manipulation techniques.  A hybrid paradigm combining the best
of both approaches seems most desirable.  In such a hybrid system, a
document has at least two representations: a {\it source\/}
representation with embedded commands that yields flexible high-level
abstractions, and a {\it target\/} representation displaying an
object's final appearance that gives precise placement and orientation
in response to direct manipulation.
```

```
VorTeX Input  {1:minibuffer} top
Translating proof selection through formatter...
```

```
VorTeX Proof *proof*-  {2:proof} 0x0
```

Powerful personal workstations with high-resolution displays, pointing devices, and windowing environments have created many new possibilities in presenting information, accessing data, and efficient computing in general. In the context of document preparation, this workstation-based technology has made it possible for the user to directly manipulate a document in its final form. The central idea is that a document is immediately reprocessed as it is edited; no syntactic constructs are explicitly used to express the desired operations. This so-called *direct manipulation* approach differs substantially from the traditional *source language model*, in which document semantics (structures and appearances) are specified with interspersed markup commands. In the source language model, a document is first prepared with a text editor, its formatting and other related processors are then executed, usually in batch mode, and the result is obtained.

The *complete* document development process involves a number of subtasks ranging from authoring, reading, filing, to printing. There are certain aspects of document development that are best-suited to a source-language approach while others are easier to deal with using direct-manipulation techniques. A hybrid paradigm combining the best of both approaches seems most desirable. In such a hybrid system, a document has at least two representations: a *source* representation with embedded commands that yields flexible high-level abstractions, and a *target* representation displaying an object's final appearance that gives precise placement and orientation in response to direct manipulation.

Figure 9.3: *Target-level font change operation in action (after).*

Document Evaluation: Quality versus Immediate Response

Quality and directness are not necessarily conflicting concepts. Quality is a property of the formatted document, while directness is a sensation involved in the process of manipulating a document. Document quality is the primary concern of VorTeX, which is achieved by the 100% TeX compatibility. At the same time, directness in VorTeX is significantly improved over the batch version of TeX, as the new formatter works incrementally on a page granularity. The same strategy, which is based on augmenting a non-incremental system, can be extended to a per-paragraph approach, while remaining compatible with TeX's criteria of generating high quality output. Under the VorTeX paradigm, quality is not compromised by enhancing the sensation of directness through incremental processing.

In other words, directness is achieved without sacrificing quality.

Moreover, VORTEX represents a true WYSIWYG system, in which the target representation is an exact replica of what will eventually appear on the hard-copy. By contrast, galley-based direct-manipulation document editors are not exact WYSIWYG systems. For either convenience or performance reasons, they are unable to support pagination on the fly and sometimes must settle for more straightforward line breaking strategies, and thereby must rely on an off-line batch postprocessor to produce the final high-quality output. There is no approximation involved in VORTEX's formatting engine. In many occasions, this precise correspondence between an editable screen representation and the final hard-copy is the most critical and desirable feature in document development.

Degree of Detail: Procedurality versus Declarativeness

VORTEX is so tightly woven with TEX's low-level processing engine that its becoming a procedural-based system is inevitable. Declarative properties, such as logical entities and document structure, are emulated in TEX by abstractions (e.g., LATEX), and can be encapsulated in VORTEX through the Lisp programming subsystem. Because VORTEX's processing engine is equivalent to that of TEX's, and because the way LATEX works is through the expansion of macros that are defined on top of TEX's low-level primitives, the VORTEX kernel can support a declarative system like LATEX at no extra overhead. Operations more intimately linked to these declarative or logical properties can be programmed in Lisp, with support through access to the internal representation.

The Glue: Strong Integration versus Weak Integration

VORTEX's integration mechanism is one that is strong enough to handle different aspects of document processing in a coherent manner — both from the system's and the user's standpoints, and yet weak enough to be extensible. In the system organization layer, the protocol-based distributed framework allows parallelism and extra computation cycles to be exploited, while at the same time, reduces memory overhead that might otherwise be unacceptable. The protocols among the principal trio are very simple and can be extended easily. Compound objects and their respective editors can be integrated with the main system through protocol extensions. In the representation and transformation layer, VORTEX achieves the capability of mapping an object from its source view to the corresponding target view, and vice versa, through tightly-coupled internal representation, an incremental formatter in the forward direction, and a systematic and extensible transformation facility in the reverse direction.

In addition to supporting reverse mapping and emulating declarative properties, VORTEX's Lisp substrate also serves in the capacity of integrating pre- and post-processing tasks with document editing and formatting. Traditionally, these pre- and post-processing tasks have been regarded as second-class citizens in interactive document processing systems, which is largely due to the absence of a programmable glue language, and consequently the inability to access external data and processors coherently. Under the VORTEX paradigm, manipulating and processing these objects are all part of the integrated system which makes it an environment specially suited to technical and scholarly work.

9.1.2 TEX Compatibility

Achieving TEX compatibility does not imply a system similar to VORTEX cannot be built from scratch. In fact, algorithms developed by TEX have become the *de facto* standard. Variants of them can be found in a number of more recent document formatting systems, such as Quill, Lilac, INF⊕R [120], etc. However, with investigating new research problems instead of rehashing known issues in mind, a software engineering decision was made in the beginning of the project to utilize these superior algorithms. Adapting from an existing program that is known for its quality seemed very sensible. The VORTEX approach has turned out to be an interesting and challenging experience.

In retrospect, VORTEX's alliance with TEX is both good and bad[1]. It is good because, by design, VORTEX inherits all the expressiveness and quality of TEX, and hence every TEX document can be processed by VORTEX with exactly the same result. The VORTEX incremental formatter has passed the "trip test", a validation suite designed by Knuth to ensure TEX compatibility [88]. Creating an incremental formatter by augmenting a non-incremental processor like TEX has some advantages. Most noticeably, it relieved the project from the necessity of creating algorithms for low-level formatting like hyphenation, line breaking, and pagination, while allowing us to concentrate on issues specific to the the interactive situation.

[1]Obviously, if it had been decided not to build the prototype on TEX, the resulting system would not be named VORTEX

document	*space*	*word*	*par*	*group*	*cseq*	*math*	*others*
macro package	5%	5%	.2%	2%	8%	.1%	ϵ
regular	14%	14%	.3%	1%	2%	.3%	ϵ
math-intensive	16%	16%	.3%	.5%	4%	2%	ϵ
average	15%	15%	.3%	1%	3%	1%	ϵ

Table 9.1: *Internal representation occurrence distribution.* This table indicates the distribution of nodes in the internal representation maintained by the VORTEX incremental formatter. The percentage is measured against the total size of a document in terms of number of characters. An ϵ means the distribution is less than .1%.

IR	IR_S	IR_O						IR_T			
node/box	*char*	*space*	*word*	*par*	*group*	*cseq*	*math*	*t_box*	*n_box*	*extra*	*total*
share	1	.15	.15	.003	.01	.03	.01	1	.15	.05	–
size	22	22	22	22	20	38	22	32	42	27	–
subtotal	22	3.3	3.3	.07	.2	1.14	.22	32	6.3	1.35	69.88

Table 9.2: *Internal representation overall heap consumption.* This table lists the overall run-time memory consumed by VORTEX incremental formatter's internal representation. Again, the fractions are relative to the total size of a document in terms of number of characters. The size is in bytes.

Augmenting a complex program like TEX, from an implementation standpoint, is not a programmer's most desirable and memorable experience. It would be much easier, in many cases, to develop code from scratch than to understand the original program first, find the right spot next, and finally insert the extension for the new functionality. Sometimes the extensions involve getting around some constraints imposed by the original constructs which otherwise might not occur if the code were fresh. On the other hand, without the well-documented program listing of TEX [89], the work would have taken much longer. The various indexes available in *TEX: The Program* increased our productivity significantly.

9.1.3 Heap Consumption

By design, VORTEX's internal representation requires a large amount of memory. Also, in the prototype implementation, little attention was given to optimizing heap usage, which contributes nontrivial overhead to the overall memory consumption. Table 9.1 indicates the distribution of IR_O nodes for three types of TEX documents: macro package, regular, and mathematics-intensive. The average is biased toward the regular and math-intensive situations, because most macro packages are pre-loaded, and thereby do not contribute anything to the internal representation. What Table 9.1 illustrates is that, for a document of size N, the number of IR_O nodes of types IRo_Space and IRo_Word are, on the average, both $.15N$, $.003N$ for IRo_Par nodes, $.01N$ for IRo_Group nodes, and so on.

Table 9.2 details IR's overall memory consumption under the current prototype, which is broken down into IR's natural components: IR_S, IR_O, and IR_T. Since IR_S and IR_T each maintains a copy of the document, the two 1's on the *share* row under columns *char* and *t_box* are the two biggest contributors to memory overhead. Since syntactic sugar found in space, group, and control sequence nodes is stripped off in IR_T, the number of non-terminal boxes is roughly equivalent to the number of word nodes in IR_O. The *size* row indicates the size of each node in bytes. Actual specification can be found in Appendix C. Multiplying the *distribution* and *size* rows yields the row of *subtotal*, which, when summed up, gives the total memory overhead as a factor of the document size. Presently this ratio is somewhere between 60 and 70, which is, of course, a staggering number. This means, with 16 megabytes of memory, the maximum capacity VORTEX can handle is roughly 210 kilobytes (the formatter itself takes up 1 megabytes of memory), which is a document of 40 to 50 pages long. This is acceptable for a prototype whose purpose is to demonstrate the underlying ideas, but certainly needs optimization to be of production quality.

There are several ways to restrain this ratio. One solution is to perform memory management within the formatter. When the memory consumption exceeds a certain threshold, a portion of the IR is swapped out, which can be brought in later when it is referenced again. This is reminiscent of virtual memory in operating systems and, because of its complexity, is not recommended except as a last resort. A somewhat easier solution converts IR_S nodes and IR_T terminal boxes from their current structure of doubly-linked lists into arrays. This would save as much as 25% of the total memory overhead, simply by eliminating left and right pointers. A more elaborate scheme is needed in this case to manage insertion and deletion of nodes. If further cutback is called for, terminal character boxes on IR_T may be eliminated at the expense of a poorer selection mechanism in which the finest-grain object is a word, rather than the individual character. The tradeoff here is heap consumption versus functionality, which can be tuned according to actual requirements.

As a matter of *déjà vu*, TEX's taking over of heap management from a programming language's run-time environment can be recreated in VORTEX, which may improve the overall memory utilization, such as reducing the pointer size from 4 bytes to 3 bytes (yielding an effective heap size of 16 megabytes, which is quite enough). There is a limitation, however. Even if the heap consumption were reduced considerably, say to only a factor of 10 to 20 times the size of a document — a drastic reduction of 70 percent from the current implementation, the memory overhead is still rather substantial. A memory overhead an order of magnitude larger than the document size can be expected. From an implementation standpoint, this is a price to be paid for creating a multiple-representation system.

9.2 Related Work

A number of document-processing systems are directly related to the VORTEX research. These are either systems that explicitly focus on multiple representation issues, or ones that support features similar to some other aspects of VORTEX. A brief description of the systems of interest follows. All except INF⊕R have been introduced earlier in this dissertation.

9.2.1 Janus

Janus [30] is a two-view document editor developed in the early 1980's by scientists at IBM San Jose Research Center. A Janus source document is specified in GML, a precursor of SGML. The Janus workstation consists of an IBM 3277 display terminal for the editing of the source representation, a Tektronix 618 Williams tube display for rendering the document's target view, and a joystick as the pointing device. Like VORTEX, Janus expects its users to do most of the editing in the *markup* (source) view. Editing in the *formatted* (target) view is limited to the cut-and-paste of figures. Formatting must be triggered by the user by invoking a "SHOW" command. The formatting strategy in Janus is based on extensive two-level state checkpointing and uses coroutines. The primary coroutines are the *formatter* (galley maker) and the *packer* (page breaker) both of which save their internal state information periodically. No quiescence checkpointing is performed. Janus did not take off because the display technology was a few years behind its ideas. Powerful personal workstations were not available at its time, so it had to rely on the IBM 3081 mainframe as its back-end machine, which limited its usability.

9.2.2 Quill

Quill [32], a descendant of Janus, is a new WYSIWYG document editor based on standard workstation technology. At IBM Almaden Research Center, a Quill prototype is being implemented on top of a server/client-based window system called WHIM [62], which runs under the QuickSilver operating system [28,70] on the IBM RT/PC workstation. Quill consists of two editors: a document editor and a style editor [33]. The document editor is a monolithic program that logically decomposes into individual editors, each responsible for a particular class of objects in a compound document. Editing and formatting are strongly integrated under a spontaneous mode that demonstrates a direct-manipulation user interface. The editor's underlying document structure is represented in SGML, so is its off-line data representation. A structural view, which illustrates the type and scope of SGML tags that mark up the document under manipulation, is displayed on a pane along the side of its main editing surface. The style editor allows one to customize document styles by simple clicking and typing on a form-based interactive tool. The document editor accepts output from this tool and performs formatting accordingly.

9.2.3 EZ

EZ is the base editor of the Andrew environment. From another perspective, EZ can be viewed as the top-level object of the Andrew toolkit [107], which evolved from an earlier version described by Gosling [64]. EZ is a galley-based document editor that must call upon a postprocessor for pagination. The most unique aspect of this system is its ability to integrate arbitrarily nested objects into a compound document, which is achieved as a result of its object-oriented, dynamically-linked class mechanism, and the conversational event processing facility. There is a top-level interaction manager in the Andrew toolkit environment. Input events are passed down the object hierarchy for processing, while drawing events are handed up, so that a parent can coordinate conflicting requests from its children. Through this up-down calling convention, objects can exchange their size information and negotiate the necessary real estate. Support from a toolkit environment like this gives a great deal of flexibility to a document editor. EZ is a galley-based, direct-manipulation document editor. Although in the Andrew toolkit the class of *data* objects is separated from the class of *view* objects and a data object can be associated with multiple views, the EZ document editor is not a multiple-representation system by our standard because a document is presented only in a single target view.

9.2.4 Tweedle

Tweedle [11] is a graphics editor that tries to bridge the gap between source-language and direct-manipulation based approaches to graphics specification by using a program as its internal representation for a picture. During an editing session the user can modify either the picture itself or the program representation; the editor modifies the other to keep the two consistent. The language used by the editor contains features that allow the editor to incrementally execute parts of a program in response to a change so that the picture can be regenerated without completely re-executing the program. Its use of a procedural representation allows the user to create pictures with repetition, recursion, and calculated point values. It further allows users to define parts of a drawing as variants of other parts; these variants can differ from their original objects in quite arbitrary ways but still respond to changes made to the original. Tweedle is a rather symmetric multiple-representation system that allows editing to take place in either the source or the target representation.

9.2.5 Lilac

Lilac [27] is a two-view document editor that employs the boxes-and-glue model of typesetting, *a la* TEX. It incorporates a more constrained source language to simplify the task of document specification. Lilac's document specification language is procedure-oriented rather than macro-oriented, and it is purely functional with no side effects. There are some exceptions to deal with such things as figure numbers and footnotes. It emphasizes tight coordination between the two views. A selection or insertion point in the target view is reflected immediately in the source view, as is every keystroke of inserted text. The user may accomplish this kind of coupling in the other direction, if a structural editor is available for the source view. Currently a plain text editor like Emacs is used for the source view, so a "reparse" command is required to get source edits reflected in the target. The user is encouraged to use the target view to manipulate a document.

Lilac can keep up with a natural typing pace in ordinary paragraphed text, but not in equations and other intricate structures. The keys to this performance include: (1) an incremental interpreter, which uses a hash table to keep track of previous operations

```
\headline={
    \ifnum\pageno=1     \hfill
    \else
        \ifodd\pageno   \hfill\bf\folio
        \else           \bf\folio\hfill
        \fi
    \fi
}
```

Figure 9.4: *A running headline defined in TeX.* TeX has its own abstraction mechanism to define a conditional running headline like what is shown here.

performed on a certain object, (2) some carefully tuned incremental primitives, including the paragrapher and the semantics processor that turn character strings into lists of boxes and glue, and (3) an incremental screen redrawing algorithm that re-uses previously displayed bits, and renders only what is new.

Of all the document preparation systems, Lilac is probably is closest to VorTeX in appearance. By design, however, the two have subtle differences, as is discussed in the Section 9.3.

9.2.6 INF⊕R

INF⊕R [120] is a Lisp-based WYSIWYG document editor, in which both editing and formatting are realized exclusively in Lisp. In appearance, it looks like an Emacs front-end to TeX, because its editing is fully compatible with Emacs and its formatting resembles TeX so much that the only noticeable difference is their syntactic sugar. Consider an example of defining a headline conditionally: it is bound to nothing (a bloc of glue) on page 1, the current page number in boldface flushed right on odd pages, and the current number in boldface flushed left on even pages. In TeX, one would define this headline as what is shown in Figure 9.4, where `\hfill` emits some amount of glue that is stretchable, and `\folio` is TeX's internal register that contains the current page number. This `\headline` macro will be expanded recursively whenever a page is constructed.

In INF⊕R, a Lisp function called `headline` can be defined as in Figure 9.5, where `format` is a function that returns a string when invoked with the first argument bound

```
(defun headline ()
  (format nil
          (cond ((eql 1 *folio*) "!hfill")
                ((oddp  *folio*) "!hfill!bf ~d")
                (t               "!bf ~d!hfill"))
          *folio*)
```

Figure 9.5: *A running headline defined in INF⊕R.* INF⊕R supports a set of low-level formatting primitives and registers that is largely compatible with TEX. However, its abstraction mechanism is based on Lisp.

to nil. The second argument is the format string (like those that appear in the standard Common Lisp format control or the C `printf` statements), in which `~d` (for a numeric value) is replaced by the third argument `*folio*` when `format` returns. The escape symbol ! is the counterpart of TEX's backslash, and `*folio*` here is identical to `\folio` in TEX. INF⊕R's output routine would invoke `headline` and place the returned string in the right position on a page. In other words, low-level formatting primitives are, by and large, identical in INF⊕R and in TEX. Program abstractions, however, are built into TEX, but are accomplished by Lisp in INF⊕R.

INF⊕R also provides a host of special key bindings for the purposes of entering mathematical formulas, creating tables, adjusting layout attributes like penality and glue, initiating formatting, and so on. Each operation can be invoked with just a few keystrokes, which greatly facilitates most of the complex activities. INF⊕R is rather close to VORTEX in spirit. However, INF⊕R does not provide explicit multiple representations and is not fully-compatible with TEX.

9.3 A Comparative Analysis

This section compares some key aspects of VORTEX with those of the systems mentioned above. In-depth analysis of commercially available systems is not possible due to the lack of technical information. This is not surprising because very few vendors publish key technical information on their products. Nevertheless, based on the experience of a user, some commercial systems are included in the comparison below for the sake of completeness.

9.3.1 Multiple Representations

The combination of scripts and actions, namely a hybrid of source-language and direct-manipulation approaches to document specification, is a powerful paradigm. A source language provides high-level constructs and abstractions, while direct manipulation yields prompt visual feedback regarding placement, measurement, and orientation. The advantages of multiple representations have been recognized by many, which is evidenced by the increasing number of systems supporting multiple representations either explicitly or implicitly. Systems like VORTEX, Tweedle, and Lilac present both source and target views of a document, allowing both representations to be manipulated by the user. Janus typifies an alternative in that both views are presented, but one of the views (target) is only commissioned with very limited capability. Systems like Quill present essentially one and a half views of a document — a target and its internal structure. A source representation in SGML, which serves as its off-line representation, is implicit in a Quill session. INF⊕R exemplifies yet another approach that a source representation (in Lisp) is implicit but accessible.

The Andrew toolkit separates the notion of *view* from the actual data object, so that multiple views can be associated with the same data object. This effectively relieves individual applications from worrying about the coherence among multiple representations. The facility provided by the toolkit acts as a server that triggers the necessary transformations automatically. However, since EZ is a galley-based, direct-manipulation editor, which does not incorporate extensive multiple representations as supported by VORTEX, it remains to be determined experimentally how well this high-level approach works for complex tasks.

The trend of migrating into a world of multiple representations is not confined to research projects. A recent version of MicroSoft Word offers a simple scripting mechanism that apparently goes beyond its predecessors which were based solely on the direct-manipulation approach. HyperCard is a commercial application that demonstrates the power of merging direct manipulation with a scripting mechanism like HyperTalk. As personal computers are becoming more and more powerful, this trend will continue to grow.

From both the user's perspective and the system's design and implementation standpoint, a wide diversity can be found in these multiple-representation systems. For instance, there is no consensus as to the relative importance of the two document views. In Janus and VORTEX, the user is expected to do most of the editing in the source view, the opposite is expected in Lilac. In Tweedle, where the task is somewhat restricted (graphics only), the source and target views are more symmetric. The nature of the source languages involved also differ in many respects. VORTEX is based on the macro-oriented TEX, which introduces a number of problems in reverse mapping that are not encountered in declarative languages like SGML or the nicely crafted macro-free languages in Tweedle and Lilac.

9.3.2 Inter-Representation Transformations

Interesting commonalities exist in the underlying execution model of multiple representation systems. For instance, in systems with two explicit views, the transformation from source to target, i.e., the "execution" of the source representation as a "program", is almost always triggered by the user under the delayed-execution model (see also Section 7.1). By contrast, it is mandatory for the target view under direct manipulation that its execution (re-evaluation) be immediate and spontaneous. Generally speaking, transforming a source representation to its corresponding target representation is more involved than simply updating the target representation itself. In the source-to-target transformation, which is similar to program compilation, it is necessary to reparse the source representation's syntactic structure, and subsequently regenerate new target code.

Reflecting changes in the target view spontaneously as the source view is modified is the key to directness. The bottom line is that the spontaneous evaluation must be able to keep up with the user's typing pace on the source editor. Unfortunately, this is infeasible for several practical reasons, all having to do with processing efficiency. First, efficient reparsing of the source document is critical, which can be facilitated if a syntax-directed editor is used. Lilac does not incorporate a syntax-directed editor. Although its reparsing is incremental, it could be improved even more under syntax-directed editing. It is due to the lack of syntax-directed editing capability that Lilac requires its source-to-target transformation to be triggered asynchronously. VORTEX's source editor is able to support some notion of language structure, but the internal representation is intended for efficient code generation. The parsing of a source document under the current VORTEX prototype is non-incremental. In direct-manipulation systems that maintain only the target view, efficient reparsing is not an issue because no syntactic constructs are present.

Another important issue concerns efficient code generation. Keeping up with the user's typing pace cannot be accomplished by a speedup in reparsing alone. Code generation in document processing involves extensive computation, especially when high quality output is desired. As indicated previously, many direct-manipulation systems trade quality for directness simply because hyphenation and pagination require too much computation for a document to be formatted on the fly. Hence these direct-manipulation systems become galley-based and rely on some off-line processors to produce the final result. In Lilac, which expects the user to do most of the editing in the target view, hyphenation is avoided to save computation time. VORTEX does not suffer from this immediate-execution syndrome. Users of VORTEX are expected to do most of the fine-grain editing in the source view. A delayed-execution model is employed so that formatting quality does not need to deteriorate.

As mentioned above, immediate execution is mandatory under direct manipulation. For systems maintaining only the target view, what is involved is only the target representation itself and some internal representation. Since the two representations must have a close correspondence for the system to function well, both representations can be updated coherently. The situation is more complex in multiple-representation systems. In addition to the target and the internal representations, a source representation must be taken into account in the transformation. There are two possibilities:

1. The internal representation is updated and the target representation is recreated by evaluating the new internal representation. The corresponding source representation is generated according to some *declarative specification.* Using the notation developed in Chapter 4, this transformation scheme can be abbreviated as $T \leftrightarrow O \rightarrow S$.

2. The updates first propagate to the source representation according to some *procedural specification.* The new source representation is re-evaluated, which creates new internal representation, and the new target representation corresponding to the manipulation is reflected. Under our notation, this transformation is $T \rightarrow S \rightarrow O \rightarrow T$.

The subtle difference here is the explicit re-evaluation of the source representation. In the declarative case, such explicit re-evaluation is not required; in the procedural case, the source representation is the basis of reproducing the target image.

In pure constraint-based systems, transformations in both the forward and backward directions are fully symmetric. The natural choice is the declarative approach. Interestingly, Tweedle, Lilac, and VORTEX all follow the procedural approach. The complexity of their underlying tasks prevent the mappings from being specified as declarative constraints. Although the three multiple-representation systems have varying emphases, in terms of functionality, on the role of their dual representations (VORTEX emphasizes on the source view, Lilac emphasizes on the target view, while Tweedle's two views are more balanced), they all follow $T \rightarrow S \rightarrow O \rightarrow T$ as their underlying model of immediate execution.

Another point of comparison is the use of Lisp under INF⊕R versus its use under VORTEX. INF⊕R is a document-processing system based exclusively on Lisp in that editing abstractions as well as formatting abstractions are programmed in Lisp. In VORTEX, Lisp is used to encapsulate editing abstractions, including those introduced by reverse mapping, while formatting abstractions are programmed in TEX. Here, expressing formatting in Lisp or TEX has little substantive differences because INF⊕R was designed to use TEX's formatting model. As indicated above, INF⊕R's formatting primitives were named after those available in TEX. One noticeable variation is their syntax: the reverse-Polish syntax (with a number of parentheses) of Lisp versus the infix syntax of TEX.

INF⊕R also works under the delayed-execution model. However, INF⊕R and VORTEX differ in some fine points of their respective delayed-execution model. In VORTEX, updates to the formatting information are registered automatically. In INF⊕R, changes to the formatting information, which is encapsulated in Lisp, requires the user to explicitly force the registration. This introduces another level of indirection to the user-driven delayed-execution model.

9.3.3 Incremental Processing

The current VORTEX prototype supports incremental formatting on a per page basis. The primary focus is incremental code generation. Extensive state checkpointing is exercised to ensure that formatting can be carried out from any page in the document. Between checkpoints, a page is parsed in a non-incremental fashion. Unlike Lilac, which performs parsing/unparsing incrementally, TEX's macro-oriented language construct makes incremental processing at this level unrealistic. Incremental redisplay is not supported in the target editor under the current VORTEX prototype. This is tolerable because in most cases the redisplay is driven as a result of delayed execution from the source editor. In the future, as more functionality is extended in the target editor, a more efficient redisplay algorithm would be necessary.

9.3.4 Integration Mechanisms

Chapter 8 identified four layers of integration that must be considered in a document development system. The lowest layer of integration concerns system organization. Here, VORTEX is rather unique because it is based on a distributed framework. As such, VORTEX exploits parallelism among the principal trio, while their simple protocols based on both asynchronous message passing and synchronous RPC alleviate complex scheduling problems. A distributed system like VORTEX also allows remote computation power to be utilized, which is an important advantage under a typical networked workstation environment. None of the other document processing systems are distributed.

The next layer of integration is related to multiple representations and inter-representation transformations, which has been covered in Sections 9.3.1 and 9.3.2. One layer up is the integration of composite objects. Support for compound documents in included in the VORTEX design (see Chapter 5) but is not implemented in the current VORTEX prototype. For instance, although the rendering of POSTSCRIPT graphics is not available in the current VORTEX prototype, the hooks are in place to support it.

By design, VORTEX's integration of text and graphics is based on single-level cut-and-paste. To the text formatter, a piece of graphics, or any non-textual object, encapsulated by `\special` is simply a blob of glue specified by its vertical dimension. All the formatter does in response to this is to skip that much space in its galley. It is the rendering program's responsibility to display the actual image. More attributes can be incorporated in `\special` so that the placement and size of a picture can be adjusted by the rendering program. A good example of an extension to `\special` can be found in the *Psfig* package [16,48]. To the graphics imaging server, text included in a picture is not subject to TEX formatting.

The strengths of this one-level cut-and-paste model are its simplicity and extensibility. Because it is simple, all classes of non-textual objects can be handled in a similar fashion: the target editor sends the code encapsulated in `\special` to its corresponding server and renders the returned raster image. It can be extended to support a new class of objects without any modification to the formatter or the source editor.

A shortcoming of this simple model is that objects cannot be arbitrarily nested. Furthermore, the processing of text is non-uniform because text in graphics is processed by the graphics imaging server, instead of by the main formatter. Since text appearing in graphics is largely used as annotations, complex formatting is not absolutely necessary. Font inconsistency is a related problem, which can be solved by requiring TEX to use PostScript fonts, or vice versa. The real problem of processing inconsistency emerges when mathematical formulas are needed as annotations. The graphics imaging server is generally incapable of handling such delicate text formatting details.

By contrast, processing inconsistency does not appear to be a problem under the pipelining model of *troff* and assorted preprocessors. Under the pipelining model, each preprocessor handles its own data, which may reside anywhere in a document. However, this approach also has the limitation that arbitrary nesting is not allowed. The *troff* pipeline is a one pass effort. Arbitrary object nesting requires more intricate exchange of information among the processors involved, which is not possible under a single non-interactive pipeline.

Several monolithic document processing systems, such as EZ, Quill, and Diamond, have been designed to solve the arbitrary object nesting problem. In EZ, the interaction is built-in as part of the Andrew toolkit object class structure. Initiated from the top-level interaction manager, input events are passed down the object hierarchy for processing, while drawing events are handed up, so that a parent can coordinate conflicting requests from its children. Through this calling convention, objects can exchange their attributes in allocating the necessary space on the viewing surface, which enables their arbitrary nesting. Quill also supports a top-level shell that serves as the event manager. Each class of objects corresponds to an "editor" which implements a set of elements. Each editor must implement a special *frame* elements whose purpose is to specify a rectangular region that contains elements belonging to a foreign editor. Hence a frame element in Quill marks a transition of context. The top-level shell, together with the transitional frame element, enables Quill to handle arbitrarily nested objects. In Diamond, a top-level routine called the *framework* is responsible for dispatching input events to various "media editors" and for coordinating painting requests queued for display. Each object occupies a rectangular region, and a layout algorithm is used to gather objects into *frames*, and ultimately into composites, which are then painted on a window pane. In addition to supporting arbitrary object nesting, EZ, Quill, and Diamond all claim to be extensible in that to support a new class of objects would only cause minimal perturbation in the existing routines.

The last layer of integration is the interface to external processors that constitute the multi-dimensional task domain. In VORTEX, this level of integration is built on top of the source editor's Lisp programming subsystem. One unique feature about the use of Lisp is its role as a gluing mechanism for the integration of not only external processors, but their associated interactive activities and job control as well. These pre- and post-processing tasks are often ignored by most interactive document processing systems. However, if a system were to be considered for complete document development, it must incorporate these tasks in an integrated and coherent environment. The VORTEX approach has achieved this level of integration successfully.

Chapter 10

Conclusions and Future Work

Document development is a complex undertaking that includes a wide spectrum of tasks. A software environment supporting complete document development must take all of them into account. The VORTEX project is rather ambitious in that it not only tries to create an environment to support these tasks, but to present a document in multiple representations so that the user can take advantage of the high-level abstractions and extensibility of a source language as well as the prompt visual response provided by direct manipulation. VORTEX is not the only system that recognizes the importance of multiple representations, but its underlying source language being the macro-oriented TEX makes it a special and interesting system.

This dissertation research has made a number of contributions. The separation of task domain from representation domain creates an effective framework for modeling multiple-representation systems. The design methodology derived from this framework for multiple-representation document processing systems, as described in Chapter 4, is useful in the conceptualization of a system's functionality in terms of the representations and their transformations.

A VORTEX prototype has been implemented as a distributed application. It has proved that a hybrid approach to document development based on the combination of a source language and direct manipulation is feasible. The performance of the system is reasonable for editing and display, while the sensation of directness is achieved by incremental formatting. Although its target-level functionality is somewhat limited and POSTSCRIPT graphics is not incorporated yet, all the necessary support for these extensions are in place.

VORTEX's incremental formatting strategies concentrate on efficient code generation. These strategies have been presented in a generic way and can be applied to other document formatting situations. The implementation is based on augmenting an existing non-incremental program. This technique may have some implications to software engineering, as existing non-incremental programs comprise the bulk of the world's software and the VORTEX experience is helpful in converting them to work under an interactive environment.

10.1 Possible Enhancements

The current VORTEX prototype can still benefit from the following enhancements:

Heap Consumption

A multiple-representation system is built at the expense of extensive dynamic memory consumption. Section 9.1.3 indicated several schemes to improve the VORTEX incremental formatter's heap utilization, all of which would improve VORTEX's capacity and performance.

Permanent Checkpointing

Presently, the VORTEX incremental formatter's state checkpointing is only used for warm start situations. All the state checkpoints are removed when a VORTEX session closes. It would be desirable to checkpoint the state at the end of the session so that the world can be restored when the system begins from a cold start again.

Incorporating Graphics

As mentioned above, the necessary support for rendering and selecting graphics is in place. A PostScript interpreter has also been built by the VORTEX group. A logical extension to the target editor is to be able to establish a connection with the PostScript server and display graphics. A step further, which has been proved feasible in an earlier experiment, is to develop a more elaborate rendering protocol between the target editor and the PostScript imaging server so that dynamic animated pictures can also be displayed on the same viewing surface.

Porting the Integration Software

This involves porting the extensive Lisp-based task integration software from Emacs to VORTEX's resident Lisp subsystem which is likely to be based on Common Lisp [68] as part of the future plan. The integration software allows pre- and post-processing facilities and their associated interactive activities to be integrated coherently with VORTEX's principal trio. These programs have been in use for a few years and have proved to be quite reliable.

Target-Level Functionality

The customization mechanism is in place for the repertoire of target-level operations and associated reverse mapping routines to be expanded. One would need to write programs in Lisp that encapsulates the TEX commands corresponding to the semantics of a target-level operation.

Layout Editor

Related to target-level functionality, a layout editor can be built that would allow the user to specify a document's layout by direct manipulation. The corresponding TEX code can be generated through the same reverse mapping facility.

Hiding Procedurality

Still related to target-level functionality, one can create encapsulated style packages, based on those supported by LATEX, for instance, and effectively make TEX's notion of procedurality transparent under VORTEX.

Exploiting Procedurality

As the opposite of the previous enhancement, one can create an interface that would allow the user to control the fine placement of objects. It is often desirable that the white space (glue) inserted between text be measurable on the target editor and that the user has a handle on controlling it. With minor enhancements to the current protocols, this can be made possible in VORTEX.

Empirical Study

In essence, the current VORTEX prototype is a test-bed upon which a number of experiments can be conducted. A most interesting study is to derive test cases that would show, from a user's standpoint, whether a two view document editor is better than a single view system.

10.2 Future Directions

A number of plans have been made to carry VORTEX into the next stage. These include

Integration with Pan

It involves replacing VORTEX's source editor with Pan [13], a multilingual structure-oriented editor based on Common Lisp [68]. The *component technology* will be investigated here to understand how separately developed software can be integrated.

Integration with Symbolic Manipulation Systems

The idea here is that a symbolic manipulation system can utilize VORTEX as the front-end display, while VORTEX can utilize it as the back-end manipulator for mathematical formulas. With these connections, a VORTEX document will become active; its mathematical equations are not only correctly formatted, but may be evaluated and validated. Although the current VORTEX prototype cannot be connected with a symbolic manipulator directly due to their differences in representing mathematical terms, this problem is interesting enough to pursue further.

Redesigning A Source Language

The macro-oriented nature of TEX has contributed a number of challenging research problems in the design of VORTEX. The technique developed has corrected TEX's lack of interactive support while retaining the output quality of its algorithms. An interesting extension of this work would be to design a new source language and an associated multiple-representation environment that incorporates both TEX's superb text (including mathematics) processing capability with the powerful graphics capability of POSTSCRIPT.

Lilac appears to be a reasonable first step, but its source language is rather restrictive by our standard. It is purely functional, so the support for pre- and post-processing tasks is problematic. Furthermore, there is no provisions for graphics whatsoever. In other words, there are still a number of research problems to be tackled along this direction.

Compound Documents and Hypertext

It would be interesting to study the implications a multiple-representation system like VORTEX would have on a multimedia and/or hypertext system. HyperCard offers some clues that the hybrid paradigm is rather powerful. How does it generalize to a more elaborate authoring environment? The VORTEX experience provides a good starting point for research along this line.

Appendix A

Formatter $\leftrightarrow$ Source Editor Protocol Specification

```
#ifndef _FSCOMM_
#define _FSCOMM_
/*
 *  Formatter and source editor communications.
 *
 *  Packet format:
 *
 *      u_short    request           request code as defined below
 *      u_short    datalen           length of rest of packet
 *      u_long     id                file ID
 *      <datalen bytes>              rest of packet; zero or more bytes
 *
 *  In the request definitions below, long is four bytes, short
 *  is two bytes, char is one byte and an array is zero or more
 *  of those objects as specified by the datalen field in the
 *  packet header in bytes.
 *
 *  FSC_VERSION should be bumped each and every time this file is
 *  changed.  This is to insure that the programs will not try to
 *  communicate with different protocol specifications.
 */
#define FSC_VERSION                 14

/*
 *  FSC_FORMAT
 *      string    filename          root TeX file name
 *
 *  S -> F
 *  Format document rooted at master file fid.
 */
#define FSC_FORMAT                  (GLC_LASTREQ + 1)

/*
```

```
 *  FSC_CLOSEDOC
 *
 *  S <-> F:  close a document.
 *  Invalidate the document and free all its resources.
 */
#define FSC_CLOSEDOC                    (FSC_FORMAT + 1)

/*
 *  FSC_OPENFILE
 *      u_long    data[]                contents of TeX file
 *
 *  S -> F:  open a source file (fid given in packet header) with
 *  contents in data array.  The same file, if already exists, gets trashed.
 *
 *  Each entry in the data array is a u_long:
 *
 *                 18              7       7
 *         ------------------------------------
 *         |   ID per letter  |  fid  | ascii |
 *         ------------------------------------
 */
#define FSC_OPENFILE                    (FSC_CLOSEDOC + 1)

/*
 *  FSC_CLOSEFILE
 *
 *  S <-> F:  close a file (fid given in packet header).
 *  Invalidate the file ID specified and free all resources
 *  used by the specified file.
 */
#define FSC_CLOSEFILE                   (FSC_OPENFILE + 1)

/*
 *  These constants define insertion/deletion points for the commands
 *  below so that they may be more flexible (and perhaps easier for the
 *  formatter/proof editor to generate).
 */
#define IRS_CUR                         0
#define IRS_BOF                         1
#define IRS_EOF                         2

/*
 *  FSC_INSERT
 *      u_long    flag                  IRS_CUR, current point as reference
 *                                      IRS_BOF, BOF as reference
 *                                      IRS_EOF, EOF as reference
 *      long      offset                offset to reference point,
 *                                      +forward, -backward
 *      u_long    count                 number of chars to insert
 *      u_long    data[]
 *
 *  S -> F:  insert data in front of insertion point in file
 *  (fid given in packet header).
 *  The insertion point is determined by stepping offset nodes relative
 *  to the reference point, which is determined by flag.
 *
 */
#define FSC_INSERT                      (FSC_CLOSEFILE + 1)

/*
 *  FSC_DELETE
 *      u_long    fla                   IRS_CUR, current point as reference
 *                                      IRS_BOF, BOF as reference
 *                                      IRS_EOF, EOF as reference
 *      u_long    offset                nonnegative offset to reference point
 *      u_long    count                 number of chars to delete
 *
 *  S -> F:  delete count characters starting from deletion point in file
 *  (fid given in packet header).
 *  The deletion point is determined by steping offset nodes relative
 *  to the reference point, which is determined by flag.
 *
 */
#define FSC_DELETE                      (FSC_INSERT + 1)

/*
 *  FSC_TEXINPUT
 *      string    filename              file name as known to \input
 *
 *  F -> S:  ask for a file.
 *  Sent when the file is not found in the formatter's IR.
 */
#define FSC_TEXINPUT                    (FSC_DELETE + 1)

/*
 *  FSC_TEXOUTPUT
 *      u_long    filelen               length of file name
 *      string    filename              output filename string
 *      string    content               file's contents
 *
 *  F -> S:  write for a file to the source editor's disk space.
 */
#define FSC_TEXOUTPUT                   (FSC_TEXINPUT + 1)

/*
 *  FSC_TEXMESSAGE
 *      string    msg                   the message
 *
 *  F -> S:  report message from the formatter.
 *  Sent when TeX is verbose.
 */
#define FSC_TEXMESSAGE              (FSC_TEXOUTPUT + 1)

/*
 *  FSC_TEXERROR
 *      u_long    lineno            line number of error
 *
 *  F -> S:  report an TeX error in LINENO of FID.
 *  An error message precedes this by TEXMESSAGE.
 */
#define FSC_TEXERROR                (FSC_TEXMESSAGE + 1)

#define FS_TGT2SRC                  0  /* given target id, return source id */
#define FS_SRC2TGT                  1  /* given source id, return target id */

/*
 *  FSC_EXECUTE
 *      <args>                      arguments to the primitive
 *
 *  S -> F:  execute primitive id (given in header)
 */
#define FSC_EXECUTE                 (FSC_TEXERROR + 1)

/*
 *  FSC_RETURN
 *
 *  F -> S:  return result of executing primitive id (given in header)
 *      long        code            return code
 *      <more data>                 depending on each primitive.
 */
#define FSC_RETURN                  (FSC_EXECUTE + 1)


#define FSC_LASTREQ                 FSC_RETURN

#endif !_FSCOMM_
```

Appendix B

Formatter $\leftrightarrow$ Target Editor Protocol Specification

```
#ifndef _FTCOMM_
#define _FTCOMM_
/*
 *  Formatter and target editor editor communications.
 *
 *  Packet format:
 *
 *      u_short    request            request code as defined below
 *      u_short    datalen            length of rest of packet
 *      u_long     pageno             page number (physical)
 *      <datalen bytes>               rest of packet; zero or more bytes
 *
 *  In the request definitions below, long is four bytes, short
 *  is two bytes, char is one byte and an array is zero or more
 *  of those objects as specified by the datalen field in the
 *  packet header in bytes.
 *
 *  Note the two constants below.  FTC_VERSION should be bumped
 *  each and every time this file is changed.  This is to insure
 *  that the programs will not try to communicate with different
 *  protocol specifications.
 */

#define FTC_VERSION                   15


/*
 *  FTC_SENDPAGE
 *
 *  F <- T  for page information
 *  Send page information to the target editor to preview that page.
 */
```

```
#define FTC_SENDPAGE                    (GLC_LASTREQ + 1)


/*
 *  FTC_PAGEINFO
 *      long        globalmag           global mag (same for every page)
 *      long        count[10]           ten count registers
 *      short       fc                  number of fonts used
 *      _Font       hdr                 actual font header        ---|
 *      char        name[hdr->ln]       actual font name             |
 *      ...         ...                                              |---> fc
 *      _Font       hdr                 actual font header           |
 *      char        name[hdr->ln]       actual font name          ---|
 *      <data stream>                   flattened page tree
 *
 *  T -> P  send page information
 *  This starts a stream of page information, which consists of a header,
 *  a stream of positioning commands plus structure and char boxes.
 *  Positioning commands define where the next box in the stream
 *  is to be placed relative to the current box, and is a u_short.
 *  There may not be a right child to the PageBox, nor may there be
 *  children to the CharBoxes.
 *  The data stream (flattened page tree) looks like the following:
 *      <box_type>        (1 byte)
 *      [BOX]             (5 words)
 *      <box_type>        (1 byte)
 *      [BOX]             (5 words)
 *      ......
 *      <box_type>        (1 byte)
 *      [BOX]             (5 words)
 *      <EOP>             (1 byte)
 */

#define FTC_PAGEINFO                    (FTC_SENDPAGE + 1)


/*
 *  FTC_PAGEOKAY
 *
 *  F -> T  inform the target editor that the page requested is still valid
 *  in its work space.
 *
 */

#define FTC_PAGEOKAY                    (FTC_PAGEINFO + 1)


/*
 *  FTC_PAGENOTFOUND
 *
 *  F -> T  inform the target editor that the page requested does not exist.
 *
 */

#define FTC_PAGENOTFOUND                (FTC_PAGEOKAY + 1)


/*
 *  Structure for font specification.
 */

typedef struct _font {
        long                    size;       /* font at size */
        short                   fid;        /* font identifier. */
        short                   ln;         /* font name length. */
} _Font;


#define FT_PAGE                 80          /* 'P': begnning of a page box */
#define FT_PAR                  81          /* 'Q': a paragraph box */
#define FT_WORD                 87          /* 'W': a word box */
#define FT_CHAR                 67          /* 'C': a character box */
#define FT_RULE                 82          /* 'R': a character box */
#define FT_SPECIAL              83          /* 'S': a special box */
#define FT_EOP                  69          /* 'E': begnning of a page box */

typedef struct _tbox {                      /* terminal char box in trnasmission */
        long                    id;         /* box id */
        long                    ch;         /* char in ascii */
        long                    ft;         /* font number */
        long                    xb;         /* baseline x coordinate */
        long                    yb;         /* baseline y coordinate */
} _Tbox;

typedef struct _nbox {                      /* nonterminal box in trnasmission*/
        long                    id;         /* box id */
        long                    xc;         /* upperleft cornor x coordinate */
        long                    yc;         /* upperleft cornor x coordinate */
        long                    wd;         /* horizontal width */
        long                    ht;         /* vertical height */
} _Nbox;

#define FTC_LASTREQ             FTC_PAGENOTFOUND

#endif !_FTCOMM_
```

Appendix C

Internal Representation Specification

```
#ifndef _ALLIR_
#define _ALLIR_

/*
 *  IRf is organized as an ordinary binary tree.
 *  IRs is organized as a doubly-linked list.
 *  An empty file is one that has a single node
 *  of struct _char containing EOF.
 */

typedef struct _file {
        char                *fn;        /* filename */
        unsigned long       id;         /* unique file identifier */
        struct _file        *lt;        /* left sibling */
        struct _file        *rt;        /* right sibling */
        struct _char        *hd;        /* head of IRs */
        struct _char        *pt;        /* current update point within IRs */
} _File;


/*
 *  Node types, these all have their own structures above, among
 *  the interior nodes of the IRi or the text nodes for character
 *  leaves.  These numbers must all be distinct, of course, as they
 *  are the only way of telling what (size) structure makes up a
 *  particular node.
 */

#define NODE_NONE           0           /* unknown node */

#define NODE_CHAR           10          /* a character leaf */

#define NODE_LIG            20          /* a ligature node */
```

```
#define NODE_WORD        21          /* an word delimiting node */
#define NODE_PAR         22          /* an paragraph, as with \par */

#define NODE_DEF         30          /* a macro def node */
#define NODE_FDEF        31          /* a font def  node */
#define NODE_CSEQ        32          /* a macro invocation */
#define NODE_FONT        33          /* a font invocation */
#define NODE_RULE        34          /* a rule node */
#define NODE_SYMBOL      35          /* a TeX symbol */
#define NODE_SPECIAL     36          /* \special node */

#define NODE_MATH        40          /* a math shift node (group) */
#define NODE_DISPLAY     41          /* a display math node */

#define NODE_GROUP       50          /* group node, as with {} */

#define NODE_INPUT       60          /* an \input node */

#define NODE_SPACE       70          /* a space/comment node */

/*
 *  One of these structure is used for all nodes in the IRi that
 *  are not leaf nodes.  Leaf nodes (letters) are really text
 *  structures in the IRs (not the same as the interior nodes).
 *  All these structures have a first byte which tells their
 *  type so that one may guarantee being able to tell the type
 *  by examining the first byte.  Note that each different node
 *  type has a different (sized!) structure, but all references
 *  are through the (pointer) type, so this is somewhat less
 *  of a problem than it might otherwise be.
 *
 *  The tree is stored as a sort of directed graph.  Since
 *  nodes may have arbitrary numbers of children, a pointer
 *  is maintained to the linked list of children, although each
 *  child has a direct ``up'' pointer to the parent.  The
 *  list of children is doubly-linked so that one may reach
 *  any sibling quickly.  Some nodes, however, don't use
 *  this scheme.  Nodes that sit directly on the IRs (don't
 *  have interior nodes for children) have begin and end
 *  pointers to the IRs directly.  An example of this is the
 *  (struct _word) for the word node type.
 *
 *  The (struct _node) is the basis for all interior tree
 *  nodes, they are all the same in the fields that appear
 *  in the (struct _node); type dependent information
 *  appears later on.  All the other node types contain a
 *  (struct _node) so that changes will be less likely to
 *  mess something up, but to avoid extra work, all the
 *  fields are defined so as to make them appear to be
 *  in the top level structure.
```

```
*/

typedef struct _node {
        char                nd_type;    /* type of node in tree */
        char                nd_char;    /* char in ACII */
        struct _node        *nd_up;     /* parent of this node */
        struct _node        *nd_lt;     /* left (prevous) child */
        struct _node        *nd_rt;     /* right (next) child */
} _Node;

#define _ty                 _com.nd_type
#define _ch                 _com.nd_char
#define _up                 _com.nd_up
#define _lt                 _com.nd_lt
#define _rt                 _com.nd_rt
#define _lc                 _com.nd_up
#define _pb                 nd_up->nd_up->nd_up
#define _qb                 nd_up->nd_up
#define _wb                 nd_up
#define _rb                 nd_rt

typedef struct _char {
        struct _node        _com;       /* common node info. */
        struct _node        *_re;       /* box reference in IRt */
        unsigned long        _id;       /* unique identifier */
} _Char;

typedef struct _unode {
        struct _node        _com;       /* common node info. */
        struct _node        *_re;       /* box reference in IRt */
        struct _node        *_dn;       /* beginning in IRs */
} _Unode;

typedef struct _group {
        struct _node        _com;       /* common node info. */
        struct _node        *_dn;       /* list of children */
        short               _level;     /* nesting level */
} _Group;

typedef struct _cseq {
        struct _node        _com;       /* common node info. */
        struct _node        *_re;       /* crodd reference to IRt */
        struct _node        *_bon;      /* beginning of cs name */
        struct _node        *_eon;      /* end of cs name */
        struct _node        *_boc;      /* beginning of char params */
        struct _node        *_eoc;      /* end of char params */
        struct _node        *_def;      /* defined IRi */
} _Cseq;

typedef struct _math {
        struct _node        _com;       /* common node info. */
        struct _node        *_re;       /* box reference in IRt */
        struct _node        *_dn;       /* list of children */
} _Math;

typedef struct _input {
        struct _node        _com;       /* common node info. */
        struct _node        *_dn;       /* list of children */
        struct _node        *_bon;      /* name of file input from */
        struct _file        *_fp;       /* pointer to IRf */
} _Input;

typedef struct _space {
        struct _node        _com;       /* common node info. */
        struct _node        *_boc;      /* beginning in IRs */
        struct _node        *_eoc;      /* end in IRs */
} _Space;


/*
 * Box structure
 */

#define BOX_NONE            0           /* NULL */
#define BOX_CHAR            67          /* C: char box */
#define BOX_LIG             76          /* L: ligature box */
#define BOX_HYPH            72          /* H: hyphen box */
#define BOX_Explicit        69          /* E: text introduced by an */
                                        /*    explicit macro invocation */

#define BOX_Implicit        73          /* I: text introduced by an */
                                        /*    implicit macro invocation */

#define BOX_RULE            82          /* R: box of rule */
#define BOX_SPECIAL         83          /* S: box of special */
#define BOX_WORD            87          /* W: box of single word in IRt */
#define BOX_PAR             81          /* Q: box of a paragraph in IRt */
#define BOX_PAGE            80          /* P: box of whole page in IRt */

typedef struct _cbox {                  /* terminal char box */
        struct _node        _com;       /* common node info */
        struct _node        *_re;       /* pointer to IRs+IRi */
        unsigned long       _id;        /* box id */
        long                _xb;        /* baseline x coordinate of box */
        long                _yb;        /* baseline y coordinate of box */
        unsigned short      _ft;        /* current font number */
} _Cbox;

typedef struct _sbox {                  /* terminal special box */
        struct _node        _com;       /* common node info */
        struct _node        *_re;       /* pointer to IRs+IRi */
        unsigned long       _id;        /* box id */
        long               _xb;         /* baseline x coordinate of box */
        long               _yb;         /* baseline y coordinate of box */
        unsigned short     _ta;         /* total no of chars in arg */
        char               *_ap;        /* point to argument list */
} _Sbox;

typedef struct _rbox {                  /* terminal rule box */
        struct _node       _com;        /* common node info */
        struct _node       *_re;        /* pointer to IRs+IRi */
        unsigned long      _id;         /* box id */
        long               _xc;         /* upper-left cornor x coordinate */
        long               _yc;         /* upper-left cornor y coordinate */
        long               _wd;         /* horizontal width of box */
        long               _ht;         /* vertical height+depth of box */
} _Rbox;

typedef struct _ubox {                  /* nonterminal paragraph/word box */
        struct _node       _com;        /* common node info */
        struct _node       *_re;        /* pointer to IRs+IRi */
        struct _node       *_dn;        /* down pointer */
        unsigned long      _id;         /* box id */
        long               _xc;         /* upper-left cornor x coordinate */
        long               _yc;         /* upper-left cornor y coordinate */
        long               _wd;         /* horizontal width of box */
        long               _ht;         /* vertical height+depth of box */
} _Ubox;

#define FT_MAX  64

typedef struct _pbox {                  /* nonterminal page box */
        struct _node       _com;        /* common node info */
        struct _node       *_ec;        /* pointer to extended context */
        struct _node       *_dn;        /* down pointer */
        unsigned long      _id;         /* context id */
        long               _xc;         /* upper-left cornor x coordinate */
        long               _yc;         /* upper-left cornor y coordinate */
        long               _wd;         /* horizontal width of box */
        long               _ht;         /* vertical height+depth of box */
        long               _ok;         /* TRUE if already sent */
        unsigned short     _no;         /* physical page number */
        unsigned short     _tb;         /*  number of boxes */
        unsigned short     _tf;         /* number of fonts used */
        unsigned short     _tn;         /* number of chars in font names*/
        unsigned short     _ts;         /* number of \special's*/
        unsigned short     _ta;         /* number of chars in \special arg */
        long               _ct[10];     /* TeX's 10 count registers */
        unsigned short     _ft[FT_MAX];/* used fonts */
} _Pbox;

#endif !_ALLIR_
```

Bibliography

[1] James O. Achugbue. On the line breaking problem in text formatting. In *Proc. of ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 117–122, Portland, Oregan, June 8–10 1981. Available as *SIGPLAN Notices* 16(6) or *SIGOA Newsletter* 2(1–2).

[2] Adobe Systems Incorporated, Palo Alto, California. *Adobe Illustrator User's Manual*, 1987.

[3] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

[4] Aldus Corporation, Seattle, Washington. *PageMaker User Manual and Reference Manual, Version 2.0*, March 1987.

[5] Todd Allen, Robert Nix, and Alan Perlis. PEN: A hierarchical document editor. In *Proc. of ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 74–81, Portland, Oregan, June 8–10 1981. Available as *SIGPLAN Notices* 16(6) or *SIGOA Newsletter* 2(1–2).

[6] W. Appelt and G. Ritcher. The formal specification of the structures of the oda standard. In *Proc. of EP88 – Internal Conference on Electronic Publishing, Document Manipulation, and Typography*, Nice, France, April 1988.

[7] Apple Computers, Inc. *Inside the Macintosh, Volume I – III.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[8] Apple Computers, Inc., Cupertino, California. *MacDraw Manual*, 1984.

[9] Apple Computers, Inc., Cupertino, California. *MacPaint Manual*, 1984.

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

[11] Paul J. Asente. Editing graphical objects using procedural representations. Research Report 87/6, DECWRL, Palo Alto, California, November 1987. Also available as Ph.D. thesis, Computer Science Department, Stanford University, 1987.

[12] Robert M. Ayers, J. T. Horning, Butler W. Lampson, and J. G. Mitchell. Interscript: A proposal for a standard for the interachange of editable documents. Technical report, Xerox Palo Alto Research Center, Plao Alto, California, 1984.

[13] Robert A. Ballance. Design of the Pan language-based editor. Computer Science Division, University of California, November 1985. Unpublished manuscript.

[14] Robert A. Ballance. *Higher-Level Language-Based Editors.* PhD thesis, UC, Berkeley, California, 1988. To appear.

[15] Robert A. Ballance, Michael L. Van De Vanter, and Susan L. Graham. The architecture of Pan I. Technical Report 88/409, UC, Berkeley, California, September 1987.

[16] Ned Batchelder and Trevor Darrell. Psfig – a Ditroff preprocessor for PostScript figures. In *Proc. of 1987 USENIX Summer Conference*, pages 31–42, Phoenix, Arizona, June 8–12 1987.

[17] Jon J. Bentley, L. W. Jelinski, and Brian W. Kernighan. CHEM: A program for phototypesetting chemical structure diagrams. Computer Sciecne Technical Report No. 122, AT&T Bell Laboratories, Murray Hill, New Jersey, 1986.

[18] Jon J. Bentley and Brian W. Kernighan. Tools for printing indexes. *Electronic Publishing*, 1(1), June 1988. Also available as Computer Science Technical Report No. 128,AT&T Bell LaboratoriesMurray Hill, New Jersey, October 1986.

[19] Jon L. Bentley and Brian W. Kernighan. GRAP — a language for typesetting graphs. *Communications of the ACM*, 29(8):782–792, August 1986.

[20] Abhay Bhushan and Michael Plass. The Interpress page and document description language. *IEEE Computer*, 19(6):72–77, June 1986.

[21] Eric Bier and Maureen Stone. Snapping dragging. *ACM Computer Graphics*, 20(3):233–240, August 1986.

[22] A. D. Birrel and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[23] Lawrence Bohn and David Weinberger. Why not have it all. *UNIX Review*, 5(7):29–34, July 1987.

[24] Alan H. Borning. Defining constraints graphically. In *Proc. of ACM SIGCHI'86 Conference*, pages 137–143, Boston, Massachusetts, April 1986.

[25] Alan H. Borning and Robert A. Duisberg. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics*, 5(4):345–374, October 1986.

[26] David F. Brailsford. Interaction vs. abstraction in the preparation of high-quality text and graphics. In J. J. H. Miller, editor, *Proc. of 1st International Conference on Text Processing Systems*, pages 94–97, Dublin, Ireland, Oct. 24–26 1984.

[27] Kenneth P. Brooks. *A Two View Document Editor with User-Definable Document Structure.* PhD thesis, Computer Science Department, Stanford University, Stanford, California, in preparation.

[28] Luis Felipe Cabrera and Jim Wyllie. QuickSilver distributed file services: An architecture for horizontal growth. In *Proc. of 2nd IEEE Conference on Computer Workstations*, pages 23–37, Santa Clara, California, March 1988.

[29] Donald D. Chamberlin. Document convergence in an interactive formatting system. *IBM Journal of Research and Development*, 31(1):58–72, January 1987.

[30] Donald D. Chamberlin, O. P. Bertrand, Michael J. Goodfellow, James C. King, Donald R. Sultz, Stephen J. P. Todd, and Bradford W. Wade. JANUS: An interactive document formatter based on declarative tags. *IBM Systems Journal*, 21(3):250–271, 1982.

[31] Donald D. Chamberlin and Charles F. Goldfarb. Graphic applications of the standard generalized markup language (sgml). *Computers and Graphics*, 11(4), 1987. Also avialable as Techincal Report RJ 5440 (55569), IBM Almaden Research Center, San Jose, California.

[32] Donald D. Chamberlin, Helmut F. Hasselmeier, Allen W. Luniewski, Dieter P. Paris, Bradford W. Wade, and Mitch L. Zolliker. Quill: An extensible system for editing documents of mixed type. In *Proc. of the 21st Hawaii International Conference on System Sciences*, pages 317–326, Kailua-Kona, Hawaii, Jan 5–8 1988. Also available as Technical Report RJ 5775 (58114), IBM Almaden Research Center, San Jose, California, Aug. 1987.

[33] Donald D. Chamberlin, Helmut F. Hasselmeier, and Dieter P. Paris. Defining document styles for WYSIWYG processing. In *Proc. of EP88 – Internal Conference on Electronic Publishing, Document Manipulation, and Typography*, Nice, France, April 1988. Also available as Technical Report RJ 5812 (58542), IBM Almaden Research Center, San Jose, California, Aug. 1987.

[34] Eugene Charniak, Christopher K. Riesbeck, Drew V. McDermott, and James R. Meehan. *Aritficail Intelligence Programming*. Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 2nd edition, 1987.

[35] Pehong Chen. GNU Emacs *BibTeX-mode*. Technical Report 87/317, Computer Science Division, University of California, Berkeley, California, October 1986.

[36] Pehong Chen. GNU Emacs *TeX-mode*. Technical Report 87/316, Computer Science Division, University of California, Berkeley, California, October 1986.

[37] Pehong Chen, Yih-Farn Chen, and Wen-Mei Hwu. On the duality of distributed interprocess communication. In *Proc. of the 1984 International Computer Symposium*, Taipei, Taiwan, December 1984.

[38] Pehong Chen, John L. Coker, Michael A. Harrison, Jeffrey W. McCarrell, and Steven J. Procter. The VorTeX document preparation environment. In *Proc. of the 2nd European Conference on TeX for Scientific Documentation*, pages 32–24, Strasbourg, France, June 19–21 1986. Published as *Lecture Notes in Computer Science No. 236* by Springer-Verlag, 1986.

[39] Pehong Chen and Michael A. Harrison. Integrating noninteractive document processors into an interactive environment. Technical Report 87/349, Computer Science Division, University of California, Berkeley, California, April 1987. Submitted for publication.

[40] Pehong Chen and Michael A. Harrison. Index preparation and processing. *Software—Practice & Experience*, 1988. To appear.

[41] Pehong Chen and Michael A. Harrison. Multiple representation document development. *IEEE Computer*, 21(1):15–31, January 1988.

[42] Pehong Chen, Michael A. Harrison, John L. Coker, Jeffrey W. McCarrell, and Steven J. Procter. An improved user environment for TEX. In *Proc. of the 2nd European Conference on TEX for Scientific Documentation*, pages 45–54, Strasbourg, France, June 19–21 1986. Published as *Lecture Notes in Computer Science No. 236* by Springer-Verlag, 1986.

[43] Pehong Chen, Michael A. Harrison, and Ethan V. Munson. Enhancements to integrated bibliography management. Submitted for publication.

[44] E. Jeffrey Conklin. Hypertext: An introduction and survey. *IEEE Computer*, 20(9):17–41, September 1987. A more detailed version is available as Technical Report STP-356-86,Microelectronics and Computer Technology CorporationAustin, Texas-Dec. 1986.

[45] Cricket Software Incorporated, Philadelphia, Pennsylvania. *CricketDraw: Advanced Graphics with PostScript*, 1987.

[46] Malcolm Crowe, Clark Nicol, Michael Hughes, and David Mackay. On converting a compiler into an incremental compiler. *ACM SIGPLAN Notices*, 20(10):14–22, October 1985.

[47] Terrence R. Crowley, Harry C. Forsdick, Matt Landau, and Virginia M. Travers. The Diamond multimedia editor. In *Proc. of 1987 USENIX Summer Conference*, pages 1–18, Phoenix, Arizona, June 8–12 1987.

[48] Trevor Darrell. *Psfig/TEX Users Guide*. Department of Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania, July 1987. Available in the Psfig release.

[49] Susan A. Dart, Robert J. Ellison, Peter H. Feiler, and A. Nico Habermann. Software development environments. *IEEE Computer*, 20(11):18–28, November 1987.

[50] Robert J. Ellison and Barbara J. Staudt. The evolution of the GANDALF system. *The Journal of Systems and Software*, 5(2):107–119, May 1985.

[51] Stuart I. Feldman. Make — a program for maintaining computer programs. *Software—Practice & Experience*, 9(7):255–265, July 1979.

[52] Frame Technology Corporation, San Jose, California. *Frame Maker Reference Manual, Version 1.0*, February 1987.

[53] David Fuchs. Device independent file format. *TUGboat*, 3(2):14–19, October 1982.

[54] George W. Furnas. Generalized fisheye views. In *Proc. of ACM SIGCHI'86 Conference*, pages 16–23, Boston, Massachusetts, April 1986.

[55] Richard K. Furuta. *An Integrated, but not Exact-Representation, Editor/Formatter.* PhD thesis, Computer Science Department, University of Washington, Seattle, Washington, September 1986. Published as Technical Report No. 86-09-08.

[56] Richard K. Furuta, Jeffrey Scofield, and Alan Shaw. Document formatting systems: Survey, concepts, and issues. *ACM Computing Surveys*, 14(3):417–472, September 1982.

[57] Carlo Ghezzi and Dino Mandrioli. Incremental parsing. *ACM Transactions on Programming Languages and Systems*, 1(1):58–70, January 1979.

[58] Carlo Ghezzi and Dino Mandrioli. Augmenting parsers to support incrementalilty. *Journal of the ACM*, 27(3):564–579, March 1980.

[59] Joseph A. Goguen and José Meseguer. Order-sorted algebra solves the contructor-selector, multiple representation and coercion problems. In *Proc. 1987 III Symposium on Logic in Computer Science*, 1987.

[60] Charles F. Goldfarb. A generalized approach to document markup. In *Proc. of ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 68–73, Portland, Oregan, June 8–10 1981. Available as *SIGPLAN Notices* 16(6) or *SIGOA Newsletter* 2(1–2).

[61] Charles F. Goldfarb, editor. *Information Processing — Text and Office Systems — Standard Generalized Markup Language (SGML).* International Organization for Standardization, Geneva, Switzerland, 1986. International Standard ISO 8879.

[62] Michael J. Goodfellow. WHIM, the window handler and input manager. *IEEE Software*, 6(5):46–52, May 1986.

[63] Danny Goodman. *The Complete HyperCard Handbook.* Bantom Books, New York, New York, 1987.

[64] James Gosling. An editor-based user interface toolkit. In J. J. H. Miller, editor, *Proc. of 1st International Conference on Text Processing Systems*, pages 126–132, Dublin, Ireland, Oct. 24–26 1984.

[65] Richard D. Greenblatt, Jr. Thomas F. Knight, John Holloway, David A. Moon, and Daniel L. Weinreb. The LISP machine. In David R. Barstow et al., editor, *Interactive Programming Environments*, pages 326–352, New York, New York, 1984. McGraw-Hill Book Company.

[66] J. Gutknecht. Concepts of the text editor Lara. *Communications of the ACM*, 28(9):942–960, September 1985.

[67] J. Gutknecht and W. Winiger. Andra: The document preparation system for the personal workstation Lilith. *Software—Practice & Experience*, 14(1):73–100, January 1984.

[68] Guy L. Steele Jr. *Common Lisp, The Language.* Digital Press, Billerica, Massachusetts, 1984.

[69] Michael Hammer, Richard Ilson, Tim Anderson, Edward J. Gilbert, Michael D. Good, Bahram Niamir, Larry Rosentein, and Sandor Schoichet. The implementation of Etude, an integrated and interactive document production system. In *Proc. of ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 137–146, Portland, Oregan, June 8-10 1981. Available as *SIGPLAN Notices* 16(6) or *SIGOA Newsletter* 2(1–2).

[70] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. Recovery management in QuickSilver. In *Proc. of 11th ACM Symposium on Operating System Principles*, Austin, Texas, September 1987.

[71] Peter Hibbard. *User Manual for MINT: The Spice Document Preparation System, Version 2a(21).* Spice Project, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, April 1983.

[72] Alan Hoenig. TEX does windows — the conclusion. *TUGboat*, 8(2):211–215, July 1987.

[73] Gerard H. Holzman. Pico — a picture editor. *AT&T Bell Laboratories Technical Journal*, 66(2):2–13, March/April 1987.

[74] Wolfgang Horak. Office document architecture and office document interchange formats: Current status of international standardization. *IEEE Computer*, 18(10):50–60, October 1985.

[75] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct manipulation interfaces. In D. A. Norman and S. W. Draper, editors, *User-Centered System Design*, pages 87–124 (Chapter 5), Hillsdale, New Jersey, 1986. Lawrence Erlbaum Associates, Inc.

[76] Richard Ilson. An integrated approach to formatted document production. Master's thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts, 1980. Available as technical report MIT/CSL/TR-253.

[77] Imagen Corporation, Santa Clara, California. *Document Description Language (Revision 1.1): Reference Manual and Tutorial*, November 1986.

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

[79] Jeff Johnson and Richard J. Beach. Styles in document editing systems. *IEEE Computer*, 21(1):32–43, January 1987.

[80] Vania Joloboff. Trends and standards in document representation. In J. C. van Vliet, editor, *Proc. of the International Conference on Text Processing and Document Manipulation*, pages 107–124, University of Nottingham, April 14–16 1986. Published by the Cambridge University Press.

[81] William Joy. *An Introduction to Display Editing with Vi*, 1980. Appears in UNIX User's Manual.

[82] Gail E. Kaiser and Elaine Kant. Incremental parsing without a parser. *The Journal of Systems and Software*, 5(2):121–144, May 1985.

[83] Brian W. Kernighan. PIC — a language for typesetting graphics. In *Proc. of ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 92–98, Portland, Oregan, June 8–10 1981. A similar version under the same title appears in *Software: Experience and Practice*, 12(1), pp. 1–20, January 1982.

[84] Brian W. Kernighan. A typesetter-independent TROFF. Computer Science Technical Report No. 97, AT&T Bell Laboratories, Murray Hill, New Jersey, March 1982.

[85] Brian W. Kernighan. The UNIX document preparation tools – a retrospective. In *Proc. of 1st International Conference on Text Processing Systems*, pages 12–25, Dublin, Ireland, Oct. 24–26 1984.

[86] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. A correction appeared in Mathematical Systems Theory, 5(1): 95–96.

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

[88] Donald E. Knuth. A torture test for TEX, version 1.3. Technical Report STAN-CS-84-1027, Computer Science Department, Stanford University, Stanford, California, November 1984.

[89] Donald E. Knuth. *TEX: The Program*, volume B of *Computers & Typesetting*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[90] Donald E. Knuth and Michael F. Plass. Breaking paragraphs into lines. *Software—Practice & Experience*, 11(11):1119–1184, November 1982.

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

[92] Butler W. Lampson. *Bravo Manual.* Xerox Palo Alto Research Center, Palo Alto, California, 1978. Appears in *Alto User's Handbook*, Butler W. Lampson and Edward A. Taft (eds.).

[93] Frank Mark Liang. *Word Hyphenation by Computer.* PhD thesis, Computer Science Department, Stanford University, Stanford, California, June 1983. Available as technical report STAN-CS-83-977.

[94] Allen W. Luniewski. Intent-based page modeling using blocks in the Quill document editor. In *Proc. of EP88 - Internal Conference on Electronic Publishing, Document Manipulation, and Typography*, Nice, France, April 1988. Also available as Technical Report RJ 5811 (58541), IBM Almaden Research Center, San Jose, California, Aug. 1987.

[95] Manhattan Graphics Corporation, Valhalla, New York. *Ready,Set,Go! User Manual*, November 1986.

[96] Lee E. McMahon. Sed - a non-interactive text editor. Computer Science Technical Report No. 77, AT&T Bell Laboratories, Murray Hill, New Jersey, August 1978. Also available in UNIX User's Manual.

[97] Raul Medina-Mora. *Syntax-Directed Editing: Towards Integrated Programming Environments.* PhD thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, March 1982.

[98] Norman Meyrowitz and Andries van Dam. Interactive editing systems: Parts I and II. *ACM Computing Surveys*, 14(3):321–415, September 1982.

[99] Microsoft Corporation, Seattle, Washington. *Reference to Microsoft Word, Word Processing Program for the Apple Macintosh, Version 3.0*, January 1987.

[100] James H. Morris, Mahadev Satyanarayanan, Michael H. Corner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith. ANDREW: a distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[101] Joseph M. Morris and Mayer D. Schwartz. Design of a language-oriented editor for block-structured languages. In *Proc. of ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, Portland, Oregan, June 8–10 1981. Available as *SIGPLAN Notices* 16(6) or *SIGOA Newsletter* 2(1–2).

[102] S. L. Murrel and D. DeBaer. An interactive WYSIWYG table editor. In *Proc. of 1987 USENIX Summer Conference*, pages 19–29, Phoenix, Arizona, June 8–12 1987.

[103] Brad A. Myers. Visual programming, programming by example, and program visualization: A taxonomy. In *Proc. of ACM SIGCHI'86 Conference*, pages 59–66, Boston, Massachusetts, April 1986.

[104] Greg Nelson. Juno, a constraint-based graphics system. *ACM Computer Graphics*, 17(3):235–243, July 1983.

[105] Mark Opperman, James Thomson, and Yih-Farn Chen. A GREMLIN tutorial. Technical Report 87/322, Computer Science Division, University of California, Berkeley, California, December 1986.

[106] 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.

[107] Andrew J. Palay, Wilfred J. Hansen, Mark Sherman, Maria G. Wadlow, Thomas P. Neuendorffer, Zalman Stern, Miles Bader, and Thom Peters. The Andrew toolkit – an overview. In *Proc. of 1988 USENIX Winter Conference*, pages 9–21, Dallas, Texas, January 1988.

[108] Oren Patashnik. *BibTEXing*. Computer Science Department, Stanford University, Stanford, California, January 1988. Available in the BIBTEX release.

[109] Charles L. Perkins. The multiple representation problem. Master's thesis, Computer Science Division, University of California, Berkeley, California, December 1984.

[110] Kenneith Pier, Eric Bier, and Maureen Stone. Gargoyle: An interactive illustration tool. In *Proc. of EP88 – Internal Conference on Electronic Publishing, Document Manipulation, and Typography*, Nice, France, April 1988.

[111] Micahel F. Plass. *Optimal Pagination Techniques for Automatic Typesetting Systems.* PhD thesis, Computer Science Department, Stanford University, Palo Alto, California, June 1981.

[112] 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.

[113] Brian K. Reid. Procedural page description languages. In *Proc. of the International Conference on Text Processing and Document Manipulation*, pages 214–223, University of Nottingham, England, April 14–16 1986. Published by the Cambridge University Press.

[114] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.

[115] Thomas W. Reps and Tim Teitelbaum. Language processing in program editors. *IEEE Computer*, 20(11):29–40, November 1987.

[116] Marc J. Rockind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, April 1975.

[117] Hanan Samet. Heuristics for the line division problem in computer justified text. *Communications of the ACM*, 25(8):564–571, August 1982.

[118] Erik Sandewall. Theory of information management systems. Technical Report LITH-IDA-R-83-03, Department of Computer and Information Science, Linköping University, Linköping, Sweden, September 1983.

[119] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[120] William F. Schelter. Sample I$_\text{N}$F$\oplus$R display. Unpublished manuscript, 1987.

[121] Mayer D. Schwartz, Norman M. Delisle, and Vimal S. Begwani. Incremental compilation in Magpie. In *Proc. of ACM SIGPLAN Symposium on Compiler Construction*, June 1984.

[122] Dan Shafer. *HyperTalk Programming*. Sams Books, 1988.

[123] Ben Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, August 1983.

[124] Michael D. Spivak. *The Joy of TEX*. American Mathematical Society, 1985.

[125] Richard M. Stallman. EMACS: The extensible, customizable self-documenting display editor. In *Proc. of ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 147–156, Portland, Oregan, June 8–10 1981. A somewhat extended version appears in *Interactive Programming Environments*, Barstow et al. (eds.), McGraw-Hill Book Company, 1984, pp. 300–325.

[126] Richard M. Stallman. *GNU Emacs Manual, Fifth Edition, Version 18*. Free Software Foundation, Cambridge, Massachusetts, December 1986.

[127] Sun Microsystems, Mountain View, California. *NeWS Manual*, March 1987.

[128] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, October 1986.

[129] Tim Teitelbaum and Thomas W. Reps. The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, September 1981.

[130] Warren Teitelman. A tour through Cedar. *IEEE Software*, 1(2):44–73, April 1984.

[131] Robert H. Thomas, Harry C. Forsdick, Terrence R. Crowley, Richard W. Schaaf, Raymond S. Tomlinson, Virginia M. Travers, and George G. Robertson. Diamond: A multimedia message system built on a distributed architecture. *IEEE Computer*, 18(12):65–78, December 1985.

[132] Walter F. Tichy. RCS — a system for version control. *Software—Practice & Experience*, 15(7):637–654, July 1985.

[133] Howard Trickey. Drag: A graph drawing system. In *Proc. of EP88 - Internal Conference on Electronic Publishing, Document Manipulation, and Typography*, Nice, France, April 1988.

[134] Stephen Trimberger. Combining graphics and a layout language in a single interactive system. In *Proc. of the 18th Design Automation Conference*, pages 234–239, Nashville, Tennessee, June 1981.

[135] Ventura Software, Inc., Salinas, California. *Ventura Publisher — Professional Publishing Program, Reference Guide, version 1.1*, July 1987.

[136] Janet H. Walker. The Document Editor: A support environment for preparing technical documents. In *Proc. of ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 44–50, Portland, Oregan, June 8–10 1981. Available as *SIGPLAN Notices* 16(6) or *SIGOA Newsletter* 2(1–2).

[137] Mark N. Wegman. Parsing for structural editors (extended abstract). In *Proc. of 21st Annual IEEE Symposium on the Foundations of Computer Science*, pages 320–327, Long Beach, California, 1980.

[138] Mark N. Wegman and Cyril N. Alberga. Parsing for a structural editor (part ii). Technical Report RC 9197, IBM Watson Research Center, January 1982.

[139] Christopher J. Van Wyk. A graphics typesetting language. In *Proc. of ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 99–107, Portland, Oregan, June 8–10 1981. Available as *SIGPLAN Notices* 16(6) or *SIGOA Newsletter* 2(1–2).

[140] Xerox Office Systems, El Segundo, California. *8010 STAR Information System Reference Library, Release 4.2*, 1984.

[141] Nicole Yankelovich, Morman Meyrowitz, and Andries van Dam. Reading and writing the electronic book. *IEEE Computer*, 18(10):15–30, October 1985.