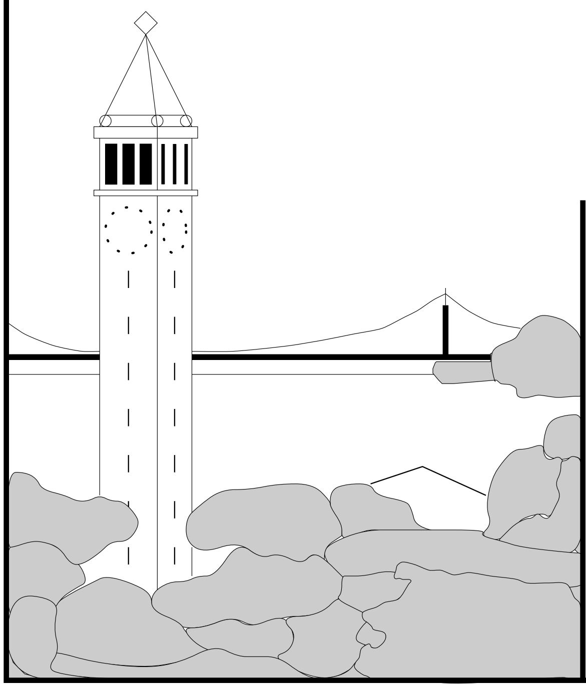
Proteus: An Adaptable Presentation System for a Software Development and Multimedia Document Environment

Ethan Vincent Munson



Report No. UCB/CSD-94-833

September 1994

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Proteus: An Adaptable Presentation System for a Software Development and Multimedia Document Environment

by

Ethan Vincent Munson

B.A. (University of California, San Diego) 1978
B.A. (University of California, San Diego) 1986
M.S. (University of California, Berkeley) 1989

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 at BERKELEY

Committee in charge:

Professor Michael A. Harrison, Chair
Professor Susan L. Graham
Professor Jim Pitman

1994

This dissertation is based in part upon work supported by the Defense Advanced Research Projects Agency (DoD), monitored by Space and Naval Warfare Systems Command under Contract N00039-88-C-0292 and by the National Science Foundation under Infrastructure Grant No. CDA-8722788.

Proteus: An Adaptable Presentation System for a Software Development and Multimedia Document Environment

Abstract

Proteus: An Adaptable Presentation System for a Software Development and Multimedia Document Environment

by

Ethan Vincent Munson
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Michael A. Harrison, Chair

The many different documents produced by a large software project are typically created and maintained by a variety of incompatible software tools, such as programming environments, document processing systems, and specialized editors for non-textual media. The incompatibility of these tools hinders communication within the project by making it difficult to share the documents that record the project's plans, design history, implementations, and experiences. An important factor underlying this incompatibility is the diversity of presentation models that have been adopted. Each system's presentation model is well-suited to the document types and media it supports, but is difficult to adapt to other types and media.

This dissertation describes a new model of presentation that is designed to be applied to a diverse array of documents drawn from many different media. The model is based on four simple services: attribute propagation, box layout, tree elaboration, and interface functions. Together, these services are powerful enough to support the creation of many novel and visually rich document presentations. Furthermore, because the model is based on a new understanding of the fundamental parameters defining media, the four services can be adapted for use with all media in common use.

The utility of this presentation model has been explored through the design and implementation of Proteus, a system for handling presentation specifications that is part of Ensemble, an environment for developing both software and multimedia documents. Proteus interprets specifications that describe how the four presentation services should be applied to individual documents. Proteus has a medium-independent kernel that provides the specification interpreter and runtime support for the four presentation services. The kernel is adapted to different media via the addition of a shell specifying the medium's parameters. Proteus's adaptability significantly eases the task of extending Ensemble to support new media. Proteus is also an important part of Ensemble's support for multiple, synchronized presentations.

In summary, this research develops a new, medium-independent model of presentation and shows that a specification-driven presentation system based on this model can form the basis of a software environment supporting multiple presentations and a variety of media.

To my parents,

Mary Ellen Jones and Paul Lewis Munson,

who created an intellectual

and then waited, patiently, for him to become an academic.

Contents

List of Figures

List of Tables

Acknowledgements

The Ensemble system is the product of many hands. Brian Dennis, Roy Goldman, Vance Maverick, and Tim Wagner have worked on the hairy beast for a long time now and together we have created something important. Wayne Christopher and Robert Wahbe also deserve special mention because they got us started actually building things. Other important contributions have come from Doug Banks, Kannan Muthukkaruppan, Takashi Ohtsu, Derluen Pan, John Pasalis, Tom Phelps, Steven Procter, and Boris Vaysman.

For me, the Berkeley Computer Science Division was a wonderful community. Among the graduate students, Mark Sullivan and Vance Maverick stand out as constant friends. Vance deserves particular thanks for doing the legwork to get this document signed and filed. Others who contributed to my success through advice, companionship, or friendship were Bob Ballance, Chris Black, John Boyland, Jacob Butcher, Charlie Farnum, John Hauser, Dan Jurafsky, Joe Konstan, Ethan Miller, Jim Ruppert, Dain Samples, Margo Seltzer, Luigi Semenzato, and Michael Van De Vanter.

Among the faculty, Domenico Ferrari, Robert Wilensky and Kathy Yelick helped with advice and insight at crucial times. Dave Patterson and Doug Terry showed me how to teach. Susan Graham gave valuable technical advice throughout my research and her *modus operandi*, the use of specifications, can be found in this work. Michael Harrison pushed me to work hard and smart and taught me how to be a computer scientist, a researcher, and a professor.

Special thanks go to my friends on the staff. Kathryn Crabtree, Liza Gabato, Jean Root, and Ruth Tobey minimized my adminstrative problems. Gail Gran and Gwen Lindsay made sure I always had the resources I needed and I'll let Phil Loarie fix my computer anytime. Eric Allman and the SWW folks significantly improved my software environment. Craig Lant and Brian Shiratsuki kept my system running and sensibly configured.

Of my colleagues outside Berkeley, I owe the greatest debt to Vincent Quint, Irène Vatton, Cécile Roisin and the other members of the Grif project. They gave me access to their software and papers and were always willing to answer my questions. I hope I can repay the debt. Among my other friends in computer science, Rick Furuta, Dick Taylor and Michal Young stand out for their generosity and openness.

My dissertation committee, Michael Harrison, Susan Graham, and Jim Pitman, made sure I described my work clearly and sensibly. Thanks to their thoughtful comments, this is a much better document.

My greatest thanks go to Barbara Carr, who put up with, well, with all the stuff a graduate student's spouse puts up with. What you see before you exists only because her love and support were constant.

Chapter 1

Introduction

1.1 Software Documents

Large software projects produce many different types of documents. In a sense, program source code is the most important type of document produced, since it is the most direct expression of the nature of the program itself. But it is often the case that the majority of software documents are not program source code. Table 1.1 lists fifteen types of documents that might be produced in the course of creating a program, categorized by the phases of the "waterfall" model of the software development process [17]. Program source code is only one of these fifteen document types, though it does appear in both the implementation and maintenance phases.

Three classes of programs are used in the production of these documents: programming environments, document processing software, and multimedia editors. Programming environments are used to edit, analyze, compile, and debug programs. Document processing software is used to edit and format natural language text documents. Multimedia editors are used to produce non-textual material, primarily for inclusion in text documents, but also at times for stand-alone non-text documents. Each of these types of programs does its job well, but with rare exception, they do not work together. Put another way, the various software documents do not *interoperate*.

This dissertation research was conducted as part of a larger effort to produce a single, unified system for *all* software documents. This system, called Ensemble, is motivated by the belief that by bringing the tools for creation, maintenance, browsing, and analysis of all these types of documents into the same system, barriers that currently prevent the flow of information (as represented by those documents) among the people involved in a software project will be removed.

1.2 A Hypothetical Scenario

To understand why a unified system might enhance software development, it is helpful to consider a hypothetical scenario of the system's use. Figure 1.1 shows a mock-up of a yet-to-be-created, mature version of Ensemble. Four different windows, all managed by Ensemble, are shown on the computer screen.

Phase	Document Types	Formal	Natural	Multimedia
Requirements	system definition			
	project plan		✓	✻
	software requirements specification		✓	✻
Design	architectural design specification		✓	✻
	detailed design specification		✓	✻
	preliminary user's manual		✓	✻
	software verification plan		✓	
Implementation	program source code	✓		
	walkthrough and inspection reports		✓	
Testing	test plans		✓	
	test data	✓		
	testing scripts	✓		
	bug reports		✓	✻
Maintenance	bug reports		✓	✻
	change reports		✓	
	program source code	✓		
	errata for user documentation		✓	✻

Table 1.1: Software document types categorized by the phases of the waterfall model of the software development process. (Some of the document types listed here were taken from Fairley's book [17].) The document types are divided into two categories, formal and natural, shown by check marks in the third and fourth columns. Formal documents are those, such as programs, that are designed to be formally analyzed. All other documents are natural. Some natural documents include elements from other non-textual media. Document types that are likely to have multiple media are marked with a star in the fifth column.

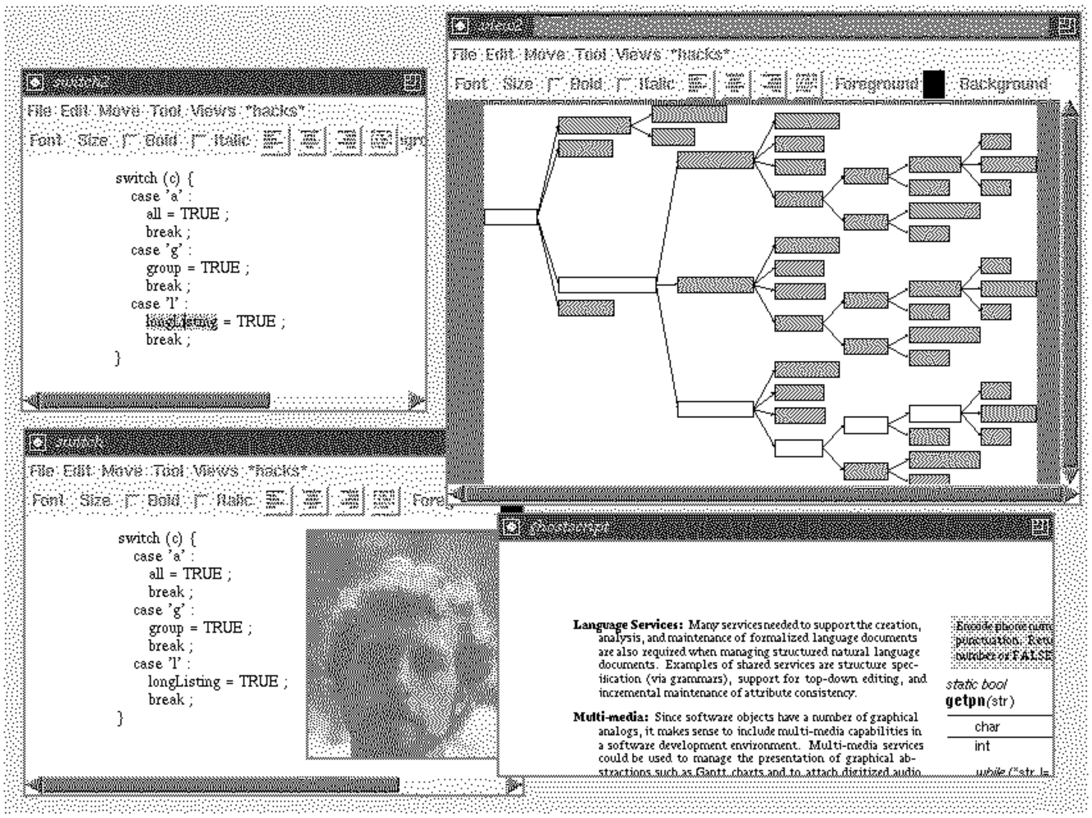


Figure 1.1: A mockup showing a hypothetical scenario of the use of Ensemble.

The upper-left window is a program-editing window. It shows a more-or-less standard view of some program source. One identifier is highlighted because program analysis has detected a problem with it. The Ensemble user can query the system for the nature of the problem with a few keystrokes. Upon learning that the identifier is undeclared, standard text-editing operations can be used either to add a declaration or to change the name of the identifier to one that is already declared. Re-analysis of the program can be easily invoked and will show, by removing the highlighting around the identifier, that the problem has been fixed.

The window in the upper-right shows a presentation of the same program's parse tree. The nodes filled with white are the ones on the path from the root of the parse tree to the identifier. When the user changes the text of the program in ways that alter the parse tree, this presentation is automatically updated to reflect those changes. A parse-tree presentation like this can have several uses:

- it can aid understanding of the program's syntactic structure,
- it can be used to navigate through the program via its syntactic structure (and other windows can be made to scroll in synchrony with it),
- and it can be used to make structurally-based modifications of the program.

The window in the lower-left also shows a presentation of the same program. This presentation shows the same program source code as the upper-left window, but it includes multimedia documentation that was elided in the other presentations. The particular documentation shown is a video clip of the program's original author which can be played back in order to hear a description of the rationale behind the design of the adjacent source code. A "full-documentation" presentation like this might include other kinds of multimedia elements like graphic figures, animations, audio clips, and bitmaps. It might also include quotations from the design documents that motivated the creation of this part of the program.

The lower-right window shows one of these design documents. It is presented using a wide variety of high-quality text formatting operations like multiple columns, hyphenation, justification, and fine-grained control of paragraph layout. This window could have been brought up by a search query that matched its contents or by following a hypertext link from the program source. Thus, the user can access the program's design documents without leaving the Ensemble environment, which reduces the effort needed to ensure that the design documents and the program stay in agreement.

Several system features stand out in this hypothetical scenario:

Multiple presentations: Three different presentations of the program were shown. None of the three is treated as the "best" or "most important" presentation. They are simply different ways of looking at the program, each one providing a viewpoint or a set of services that may be particularly useful in certain circumstances and distracting or unhelpful in others.

Multimedia: The system supports a variety of media, including text, two-dimensional graphics, and digital video. A document can contain elements from many media

Feature	Programming Environments	Document Systems	Multimedia Systems
Multiple presentations		$\pm$	
Multimedia		$\pm$	✓
High-quality formatting		✓	✓
Interconnections	$\pm$		$\pm$
Analysis	✓		

Table 1.2: This table illustrates the inconsistent support for important features of Ensemble in existing programming, document and multimedia software. A check indicates full support. A plus/minus symbol indicates partial or limited support.

integrated seamlessly. Alternately, it may be possible to display the same document in different media, when doing so makes sense.

High-quality formatting: The system supports high-quality formatting services suitable for the creation of production-quality documentation.

Interconnections: The system has facilities for representing the connections among a collection of documents. These connections can be used for navigation between related documents or for understanding the relationships.

Analysis: The system includes integrated analysis services for programs and other types of documents with formal semantics. The user can query the system about analysis results.

One might wonder whether these features are novel. Table 1.2 illustrates the extent to which these features can be found in existing software and shows that no existing system is able to bring together all of these features.

1.3 A Thesis

The thesis of this dissertation is:

- that it is possible to build an interactive document system providing general support for multiple presentations of high quality in a variety of media;
- that there exists a set of presentation services that can be usefully applied to a variety of media; and
- that a single, adaptable software module providing these services can be created and used to manage presentation for a variety of media.

The research supporting this thesis has focused on a single subsystem of Ensemble, called Proteus, that manages formal descriptions of appearance for the documents manipulated by Ensemble's users. Proteus is a single, adaptable software module providing presentation services that are useful to several media. It is the key element in Ensemble's support for multiple high-quality presentations.

For an interactive document system to support multiple presentations, it must first make a clear separation between the representation of the document and the representation of its presentations. The document representation must include everything that is shared between presentations or that must persist from one editing session to the next. The presentation representation must embody the more ephemeral differences in how the document may be viewed.

One could design a document system in which users defined each presentation of a particular document by hand, but in practice, the task of defining a new presentation is too time-consuming. Thus, to make multiple presentations practical, a presentation's appearance must be based on a formal specification that applies to a large set of documents (such as the set of C programs, or the set of bug reports). In such a system, the module that handles these formal appearance specifications is a critical link in the system's support for multiple presentations. Proteus is that module.

Presentation quality is also important, because while it is easy to build a low-quality document system, high-quality formatting requires fine-grained control over document appearance along with other complex features. Thus, the formal appearance specifications must describe the application of presentation services that are powerful enough to support high-quality formatting. Proteus provides four simple services that together are powerful enough to support high-quality formatting in several media.

One approach to supporting multiple media is to identify a presentation model that subsumes all the qualities of the media to be supported. Unfortunately, different media have grown out of different artistic or technical traditions and it is difficult to do them all justice with a single model. Also, if a new medium is identified for inclusion in the system, it may not fit the model at all. The alternative is to create a system whose services are designed for adaptation to the needs of different media. This requires a model of the concept of "medium" around which to base the adaptation. Ensemble supports such a model and Proteus uses it as the basis for adapting to different media.

1.4 Overview of the Dissertation

This dissertation is organized into six chapters that present the motivation, design, and implementation of Proteus and then evaluate its successes and failures.

Chapter 2 examines the systems that have been used to present software documents: programming environments, document systems, multimedia systems, and user interface software. These systems point the way to the solutions sought in this dissertation, but none of them bring all of the right elements.

Chapter 3 describes the Ensemble system. It begins by describing the two systems, Pan and VORTEX, whose development directly stimulated the design of Ensemble. Pan is a language-based editing system for a variety of different programming languages. VORTEX was a multiple-representation editor for TEX documents. Both systems had successes and failures and the Ensemble project was undertaken as an attempt to bring together the most desirable elements of each system in order to create a new system supporting a broader array of documents (particularly software documents).

The design of Proteus is presented in Chapter 4. This chapter starts by describing the architecture of Ensemble's presentation subsystem. Then Proteus's design is described in detail, beginning with the architecture that enables it to adapt to different media and then the syntax and semantics of the appearance specifications are covered in detail. Examples are used throughout to make the abstractions more understandable.

Proteus's implementation is described in Chapter 5. This chapter discusses the data structures of Proteus, the modules that operate on them and their relationship to the design concepts presented in Chapter 4. Also described is the interface between Proteus and the rest of Ensemble.

The next chapter is an evaluation of Proteus, both on its own terms and in comparison to other systems providing similar or related services. Experience with Proteus is evaluated, highlighting both positive and negative aspects of that experience. The comparisons with other systems make clear both the areas where Proteus represents an advance over earlier work and those areas where improvement is needed.

Finally, Chapter 7 summarizes the contributions of the research and suggests directions that future work should take.

Chapter 2

Previous Work

This chapter explores the current state of the art in systems used to produce different kinds of software documents and evaluates the extent to which these systems provide

- general support for multiple presentations,
- adaptability to different media, and
- high-quality formatting in those media.

This chapter also looks at some tools that can be used to build such systems: user interface systems and compound document systems.

2.1 Programming Environments

The term *programming environment* can be applied to a wide variety of different systems used to create, analyze, and maintain programs. A programming environment may be one program or a collection of programs. Environments generally provide a program editor and a language interpreter or compiler. They may also include debuggers, configuration management tools, and tools for examining the results of program analyses (e.g. program profiles).

Contemporary commercial environments, like SMARTsystem[1] [63], support only a single programming language and provide no particular support for non-program documents. They provide an editor for the raw text files that are the input of the interpreter or compiler. The editor may be custom-made or may be based on a standard text editor like GNU Emacs [70]. While these editors may allow the program to make simple changes of character style or color, they do not allow the user to make persistent changes of appearance to the document, as a document-processing system would. Also, they provide none of the more advanced formatting services, like multiple columns, line-breaking or arbitrary font choices, that are required for natural-language text documents.

The Cornell Synthesizer Generator [67] is an editor generator that supports a wide variety of programming languages. Given an attribute grammar specification for the syntax and

[1] SMARTsystem is a registered trademark of the ProCase Corp.

static semantics of a programming language, it generates a language-specific, stand-alone editor. This editor analyzes the text of a program, creating an attributed abstract syntax tree. The user sees and edits an *unparsing* of this tree. An unparsing is an expansion of the tree and its attributes into a text stream and is defined by *unparsing rules* in the attribute grammar. Syntax and semantic errors are announced to the user by unparsing rules that produce special, highlighted comments describing the errors. Because each language can have only one attribute grammar, the Synthesizer Generator can only support a single presentation. It has no support for multimedia.

Centaur [33] is another editor generator. In Centaur, the appearance of a program document is specified separately from the specification of its analysis, using a specification language called PPML (Pretty-Printing Meta-Language) [32]. A PPML specification defines unparsing rules for *tree patterns*. The abstract syntax trees of program instances are compared to these tree patterns and when a match is found, the corresponding unparsing rule is activated. Centaur supports multiple variable-width fonts and changes to foreground and background color. Most importantly, because the PPML description is separate from the language specification, Centaur can support multiple presentation styles for a programming language. However, it is not possible to view the same program in two styles simultaneously and switching pretty-printing styles is awkward.

The key point to note here is the complete inadequacy of programming environments for any task other than program editing. Their formatting model is so restricted that no one would even consider using them for WYSIWYG document editing. In fact, these systems are insufficient for advanced program typography like that proposed by Baecker and Marcus [4].

2.2 Document Systems

Non-program software documents are generally produced using commercially-available document processing systems that have direct-manipulation, WYSIWIG user interfaces. A widely-used example is Framemaker [19]. Framemaker supports the creation and editing of conventional text documents, tables, equations, and two-dimensional graphics. It can import bitmaps and arbitrary PostScript figures. Its text formatting services include support for multiple columns, arbitrary font choices, cross-references, automatic component numbering (e.g. figures, itemized lists, sections), and fine-grained control over spacing between paragraphs, lines, words, and characters.

Framemaker treats a text document as a sequence of paragraphs that have named types. Each paragraph type has an associated set of formatting attributes that determine the appearance of paragraphs of that type, though the defined values may be overridden for individual paragraphs. These attributes control font choice, spacing, tabs, and hyphenation. Sets of paragraph type definitions can be stored in files known as *templates*. Templates can be loaded into Framemaker documents at any time in order to establish or change values for formatting parameters. However, a document's connection with its template is not active: if the template file changes, documents that were based on the previous version must be updated manually. Since the description of a document's appearance is stored with the document itself, there is no way for Framemaker to support multiple presentations.

Framemaker, like almost all other commercial document processing systems, is completely sufficient for the creation of most textual software documents. However, its formatting model is not adequate for programs for two reasons. First, the best basis for formatting programs is the abstract syntax tree. Framemaker's sequential document model is difficult, if not impossible, to adapt to a tree representation. Secondly, Framemaker's formatting model provides no mechanism by which two "paragraphs" can be placed on the same line. In programs, the smallest typed elements (which Framemaker must represent as paragraphs) are individual tokens. Since a single line can easily contain five or more tokens, Framemaker's style mechanism cannot express their layout. Also, Framemaker provides no mechanism for connecting program documents to analysis tools.

Document processing systems that support tree-structured documents do exist. Almost all of them have been designed to support SGML, the Standardized Generalized Markup Language [22]. SGML was designed as a document interchange format: a storage representation that could act as the *lingua franca* between otherwise incompatible document systems. The SGML standard actually defines two languages, SGML proper and a meta-language for creating Document Type Definitions (DTDs). A DTD is a regular-right-part grammar [34] with supplemental specifications for cross-references and for attributes that can be attached to the tree nodes defined by the grammar. A DTD defines a *document type*, which is the set of documents that conform to the DTD's grammar. Instances of a document type, that is *documents*, are written as specifications in SGML, which is a language for describing tree instances and the data (such as the text of a document) attached to them.

SGML is designed to describe only the *logical structure* and *content* of a document, not its appearance. The intention of SGML's designers was that appearance would be specified separately, either using proprietary mechanisms or using DSSSL, the Document Style Semantics and Specification Language [1]. Thus, SGML systems should be able to support simultaneous multiple presentations, since the specification of the document's appearance is stored separately from the document itself. In practice though, they don't. For example, Framebuilder [20], which can import and export SGML documents, intertwines its structure and appearance specifications, a design choice that precludes multiple presentations. Furthermore, Framebuilder uses the same formatting model as Framemaker and, thus, lacks the layout capabilities required for programs.

One system which does support multiple presentations is the Grif structured document editor [66].[2] Development of Grif began before the use of SGML became widespread, so Grif has its own version of the DTD, called a *structure schema*. Document appearance is defined by a separate specification called a *presentation schema*. Grif's presentation schemas allow very fine-grained control over the layout and formatting of document elements. The layout model can support multiple elements on the same line and so is sufficient for program formatting.

Grif also supports a limited form of multiple presentation, called *views*. A single presentation schema can define several views. For instance, a presentation schema for Report

[2]Furuta [21] makes clear that any document model with typed elements (such as Framemaker's model) should be considered "structured". In practice, though, the term *structured document* usually means "tree-structured document" and the dominance of SGML in the marketplace is beginning to make the term equivalent to "SGML document".

documents can define "Full Text" and "Table of Contents" views. The Full Text view would show the entire document, while the Table of Contents view would show only the chapter and section headings. The user can see both views simultaneously and changes made in one will appear in the other. However, Grif does not provide the same synchronized viewing for presentations defined by different schemas.

Grif's development began in a research environment. It has since been commercialized and the commercial version is a true SGML system using SGML DTDs instead of structure schemas [24].

Grif does have some serious limitations.

- It is difficult to extend Grif to support media other than text. The technical reasons have never been described in detail, but currently Grif supports only text, bitmaps, and restricted forms of mathematics and two-dimensional graphics. One obvious problem is that Grif's presentation schema language, P, is not modularized with respect to media. So, the addition of a new medium requires modification of Grif's core presentation services.

- Grif's formatting model for text is not as rich as those of commercial document processing systems. For instance, it cannot produce multiple columns.

- The P language is idiosyncratic. It lacks unifying principles around which a presentation designer can come to a general understanding of the language. For instance, the language has some boolean parameters. It also supports features (e.g. control over page breaks) that seem to be equivalent to booleans, but are invoked with a command-like syntax, rather than a parameter-setting syntax.

- The P language lacks power. In particular, it is hard to write context-sensitive rules for node types because the language does not support conditionals.

In spite of these limitations, Grif remains the most important single influence on this dissertation. It demonstrated the potential of multiple presentations and identified a formatting model for tree-structured documents that appears to be sufficient for all textual documents, including programs.

2.3 Multimedia Systems

Software for editing multimedia documents is both very sophisticated and very primitive. It is possible to buy software for personal computers (albeit *expensive* personal computers) that can bring much of the power of professional studio editing equipment into the home. But while this software is powerful and provides the ability to construct complex and sophisticated multimedia presentations, its approach is, in many ways, primitive. In these programs, multimedia documents are created and edited at a very low level. There is no high-level document model, comparable to SGML, that separates document structure from document presentation. Instead, each document is a stand-alone object described almost entirely in terms of the way it looks or plays.

For example, in MacroMedia's Action! program for multimedia presentations [45], every document element has start and stop times that determine how long it appears on the screen. Each element also has manually set positions, movements, and drawing styles. These values must be set explicitly for each object and there is no easy way to define structures that can be shared by multiple documents.

Interestingly enough, the Action! program does provide multiple presentations of its documents, though the set of possible presentations is fixed. These different presentations help the user see and manipulate all the elements in the document, even though there is no single time point at which all of them are on the screen. The various presentations provide different abstractions that either ignore time (the "Compressed View") or map it into the spatial dimensions (the "Timeline" and "Scene Sorter" tools). Because of the complexity of multimedia documents, a similar variety of presentations can be found in other commercial systems such as PowerPoint [49] and Director [46].

An important research trend in multimedia systems is the growing use of spatial layout techniques for automatic layout of multimedia objects in the time dimension. Most multimedia programs require that objects be placed explicitly on a time line and do not support the kinds of automatic layout services that are common in document systems and in user interface software. Buchanan and Zellweger [9] discuss this idea explicitly and describe how they use spatial techniques for automatic temporal layout in the Firefly system. The Xavier multimedia toolkit [26] extends the boxes-and-glue layout mechanism of the Interviews user interface toolkit [44] to the time dimension. The boxes-and-glue model was originally introduced in the TEX document formatter [38] with no particular thought about its applicability to temporal layout. This work makes clear that, for the purposes of document layout, the widely-perceived distinction between the time dimension and the spatial dimensions is more apparent than real.

2.4 User Interface Software

Since the appearance of documents is part of their user interface, it is natural to ask whether user interface systems provide useful services for software documents. As with the systems examined earlier, the answer is that they offer partial solutions at best. Most user interface toolkits (e.g. OSF/Motif [53] and Tk [58]) focus their services on classic user interface objects such as windows, buttons, menus, and scrollbars. Typically, they also provide a few more heavyweight objects, such as Motif's text widget or Tk's canvas. These tools are more complex and can be used to build specialized document interfaces, but their formatting capabilities never approach those of a real document or multimedia system and they are hard to extend.

The Garnet [52] toolkit for X11 is consciously designed to provide services for the application window (as opposed to the user interface wrapper around the application window). It manages layout of graphic and text objects with a layout system based on uni-directional constraints. However, Garnet lacks support for high-quality text.

The Interviews toolkit [44] has better support for text because it uses the boxes-and-glue layout model. In fact, the toolkit has been used as the basis for a text document editor [10] that uses the optimal line-breaking algorithm of Knuth and Plass [40].

Both Interviews and Garnet are constructed in object-oriented languages, so they should be relatively easy to extend to new media. The Xavier toolkit mentioned in the previous section shows that this promise can be realized under Interviews. However, neither system provides any support for multiple presentations. In fact, in both systems, there is little separation between the representation of an object's meaning and its appearance.

The Chiron [35, 76] user interface development system does support multiple presentations. The central concept in Chiron is that of the *artist*, which is an agent responsible for managing the appearance of an abstract data type (ADT) and the user's interaction with the ADT. A single ADT may have any number of artists attached to it. Artists are implemented in an extended version of Ada and are compiled into the system. This means that it is unlikely that new presentations will be defined by end users.

The Chiron system provides a server that renders user interface objects, graphics objects and text. It also multiplexes user input to its various artist clients. Much of the code needed to build an artist is automatically generated. Chiron provides a set of drawable representations implemented as a hierarchy of C++ classes. This set, called the ADL hierarchy, is sufficient for a user interface system but is not sufficient to support high-quality multimedia documents [11].

2.5 Summary

Looking back at the systems described in the last four sections, a number of facts become clear:

- No existing system provides a general mechanism for multiple presentations of high quality, though Grif and Chiron come close. Grif's mechanism for multiple presentations is not fully general. Chiron has a very general mechanism, but does not provide services for producing high-quality output and is implemented in a manner that discourages style modification by end users.

- So far, the only presentation services that have been shown to apply to a variety of media are layout services (constraints in Garnet and boxes-and-glue in Interviews/Xavier). While layout is very important, document presentation involves much more than just the placement of objects.

- No system other than Interviews/Xavier has made a clear demonstration of its extensibility to different media. Even Interviews lacks an explicit notion of "medium" around which to base this extensibility.

Thus, it is clear that the state of the art in handling multimedia documents in general, and software document in particular will be advanced by the creation of a system achieving these goals.

Chapter 3

Ensemble

Ensemble is an environment for the creation, maintenance, analysis, and browsing of *all* types of software documents. In Ensemble, these varied documents will interoperate seamlessly, bringing down technical barriers that currently hinder effective communication between the designers, implementors, users, maintainers, and extenders of a system.

Research on Ensemble has focused on the development of generic services that can be usefully applied to many different document types. In and of itself, this is not an unusual goal. What is unusual about Ensemble is its commitment to using specification-driven services. Ensemble currently accepts six different kinds of specifications for:

- lexical analysis,
- parsing,
- static semantic analysis,
- document structure,
- user interface control,
- and document presentation.

3.1 Ensemble's Ancestors

Ensemble grew out of experience with two systems developed earlier at Berkeley, Pan and VORTEX. Pan [5] was a programming editor with integrated support for program analysis. VORTEX [13, 14] was an editor for TEX documents that supported both a source view and a direct-manipulation view of the document. This section describes these systems in some detail and shows how the goal of merging their features resulted in the creation of Ensemble.

3.1.1 Pan

Pan is a multi-lingual programming environment that combines a window-based user interface with integrated program analysis services. What sets Pan apart from other programming environments, such as Centaur or the Cornell Program Synthesizer [33, 67], is that it is a syntax-recognizing editor. A syntax-recognizing editor allows the user to perform unrestricted text editing and then reanalyzes the program as needed. In contrast, syntax-directed editors require that the program be constructed in a top-down fashion. Such editors typically only allow free text editing in restricted portions of a program, such as within one expression or statement.

The syntax-recognizing approach can be found in commercial single-language programming environments. However, it is rare in multi-lingual environments because there are still open research questions about how to efficiently provide analysis services and how to best handle syntactic and semantic errors in the program. Pan adopted the syntax-recognizing model not because it was easy to implement but because users of syntax-directed environments often find their top-down editing style frustratingly rigid for practical software development. Pan is still used today as an experimental programming environment though development of core services for Pan has largely halted and there is only a small number of regular users.

A major portion of the research on Pan focused on improving user interfaces for programming [72]. Rather than attempting to meet the needs of novice programmers, Pan's designers worked to identify techniques that could maximize the amount of useful information available to expert programmers. An early engineering choice limited the options available for doing this: the representation used for editing and program analysis was also the display representation. This data structure was similar to an ASCII text file, but included support for incremental changes to the text stream. The appearance of a program could be manipulated by modifying the number and type of whitespace characters in the text stream and by setting the font or color (both foreground and background) of the characters.

There were two problems with the combined editing/display representation. The first problem was the lack of separation between the fundamental representation of the program and the presentation. This made it difficult to create alternate views of the program, particularly ones from which the program could be edited. The second problem was that the representation's limited typographic information made it impossible to create presentations of the program that deviated from the traditional line-oriented style to any great degree. For instance, there was no way to place text from non-contiguous sections of a program on the same line.

3.1.2 VORTEX

Ensemble's other ancestor was VORTEX, an interactive environment for editing TEX documents that was developed at UC Berkeley between 1986 and 1989. VORTEX had three main parts: a source editor, a proof editor, and an incremental formatter. VORTEX had a window-based interface built using the X10 window system. VORTEX was designed to be compatible with all TEX documents and macro sets.

The source editor was a screen-based text editor similar to GNU Emacs and to Pan (but without Pan's program analysis services). It had an embedded Lisp interpreter which was used to extend the editor's operations beyond a basic set of simple character-based and word-based operations. Every available editing operation was accessible as a Lisp function. Except for its ability to communicate with both the formatter and proof editor, the source editor was little different from other text editors.

The proof editor showed the document in its final (or WYSIWIG form), but could not be used to edit the document directly. Rather, operations invoked in the proof editor were sent as messages to the source editor, which changed the TEX source and notified the incremental formatter of the change. The formatter would reformat as much of the document as was necessary and send the changes to the proof editor, which would display them. The set of operations available in the proof editor was more restricted than in the source editor, primarily because the user could only see the effects of the TEX macros, not the macros themselves.

The incremental formatter converted the text stream of the source editor into the display structures (pages, lines, and characters) of the proof editor. It also answered queries from the proof editor about the correspondence between the display structures and the text stream so that the proof editor could send the correct arguments for operations to the source editor.

The correctness of the VORTEX implementation was shown by its ability to correctly format the rigorous `trip` test for TEX formatters [39]. It passed this test running in both batch and incremental modes [12].

However, several problems in the design of VORTEX became clear as experience with the system grew. In a sense, the problems all arose from the design of the incremental formatter, which had been implemented by adding a checkpointing system to an existing batch formatter for TEX. Taking this approach had made it easier to assure that all TEX documents would be correctly formatted, because the batch formatter had already been certified correct. The checkpointing code simply saved the state of the TEX interpreter at various points in the processing of the document. Unfortunately, this choice had several negative consequences:

- The formatter was page-incremental; that is, the smallest unit of reformatting was a page. This made the proof editor too slow to be used for regular text entry. The design had assumed that almost all text entry would be done in the source editor, but it quickly became clear that even small error corrections in the proof editor might require three to six keystrokes (for deleting incorrect characters and entering new ones). Immediate feedback about the effect of each keystroke is important for the user to be certain that the correct modification is being made. The substantial pause that occurred between each keystroke was not acceptable.

- It was difficult to determine when changes on one page of the document stopped producing changes on other pages. This problem is called *quiescence detection* and is discussed in detail by Harrison and Pan [27]. Quiescence detection was particularly difficult for LATEX documents which require two passes of the batch formatter to resolve cross-references, though the problem was not restricted to LATEX documents.

  The fundamental problem was that in a batch formatting system driven by a macro language, dependencies between distant parts of the document are represented by changing the state of the formatting engine (or macro language parser). Many changes in the state of the formatting engine are not important, but without a deeper understanding of the engine's state, the many irrelevant changes cannot be ignored.

- Changes to macro definitions could not be handled incrementally because the checkpointing mechanism had no knowledge of the *meaning* of a TEX macro. So, when a macro definition was changed, correctness could only be assured by reformatting all parts of the document that came after the definition.

3.1.3 Merging Pan and VORTEX

The Pan and VORTEX systems were intended to solve rather different problems, but shared some features. Both systems

- had an extensible text editor modeled after GNU Emacs,
- used a Lisp dialect as the extension language,
- had a window-based user interface,
- attempted to provide better interaction for authors of certain kinds of textual documents (i.e. programs and natural language documents, respectively),
- had adopted a multiple-representation approach,
- and required the development of incremental algorithms in order to provide sufficient performance.

These similarities alone were enough to give the two projects reason to join forces. But the problems experienced with each system provided even stronger motivation to collaborate.

Pan had lacked sophisticated document services. As a result, many interesting program presentations simply were not possible. Pan needed a richer document model that separated presentation from the core representation of a program. It also needed the kind of high-quality typographic effects that are considered standard for document processing systems.

VORTEX had lacked language analysis services and so had been forced to use checkpointing to achieve incrementality. Because VORTEX never really *understood* a TEX document, it was difficult to improve its incrementality. Furthermore, TEX's document language was never intended for use in an interactive setting. So, some tasks (e.g. cross-references) required multiple runs of the formatter when, in an interactive system, these problems can be solved without multiple runs by adding data structures. Since this implied the design of a new document language, it would make sense to design one that was well suited to the language analysis services of Pan.

Thus, the initial vision for a merger of Pan and VORTEX was to produce a new multiple-representation document system providing language analysis and high-quality document processing services that would be used for both general-purpose programs and for natural-language documents.

It soon became clear that adopting the structured document model was a better approach. The structured document model was sufficiently general to support both natural language and program documents. It provided a general mechanism supporting multiple presentations of the same document and did not prevent the use of the multiple-representation approach, which would still be required for the analysis for programs. Grif had shown that it was possible to build an interactive structured document system. A new system based on the same technology could show the benefit of bringing document and language technology together. This new system came to be called Ensemble.

3.2 The Architecture of Ensemble

Ensemble's architecture has three main components: the control subsystem, the document subsystem, and the presentation subsystem. Each of these subsystems has a well-defined domain to which no other subsystem can lay claim. The relationship among them and with the window system is shown in Figure 3.1.

Document Subsystem: Documents are the fundamental objects of the Ensemble system. They are the objects that users navigate, modify, and see on the computer screen. In Ensemble, all documents are represented by a tree structure with the content appearing at the leaves. The document subsystem

- provides a set of low-level navigation, access, and modification operations;
- manages the storage and retrieval of documents from the persistent store (currently the UNIX file system); and
- announces document modifications to other subsystems via *change events*.[1]

For program documents (and other documents written in languages with formal semantics), the document subsystem also provides language analysis services. These include services for lexical analysis, syntactic analysis, and static-semantic analysis.

Control Subsystem: The process of responding to user actions (as described by X11 events from Tcl/Tk [57]) is managed by the control subsystem. The core of this subsystem is a modified interpreter for the XLISP dialect of Lisp [6]. This interpreter provides a high-level mechanism for managing the binding between user actions (mouse and keyboard events) and internal Ensemble operations. It also manages those parts of the user interface that are not directly related to documents themselves, such as the main control window and the creation of buttons, menus and scrollbars. Finally, it provides the means by which developers and end-users can customize or extend the behavior of Ensemble. Every user action corresponds to a function in the extension language. Communication between the extension language subsystem and the rest of Ensemble falls into three categories:

Position Queries: Certain user actions (such as cursor placement or selection) must be mapped to a portion of the underlying document. The control subsystem has

[1] In Ensemble, events are simply one-to-many procedure calls with runtime binding. They are used to announce interesting changes in the state of objects. Clients interested in the state of a particular object register a handler for that object's event(s). When the object's state changes, the event is "fired" and each registered handler is invoked. If multiple handlers are registered, they are invoked one at a time, because Ensemble is a synchronous system. However, no guarantees are made about the order in which events are fired. Because no more than one handler can be invoked at a time, the problems common to asynchronous systems (such as deadlock) cannot occur.

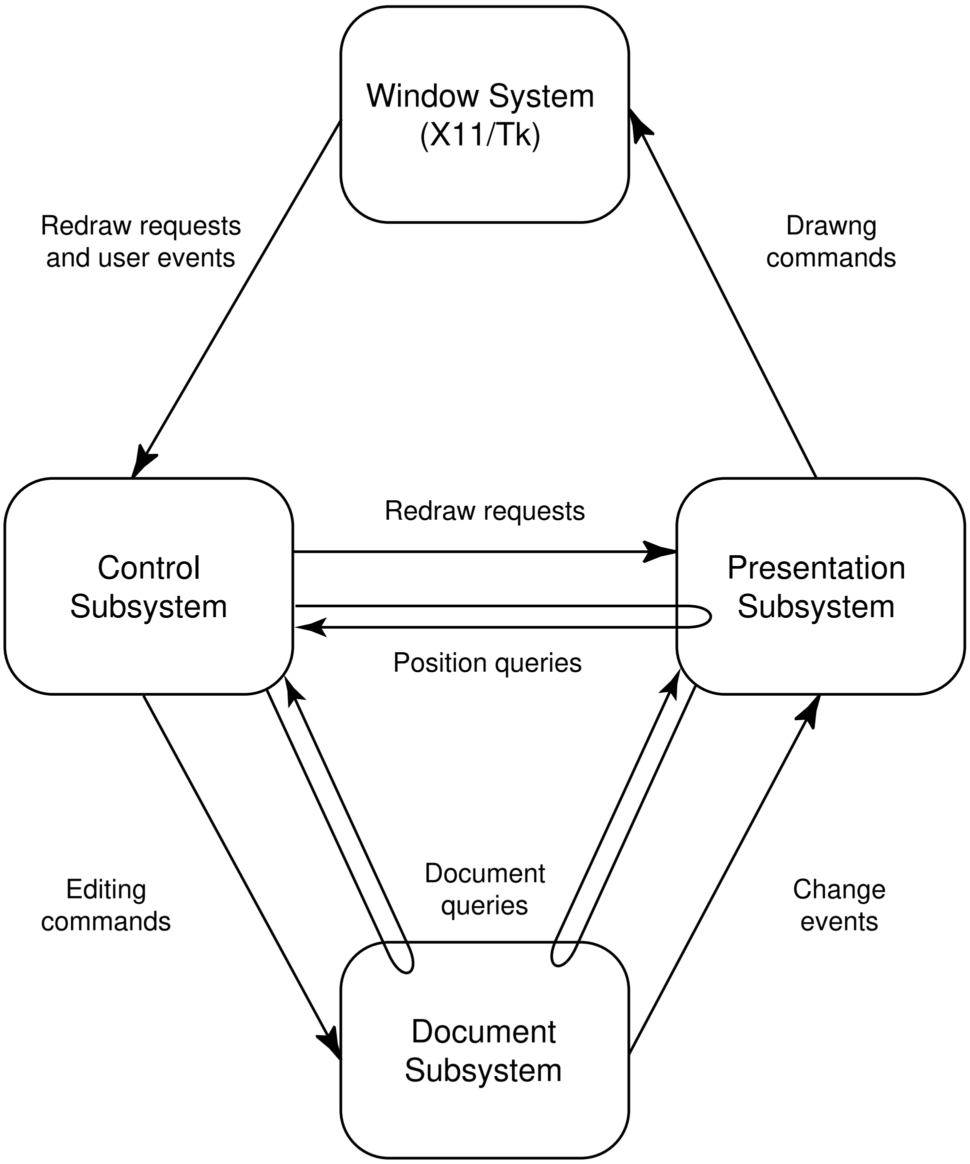


Figure 3.1: **Overview of Ensemble's architecture.** Single-pointed arrows represent commands. U-shaped arrows represent query-and-response interactions.

no knowledge of the position of document elements on the screen. To obtain this information it sends *position queries* to the presentation subsystem and uses the descriptors that are returned as arguments for operations on the document.

Redraw requests: Other user actions do not change the content of the document but simply change what part of the document is displayed. Examples of such user actions are scrollbar operations and the exposure of previously occluded (or just created) windows. When this happens, the control subsystem issues redraw requests to the presentation subsystem. These redraw requests are expressed in fairly abstract form. For instance, a possible request would be to show the full width and 5% of the vertical height of the document, centered on the position of a cursor.

It may seem strange that events that require drawing should not go directly to the presentation subsystem. By having the control subsystem catch *all* window system events, Ensemble is able to support customization of the system's response to those events. Thus, synchronization of scrolling in different windows can be specified in the extension language.

Editing Commands: Many user actions are bound to operations that modify the underlying document. In this case, the control subsystem invokes one of the many lower-level editing functions provided by the document subsystem. Other editing commands are used to create useful pieces of editing state, such as cursors and selections.

Presentation Subsystem: The presentation subsystem is responsible for drawing documents on the screen. It monitors the change events for each document. If an event indicates that the document has changed in a way that requires reformatting, the presentation subsystem traverses the altered portions of the document and reformats the material there. If any of the changes are visible on the screen, the presentation subsystem issues drawing commands that bring the screen up to date. The architecture of the presentation subsystem is described in detail in the next chapter.

Ensemble differs from other interactive systems in the separation between the control subsystem and the presentation subsystem. The control subsystem embodies all the control aspects of the user interface. It parses user events and makes decisions about how Ensemble should respond to them. The presentation subsystem primarily draws documents. Its only other output comes in the form of responses to position queries from the control subsystem.

This kind of separation between presentation and control has long been endorsed by user interface designers. It lies at the heart of the Seeheim user interface model [62] and the Smalltalk Model-View-Controller paradigm [41] (often referred to by its initials, MVC). But, in practice, most interactive systems merge these two functions. For instance, in response to a question about whether the Smalltalk community had moved beyond the MVC paradigm, L. Peter Deutsch, one of the original Smalltalk designers, said:

> There is no consensus in the Smalltalk community about this. Digitalk uses a watered-down form of MVC that feels more familiar to people used to programming with the lamentable Windows API. I believe both Digitalk and

> ParcPlace have created "application" objects that are places to put "all the stuff that doesn't fit anywhere else"; I think this usually is a mistake. There have been quite a number of research papers investigating variants of MVC; the project I've heard most about is the work by Coutaz somewhere in France.
>
> I think at this point there is substantial agreement that it is hard to make reusable views and controllers that are significantly more than the "controls" or "widgets" in toolkit libraries; that separating the V and the C for widgets only has modest value (although it does have some); that for more domain-specific code, design dependencies inevitably arise between the M, V, and C; that the idea of multiple simultaneous Vs/Cs for a single M can run into problems with synchronization (which, indeed is a major problem for all UI frameworks, not just MVC); and that despite all these shortcomings, separating the M, V, and C is good design discipline that can produce substantially more understandable systems. [15]

The key point here is that the only *compelling* reason to separate the implementation of model, view, and controller is for a system that

- supports multiple simultaneous views, and
- does not have a simple one-to-one mapping between views and controllers.

Most systems don't have these requirements. The SUIT user interface toolkit [60] has multiple views of a small set of simple widgets, but each view has a different interaction style. The Chiron user interface management system has a more general mechanism that supports multiple *agents* for the same data object. Each artist manages both presentation and control for its object. Thus, in both SUIT and Chiron, the model is separated, but view and controller are intertwined. This makes sense, because for these systems, the differences between two presentations of the same object are likely to be fairly substantial. For instance, an integer object might be controlled by a slider widget or a dial widget. These two widgets are operated by substantially different mouse motions, so each one will probably need its own, specialized controller.

In contrast, two presentations of the same document are probably not so different from each other. For instance, one-column and two-column presentations of the same text document have substantially different appearances, but the text-editing and formatting commands that are available for them could be nearly identical. While the presentation part of the user interface is different, the control part is the same. Thus, it makes sense for Ensemble to separate presentation and control.

Chapter 4

The Design of Proteus

Proteus[1] is the module of Ensemble that interprets appearance specifications for documents. Proteus is central to Ensemble's support of multiple presentations and is the first system for handling formal appearance specifications that is designed to be adapted to the needs of different media.

This chapter describes Proteus in detail. It begins with a description of Ensemble's presentation subsystem, of which Proteus is a part. Then Section 4.2 presents the architecture of Proteus. The language in which presentation schemas are written is described in Section 4.3. Finally, Section 4.4 describes the fundamental elements of media and how they are exploited to allow Proteus to adapt to the needs of different media.

4.1 Presentation Subsystem Architecture

The role of Ensemble's presentation subsystem is to make formatting decisions and then display the results by putting pixels on a screen or ink on a page. The design of the presentation subsystem is centered around two goals:

- support for multiple presentations, and
- minimizing the effort needed to add new media to Ensemble.

From a user's point of view, the presentation process involves three related abstractions: documents, presentations, and views. Their relationship is illustrated by Figure 4.1.

In Ensemble, a *document* is a tree-structured collection of content material. Each node of the document tree has a type (such as "procedure keyword" in a program document or "section title" in a book document) and the document's content (e.g. text, video clips) is

[1]Proteus was a sea god of the Greeks who lived either on the island of Pharos (according to Homer) or on the island of Carpathus (according to Virgil). He was completely omniscient but also secretive. In order to consult him it was necessary to surprise and bind him during his noontime nap. Upon awakening he would attempt to escape by assuming a variety of shapes such as that of a lion, a serpent, a leopard, a boar, a tree, fire, or water. If unable to escape by shape-changing, he would return to humanoid form, answer the question and dive into the sea. [16]

The Proteus system was named for its adaptability, but unlike its namesake, it is also cooperative.

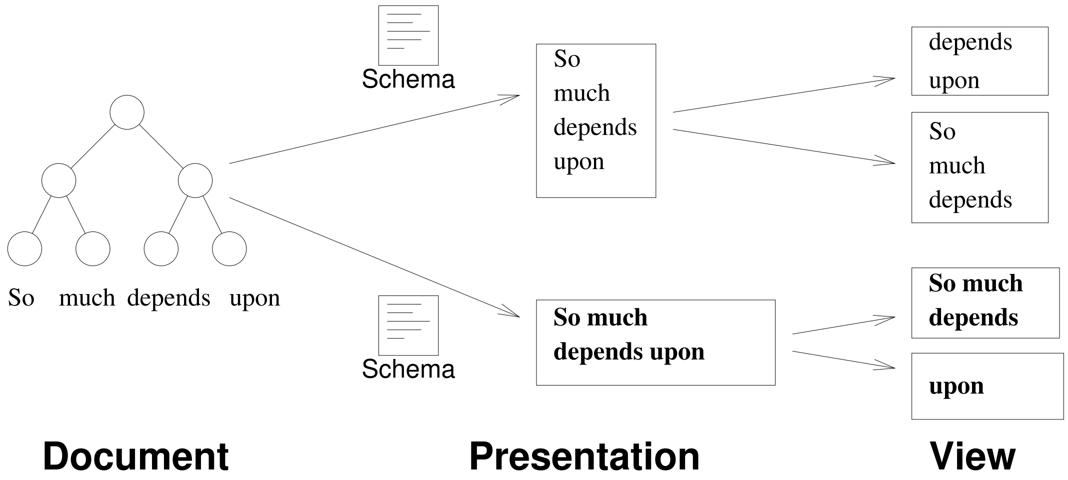


Figure 4.1: The relationship between documents, presentations, and views in Ensemble.

kept in the leaves of the tree. Documents are persistent; that is, they can be saved in one Ensemble session and restored in the identical state in another Ensemble session.

Each document belongs to a *document class*, such as letter, article, television advertisement, or C program. Most document classes are defined formally by context-free grammars called *structure schemas*. A structure schema specifies the possible configurations of members of the document class. Structure schemas may define ambiguous grammars. This is not a problem for Ensemble because document classes defined by structure schemas can only be created by top-down editing operations. Programs, and other document classes whose tree structure is determined by analyzing a text stream, are defined using unambiguous grammar specifications that are somewhat more complex than structure schemas. However, the user can usually ignore this distinction.

As an example, Figure 4.2 shows the structure schema for a simple memorandum document class. It states that a memorandum (called "Memo") is composed of four elements called "To", "From", "Subject", and "Body". The Body element is composed of one or more "Paragraph" elements. The To, From, Subject, and Paragraph nodes are *primitives*, meaning that they represent primitive content data that can be presented to the user. They are all of the "text" primitive type.

Presentations are objects that embody a particular way of drawing a document. Each one is essentially a different representation of the document. Any number of presentations may be active at the same time for a particular document. Presentations might differ only in fairly subtle ways, such as the use of different fill colors in a graphical presentation, but it is also possible to present the same document in multiple media at the same time. For instance, a document containing only text can be formatted into paragraphs or it can be formatted as a tree (i.e. a two-dimensional graphics presentation) in order to see its hierarchical structure more directly.

Presentations "draw" the document abstractly in a virtual space that has no actual connection with the user's display or a particular printer. The appearance of the presentation is primarily determined by a *presentation schema*, which is a formal specification describing

```
SCHEMA Memo;

Memo: To From Subject Body;
To = text;
From = text;
Subject = text;
Body: Paragraph+;
Paragraph = text;
```

Figure 4.2: The structure schema for the memorandum document class.

a formatting style for the document's class. Presentation schemas are handled by a module of the presentation subsystem called Proteus, which is described in detail in Section 4.3.

The user sees each presentation through one or more *views*. A view is a window onto a presentation drawn on a particular device. Due to constraints on the space available on a device, a view may be able to show only a small portion of the entire presentation. Naturally, views are also the objects to which user interface elements like menus, scrollbars, and buttons are attached. The presentation subsystem manages only the drawing of the presentation on the view. It leaves the management of these user interface objects to the control subsystem.

Figure 4.3 shows how information flows between presentations, views, and the rest of Ensemble during a user's session. Whenever a document changes, an event is fired announcing the change. Each presentation handles the event and determines whether the change is relevant to it. If so, it performs any required reformatting, keeping track of which areas of its virtual space have changed in the process. Once the reformatting is complete, the presentation fires an event of its own. This event specifies only the bounding box of the changed portion of the virtual space.

This "bounding box changed" event is handled by each of the views on that presentation, which compare their own bounding boxes to the one specified by the event. If, for a particular view, these two bounding boxes overlap then the view must redraw the region of the overlap. It does this by asking the presentation for all the formatted data that falls in the overlap area. The view translates the formatted data from the device-independent (or virtual space) terms of the presentation to the terms of the device (e.g. X11/Tk).

In addition to responding to changes in the document, the presentation subsystem must also respond to rendering and backmapping requests from the extension language subsystem. Both types of requests are sent directly to a single view. When this view receives a rendering request, it responds by simply drawing the appropriate portion of the document on the screen. It may do this using cached data or by asking its presentation for the appropriate formatted data.

The process of responding to a backmapping request is somewhat more complex. A backmapping request is always made in terms of the coordinate system used by the device (e.g. in pixel coordinates for X11). The view translates these device coordinates into the coordinates of the presentation's virtual space and then passes the backmapping request

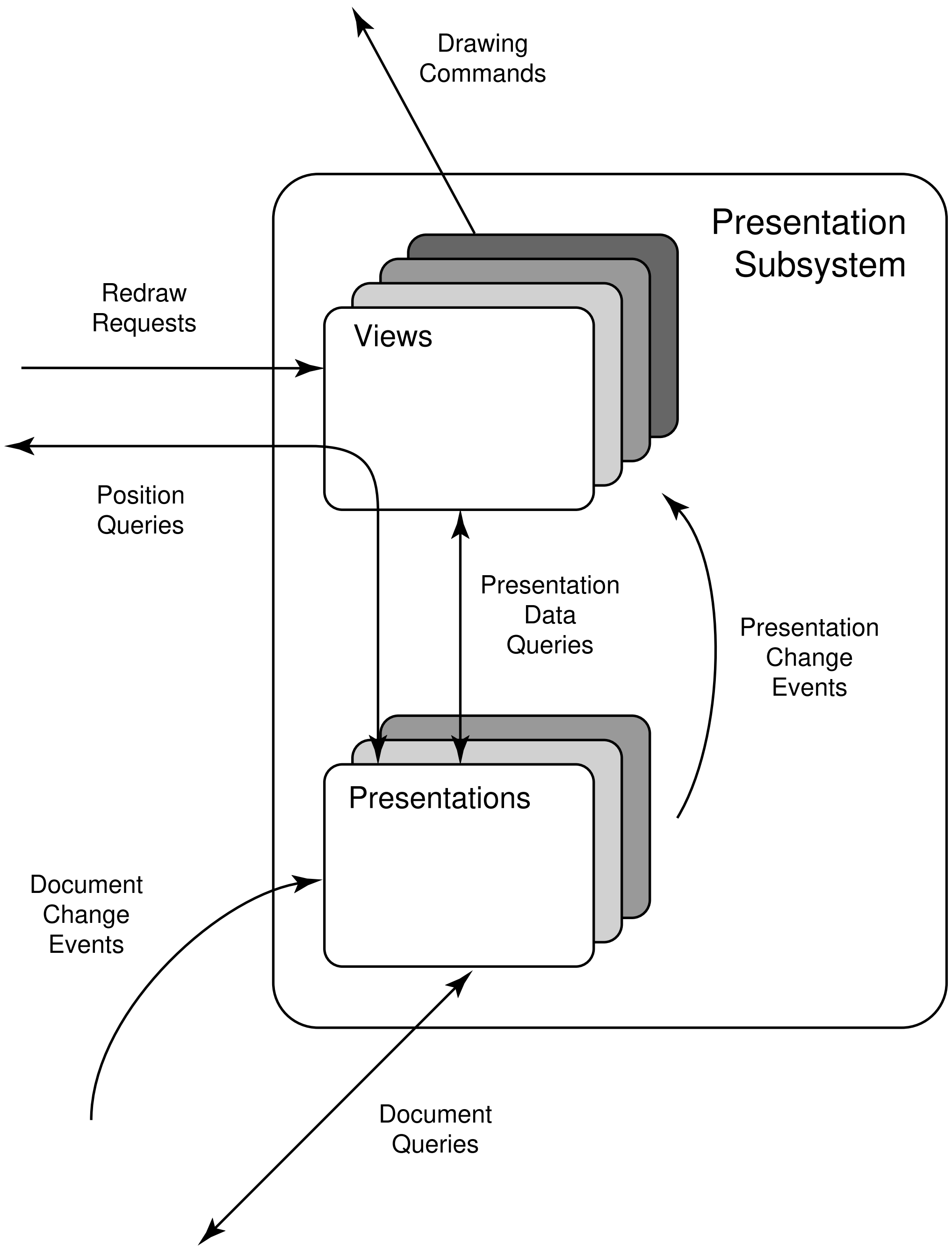


Figure 4.3: **Information flow within Ensemble's presentation subsystem.** Single-pointed arrows represent commands. Double-pointed arrows represent call-and-return interactions. Proteus is not shown.

on to the presentation. The presentation determines what component(s) of the document are under the point or in the region described in the request and returns a handle describing those objects to the view, which passes this result through to the extension language subsystem.

As in the overall architecture of Ensemble, separation of concerns is the central paradigm of the architecture of the presentation subsystem. Presentations are only concerned with creating an abstract layout of the document in a virtual space. They announce changes in their abstract layout but have no idea whether any other entities are consuming the announcements. Views are concerned with translating the abstract formatting data created by presentations into the concrete form required by a device. They have no knowledge of the relationship between the formatting data and the underlying document.

4.1.1 Compound Documents

The previous discussion of the relationship between presentations and views was incomplete because it ignored an important problem: compound documents. *Compound documents* are documents that include other documents. Support for compound documents is currently an area of active commercial development with Microsoft's Object Linking and Embedding (OLE) [48] and Apple's OpenDoc [54] standards competing for primacy in the marketplace. Ensemble has a simple compound document model that bears some resemblance to these standards, but was designed independently. Where Ensemble moves ahead of these standards in its adherence to a single model for all documents. In contrast, OLE and OpenDoc only define a protocol for interaction between documents created by different programs.

The problem that compound documents present to Ensemble's presentation subsystem lies in the handling of elements of the user interface like menu bars, scrollbars, and buttons. Consider a compound document like the one whose tree is shown in Figure 4.4a. The root document contains a total of 3 sub-documents. Together these four documents form a smaller tree called the *compound document tree*, seen in Figure 4.4b. The formatting of a compound document is performed by an isomorphic *compound presentation tree*, shown in Figure 4.4c. The bounding boxes of these presentations in virtual space have a nesting relationship corresponding to the structure of this tree.

The natural next step would be to have another isomorphic tree, the *compound view tree*, through which each of the presentations would be rendered on the screen. The problem with this approach is that Ensemble's view objects also include the support for menu and scroll bars and other user interface elements. A naive composition of compound views would place these elements around the rendering of every subdocument (as shown in the mock-up in Figure 4.5). This would prevent users from seeing compound documents in their final form, since a printed version of the document should certainly not include user interface gadgets.

What was needed was to create a separation between the two tasks performed by the view objects. This was done by creating a new type of object called a *Rendition*. Renditions embody the mapping between a presentation and an output device and handle all drawing that is performed in the application window. Thus, there is a *compound rendition tree* that is isomorphic to the other compound trees (shown in Figure 4.4d). The *View* objects

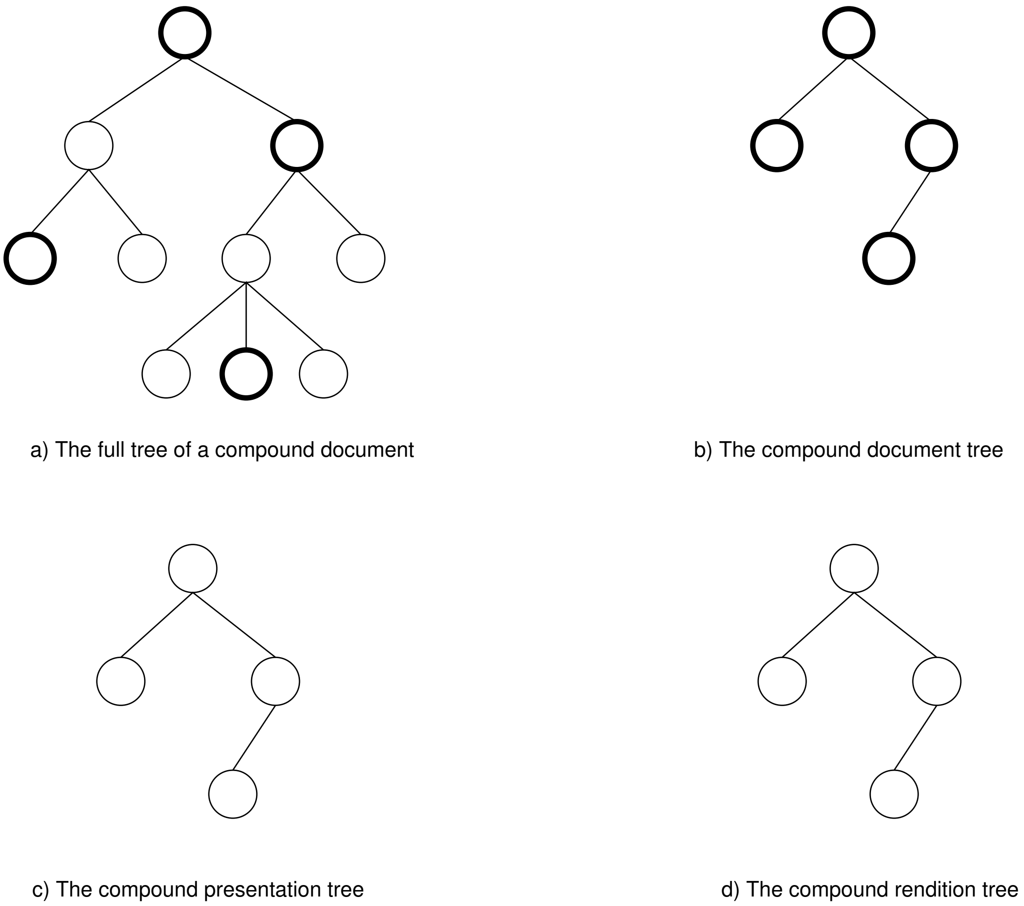


Figure 4.4: The isomorphic relationship between the trees of documents, presentations, and renditions. Part a) shows a compound document's full tree. The root nodes of the various sub-documents are drawn with thick outlines. Part b) shows the compound document tree that is embedded in the full tree. Parts c) and d) show the isomorphic compound presentation and compound rendition trees, respectively.

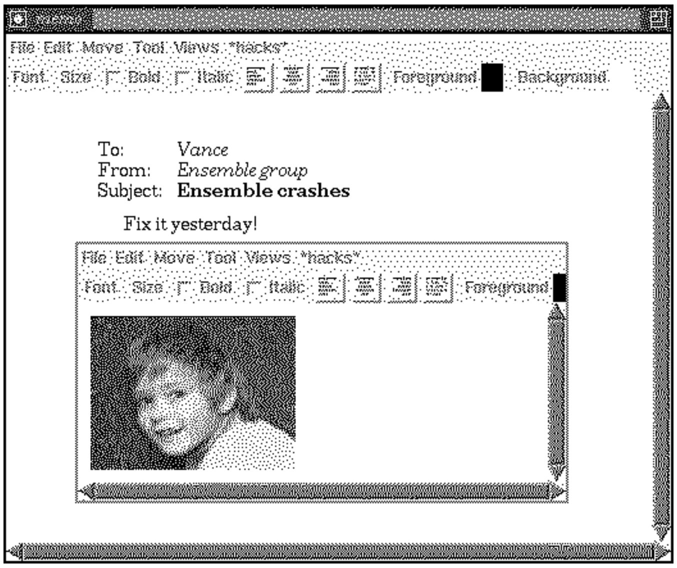


Figure 4.5: Mock-up of how Ensemble would look without rendition objects.

remain, but there is only one View per top-level user interface window. A View embodies the connection between the presentation subsystem and an output device. It provides the application window on which the various Renditions draw the document and it provides the user-interface wrapper of menus, buttons, and scrollbars.

4.2 Architecture

Proteus is a service used by most, but not all, Ensemble presentations. Architecturally speaking, Proteus sits between the presentation and the document, as shown in Figure 4.6. The inputs to Proteus are the document and a presentation schema. Based on the specifications in the presentation schema, Proteus performs two primary actions:

- It creates a *presentation tree*, which is an elaborated version of the original document tree. Every node in the document tree will be found in the presentation tree. In addition, the presentation tree may contain other nodes that are not found in the document tree, but rather are generated according to *tree elaboration* specifications in the presentation schema.[2]

- It maintains values of formatting and layout attributes for all the nodes of the presentation tree. Proteus's presentation client can query Proteus for these values at any time.

[2]It should be noted that even though the presentation tree contains every node in the document tree, some of those nodes may not be visible in the presentation because the presentation schema specifies that they should be elided.

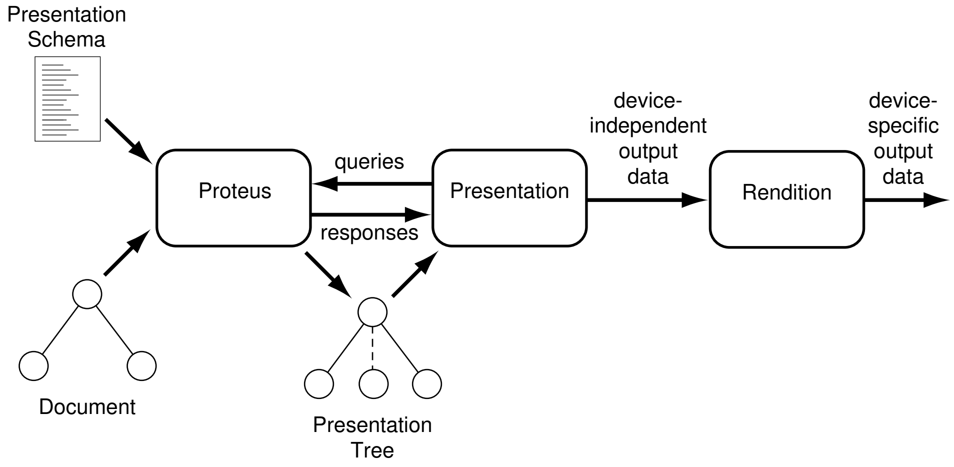


Figure 4.6: The relationship between Ensemble documents, Proteus, and presentations.

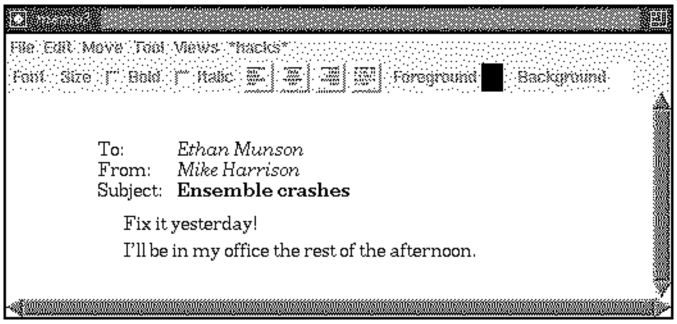


Figure 4.7: A presentation of the memorandum document.

4.3 Presentation Schemas

Proteus is most easily understood by examining what presentation schemas can specify. So, this section describes the syntax and semantics of the presentation schema language for the text medium. It uses a presentation of a simple memorandum as a running example. The presentation is shown in Figure 4.7. The tree of the document being presented can be seen in Figure 4.8. Note that this document tree corresponds to the structure schema given in Figure 4.2.

The application of Proteus to the presentation of programs is illustrated by a separate example in Appendix A.

4.3.1 Overview

Proteus provides three primary services: attribute propagation, box layout, and tree elaboration.

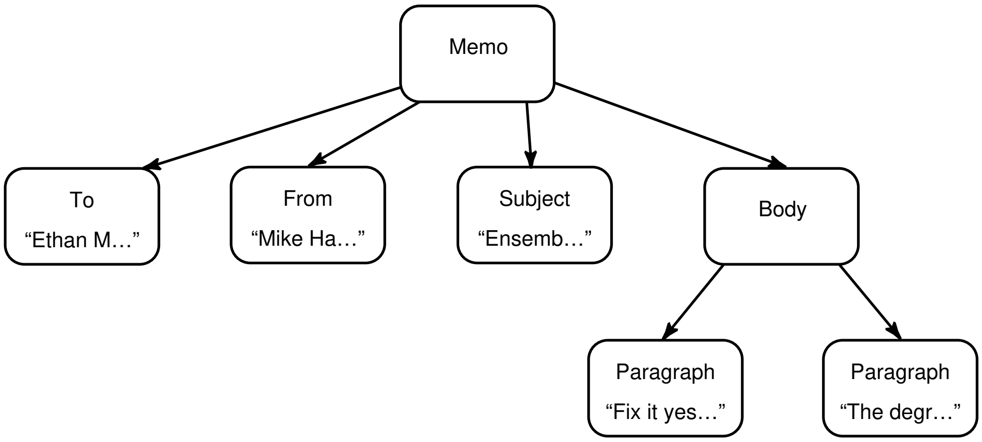


Figure 4.8: The document tree for the memorandum document.

- The *attribute propagation* service is used to specify parameter values for formatting operations. In the text medium, there is only one formatting operation, the *line-breaking* operation, but it has many parameters. The parameter values of one node can be based on the values of other nodes. This process is called *propagating* the parameter values.

- The *box layout* service is used to lay out the elements of the document in the medium's k-dimensional space. The text medium is two-dimensional, so box layout is performed in the horizontal and vertical dimensions.

- The *tree elaboration* service is used to automatically generate material that needs to be visible, but is not part of the document itself. For instance, in the memorandum presentation example of Figure 4.7, the labels for the To, From, and Subject fields are generated. The generated material is represented by nodes that are added to the presentation tree but not to the document tree.

Proteus provides a fourth, subsidiary service, the *interface function* service, which can be used to give presentation schemas access to information maintained by other parts of the Ensemble system.

A presentation schema is a simple, declarative specification of appearance for a document class defined separately by a structure schema. A presentation schema specifies how the services of Proteus should be applied to members of the document class. The outline of the presentation schema for the sample memorandum presentation is shown in Figure 4.9. The elided portions of the presentation schema are given subsequently. A presentation schema has four sections: header, defaults, boxes, and rules. The header section declares the schema's medium, its name, and the name of the structure schema to which it corresponds. The DEFAULTS section defines default presentation rules to be used when there are no specific rules for a node. The BOXES section declares the types of nodes that can be generated by tree elaboration. Finally, the RULES section is used to define specific presentation rules for each of the node types defined in the document class.

```
MEDIUM text;
PRESENTATION memo FOR memo;

DEFAULT
 ...

BOXES
 ...

RULES
 Memo :
  ...
 To :
  ...
 From :
  ...
 Subject :
  ...
 Body :
  ...
 Paragraph :
  ...
END
```

Figure 4.9: Outline of a presentation schema for the memorandum presentation.

```
BOXES
 ToLabel : Text("To:") :
  BEGIN
  END;

 FromLabel : Text("From:") :
  BEGIN
  END;

 SubjectLabel : Text("Subject:") :
  BEGIN
  END;
```

Figure 4.10: The tree elaboration rules for the memorandum example.

In general, the language for presentation schemas is case-insensitive. The only effect of whitespace in the schema is to separate tokens: line breaks and indentation have no other meaning.

4.3.2 Tree Elaboration

The tree elaboration service creates and maintains the elaborated version of the document tree called the presentation tree. The presentation tree is identical to the document tree except that it contains additional nodes that are generated in accordance with specifications in the presentation schema. The tree elaboration rules for the sample memorandum presentation are shown in Figure 4.10. The presentation tree that results from the application of these rules is shown in Figure 4.11. Tree elaboration is specified in two steps. The first step is the declaration of the nodes that can be generated in the BOXES section of the

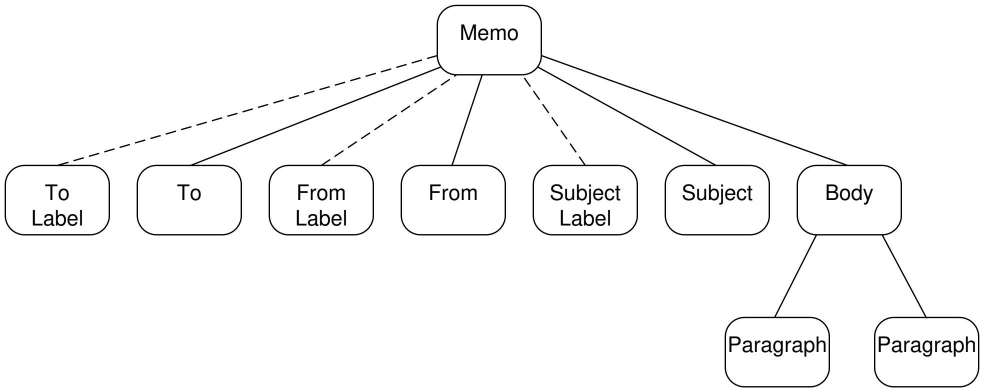


Figure 4.11: The presentation tree for the memorandum example. Arcs that are present in both the document and presentation trees are solid. Arcs that are only found in the presentation tree are dashed.

presentation schema. In the memorandum presentation schema, this section declares three new node types: ToLabel, FromLabel, and SubjectLabel. The ToLabel node is declared as follows:

```
ToLabel : Text("To:") :
     BEGIN
     END;
```

The first line declares a new node type called "ToLabel" which is a primitive text node whose content is the string "To:". The name of the new node type can be any name that is neither a schema language keyword or already used as a node name. Any type of primitive node supported by the schema's medium may be generated. The text medium of Ensemble only supports one type of primitive, text. The specification of the node's content could have used any expression whose value was a string.

The remainder of the declaration is for defining attribute propagation and box layout rules. In this particular schema, the generated nodes don't have attribute or box layout rules, so nothing appears between the `BEGIN` and `END` keywords.

The generated nodes actually appear in the presentation tree because of the presence of creation commands in the RULES section of the schema. The ToLabel node is added to the tree because the RULES section contains a creation command for the To node type:

```
To :
     BEGIN
     CreateBefore(ToLabel);
     ...
     END;
```

The `CreateBefore` command causes a node of type ToLabel to be created as a left sibling of the To node. The same command is used to create the labels for the From and Subject nodes. There are three other creation commands. The `CreateAfter` command creates right

```
RULES
   Memo :
        BEGIN
        FontFamily = "new century schoolbook";
        Size = 14;
        Bold = No;
        Italic = No;
        END;
   To :
        BEGIN
        Italic = Yes;
        END;
   From :
        BEGIN
        Italic = Yes;
        END;
   Subject :
        BEGIN
        Bold = Yes;
        END;
```

Figure 4.12: The attribute rules affecting font choice from the memorandum presentation schema.

siblings, while the `CreateFirst` and `CreateLast` commands create first and last children, respectively.

4.3.3 Attribute Propagation

The central operation of the text medium is the line-breaking of paragraphs. The line-breaking operation requires some text and a number of parameters that specify font, hyphenation, justification, line spacing, and the width of the line. From these inputs, it converts the words of the text into lines of characters to be drawn on a screen or page. The values of the line-breaking operation's parameters are defined using attribute rules that control the behavior of Proteus's attribute propagation service.

For instance, the rules that defined the fonts used in the sample memorandum presentation are shown in Figure 4.12. These rules set the font of the root node of the `Memo` document to be 14 point New Century Schoolbook in the Roman style (which is neither bold nor italic). These values are propagated to the rest of the presentation tree by simple inheritance. Certain nodes override their inherited values: the To and From nodes give the Italic attribute a value of `Yes`; the Subject node sets the value of the Bold attribute to be `Yes`.

Attribute Rules

Attribute rules have a simple syntax:

<*attribute name*> `=` <*expression*> `;`

The attribute named in the left hand side of the rule must belong to the presentation schema's medium. Each medium has its own set of attributes, determined by the demands of its formatting operations. For instance, Ensemble's text medium currently has 15 attributes controlling font (FontFamily, Size, Bold, and Italic), hyphenation (Hyphenate, MinHyph, MinLeft, MinRight), justification, indentation, line-spacing, visibility, foreground color, background color, and the position of the right margin. Each attribute has a type, which is either boolean, string, real, or a medium-specific enumeration type. The set of attributes available for each medium is specified by that medium's designer (who is a programmer whose code is part of the Ensemble system). This designer is expected to document the presentation features of the medium so that presentation schemas may be written for it.

The expression on the right hand side of the attribute rule can be any expression whose type is the same as the attribute named on the left hand side. Expressions can be constructed using a variety of operations common to general-purpose programming languages:

arithmetic operators: addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and modulus (`%`) ;

comparison operators: greater than (`>`), greater than or equal (`>=`), less than (`<`), less than or equal (`<=`), equality (`==`), inequality (`!=`), case-insensitive string equality (`=`), case-insensitive string inequality (`!`);

boolean operators: logical and (`&&`), logical or (`||`) ;

trigonometric functions: Sin, Cos, Tan, Arcsin, Arccos, Arctan ;

miscellaneous real functions: Min, Max, Trunc, Round, Abs ;

Defining Attribute Constraints

Attribute values can be constrained to depend on other node's attribute values. This is done using the attribute access expression, for which the syntax is:

<*node expression*> `.` <*attribute name*>

The value of an attribute access expression is the value of the named attribute for the node returned by the expression on the left hand side of the dot. For instance, the following expression gives the value of the parent node's `Size` attribute:

```
Parent . Size
```

There are no attributes whose type is "node", but there are many functions that return nodes, all of which can appear in the left hand side of an attribute access expression.

Tree Navigation Functions: The `Parent`, `LeftSib`, `RightSib`, `FirstChild`, and `LastChild` functions return the corresponding neighbors in the presentation tree. The tree navigation functions have two forms. One form takes a single argument of type node. The other form doesn't require an explicit argument. Instead, it uses the *defining node* as its argument. The defining node is the node for which the attribute was defined.

Creator: If the argument to the `Creator` function is a generated node, it returns the node whose creation operation caused the generation. The `Creator` function has the same two forms as the tree navigation functions.

Self: The `Self` function take no arguments and returns the node for which the attribute is defined. The `Self` function is typically used to define a constraint between attributes of a single node or to pass the defining node to a function requiring an explicit node argument.

Root: The `Root` function returns the root node of the tree.

NthChild: The `NthChild` function takes two arguments, a node and a number, and returns the child of the node whose zero-based index equals the number.

AncestorOfType: The `AncestorOfType` function takes two arguments, a node and a string. It searches the path from the node to the root of the tree for an ancestor whose type name matches the string. It returns that ancestor node.

These functions are designed to allow the specification of constraints between the defining node and every other node in the tree. It is easiest to define constraints with neighboring nodes, since the tree navigation functions and the `NthChild` function can be used to specify all immediate neighbors. More distant nodes in the tree can be specified by composing those functions, as in

```
FirstChild(LeftSib(Parent)) . Size
```

which specifies the `Size` attribute of a "cousin" node. Distant nodes can also be specified using the `Root` and `AncestorOfType` functions and, for generated nodes, the `Creator` function. The most common use of these functions is to define the layout constraints of the documents. This process is described in Section 4.3.4.

Conditional Rules

All the presentation rules (attribute, tree elaboration, and box layout) for a single node type are defined together in a rule block. The formal syntax for a node type's rules is:

<*node type name*> `: BEGIN` <*rule list*> `END ;`

Typically, the rule block is composed of a series of rules, each terminated by a semi-colon.

It is also possible to define sets of alternate presentation rules using a conditional syntax. For instance, the following rules make a Paragraph node's text appear in red whenever the Paragraph is formatted in a bold font:

```
Paragraph:
   BEGIN
        FgColor = "Black" ;
        IF (Self . Bold == Yes) THEN
           FgColor = "Red" ;
        ENDIF
   END;
```

Conditionals may be nested and have a simple syntax.

```
IF <condition> THEN <rule list>
[ ELSIF <condition> THEN <rule list> ] +
[ ELSE <rule list> ]
ENDIF
```

The semantics of attribute rules that appear in conditionals are analogous to those for assignment statements in imperative programming languages. The conditions are tested in the order written. The first block whose condition is true has its rule list activated and the remaining blocks are ignored. The `ELSE` block is activated if no preceding conditions are true. Any attribute rules that appear in the activated rule list override all previous definitions of the same attributes. Any unguarded rule overrides all rules that appear before it.

When tree elaboration commands appear inside conditionals, they *do not* override previous tree elaboration commands. Instead, all unguarded creation commands are performed and any creation commands that appear in the activated rule list of a conditional are also performed. This treatment of conditional tree elaboration makes creation commands analogous to output statements in an imperative programming language and allows a single node to generate multiple nodes. The utility of this feature in described in Section 6.1.1.

The most important use of conditionals is to make presentation rules that are context sensitive. The RULES section of the presentation schema allows attribute rules to be defined for each node type. However, the document's structure schema may allow the same node type to appear in many different contexts. For instance, suppose that the structure schema for the Article document class included the following rules:

```
Section : Title Preamble Subsection+ ;
Preamble : Paragraph+ ;
Subsection : Title Paragraph+ ;
Title = text ;
Paragraph = text ;
```

These definitions allow Title nodes to appear as children of either Section or Subsection nodes. It is probably desirable to have the Section titles be larger than the Subsection titles. This effect could be achieved with the following conditional rules:

```
Title :
   BEGIN
        IF (TypeOf(Parent) == "Section") THEN
```

```
            Size = 18;
        ELSE
            Size = 14;
        ENDIF
   END;
```

The `TypeOf` function returns the type name of its node argument and is important for constructing context-sensitive conditionals.

Default Rules and the Order of Evaluation

Proteus's presentation schema language has a system of default rules that help reduce the size of presentation schemas. There are two sets of default rules, explicit and implicit.

Explicit default rules are defined in the DEFAULTS section of the schema. The syntax of this section is:

DEFAULTS BEGIN <*rule list*> END ;

The rule list has the same syntax and semantics as rule lists for specific node types.

The rules defined in the DEFAULTS section are used primarily when the node type does not have a rule defined for the attribute. They are also used when the node-specific rule *fails*. A rule fails when its expression cannot be computed for some reason. This could occur because of an arithmetic error (such as division by zero), but most commonly it results from tree navigation. For instance, a node's first child has no left sibling, so if the `LeftSib` function is invoked with a first child as its argument, the function fails.

Figure 4.13 shows a fragment of a possible presentation schema for memorandum documents. In this example, the `Paragraph` node type doesn't have a `Size` rule of its own, so the explicit default rule for `Size` is used, giving a value of 12.

The *implicit default rule* is used when either there is no explicit default rule for an attribute or the rule fails. The implicit default rule uses simple inheritance, which is equivalent to a rule of the form

```
Attribute = Parent . Attribute ;
```

In the example in Figure 4.13, the implicit default rule causes `Paragraph` nodes to have a value of "times" for the `FontFamily` attribute, since, in memoranda, `Paragraph` nodes are always children of `Body` nodes.

So far, a three-stage process for finding an attribute's value has been described, but the complete process has five stages. The attribute evaluation algorithm tries each of the five stages in order, stopping as soon as any stage returns a valid value. The five stages are:

- User override,
- Node-specific rule,
- Explicit default rule,
- Implicit default rule,
- Global default value.

```
DEFAULTS
   BEGIN
   Size = 12 ;
   END;
RULES
   Body:
        BEGIN
        FontFamily = "times";
        END;
   Paragraph:
        BEGIN
        Bold = No;
        END;
END
```

Figure 4.13: Sample presentation schema code illustrating use of explicit and implicit default rules.

A *user override* is an attribute value specified explicitly by the user for a particular node of a document. Overrides are stored with the document object, so they persist from one Ensemble session to another. The first stage of the attribute evaluation algorithm looks for a user override. If one is found, its value takes precedence over any specifications in the presentation schema.

The behavior of the next three stages has already been described. In the second stage, the algorithm checks for a node-specific rule for the attribute. If there is one and it doesn't fail, the value is returned. Otherwise, the third stage follows the same procedure with the explicit default rule. If there is no explicit default rule or the rule fails, the implicit default rule is used.

The final stage exists because even the implicit default rule can fail. This can only occur for the root node of the document and it is a rare event for sensibly-designed presentation schemas. Still, in order to assure that a valid value will always be returned, the fifth stage returns a *global default value*. Each attribute has a global default value specified by the medium's designer. For instance, in Ensemble's text medium, the global default values for the `Size` attribute and `FontFamily` attributes are 12 and "new century schoolbook", respectively.

4.3.4 Box Layout

In Proteus, the *box layout* service is used to specify the positions of the elements of the document. The box layout service is based on a model of nested boxes. In this model, each node in the presentation tree has a bounding box, which, for the text medium, is the smallest rectangle that encloses all of the text of the node. The nodes are laid out by defining constraints between these bounding boxes.

The box layout service defines four attributes for each dimension supported by the medium. The generic names of these attributes are *extent*, *minimum*, *maximum*, and *center* but each medium renames them. For example, the text medium has two dimensions *Horizontal* and *Vertical*. The attributes of the Horizontal dimension are `Width`, `Left`, `Right`, and `HMiddle` while those of the Vertical dimension are `Height`, `Top`, `Bottom`, and `VMiddle`.

While each dimension logically has four attributes, there are only two degrees of freedom among them. Thus, some special handling is required for dimensional attributes. Proteus adopts the solution used in Grif's presentation language, P [65], which is to give each dimension only two real attributes, *extent* and *position*. The three non-extent attributes (minimum, maximum and center) are given a subsidiary role to the position attribute and are called *points*. In the text medium, the position attributes for the horizontal and vertical dimensions are called `HorizPos` and `VertPos`, respectively. The points corresponding to `HorizPos` are `Left`, `Right`, and `HMiddle`.

Extent attributes are handled just like other attributes in the presentation schema language, but rules for position attributes are defined with a special syntax:

<*position name*> : <*point name*> = <*expression*> ;

In the text medium's horizontal dimension, a possible rule would be:

```
HorizPos: Left = LeftSib . Right;
```

This rule would constrain its node's left edge to be aligned with the right edge of its left sibling. Notice that point names, not position names are used on the right-hand side of attribute access expressions.

The box layout service supports one other special feature, the `AllChildren` function. A common rule definition using this function is:

```
Width = AllChildren . Width ;
```

The informal meaning of this rule is that "This node is as wide as its children are". The technical meaning is this

- The `Left` and `Right` points of each of the node's children are computed.
- Among these values, the most extreme left edge and most extreme right edge are identified.
- The difference between these two values is returned.

The `AllChildren` function was originally conceived as one of a class of functions returning *node sets*, which are unordered sets of nodes. However, to date, no other uses of this concept have been found to be critical for use in presentation schemas. Furthermore, no consistent semantics has been found to date for applying the `AllChildren` function to non-extent attributes. As a result, the `AllChildren` function remains a special case in the language for which a better alternative would be desirable.

```
DEFAULT
   BEGIN
   Width = AllChildren . Width;
   Height = AllChildren . Height;
   VertPos: Top = LeftSib . Bottom;
   END;
RULES
   Memo :
BEGIN
Width = 432;
HorizPos: Left = 72;
VertPos: Top = 36;
END;
   To :
BEGIN
HorizPos: Left = RightSib(RightSib) . Left;
VertPos: Top = LeftSib . Top;
END;
   From :
BEGIN
HorizPos: Left = RightSib(RightSib) . Left;
VertPos: Top = LeftSib . Top;
END;
   Subject :
BEGIN
HorizPos: Left = LeftSib . Right + 12 ;
VertPos: Top = LeftSib . Top;
END;
   Body :
BEGIN
VertPos: Top = LeftSib . Bottom + 10 ;
END;
   Paragraph :
BEGIN
VertPos: Top = LeftSib . Bottom + 5 ;
END;
```

Figure 4.14: Box layout rules from the memorandum presentation schema

Layout of the Memorandum Example

The box layout rules that created the memorandum presentation are shown in Figure 4.14. This section examines these rules closely in order to illustrate how the box layout is typically used.

The left edge of the `Memo` was set explicitly to be one inch (72 points) from the left edge of the page with the rule

```
HorizPos: Left = 72;
```

The three label nodes and the two `Paragraph` nodes get their horizontal position using the implicit default rule, so they inherit their position from the `Memo` node.

The presentation aligns the left edges of the `To`, `From`, and `Subject` nodes and puts them far enough to the right that they don't overlap their labels. To achieve this effect, it is first necessary to recognize that the `SubjectLabel` node is the widest of the three labels. So, the next constraint places the `Subject` node's text a bit to the right (12 points) of the `SubjectLabel` node's right edge.

```
HorizPos: Left = LeftSib . Right + 12;
```

Then, the left edges of the `To` and `From` nodes are constrained to line up with the left edge of the `Subject` node by giving them rules of the form:

```
HorizPos: Left = RightSib(RightSib) . Left ;
```

This, constrains the left edge of the `From` node to be aligned with the left edge of the `Subject` node and then transitively aligns the `To` node with the `From` node.

The vertical positions of the presentation's elements are based on a more linear chain of dependencies. The default rule for vertical positioning is

```
VertPos: Top = LeftSib . Bottom;
```

This rule fails for the `ToLabel` node (because it has no left sibling), so the implicit default rule is used, which aligns the top of the `ToLabel` node with the top of the `Memo` node. The `To` node's top is aligned with that of the `ToLabel` node via the rule

```
VertPos: Top = LeftSib . Top;
```

The same two rules apply to the `FromLabel` and `From` nodes and the `SubjectLabel` and `Subject` nodes. The only difference from the first line is that the default vertical position rule doesn't fail, so the `FromLabel` node's top is aligned with the bottom of the `To` node and similarly with the Subject line.

The vertical position rule for `Paragraph` nodes is

```
VertPos: Top = LeftSib . Bottom + 5 ;
```

This rule places 5 points of space between the two `Paragraph` nodes. However, it fails for the first `Paragraph` node, as does the explicit default rule. So, the implicit default rule is used which makes the first `Paragraph` use the value of the `Body` node's top. This value is determined by the rule

```
VertPos: Top = LeftSib . Bottom + 10 ;
```

The `Body` node's left sibling is the `Subject` node, so the first `Paragraph` is placed 10 points below the bottom of the `Subject` node. The net result is an attractive alignment of the paragraphs of the memorandum body, with a little extra space between the paragraphs and slightly more space between the memorandum header and the first paragraph.

4.3.5 The Complete Presentation Schema

By incorporating the specifications of tree elaboration, attribute propagation, and box layout (given respectively in Figures 4.10, 4.12, and 4.14), we get the complete presentation schema for the memorandum example given in Figure 4.15.

4.3.6 Interface Functions

The three services of Proteus that have been described so far are quite powerful, but there is also a considerable amount of information that can be relevant to presentation that they don't manage. So, Proteus provides a fourth service that allows the presentation schema language to be extended. It is called the *interface function* service because it provides support for functions that cross the interface between Proteus and its presentation clients.

As an example, Ensemble's text medium defines an interface function called `URoman` that takes a numeric argument and returns a string containing the upper-case roman numeral representation of the number. If the number is not an integer, it is rounded to the nearest integer before conversion. This function can be used to number the paragraphs of the memorandum document by declaring a generated `ParaNumber` node whose content is the upper-case roman numeral representation of its creator's child number (incremented by one to compensate for the zero-based child number values):

```
ParaNumber : Text(URoman(ChildNum(Creator) + 1)) :
```

and then creating the ParaNumber as the left sibling of each Paragraph in the memorandum's body:

```
Paragraph :
     BEGIN
     CreateBefore(ParaNumber);
     END;
```

In presentation schemas, interface functions can be used just like any other expression. The `URoman` function can be used in any context where a string-valued expression is appropriate. Its argument can be any expression returning a number.

In the current version of Ensemble, the text medium defines `URoman` and two similar functions, `LRoman` and `Arabic`, but otherwise, interface functions are not widely used. However, three types of uses are expected:

```
MEDIUM text;
PRESENTATION memo FOR memo;

DEFAULT
 BEGIN
 Width = AllChildren . Width;
 Height = AllChildren . Height;
 VertPos: Top = LeftSib . Bottom;
 END;

BOXES
 ToLabel : Text("To:") :
  BEGIN
  END;

 FromLabel : Text("From:") :
  BEGIN
  END;

 SubjectLabel : Text("Subject:") :
  BEGIN
  END;

RULES
 Memo :
  BEGIN
  Width = 432;
  HorizPos: Left = 72;
  VertPos: Top = 36;
  FontFamily =
      "new century schoolbook";
  Size = 14;
  Bold = No;
  Italic = No;
  LineSpacing = 1.2;
  Justify = LeftJustify;
  Indent = 0;
  Visible = Yes;
  RightMargin = 504;
  END;
 To :
  BEGIN
  CreateBefore(ToLabel);
  HorizPos: Left =
        RightSib(RightSib) . Left;
  VertPos: Top = LeftSib . Top;
  Italic = Yes;
  END;
 From :
  BEGIN
  CreateBefore(FromLabel);
  HorizPos: Left =
        RightSib(RightSib) . Left;
  VertPos: Top = LeftSib . Top;
  Italic = Yes;
  END;
 Subject :
  BEGIN
  CreateBefore(SubjectLabel);
  HorizPos:
      Left = LeftSib . Right + 12 ;
  VertPos: Top = LeftSib . Top;
  Bold = Yes;
  END;
 Body :
  BEGIN
  VertPos:
      Top = LeftSib . Bottom + 10 ;
  END;
 Paragraph :
  BEGIN
  VertPos:
      Top = LeftSib . Bottom + 5 ;
  Justify = BlockJustify;
  Indent = 20;
  END;
END
```

Figure 4.15: Complete presentation schema for the memorandum presentation.

Medium-Specific: Some kinds of information that are important for presentation are peculiar to specific media. Since Proteus is medium-independent, it is inappropriate to support these directly. For example, in certain graphics documents it might be useful to base an object's color on its area. However, computing the area of irregularly-shaped objects, like closed spline paths, is outside the scope of Proteus's mathematical operations. So, the graphics medium could define an `Area` interface function that takes a node as its argument and returns the node's area. The text medium's `URoman`, `LRoman`, and `Arabic` interface functions are also medium-specific.

Editing State: The state of a user's editing session is maintained by the extension language subsystem. Examples of editing state are the current selection and the current cursor location. If the appearance of the selection is to be under the control of presentation schemas, it must be possible to find out which document elements are selected. Interface functions can be used to make that information available.

Analysis: Certain kinds of analysis are outside the scope of Proteus's presentation services and are not reflected in the document tree. Static semantic analysis of programs, which in Ensemble will be performed by a part of the language kernel called Colander II, is a good example. The results of analysis can be used to guide presentation. For example, a presentation schema might invoke a `Declared` function which, when passed an `Identifier` node, would return a boolean value representing whether or not the identifier was declared.

4.4 Extensibility to New Media

Proteus and its presentation schema language can be adapted to new media because of a new understanding of the fundamental elements that define a medium. Existing dictionary definitions of the "medium" concept are not sufficiently operational to form the basis of an adaptable software system. For instance, one such definition states that a medium is "a mode of artistic expression or communication." [75] This definition is intuitively correct, but it doesn't provide the information that Proteus needs to adapt to different media.

4.4.1 The Elements of Media

What Proteus needs is a specification of the elements that make one medium different from another. Careful examination of the demands of formatting documents in different media led to the identification of these elements. Based on this model, Proteus has already been adapted to three different media: text, two-dimensional graphics, and digital video.

The elements of a medium are:

- a set of primitive object types,
- a set of dimensions in which such objects are laid out, and
- a set of parameterized formatting operations.

In Ensemble, these elements are embodied in a `C++` object called a `Medium`. Currently, `Medium` objects are created by hand, but they are designed to be generated automatically from a specification. The remainder of this section describes the elements of media in more detail and how these elements are represented in the `Medium` objects.

Primitive Objects

Each medium supports a set of primitive object types. These are the fundamental data objects that can be formatted and rendered on a computer screen or printed page. For example, the primitive data types of Ensemble's current media are:

Text Medium: text (which is a sequence of ASCII characters);

Graphics Medium: point, line, ellipse, rectangle, regular polygon, and text;

Video Medium: video (a sequence of equal-sized bitmaps) and text.

A medium may support many types of primitive data (as the graphics medium does) or only one (as the text medium does). Different media may support the same data types, as shown above where the text data type is supported by all three media.

Proteus needs to know about the primitive data types in order to support tree elaboration. Each Ensemble `Medium` object includes a set of canonical instances of its primitive nodes. Other parts of Ensemble can ask which types are supported or whether there is a supported primitive with a particular name (e.g. "Rectangle"). The tree elaboration service uses this information to check the correctness of generated node declarations. That service also needs to know, for each primitive node type, the type signature of the function used for setting the primitive node's data slots.

Dimensions

The primitive objects of a medium are laid out in a k-dimensional space. Media differ in the number of dimensions they support and in the names and types of their dimensions. Ensemble's text and two-dimensional graphics media have the same two dimensions, horizontal and vertical, both of which are spatial. Ensemble's digital video medium also uses these two spatial dimensions and adds the time dimension.

It is easy to conceive of media having anywhere from one to four dimensions. For instance, digital audio has a single dimension of time, while three-dimensional animation has three spatial dimensions and one time dimension. However, it is difficult to conceive of a medium supporting more than one time dimension or more than three spatial dimensions.

There are some more fine-grained distinctions between the dimensions of media that can be important. Often, media have different design communities with their own conventions. For instance, while text and two-dimensional graphics logically have the same dimensions, they have different conventions for the natural location of the origin. Typographers usually place the origin at the upper left hand corner of the drawing area and "grow the document down" from that point. In contrast, some graphical domains (drafting for instance) are likely to put the origin in the lower left corner. Another area in which media have different

Parameter Name	Type	Description
FontFamily	string	font family
Size	numeric	font size (in points)
Bold	boolean	bold font style
Italic	boolean	italic or oblique font style
Hyphenate	boolean	whether to hyphenate
MinHyph	numeric	minimum word length
MinLeft	numeric	minimum characters left of hyphen
MinRight	numeric	minimum characters right of hyphen
Justify	enumeration	how to justify lines
Indent	numeric	first line indentation
RightMargin	numeric	position of right margin
LineSpacing	numeric	spacing between lines
FgColor	string	foreground color
BgColor	string	background color
Visible	boolean	whether element is visible

Table 4.1: Parameters of the line-breaking operation of Ensemble's text medium.

conventions is in units of measure. An example is typographer's points, which are widely used in typography, but not in other domains.

Proteus needs to know two things about the dimensions of a medium. First, it needs the names of the attributes (extent and position) and points (minimum, maximum, and center) of each dimension. The designer of a medium may assign these names in any way that is useful. This allows Proteus to support the various origin conventions. The text medium can bind the name `Top` to the minimum point, which tends to put the origin at the top of the page or screen. The graphics medium can place the origin at the bottom of the drawing area by binding the name `Bottom` to the minimum. The second thing Proteus needs to know is whether the medium has any special units and to which dimensions they apply.[3]

Formatting Operations

Media have distinctive formatting operations. These operations are controlled by a set of typed parameters. For example, Ensemble's text medium has one formatting operation, line-breaking, whose parameters are shown in Table 4.1.

It is tempting to think that a medium has one formatting operation for each primitive data type it supports. However, there are counter examples to this claim. The two-dimensional graphics medium can use the same stroking and filling operation for all its objects. The text medium also has a page-breaking operation that operates on the elements of the documents largely without regard to their type.

[3]Support for custom units has not been implemented yet.

For the purposes of Proteus, what is important is not the number of operations, but rather their parameters. The parameters of the formatting operations are, along with the layout decisions, the key determinants of the appearance of the document. So, Ensemble `Medium` objects store a description of the medium's set of formatting parameters. That is, for each parameter, the name and type of that parameter are stored. Proteus uses this information to determine which attribute names appear in presentation schemas and to check their types.

Chapter 5

The Implementation of Proteus

Proteus has been implemented in `C++` using the `g++` compiler as part of the current Ensemble prototype.. Currently, it is supported on both Sun's Sparc architecture (running SunOS 4.1) and on the HP PA-RISC architecture (running HP-UX 9.0). This chapter describes its implementation.[1]

5.0.2 Architecture

Proteus has a kernel/shell architecture (shown in Figure 5.1) that makes a clear separation between the medium-independent and medium-specific aspects of its operation. Proteus's kernel is independent of any particular medium and contains two agents, the schema translator and the presentation engine. The schema translator translates presentation schemas into presentation tables. The presentation engine uses these tables to maintain the presentation tree and to answer the queries of its presentation clients.

The medium-specific shells are designed to be generated automatically from simple specifications. However, they are currently created by hand. The shells have two primary components, a *translator configuration* and a *custom interface*. The translator configuration for a medium adapts the schema translator to the domain of that medium (attributes, dimensions, primitive data types). The custom interface provides the same medium-specific-to-generic mapping at the boundary between the presentation engine and its presentation clients.

5.0.3 Class Hierarchies

The expression of this architecture in `C++` is based primarily on two class hierarchies, the `Presentation` hierarchy and the `PresSchema` hierarchy, which are shown in Figure 5.2.

The `ProteusPres` and `PresSchema` classes implement Proteus's kernel. The `ProteusPres` class provides a virtual tree abstraction used by the tree elaboration service and a value caching system used by the attribute propagation and box layout services. The `PresSchema` class embodies the schema translator and holds the runtime representation of the presentation schema.

[1] Some familiarity with object-oriented programming languages, and with `C++` in particular, is helpful for understanding this chapter.

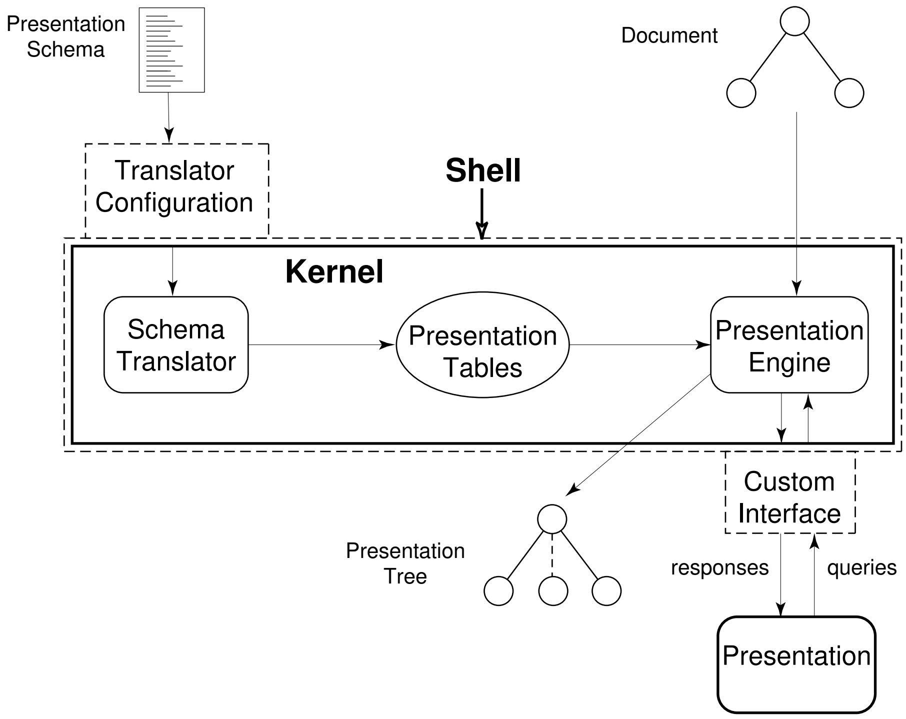


Figure 5.1: The kernel/shell architecture of Proteus which forms the basis of its adaptability to different media. The kernel appears in the center surrounded by the thick, solid border. The shell has a thin, dashed border and consists of a translator configuration and a custom interface.

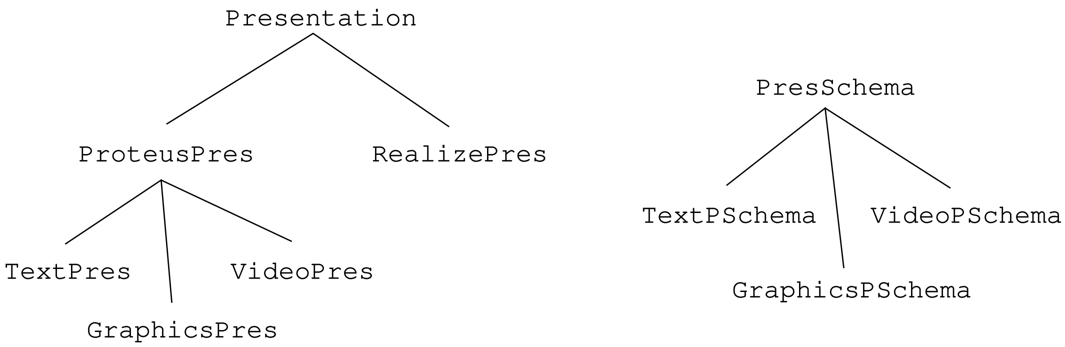


Figure 5.2: The two `C++` class hierarchies that implement the kernel/shell architecture of Proteus.

The medium-specific shells are implemented by instances of the `Medium` class and by each medium's specialized subclass of the `PresSchema` class. Instances of the `Medium` class, described in some detail in Section 4.4.1, are the translator configurations for each of the media. The `TextPSchema`, `GraphicsPSchema`, and `VideoPSchema` classes provide the custom interface for each medium.

The `Presentation` Class

The `Presentation` class defines the runtime representation of presentations. Ensemble presentations are not required to use Proteus, which allows tools developed outside the Ensemble framework to be brought into the system as "black boxes". For example, the Realize data visualization tool [59] has been integrated into Ensemble via the `RealizePres` class. The primary requirement for such presentations is that they adhere to Ensemble's protocol for compound presentations. The root `Presentation` class defines this protocol.

The `ProteusPres` Class

The `ProteusPres` class is the root class for all presentations that use Proteus. Its primary addition to the compound document services of the `Presentation` class is support for the presentation tree. The presentation tree is a *virtual copy* of the document tree to which nodes generated by the tree elaboration service are added. The copy is virtual in the sense that the nodes from the document tree are not actually copied. Instead, the generated nodes are stored in *attachment records*. The attachment records are themselves stored in a hash table indexed by the node to which they are attached.[2] The `ProteusPres` class provides an interface for navigating the presentation tree that hides the attachment mechanism.

The `ProteusPres` class also supports a cache of attribute values used by the attribute propagation and box layout services. The details of the use of this cache are described later in Section 5.0.4.

Medium-specific Presentation Classes

A medium designer who wishes to use Proteus does so by creating a subclass of the `ProteusPres` class. Ensemble currently has three such subclasses called `TextPres` (supporting the text medium), `GraphicsPres` (for two-dimensional graphics), and `VideoPres` (for video). Each of these classes provides:

- Data and functions for the medium's formatting operations,
- Handlers for document change events which invoke the formatting functions as appropriate,
- Initialization functions invoked when a presentation is first created,
- Hit detection functions used to determine whether mouse clicks actually "touch" particular objects, and

[2] Credit for the design and implementation of the virtual tree system goes to Vance Maverick.

```
double TextPSchema::fontSize (ProteusPres* tree, PNODE* node) { //
     Get attribute identifier, but only on the first invocation
     static int attrId = getAttrId("SIZE");

     // Evaluate attribute and extract return value from union
     PslValue v = attrValue(tree, node, attrId);
     return (v.doubleVal());
}
```

Figure 5.3: The implementation of the `fontSize` function of the `TextSchema` class.

- Functions that compute the size of the presentation.

5.0.4 The `PresSchema` Class Hierarchy

The `PresSchema` class is the central element in the implementation of Proteus. Its data slots embody the presentation tables which are the runtime representation of a presentation schema. Its functions implement the entire schema translator and most of the presentation engine. The most important function defined by the `PresSchema` class is the `attrValue` function, which, given a node in the presentation tree and an attribute identifier, returns the value of the attribute for that node. The value is returned in the form of a union called a `PslValue`.

The `attrValue` function is never called directly by the presentations that use the `PresSchema`. Instead, it is invoked indirectly by member functions of its subclasses. These subclasses specialize the behavior of the `PresSchema` class to the needs of each of Ensemble's media. Currently, there are three such subclasses called `TextPSchema`, `GraphicsPSchema`, and `VideoPSchema`.

The subclasses specialize the behavior of the `PresSchema` class in two ways. First, they specify which medium they are implementing (as represented by the `Medium` class). The `PresSchema` class uses this information to determine the correct allocation of its data structures. Secondly, they provide the custom interface of each medium's shell. One such function is the `fontSize` function of the `TextPSchema` class (shown in Figure 5.3). It simply marshals the arguments for a call to the `attrValue` function and extracts the correct return value from the result. The `TextPSchema` class defines one such function for each of the fifteen non-dimensional attributes in the text medium plus functions for each of the four attributes of each dimension.

Attribute Rule Tables

The primary data structure supporting attribute propagation and box layout is a table holding pointers to expression parse trees. The table has one entry for every attribute/node-type pair, plus an entry for each attribute's default rule. Every node type named in the presentation schema's `BOXES` and `RULES` sections has table entries. The number of attributes represented in the table is determined by the schema's medium. For instance, the

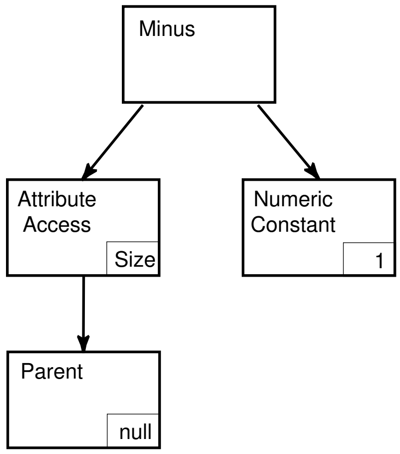


Figure 5.4: The parse tree of an attribute rule. Each node in the tree is an evaluable expression. The small box in the lower right corner of some of the nodes is a data slot holding data specific to the expression's type. Because the Parent has no child, it will evaluate to the parent of the defining node.

text medium currently has fifteen non-dimensional attributes and two dimensions. Each dimension requires two attribute slots. So, instances of the `TextPSchema` class have nineteen entries for each node type named in the schema.

Each table slot holds a pointer to an expression parse tree. This parse tree has a straightforward relationship with the text of the presentation schema. For instance, a rule of the form

```
Size = Parent . Size - 1 ;
```

would be represented by the parse tree shown in Figure 5.4.

When the presentation schema does not define a rule for a particular attribute/node-type pair, the table entry for that pair is a null pointer. In the memorandum presentation schema, the full set of attribute and box layout rules for the `Paragraph` node type is

```
Paragraph :
     BEGIN
     VertPos: Top = LeftSib . Bottom + 5 ;
     Justify = BlockJustify;
     Indent = 20;
     END;
```

Thus, there are only three non-null table entries for the `Paragraph` node type.

Evaluating Attribute Rules

Attributes are evaluated whenever a presentation needs parameters for its formatting operations. To get an attribute value, the presentation invokes the corresponding function in its custom interface, which in its turn, invokes the `attrValue` function.

```
PslValue PresSchema::attrValue (ProteusPres* tree, PNODE* node,
    int attrId) {
    If there is a valid entry in the cache of tree, return it

    If there is a user override for node, return it

    If there is a node-specific rule for the attribute, then {
        evaluate the expression
        if evaluation does not fail, return the result
    }
    If there is an explicit default rule for the attribute, then {
        evaluate the rule's expression
        if evaluation does not fail, return the result
    }
    If node is not the root of the presentation tree
        Call attrValue on the parent of node.

    Otherwise, return the global default value for the attribute.
}
```

Figure 5.5: Implementation of the `attrValue` function (in pseudo-code).

The `attrValue` function implements the five stage evaluation algorithm described in Section 4.3.3. In addition, any values that it returns are stored in a cache maintained by the `ProteusPres` class. The entire cache is invalidated whenever the size of a primitive object changes, the structure of the presentation tree is altered, or a user override is added or deleted. The coarse-granularity of cache invalidation results in some performance problems that are discussed in Section 6.1.2. The implementation of the `attrValue` function is sketched in Figure 5.5.

Of the five stages in attribute evaluation, the two most frequently invoked are the ones that evaluate node-specific rules and explicit default rules. These are the rules whose expression trees are stored in the `PresSchema` class and the process of evaluating them is quite straightforward. Each node in the expression parse tree is an instance of the `PslExpr` class. This class defines a function called `eval` that takes as its arguments

- the `ProteusPres` instance, which provides the presentation tree for attribute access operations,
- the node for which the attribute value is being determined, and
- a boolean reference argument called `valid`, that is set to false when evaluation fails.

The `eval` function returns a union type (called `PslValue`) from which the actual return value is extracted. Each node in the expression parse tree runs the `eval` function recursively on its children and then combines their return values to determine its own return value.

For instance, the plus node in Figure 5.4 evaluates its two children (the attribute access expression and the constant) and then returns the sum of their values.

Handling Box Layout Rules

The box layout services makes extensive use of the implementation of attribute propagation. This is only sensible since box layout is based on a similar model of propagating attributes. However, because there are only two degrees of freedom among the four attributes of each dimension, certain cases must be handled specially. The approach taken in the current implementation of Proteus is to treat each dimension's extent attribute (e.g. `Width` or `Height`) normally. The remaining three attributes are demoted to the role of *points* which can be used to define a single `position` attribute. Thus, the text medium provides a `HorizPos` attribute (for *horizontal position*) which can be set using any one of the `Left`, `Right`, or `HMiddle` points.

Internally, Proteus represents a node's position by its minimum. So, any rules that are based on the minimum point can simply return the value of the position attribute. Rules that involve the middle or maximum of a dimension require special handling, based on the mathematical relationship between extent and the three points:

$$\begin{aligned} \text{middle} &= \text{minimum} + \frac{\text{extent}}{2} \\ \text{maximum} &= \text{minimum} + \text{extent} \end{aligned}$$

When an attribute access expression for the middle or maximum points appear in a schema, the translator replaces it with an equivalent expression computing the value from the minimum and extent. A rule of the form

```
HorizPos: Left = Parent . HMiddle ;
```

is equivalent to

```
HorizPos: Left = Parent . Left + (Parent . Width / 2) ;
```

When position is specified in terms of the middle or maximum, the inverse procedure is followed. In this case, a rule like

```
HorizPos: HMiddle = Parent . Left ;
```

is equivalent to

```
HorizPos: Left = Parent . Left - (Self . Width / 2) ;
```

Note that when the maximum or middle point appears on the left hand side of a rule, then the defining node's extent is used. When the maximum or middle point appears in an expression that will be on the right hand side of the rule, the accessed node's extent is used.

Conditional Attribute Propagation

When attribute rules appear inside conditionals, their internal representation becomes somewhat more complex. Consider the following specification for the `Title` node:

```
Title :
   BEGIN
        Bold = Yes;
        IF (TypeOf(Parent) == "Section") THEN
            Size = 18;
        ELSE
            Size = 14;
        ENDIF
   END;
```

The presentation table entry for the `Title` node's `Bold` attribute is unaffected by the presence of the conditional: it is just a simple constant expression. The table entry for the `Size` attribute, however, is an *if-expression* that is composed of a sequence of *guard/expression pairs* and a single *else expression*. The guard/expression pairs correspond to the `IF` and `ELSIF` blocks of the conditional. The `guard` is the boolean expression determining whether the block applies. Nested conditionals are represented by nested if-expressions.

Evaluation of if-expressions is best described as having two stages. In the first stage, the guards are evaluated. The expression corresponding to the first guard that evaluates to true is used. If no guards are true, then the else expression is used. The failure of a guard expression is considered equivalent to a false value. In the second stage, the chosen expression is evaluated and the result is returned. If evaluation of this expression fails, then the evaluation of the entire if-expression fails.

Proteus assumes that attribute rules do not have side effects. In general this is a safe assumption because the inherent operations of Proteus do not have side effects. However, it would be possible to define interface functions that did (see Section 5.0.4 for a discussion of interface functions). Thus, the creation of interface functions with side effects is strongly discouraged. An example showing a case where side effects could be important is:

```
   BEGIN
        Size = 12;
        IF (TypeOf(Parent) == "Section") THEN
            Size = 18;
        ELSE
            Size = 14;
        ENDIF
   END;
```

The initial "`Size = 12;`" rule can never take effect because there are definitions for the `Size` attribute in every block of the conditional that, together, guarantee that the first rule will be overridden. When Proteus translates a rule block having this form, it "throws away" the initial, irrelevant attribute rule. This approach to conditionals can cause problems if the right hand side of an attribute rule can have side effects.

Tree Elaboration

The implementation of the tree elaboration service, like that for attribute propagation, is a fairly straightforward translation into `C++` data structures of the concepts expressed in the presentation schema. Just as the schema specifies tree elaboration in two stages, the service's implementation is built around has two types of objects.

The first object type is the *box specification* (class `PslBoxSpec`), which is the internal representation of a generated node declaration (from the schema's BOXES section). A box specification holds the name of the generated node type, the name of the node's primitive type, and a list of the expressions used as arguments to the function that sets the value of the node at creation time. The box specifications are stored in a table in the `PresSchema` object.

The second object type is the *creation* (class `PslCreation`), which is the internal representation of a creation command. It has three slots: an expression that specifies the node to which the generated node will be attached, an enumeration value specifying where the generated node will be attached (before, after, first child, or last child), and the name of the node to be generated. Because many different creations can be defined for a single node type, each node type's creations are collected into a set called a *creation specification* (class `PslCreationSpec`). The creation specifications are kept in another table in the `PresSchema` object.

When a presentation is created, the tree elaboration service performs a pre-order traversal of the document tree. The creation specification corresponding to each node is checked. If the specification is empty, nothing is done. Otherwise, for each creation in the specification:

1. The creation's node expression is evaluated to find the *attachment node*. If the evaluation fails, the creation process aborts.

2. Otherwise, the creation's box specification is looked up in the `PresSchema`'s table, and

   (a) The arguments to the value-setting function are evaluated. If any of the evaluations fail, the creation process aborts.

   (b) A node of the specified primitive type is generated and given its specified name.

   (c) The generated node is attached to the attachment node (computed in step 1) in the relative position specified in the creation object.

   (d) The value-setting function for the primitive type is called with the correct values computed in step 2(a).

Once the initial elaboration of the presentation tree is complete, the presentation engine must maintain correct elaborations in the face of changes in the structure of the underlying document. This is done by monitoring the change events of the document and, when the changes announced by those events affect generated nodes, making appropriate updates. There are three important update classes:

Insertion: Whenever a node is inserted, the creation specification tables are checked. If there is a creation specification for that node, then the creation procedure described above is performed.

Deletion: Deletion can affect tree elaboration in two ways because both the creating node and the attachment node can be deleted. In most cases, the same node has both roles, but it is possible to write creation rules that attach generated nodes to remote parts of the presentation tree.

Creating Node: If the creating node is deleted, then all nodes that were generated because of that node's insertion must be removed from the presentation tree. This operation is performed using a table in the `ProteusPres` class that stores for each node N, the set of nodes that were created because N was inserted into the presentation tree.

Attachment Node: If a generated node's point of attachment is deleted, then the generated node may need to be reattached at another location. The current implementation first deletes the generated node and the attachment node. Then, the original creation that produced the generated node is invoked again for the creating node, according to the procedure above.

This process behaves correctly for the majority of situations. However, it is not correct for some important cases. The central source of the problem is *argument failure*. Each generated node has its value (e.g. the string of text associated with a text primitive) computed from a set of arguments. Each argument is a general expression whose value may change as the document changes. The presentation engine does not keep track of changes that affect these expressions. As a result, the values of generated nodes are not correctly updated in response to changes to other parts of the document. This problem is part of a larger problem with Proteus's current implementation that is discussed in Section 6.1.2.

Conditional Tree Elaboration

Because tree elaboration commands that appear inside conditionals do not override earlier tree elaboration commands, they require a slightly different runtime representation. The last section described how non-conditional creations were collected into creation specifications. To support conditional tree elaboration, these creation specifications also contain a list of *guarded creations*, each of which describes the creation commands of a conditional block.

A guarded creation contains

- a sequence of guard conditions corresponding to the `IF` and `ELSIF` sections of the conditional
- a matching sequence of creation specifications that describe the unguarded creation commands of each section and any nested conditionals, and
- a creation specification for the `ELSE` section of the conditional.

When a creation specification is invoked, its guarded creations are checked in a straightforward manner to determine if any additional node should be generated.

Interface Functions

Proteus's interface function service is implemented by two `C++` classes: `PslFunction` and `FuncCall`. Each medium contains a (possible empty) set of `PslFunction` objects, which specify the interface functions that can be used in presentation schemas for that medium. Each `PslFunction` object holds the name by which the function can be invoked in presentation schemas, a list of the types of the function's arguments and a pointer to the implementation of the function (i.e. a `C++` function pointer).

`FuncCall` objects represent actual invocations of the function in a presentation schema. The `FuncCall` class is a subclass of the `PslExpr` class and, thus, defines an `eval` function which marshals the interface function's arguments, calls the function, and returns its value.

Chapter 6

Evaluating Proteus

6.1 Experience with Proteus

At this time, Proteus has been in use as part of Ensemble for about two years. Although Ensemble is not yet used on a daily basis, it has still been possible to create and test a variety of presentations in different media. This section discusses the lessons learned from experience with the system.

6.1.1 Presentation Examples

The power and range of Proteus's presentation schemas is best understood through examples. A straightforward example of presentation in the text medium was discussed earlier, but Proteus has also been used to create presentations of programs and of multimedia documents.

Program Presentation

Effective presentation of programs has long been a concern in computer science. The Algol 60 report [61] included the notion of a *publication language* that was separate from the *reference language* used as input to the compiler. In particular, the designers of Algol 60 wanted to be able to express their programs using traditional mathematical notation (with Greek letters, subscripts and superscripts), but also wanted to conform to the restricted character sets supported by the computers of the time.

In the intervening years, there has been considerable discussion of both mechanisms and stylistic standards for the pretty-printing of programs. The term "pretty-printing" has been given several meanings. Oppen provides a minimalist definition:

> A *pretty-printer* takes as input a stream of characters and prints them with aesthetically appropriate indentations and line breaks [55].

Although this definition emphasizes layout, the term is often extended to include changes of character font and other typographic effects. In fact, Oppen's examples include widespread use of font changes to convey the lexical class of program elements.

The search for a superior pretty-printing style for Pascal has produced a large literature. L'Ecuyer [43] provides a good example of this research while many other such works are cited by Baecker and Marcus [4, p. 18] as part of their exploration of the design space of styles for C programs.

While the particulars of program presentation style can be quite important, this dissertation's focus is on mechanism rather than policy. Traditionally, programs have been formatted by hand-editing of the whitespace and comment characters in the source code. This makes the author of the source code into both a programmer and a typographer. Furthermore, the ASCII text representation of programs allows only the most primitive formatting.

Knuth [37, 36] extended this approach in his WEB system. Users of WEB specify both the program source and how it should be formatted using a special macro language. The WEB system includes two filters, `tangle` and `weave`. `Tangle` extracts the program source for input to a compiler. `Weave` extracts formatting commands for the TeX formatter. The system provides support for complex indexing and cross-referencing operations that can be used to produce excellent documentation. Unfortunately, WEB's macro language is difficult to learn, to read, and to write. While it can be used to create beautiful and effective documentation, it does little to ease its production.

An alternative is to format programs automatically using some sort of analysis of their text. Early attempts at automatic pretty-printing were made by the Lisp community. The culmination of this process is Waters's XP [74], which is now part of the Common Lisp standard. It provides a rich set of primitives for defining line-breaking and indentation style. However, its concepts are difficult to separate from the Lisp environment.

Much of the pretty-printing research focuses on language-independent algorithms for line-breaking. Oppen [55] presented an efficient, language-independent algorithm for batch pretty-printing. The basic technique is to put grouping markers in the text stream, which the line-breaking algorithm later replaces with newline and white-space characters. Pugh and Sinofsky [64] modified Oppen's algorithm to be more incremental and enriched the set of text stream markers. Mikelsons [50] developed a different approach based on nested boxes which appears to be language-independent, but his implementation is not. Knuth and Plass [40] showed that the line-breaking algorithm of TeX can be used to good effect with Pascal programs.

In order to exploit language-independent line-breaking models, other research in pretty-printing has explored ways of formally specifying style. Rubin [68] describes a grammar-based specification for pretty-printing intended for use in a syntax-directed editor. The grammar is decorated with rules describing when to break lines and how to indent. A similar approach is found in the Synthesizer Generator [67], which uses unparsing directives in an attribute grammar. Pugh and Sinofsky's work on algorithms was intended for use with the Synthesizer Generator. PPML [32], the pretty-printing meta-language for the Centaur environment, uses pattern-matching to bind line-breaking rules to the internal representation of a program. Unlike Rubin's work and the Synthesizer Generator, PPML specifications are separate from the parsing grammar for the language, thus supporting multiple styles.

This work on automatic pretty-printing all shares a common flaw; there is an implicit

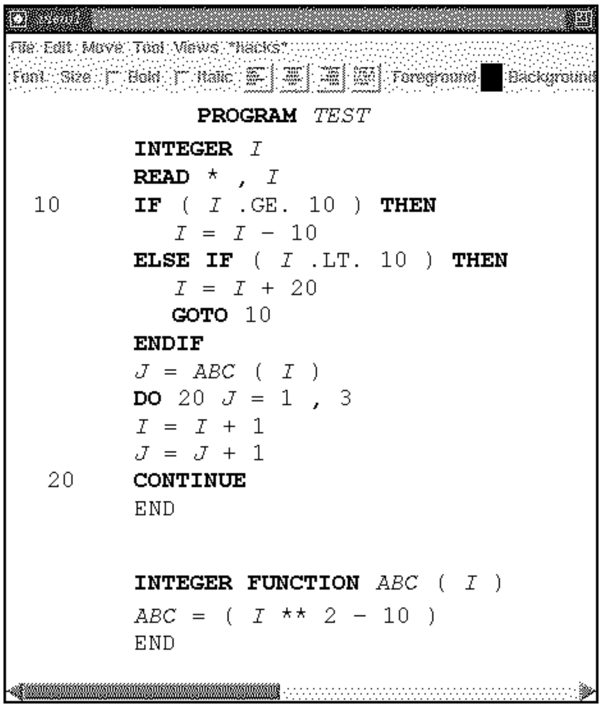


Figure 6.1: A small FORTRAN program as formatted by Ensemble.

assumption that the output device has the characteristics of a "dumb" terminal (e.g. fixed line height, monospaced fonts). These systems all work by transforming one stream of text into another stream of text. Printing characters are laid out by placing newline, space and tab characters in the text stream. This assumption fails to take advantage of improvements in monitors and printers, which now support a variety of more sophisticated typographic effects. A second weakness of much of this work is the close coupling of the pretty-printing mechanisms to the concrete syntax of the language, making it difficult to base aspects of presentation on static semantic or execution-based information.

Ensemble and Proteus treat programs differently. From the point of view of Ensemble's presentation subsystem, a program is just another text document. Thus, the full power of Ensemble's text medium can be applied to programs. Instead of treating the program text as a stream of characters, the leaves of the program tree (the program's lexemes) are considered to be independent paragraphs that can be placed at any position in the text medium's two-dimensional space. Of course, in most cases, Ensemble's text medium will only be used to produce more-or-less standard program presentations like the FORTRAN example shown in Figure 6.1.

However, this formatting power can be applied to produce novel presentations of program text that have never been achieved in stream-oriented systems. As an example, consider the C switch statement shown in Figure 6.2. That figure shows a very normal presentation of the switch statement. In general, each statement is on a separate line and indentation is used to highlight the block structure of the code.

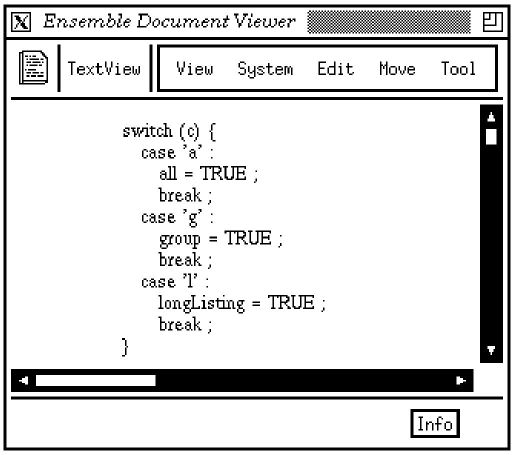


Figure 6.2: A switch statement presented in normal, line-oriented style.

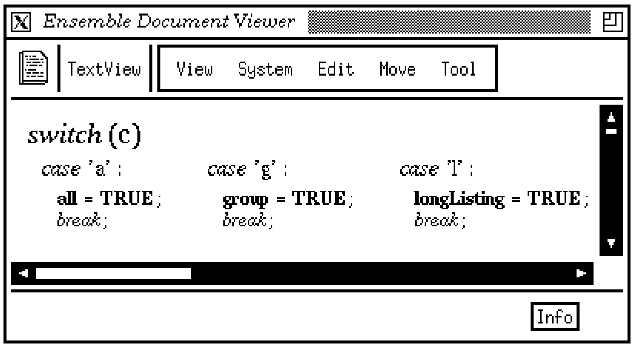


Figure 6.3: A switch statement presented in a tabular style.

An alternate, tabular presentation of the same document is shown in Figure 6.3. The most striking difference between the two presentations is the layout of the cases. The normal line-oriented style arranges the cases vertically, but the tabular presentation lays the cases out horizontally, as if they were columns in a table. This might be a useful style when trying to understand the differences between similar cases. The tabular presentation has other differences from the normal presentation. The header portion of the switch statement uses a larger font than the body. Keywords are italicized, while identifiers are displayed in boldface. In emulation of text document conventions, semicolons are placed a little closer to the lexeme to their left than are most other lexemes.

No existing program pretty-printer can reproduce the tabular switch presentation because these other systems' adherence to the text-stream formatting approach precludes placing non-contiguous text on the same line. In addition, programming systems do not support the kind of fine-grained control over presentation that is required to adjust the positions of the semicolons.

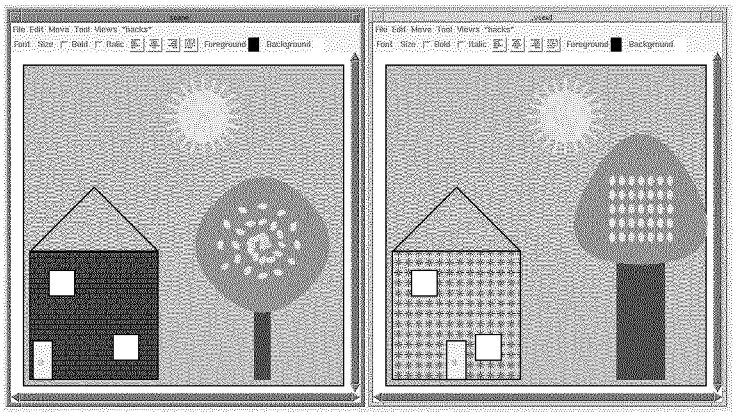


Figure 6.4: Two presentations of the same two-dimensional graphics document, a simple illustration of a house, a tree, and the sun.

Graphics Presentation

Ensemble's two-dimensional graphics medium provides a number of interesting examples showing the power of presentation schemas and the value of multiple presentations.

The graphics medium differs from the text and video media in having "data-less" primitive types. For instance, nodes of the *ellipse* primitive type have only their type associated with them. Every other aspect of an ellipse is considered a presentation attribute and is under the control of Proteus. Other data-less primitive types are the line, rectangle, point, and regular polygon types. They are contrasted with the text and video clip primitives, whose instances contain sequences of data (characters and bitmaps, respectively).

The effects of using data-less primitives can be seen in Figure 6.4 which shows two different presentations of a simple graphics document whose content is similar to a child's drawing of a house, a tree, and the sun. The presentations differ in the position of the house's door, the size of the tree trunk, the number of sides in the tree's leaf area, and the arrangement of the tree's fruit. In most interactive graphics editors (e.g. Adobe Illustrator [2]), position, size, and orientation are treated as fundamental qualities of an object. They can be changed by editing operations, but they are handled separately from elements of "style" like stroke width and fill color. The approach taken in Ensemble's graphics medium gives full rein to the designers of new presentations, but the results may surprise those whose experience is based on the use of conventional graphics editors.

Of the differences between the two presentations in Figure 6.4, the most interesting is the arrangement of the fruit, which nicely illustrates the value of having support for advanced mathematical operations in the presentation schema language. In the first presentation, the fruit are arranged in a spiral around the center of the tree's leaf area by box layout rules that use the `ChildNum` function and simple trigonometry. The rules are:

```
HorizPos: HMiddle = LeftSib(Parent).HMiddle + 2 * ChildNum(Self) *
        SIN(ChildNum(Self) * 30.0);
```

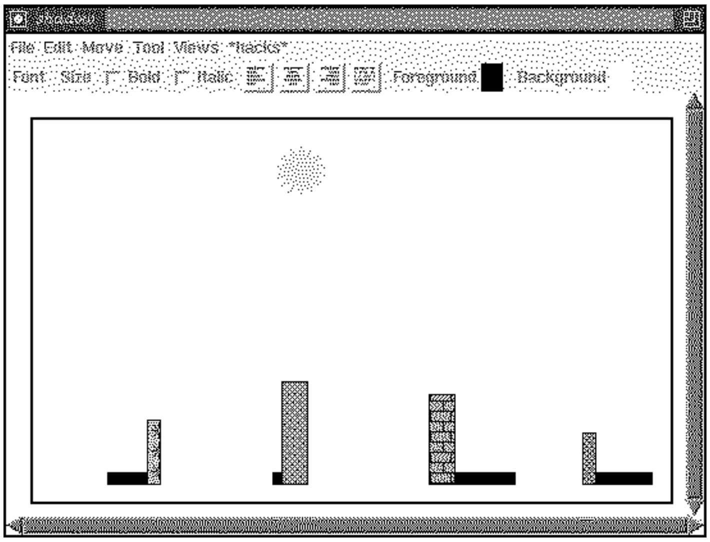


Figure 6.5: A simple graphics document showing the power of constraint-based layout. The document contains a sun and four poles. Each pole has a shadow whose size depends on the relative positions of the sun and the poles.

```
VertPos: VMiddle = LeftSib(Parent).VMiddle + 2 * ChildNum(Self) *
        COS(ChildNum(Self) * 30.0);
```

In these rules, the attribute access expression

```
  LeftSib(Parent) . HMiddle
```

refers to the horizontal center of the tree's leaf area. The box layout rules of the second presentation use the `DIV` and `MOD` functions to lay the fruit out in neat rows and columns.

```
VertPos: VMiddle = LeftSib(Parent).VMiddle - 40 +
        22 * MOD(ChildNum(Self), 5) ;
HorizPos: HMiddle = LeftSib(Parent).HMiddle - 45 +
        15 * DIV(ChildNum(Self), 5);
```

Box layout constraints are also used to present the document seen in Figure 6.5. This document contains a sun and four poles. Each pole has an attached shadow whose size is determined by the position and size of its pole and the position of the sun. If the user moves the sun (or a pole), the shadows change automatically to maintain the correct appearance. Figure 6.6 shows the same document after the position of the sun has been changed.

The claim here is not that this kind of support for constraint-based graphics is revolutionary, because it is not. The value of the constraint approach for managing the appearance of graphics has already been demonstrated by earlier graphics systems, notably Sutherland's Sketchpad [71] and Borning's Thinglab [7]. But these systems were designed for the domain of graphical simulation. Constraint-based presentation is largely unheard of in software for graphics *documents*. Proteus brings some of the power of a simulation system to this more mundane domain.

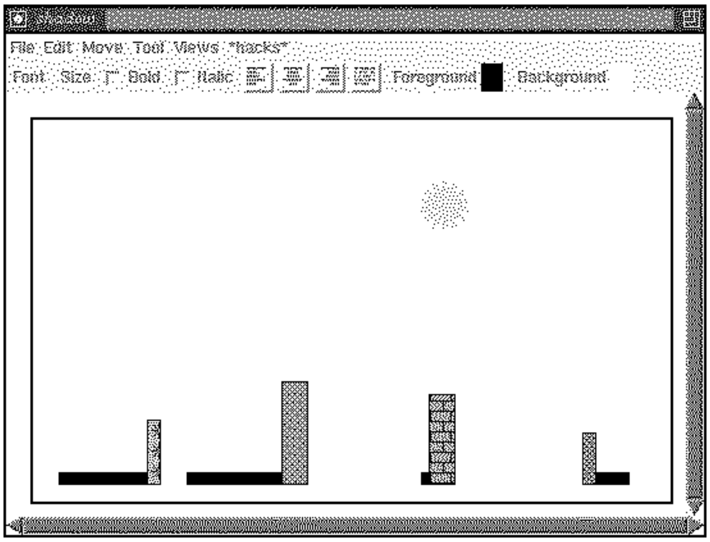


Figure 6.6: The same graphics document shown after the sun's position has been changed.

The multiple-presentation example shown in Figure 6.4 demonstrated that multiple presentations are possible in the graphics medium. However, it did not make a clear demonstration of their utility. The benefits of multiple presentations become more clear when considering Figure 6.7, which shows four different presentations of a simple "sequence" document. This document's tree has a root with four children, each of which is an ellipse primitive. This simple sequence is then given four different presentations, each corresponding to different implementations of the sequence abstraction: endogenous singly-linked lists, exogenous singly-linked lists, vector, and exogenous doubly-linked lists. These four presentations could be used in an educational setting to illustrate the relationship between abstraction and implementation.

Video Presentation

Ensemble has a simple video medium that supports two types of primitives: video clips and text. Video clips are sequences of bitmaps (stored in MPEG-encoded disk files). The text primitive is the same one used by the text and graphics media and is intended for creating labels and titles.

The video medium has three dimensions: horizontal, vertical, and time. The horizontal and vertical dimensions have the same attributes as the other media. The attributes of the time dimension are `Duration`, `Start`, `Stop`, and `TMiddle`.

The medium only supports two attributes for video clips: `FrameRate` and `PlayStyle`. `FrameRate` defines the rate at which the video clip should be played in units of frames per second. `PlayStyle` is an enumeration attribute whose values are:

`Natural`: The video clip plays at the rate specified by the `FrameRate` attribute.

`Stretch`: The video clip is played one time at whatever speed makes it run for the time specified by its `Duration` attribute.

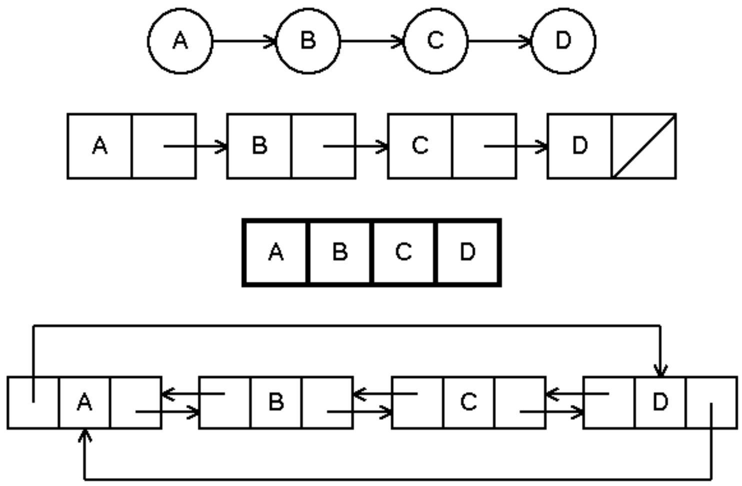


Figure 6.7: Four different presentations of a “sequence” document. These presentations could serve as examples of the conceptual equivalence of different implementations of the sequence concept.

`Loop`: The video clip is played over and over at the rate specified by the `FrameRate` attribute, for the time specified by the `Duration` attribute.

These two attributes and the box layout service have been used to create several different presentations of a test document, shown in Figure 6.8. This document contains two video clips. One is a very short clip of a boy's face, to which a “twisting” distortion is applied and reversed over the course of 15 frames. The other is a clip of woman driving, talking, and gesturing which contains about 125 frames. Both clips have a natural frame rate of about 15 frames per second. Using Proteus, it has been possible to create five different presentations of this document that differ from each other in either layout or play style. For instance, the clips can be played sequentially or they can start simultaneously. If they start simultaneously, they can play at their natural rates, or the little boy clip can played in slow motion to fill the same amount of time as the woman clip. The playback of the two clips can even be centered in time. just like a text document's title is centered on the page.

These presentations show that Proteus's services are applicable to and useful for dynamic media like video. The box layout service is useful for temporal dimensions just as it is for spatial dimensions. Like other media, the video medium has formatting operations controlled by parameters. It is easy to imagine an expanded set of attributes controlling the application of transitional effects (fades, wipes) and image processing (star and diffusion filtering). Finally, the utility of tree elaboration for generating titles and labels is clear.

6.1.2 The Presentation Engine

The performance of the current implementation of Proteus's presentation engine varies considerably from document to document. When editing small natural language documents,

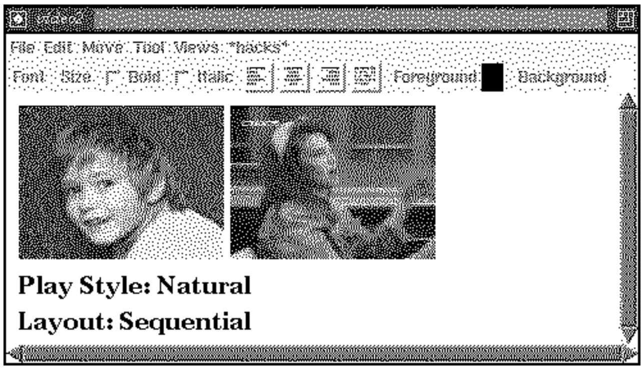


Figure 6.8: A screen dump of an Ensemble video document. The document contains two video clips. The text labels in the presentation were generated by tree elaboration and describe the style of playback.

like the memorandum example discussed earlier, Ensemble keeps up with fast typing rates. For instance, in an informal test on a SparcStation 10, I typed the delete key twenty times consecutively, each keystroke invoking the "delete-char" editing operation. It took me about three seconds to do this and Ensemble showed no noticeable delay in updating the screen. In contrast, Ensemble took about 8 seconds to perform the same twenty delete operations in a presentation of the switch document (shown in Figure 6.2). Profiling tools have shown that about two-thirds of this time is consumed by Proteus.

The primary difference between the memorandum and switch presentations was the size of their presentation trees. The memorandum's tree had 8 nodes, while the switch's tree had 49 nodes. This and other informal tests have led to the conclusion that Proteus's performance degrades to unacceptable levels as the size of the document grows.

The underlying problem is that Proteus does not store sufficient information about the dependencies between the attribute values of the presentation tree's nodes. The expressions stored in Proteus's presentation tables describe the dependencies between nodes abstractly. No representation is maintained of the dependencies between the node instances in the presentation object.

For instance, this rule for `Paragraph` nodes in memoranda

```
VertPos: Top = LeftSib . Bottom + 5 ;
```

says that every `Paragraph`'s top is dependent on its left sibling's bottom. The presentation tables store a fairly direct translation of this rule into Proteus's internal data structures. Suppose a memorandum has two paragraphs, $P1$ and $P2$, and that because of the above rule, $P2$ depends on $P1$. Proteus's data structures make it fairly easy, given $P2$, to determine that $P2$ depends on $P1$, but provide no mechanism to identify this dependency given only $P1$. So, if the value of $P1$'s `Bottom` attribute changes, the system has no way to determine which node's attribute values are affected. It must assume that *all* nodes are affected and invalidate all cached attribute values.

Pan's semantic analysis module, Colander [5], included a straightforward solution to this problem. Colander maintained, for each attribute instance, a list of all the attribute instances that were directly dependent on it. When an attribute instance changed, the dependent attributes were placed on a work list for recomputation. This approach had two disadvantages. First, the work list was not ordered, so an attribute instance could be recomputed and then be immediately returned to the work list by the recomputation of another attribute instance. Second, this approach, if applied to Proteus, increases the storage requirements of the system because the size of the dependency information would be proportional to the number of nodes in the document, rather than the size of the presentation schema. The first problem was addressed by Hudson's work on incremental attribute evaluation in general graphs [30], but the second is fundamental to the approach of storing explicit dependency information.

A better solution has been explored by my colleague on the Ensemble Project, Kannan Muthukkarruppan. He has designed and implemented a attribute grammar system called SPINE [51]. The attribute grammar approach has several advantages over both the current presentation engine and the general-purpose constraint system approach:

- Dependencies between nodes are described abstractly, in terms of the document's grammar. Thus, the storage required for dependency information is constant.
- Efficient, incremental evaluation algorithms exist for attribute grammars.
- It is possible to choose which attributes are cached and which are not. This allow sensible trade-offs to be made between the storage and execution time aspects of performance.
- Compilation of attribute grammar specifications into native machine language (via `C++`) is fairly straightforward.

Presentation specifications in SPINE have been hand-written for several different Ensemble documents and substantial improvements in performance over the original presentation engine have been observed. For example, in one test of text editing performance, Ensemble took about 1 second to respond to each keystroke when using the current presentation engine. Using SPINE, the response time was reduced to about .25 seconds. SPINE is faster than the current presentation engine because its knowledge of dependencies between the attribute values of nodes allows it to minimize the number of values that must be recomputed and because SPINE specifications are compiled into machine code. Further performance improvements will be possible once Ensemble's formatters have been rewritten to better exploit SPINE's information about which attribute values have changed. Future work will produce an automatic translator from Proteus's presentation schema language to the SPINE attribute grammar language.

One problem with this approach arises from the use of tree elaboration. The effect of tree elaboration is to create a new document tree which does not necessarily conform to the original document grammar. To cope with tree elaboration, SPINE would have to be able to alter its dependency information dynamically. SPINE cannot do this, but it can be done by higher-order attribute grammar systems, which maintain a more complete representation of the dependencies by node types and can modify this representation at runtime. However, the design and implementation of an incremental higher-order attribute grammar system remains a open research problem [73]. An alternate solution would be to take advantage of the translation step between presentation schemas and SPINE specifications. Part of this translation step could be an elaboration of the original document grammar.

Another limitation of the attribute grammar approach arises from the compilation of the specifications. A feature envisioned for Ensemble is the ability to alter a presentation schema while looking at an actual document. This would ease the creation and debugging of presentation schemas quite a bit. In an interpreted specification system (such as the current version of Proteus), it is fairly straightforward to alter specifications while the system is running, since the interpreter's data structures are readily accessible. This is more difficult to do in a compiled system (such as SPINE), where the internal representation of the specification is machine code. In a compiled system, some kind of dynamic re-linking of a running binary would be required to load a new specification.

These two concerns are the only important problems identified to date with the attribute grammar approach taken by SPINE. The expressive power of its specification language is essentially the same, though it may not be as convenient to write certain specifications in SPINE as it is in Proteus.

6.1.3 Experience with the Presentation Schema Language

The presentation examples described in Section 6.1.1 showed that Proteus's presentation schema language is powerful enough to describe a wide variety of useful presentations in several media. Anecdotal reports from the small set of users who have written presentation schemas have been generally positive. However, their experiences have identified some shortcomings in the language's design.[1]

Schema Idioms

Like all other specifications, some presentation schemas make repeated use of certain idioms. The presentation schema language provides some help in this area through the system of explicit and implicit default rules. The implicit default rule makes it easy for entire subtrees to share the rule definitions of the subtree's root. The explicit default rules allow sharing of rules between nodes that are not in the same subtree. These mechanisms are quite helpful in reducing the size and complexity of presentation schemas, but there are granularities of sharing that they do not support.

The current language has no mechanism for sharing expressions. To understand why this would be useful, consider the following two rules specifying left (`x1`) and right (`x2`) end-points for the rays of the sun in Figure 6.4.

```
x1 = Parent.HMiddle - ((Parent(Parent).major/2) *
  sin(childnum(self) * 360.0/(2.0*NumChildren(Parent))));
x2 = Parent.HMiddle + ((Parent(Parent).major/2) *
  sin(childnum(self) * 360.0/(2.0*NumChildren(Parent))));
```

[1] The work of Roy Goldman, a fellow member of the Ensemble Project, on graphics presentation has been especially helpful in pinpointing language problems.

Both rules are quite complex, but they only differ by a single character. The rule for `x1` subtracts two terms while the rule for `x2` adds the same two terms. The second of these two terms is quite complex. The schema would be both shorter and easier to understand if the second term could be specified once and then reused in both rules. This could be achieved through several different mechanisms including textual macros, user-defined functions, or local attributes (which could be used to store the term's value as an intermediate result).

Another common idiom in the presentation schemas for Figure 6.4 produces sets of rules like the ones below for the trunk of the tree:

```
Height = Parent . Height * 0.6;
Width = Parent . Width * 0.05;
```

These rules define the size of the trunk in terms of the size of its parent, which is the node representing the entire tree. The scaling relationship that they define is a typical idiom for specifying the size of subcomponents in a graphics document. Because this idiom involves multiple rules, it cannot be expressed by simple functions or through using a single local attribute. Instead, textual macros or a more specialized mechanism designed to expand into multiple rules would have to be used.

Tree Navigation

In the presentation schema language, constraints between the attributes of different nodes are defined by placing attribute access expressions in the right hand side of attribute rules. These expressions specify a node and an attribute. The node is specified using the tree navigation functions. These navigation functions have been shown to be sufficient for the task of specifying other nodes, but they are not always convenient to use. For example, consider the following rule from the presentation schema for the doubly-linked list presentation shown in Figure 6.7.

```
HorizPos: Left = RightSib(RightSib(RightSib(RightSib(
    RightSib(RightSib(RightSib(Parent))))))) . Left;
```

This rule could easily have been replaced by a slightly simpler one:

```
HorizPos: Left =
    NthChild(Parent(Parent), ChildNum(Parent) + 7) . Left ;
```

This rule describes the same node as the child of the grandparent whose index is 7 higher than the index of the parent. However, while this second rule is somewhat easier to read, it is only marginally easier to understand. Both rules are hard to understand because they are purely navigational and make no mention of the types of the nodes being traversed. Thus, the tree navigation functions provide no conceptual frame of reference.

The language would be improved if type-based constraints could be defined. For instance, in the example given above, the grandparent of a `NodeData` node is always a `Sequence` node and the seventh child of a `Sequence` node is always a `NodeShell` node. Thus, a more comprehensible rule might be

```
HorizPos: Left = Next NodeShell . Left ;
```

The precise semantics of this type of expression requires further research, but a possible semantics might be

- Return the nearest right sibling of type `NodeShell`,
- Otherwise, for each ancestor, if the ancestor has a right sibling of type `NodeShell`, return it,
- If no ancestor has a right sibling of type `NodeShell`, fail.

Being able to describe the nodes involved in constraints in terms of their types, instead of their relative position in the tree, would make presentation schemas easier to understand. It would also make them easier to modify if the structural schema changed.

Node Creation

The creation commands for the presentation schema language were originally designed with traditional text documents in mind. There is extensive experience in using tree elaboration for text documents in Grif. Among the many sample schemas distributed with Grif, there are never more than two creations for a node type. However, this rule of thumb breaks down when tree elaboration is used in the graphics domain, as is shown by an example from the presentation schema for the doubly-linked list presentation shown in Figure 6.7. In that schema, the rule for the `Node` node type includes *thirteen* different creation commands for generating the boxes and arrows of the list presentation.

```
Createafter(NextArrow2);
Createafter(NextArrow1);
Createafter(NextLine3);
Createafter(NextLine2);
Createafter(NextLine1);
Createafter(NextShell);
Createafter(NodeShell);
Createafter(PrevArrow2);
Createafter(PrevArrow1);
Createafter(PrevLine3);
Createafter(PrevLine2);
Createafter(PrevLine1);
Createafter(PrevShell);
```

This example highlights two independent problems with the schema language. First, the sheer length of this sequence of creation commands makes them confusing. Some mechanism for organizing these commands into smaller units would be very helpful. One way to do so would be to allow the creation of internal tree nodes. For this example, the schema could specify the creation of `PreviousCell` and `NextCell` internal nodes. Their children could be the existing nodes used to construct the boxes and lines for the previous and next pointer slots or an additional layer of internal nodes could be created (e.g. `PrevArrow` and `PrevLine`).

The second language problem is that the order of the commands is a critical element of the specification because nodes are created in the order that the commands appear in the schema. In this case, the nodes must be created in right-to-left order because they are being attached as right siblings of the `Node` node. When the language forces its users to be concerned with command order, it violates the declarative approach that the rest of the language adheres to fairly consistently.[2]

A possible solution for this problem is to provide a different way of specifying node generation. Tree elaboration is really just a very constrained form of tree transformation. The creation commands can be viewed as specifying ways in which the presentation tree's grammar productions are different from those of the document grammar's productions. When the differences are few in number and local, the current set of creation commands work well. When they are not, some other mechanism would probably be easier to work with. For instance, a better specification style might describe the differences between the document and presentation grammars in terms of grammar productions. In the given example, this might take the form of

```
Node : NodeData ; ==>
Node : NodeData PrevShell PrevLine1 PrevLine2 PrevLine3
    PrevArrow1 PrevArrow2 NodeShell NextShell
    NextLine1 NextLine2 NextLine3 NextArrow1 NextArrow2 ;
```

The description of tree transformations is an area of active interest among programming language researchers because of its applicability to certain problems in compilation and in specification-based programming [18, 25].

Interactions with Box Layout Attributes

The box layout service was originally designed to meet the needs of the text medium. It is well-suited to text because text paragraphs are usually aligned with the axes of the Cartesian plane. Exceptions to this rule occur primarily in text for advertising and other artistic materials, whose production is largely outside the scope of Ensemble's current text medium.

The box layout service has also been successfully applied to the graphics and video media in Ensemble, but a problem with the current implementation becomes clear when the service is used for graphics. Figure 6.9 illustrates the central problem. It shows three equal-sized ellipses — that is, the lengths of their major and minor axes are the same. However, the sizes of their bounding boxes are not the same because of differences in rotation and stroke-width. Thus, the actual width of an ellipse when it is drawn is a function not only of the lengths of its axes, but also of its stroke width and rotation.

The designer of Ensemble's graphics medium [23], Roy Goldman, coped with this problem by

1. Creating separate attributes for the specification of object size. For ellipses, these are the `Major` and `Minor` attributes.

[2] The presentation schema language is also order-sensitive when conditionals are used. However, to date the constructions using conditionals have been much less complex. So, the order-sensitivity of conditional rules has been of less practical importance.

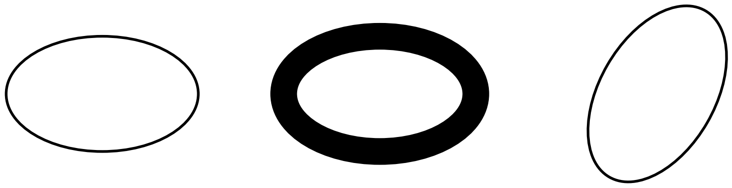


Figure 6.9: Nominal vs. actual size: The three ellipses in this figure have the same size (1 inch by .6 inch), but their bounding boxes are different because of the effects of stroke width and rotation.

2. Establishing special conventions for the extent attributes in graphics presentation schemas. Specifically, the `Width` and `Height` attributes are treated as if they are "read-only" attributes whose values are computed from the values of the attributes that specify object size and rotation.

In other words, he created a set of attributes to describe the *nominal* extent of graphic objects and then used the existing attributes of the box layout service to represent their *actual* extent. The problem with his solution is that Proteus provides no mechanism to enforce the relationship between the nominal and actual size attributes. So, correct behavior of graphics presentations depends on the authors of presentation schemas adhering to the convention.

It turns out that the same problem occurs, in a less pervasive way, in the text medium. In order to break a paragraph into lines, a maximum line width must be specified. For fully-justified paragraphs that have more than one line, the actual width of the paragraph will be the same as the specified width. But as shown in Figure 6.10, the actual width of single-line paragraphs is likely to be less than the specified maximum line width. Thus, the text medium, like the graphics medium, requires that a distinction be made between nominal and actual extent. The current implementation of the text medium addresses this problem by creating an attribute called `RightMargin`. The maximum line width for the line-breaking operation is the difference between the `RightMargin` attribute and the `Left` position attribute. This allows the `Width` attribute to be computed from the actual sizes of the document elements.

A possible solution to this problem is to define, for each dimension in a medium, both nominal and actual extent attributes. Further research is required to determine how to express this in the language and how this approach affects the position attributes.

The `AllChildren` Function

The `AllChildren` function, whose semantics were described in Section 4.3.4, is a special case in a language that is otherwise quite general. In practice, because its application is limited to extent attributes, its only use is to compute and return the size of the space occupied

[4]These paragraphs are taken from Page Smith's history of the Post-Reconstruction era of the United States [69, page 117].

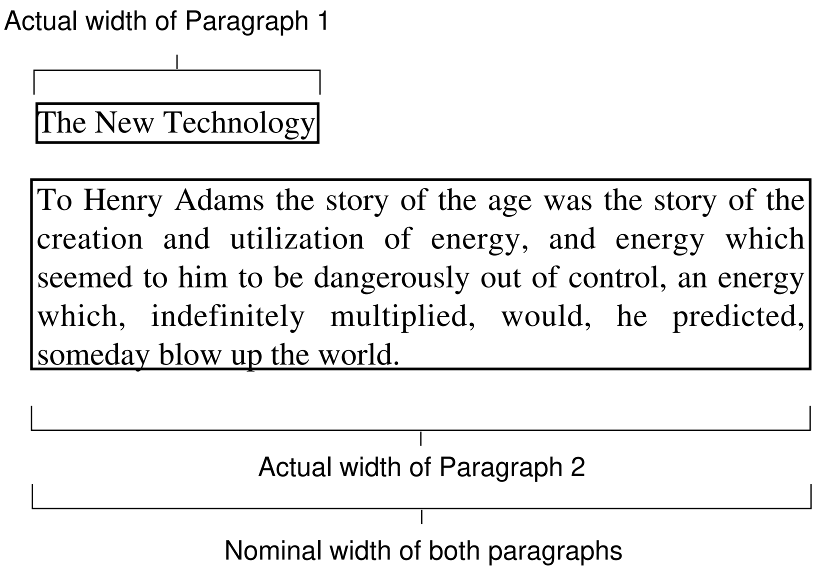


Figure 6.10: Nominal vs. actual size: The two paragraphs in this figure have the same nominal width, but different actual widths. In the text medium, nominal width is used to determine when lines are broken, but the actual width of single-line paragraphs (like the first one in this figure) can be much smaller.[4]

by a node's children. From this information, it is possible to determine the bounding box of each subtree in the document. Experience with presentation schemas has shown that this is an important feature. However, the same experience has shown that the `AllChildren` function needs to be replaced by some more general concept.

The `AllChildren` function is based on a similar feature of P, the presentation language of Grif [65]. P has an `Enclosed` operator, which, like `AllChildren`, can only be used to find the extent of a subtree and is important, but very much a special case in the language. It was in an attempt to find a more general solution, that the concept of *node set* was developed as the basis for the `AllChildren` operator. However, a consistent semantics for attribute access on node sets was never specified, which established the current special case situation.

Clearly, further research is required to find a more general and elegant replacement for the `AllChildren` function.

6.2 Comparing Proteus to Related Systems

This section compares Ensemble and Proteus to a variety of related systems, some of which were introduced in Chapter 2. Rather than examining each system in turn, it is organized around three areas in which Proteus makes a contribution: synchronized multiple presentations, style specification, and medium-independence.

6.2.1 Synchronized Multiple Presentations

Many types of systems support some limited form of multiple presentations, including document systems, interactive graphics editors, and multiple representation systems.

Batch document formatters, such as LaTeX [42] and troff [56], have supported multiple styles for documents for quite some time. As in Ensemble, this is possible because the style specifications are stored separately from the document. However, because these are batch systems, they do not support any viewing of the final output, much less viewing of synchronized multiple presentations.

Commercial interactive document systems, such as Framemaker [19] and Microsoft Word [47], also have provisions for storing style specifications separately from individual documents, but these specifications must be loaded into a document in order to take effect. Since a document can only hold one such specification, these systems are unable to support multiple presentations.

The use of multiple presentations is common in many other types of commercial software. One example is Adobe Illustrator [2] which supports two views of its graphics documents: a "Preview" view and a "Artwork" view. The Artwork view shows the PostScript paths that define the objects in the document. The Preview view shows these objects as they will be drawn on the page. If different windows on the same document are created showing both views, they will both support editing and both will be kept up to date as the document is edited. What Adobe Illustrator lacks, in comparison to Ensemble, is the ability to define additional views. The Preview and Artwork views are hard-coded into Illustrator. It is not possible for an end-user to define additional views.

This pattern of supporting a fixed set of presentations is repeated in many other types of software. Among them are research systems supporting multiple representation document editing, including VORTEX [14, 13], Lilac [8], Tweedle [3], and Juno-2 [28, 29]. VORTEX and Lilac support textual documents. Tweedle and Juno-II support graphics documents. Each of these systems provides two hard-coded presentations: a WYSIWIG presentation and a presentation showing a programming-language-based specification of the document's content and appearance. The WYSIWIG presentation supports standard direct-manipulation editing operations, while the language-based presentation supports standard text editing operations. A change made in one presentation will appear in the other presentation, though it might require an explicit request for synchronization by the user. The goal of these systems is to provide both the power of language-based document systems and the ease of use of direct-manipulation systems.

Each of these systems only supports the two hard-coded presentations, but the two presentations are *very* different. All of the systems except Juno-II support iterative primitives for which Proteus has no equivalent. The iterative primitives are typically used to generate graphical or textual elements with many repeating parts or to write explicitly procedural descriptions of the formatting process. These system's presentations are kept in synchrony using specialized incremental parsing techniques that are tuned for each system's language. There is no obvious way to map this parsing process onto Proteus's services. However, that is not Proteus's role in Ensemble. Ensemble has language analysis services that are designed to maintain the correspondence between language representations and other representations. Furthermore, what Proteus does provide (i.e. the ability to display one document in a variety of styles) is not supported in any way by these multiple representation systems.

Only two other interactive systems, Grif [66] and Chiron [35], actually provide *general* support for multiple presentations.

Grif supports multiple presentations through its "view" mechanism. The author of a presentation schema can specify several views in that schema. Different views can have very different appearances, but some presentation rules must be the same for all views. Multiple views on the same document that are specified in the same schema are kept synchronized by Grif, but presentations specified in different schemas are not. Thus, Grif's support for multiple presentations is more limited than that provided by Ensemble and Proteus.

The Chiron user interface development system supports multiple presentations in the form of *artists*. An artist is an agent responsible for both the appearance and the interaction style of an abstract data type (ADT). A single ADT instance may have many artists displaying it at the same time, so Chiron clearly provides a general mechanism for multiple presentations. However, Chiron provides this support only to a system developer because artists are implemented in a variant of Ada, compiled and statically linked into the system. In contrast, Proteus's presentation schemas are interpreted at runtime and can be created or modified by end users. Furthermore, Chiron does not provide a high-level model of presentation. The system does include a rendering server that supports a collection of drawing primitives, but it does not have a unified notion of presentation similar to that provided by Proteus's presentation services.

6.2.2 Style Specification

At the center of Proteus is its language for style specification. This language is based on P, the presentation schema language of Grif [65], but the design of Proteus's presentation schema language is much more general. As a result, the syntax and semantics of Proteus's language are much easier to describe than those of P. This generality is the critical feature allowing Proteus to adapt to different media (see next section). Some of the key advantages of Proteus's language over P are:

- In the definition of P, each presentation parameter (equivalent to an attribute in Proteus) has a separate syntax and semantics. P lacks central unifying concepts equivalent to the typed attributes of Proteus. For example, most parameters in P can be defined by constraint on the values of other nodes, but some attributes, such as those that restrict page breaking, cannot be defined by constraint.

- P does not allow arbitrary expressions on the right-hand side of its presentation rules. It does support an expression-like syntax for some numeric presentation parameters. For instance, the `Size` parameter can be specified using the following rule:

  ```
  Size : Enclosing + 2 pt ;
  ```

  This specifies that the value of `Size` should be two points larger than that of the parent node (the "`Enclosing`" node). While the right-hand side of this rule appears to be an expression, it is actually composed of a node specification ("`Enclosing`") and a difference ("`+ 2 pt`"). So, it is not possible to write the rule as

```
Size : 2 pt + Enclosing;
```

- P does not support conditional rules. It does provide a restricted form of conditional tree elaboration, but the set of possible conditions is limited.

In all fairness, it should be made clear that P (and Grif) have real advantages over Proteus (and Ensemble). P provides support for numbering document elements, such as figures and tables. Proteus is powerful enough to support high-quality formatting, but its current clients are rather rudimentary. Thus, Grif's text formatting services are much more powerful than those of Ensemble's current text formatter.

Another important language-based style specification system is DSSSL [1], which is a draft standard being developed for use with SGML documents. A DSSSL specification has two parts. The first part specifies a transformation of the document tree. There do not appear to be any restrictions on the kinds of transformations that may be performed. The second part of the specification provides "style sheets" for the elements in the transformed tree. DSSSL's designers appear to believe that formatting will not perturb the order of the document elements; that is, that the order of the nodes in a traversal of the tree will be directly reflected in their placement on the page. Thus, the nodes of the tree must be rearranged if their order on the page is to be altered. Proteus does not need the tree transformation step because its presentation model allows the placement of document elements at arbitrary positions on the page or screen. It is not clear whether DSSSL is suitable for an interactive system because its current tree-transformation module is *ad hoc*, making it hard to invert tree transformations when mapping screen locations back to the underlying document.

It is also appropriate to compare Proteus's style specifications to the languages used to specify pretty-printing in software development environments. In the Synthesizer Generator [67], the appearance of a program document is specified by "unparsing" rules embedded in the attribute grammar specifying the analysis of the program. The Synthesizer Generator's model is that the program text is analyzed to produce a parse tree which is then "unparsed" into a stream of data for presentation to the user. The stream of data contains text intermixed with formatting primitives and is displayed by a simple text formatter. It is difficult to see how this model can be applied to non-textual media or extended to support arbitrary two-dimensional layout (as was shown in Figure 6.3). Another contrast with Proteus is that each production in the attribute grammar has its own unparsing rules. Thus, each unparsing rule is defined in a setting where there is more knowledge of context than is the case using Proteus, which defines rules on a per-node-type basis. The PPML pretty-printing language [32] achieves a similar result by binding unparsing rules to tree patterns. With Proteus, conditional rules are used to make a node's presentation dependent on the production in which it appears. Experience so far indicates that Proteus's approach is acceptable because the vast majority of presentation rules are independent of a node's context.

6.2.3 Medium-Independence

Proteus is the first system for managing the appearance of objects that is based on an explicit model of media. As a result, it can be adapted to serve media that use very different formatting operations and is well-suited to the compound document model.

Now, every multimedia system must provide some common ground on which to manipulate elements of different media it supports. For instance, in the Action! [45] multimedia presentation editor, every object (text, ellipse, video clip) has a spatial position and a position on the time line. But otherwise, there is no common paradigm that is applied to the various objects.

This pattern is repeated in research multimedia systems. Both the Firefly [9] and Xavier [26] systems use layout mechanisms that are based on techniques well-known from textual document formatting. Like Proteus, Firefly uses constraints to determine the layout of multimedia data in time. Xavier uses the boxes-and-glue method (originally designed for text documents [40]) for temporal layout. But neither system provides any other facilities that are applicable to all media.

Proteus stands apart from all existing multimedia software in having an explicit model of a "medium". Each of the programs mentioned above applies a single formatting system to a variety of primitive data types. For instance, Xavier supports video clips, text, and two-dimensional graphics. It applies a uniform formatting model (boxes and glue) to lay them out in two spatial dimensions and the time dimension. We call Xavier a multimedia system because it supports primitives that are drawn from different media. But, unlike Proteus, Xavier has no explicit representation of the concept of "medium". In fact, in the terms used by Proteus, Xavier is a single-medium system because it applies the same formatting model and the same dimensions to all its objects.

6.2.4 User Interface Systems

Because the way a document is drawn is part of its user interface, Proteus can be considered a kind of user interface software and its implementation has similarities to a number of existing user interface systems [31, 44, 52, 58]. Like these systems, it provides a layout service and ways of defining the parameters that control object appearance. In fact, Proteus's use of constraints to specify document appearance is similar to their use in the Garnet toolkit [52] and the Higgens UIMS [31]. Comparisons are awkward, though, because a user interface is subtly different from a document. Some of the differences are:

- Proteus is only involved in the output aspects of the user interface, while user interface systems also manage input.

- Architecturally speaking, user interface systems sit between the program and the user, managing all aspects of input and output. In contrast, Proteus is an internal service used by parts of the program to determine how to draw the document. Proteus does not stand between the program and the user, but rather between the document and the formatting modules of Ensemble.

- Proteus is used to describe how a document should look on any device. Thus, presentation schemas describe the layout and formatting of the document in abstract, device-independent terms. User interface systems are used to describe computer interfaces using bit-mapped displays and generally use device-specific concepts like pixels.

- In order to provide maximum flexibility, most user interface systems allow an interface to be specified procedurally. In contrast, Proteus's presentation schemas are almost entirely declarative (with tree elaboration being a possible exception).

It is possible to envision Proteus being used as part of a user interface system. The world of classic user interface objects like buttons, menus, and scrollbars, can be seen as "just another medium" with its own primitive objects, dimensions, and formatting operations. So, Proteus can be adapted to this medium and provide control over the appearance of a user interface via presentation schemas. Of course, a considerable amount of work would be required to create a user interface system based on Proteus.

Chapter 7

Conclusions

The preceding chapters of this dissertation have described the design of Ensemble and its presentation system, Proteus. Together, Ensemble and Proteus have several novel features:

- multiple presentations,
- a simple, powerful set of presentation services, and
- an architecture that can provide these services to a wide variety of media.

An Ensemble user can view multiple presentations of the same document simultaneously. Changes made to the document through one presentation will be seen immediately in all other active presentations. Ensemble's support for multiple presentations is completely general. Unlike Grif [65], there are no restrictions on the set of presentations that may be viewed simultaneously. Any presentation that can be specified can be viewed simultaneously with any other presentation that can be specified.

Thus, as the subsystem that manages appearance specifications for the various presentations, Proteus is the centerpiece of Ensemble's support for multiple presentations. Proteus provides four simple services that together are powerful enough to describe a wide variety of novel and conventional presentations of text, graphics, and video documents. The application of these services to a particular document is specified in a presentation schema. The schema describes the appearance of the document in abstract terms. Unlike the formatting and rendering services of user interface software (e.g. Chiron [35]), Proteus's services are suitable to a wide variety of output devices and could be used in a production-quality document or multimedia system. In particular, Proteus's services are well-suited to natural-language text documents and to program documents, while most user interface software gives short shrift to text.

Of equal importance is Proteus's ability to adapt to a variety of media. Its four presentation services are medium-independent. Proteus has an adaptable, kernel/shell architecture that enables it to change both the presentation schema interpreter and its client interface in order to suit the needs of different media. The kernel provides generic forms of the presentation services that are adapted to different media via medium-specific shells. The behavior of each medium-specific shell is driven by an explicit representation of the medium to which it corresponds. Proteus is the first system to be based on an explicit, encapsulated notion of a medium. Many systems have a generic notion of primitive data types associated with various media (e.g. text, line, video clip). No other system has a higher level notion that brings multiple primitive data types together with formatting operations to define a "medium".

7.1 Future Directions

As with any course of research, further work remains to be done. Some areas needing additional research were identified in earlier discussions of experience with Proteus. Others are simply natural directions for extension of this work.

Performance

The poor performance shown by the current presentation engine for large documents must be addressed. Kannan Muthukkaruppan's work on SPINE [51] is a first step in this direction but an implementation of a presentation-schema-to-SPINE translator must be completed before the attribute grammar approach can be fully evaluated. It may also be worthwhile to explore other approaches to improving performance including specialized data structures for box layout.

Nominal Layout versus Actual Layout

Roy Goldman's experiments with attribute systems for Ensemble's graphics medium [23] brought the distinction between an object's nominal size and its actual size clearly into focus. It is common for objects in many media to have both nominal layout and actual layout. Nominal sizes and positions (i.e. layout) are used as input arguments to formatting operations, but often the formatting operation produces output with different sizes and positions; hence the need to represent actual size and position separately. Figure 6.9 (on page 75) showed an example of this phenomenon from the graphics medium. Proteus needs a mechanism supporting this distinction in both the presentation schema language and the presentation engine.

Presentation Schema Syntax

Experience with Proteus has found three areas in the syntax of presentation schemas that need improvement. First, the current syntax for specifying tree elaboration is awkward when a single node type specifies many different creation commands. A new syntax that helps the user understand the entire elaborative transformation for that node would be very helpful. The second area needing improvement is the tree navigation syntax. Currently, the nodes involved in constraints must be identified by relative position, which is awkward for nodes that are not nearby in the tree. Proteus needs operations that identify nodes by type, rather than by relative position. The best syntax and semantics of such operations remain to be determined. Finally, the presentation schema language needs language features that ease the definition of frequently used expressions and other specification idioms.

Applying Proteus to Additional Media

To date, Proteus has been used with three media: text, graphics, and video. The implementations of the text and graphics media are moderately sophisticated, but are still quite far from the standards of commercial software. The video medium implementation is extremely primitive and stands only to demonstrate that Proteus is applicable to a dynamic medium.

So, Proteus needs additional clients to promote the claim that it is suitable for all media. One set of potential new clients would more sophisticated graphics and text media. In particular, a page-breaking text medium supporting multiple columns would be a substantial advance and would clearly demonstrate the sufficiency of Proteus's services for high-quality formatting. Another set of potential clients would be media not yet supported in any form. Examples might include computer music, digital audio, synthesized speech, three-dimensional graphics, and animation (either two- or three-dimensional).

Interaction within Compound Documents

The current implementation of compound documents in Ensemble does not allow the non-dimensional presentation attributes of a sub-document to depend on those of its ancestors. This contrasts with Grif, which because it has a single set of presentation attributes for all documents (i.e. it is a single medium system), is able to pass attribute values freely through the compound document tree. Providing the same service in Ensemble is more difficult because Ensemble documents do not have a uniform presentation model. So, Ensemble and Proteus require a different protocol that can allow subdocuments to take advantage of their context (i.e. their ancestors in the compound document tree), without unnecessarily restricting the ways that compound document may be composed.

Schema Modularity

At this time, presentation schemas are stand-alone specifications. So, if two schemas have a great deal in common, there is no way for them to share the common parts of their specifications. Clearly, the effort to produce related presentation schemas and to keep them in agreement would be reduced if some sharing mechanism could be introduced.

PPML [32] has an approach that might make sense for Proteus. PPML allows its specifications to be composed into a logical stack of specifications. Specifications at the top of the stack take precedence over those lower in the stack. A shared base specification can be put at the bottom of the stack and then variations from that base can be placed higher up There are implementation issues that may make this approach easier with PPML than with Proteus, but a mechanism of a similar type might ease the burden of schema creation.

Proteus without Ensemble

While Proteus was conceived to meet the needs of Ensemble, it is actually a service with very general applications. It could be used to manage the appearance of X11 user interfaces. It could be used by any system that can usefully apply a tree-structured model to the objects it displays. In a system using Proteus, the appearance of objects (e.g. documents, data structures, user interfaces) would be specified by external specifications, rather than source code in the system's development language. If these specifications are made properly accessible, end users can gain more control over the interfaces to the objects they interact with. Also, because Proteus's services can manage many aspects of the output process, systems using Proteus should be simpler, with all the attendant benefits of simplicity (easier maintenance, fewer bugs, etc.).

So, it makes sense to try to make Proteus independent of Ensemble. This will require defining the interfaces that Proteus requires to walk the document tree and manage the presentation tree. Other interfaces will also have to be defined but may not be apparent until the task is attempted. The creation of `Medium` objects and of each medium's custom interface will have to be automated. Then, having made it possible to use Proteus with other systems, the utility to do so will have to be demonstrated with examples.

Bibliography

[1] Sharon Adler and Anders Berglund. DSSSL tutorial: Introduction & overview to ISO DIS 10179. Distributed to tutorial participants, 1993.

[2] Adobe Systems, Mountain View, CA. *Adobe Illustrator v. 5.0 User Guide*, 1993.

[3] Paul J. Asente. Editing graphical objects using procedural representations. WRL Research Report 87/6, Digital Equipment Corporation Western Research Laboratory, Palo Alto, California, November 1987.

[4] Ronald M. Baecker and Aaron Marcus. *Human Factors and Typography for More Readable Programs.* Addison-Wesley, Reading, Massachusetts, 1990.

[5] Robert Allan Ballance. *Syntactic and Semantic Checking in Language-Based Editing Systems.* PhD dissertation, University of California, Berkeley, Berkeley, CA 94720, December 1989.

[6] David Michael Betz. XLISP: An object-oriented Lisp, version 2.0. Available with the source distribution and from the author at P.O. Box 144, Peterborough, NH 03458, 1988.

[7] Alan H. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

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

[9] M. Cecelia Buchanan and Polle T. Zellweger. Automatic temporal layout mechanisms. In *Proceedings of ACM Multimedia '93*, pages 341–350, Anaheim, CA, August 1993. ACM Press.

[10] Paul R. Calder and Mark A. Linton. Glyphs: Flyweight objects for user interfaces. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 92–101. ACM Press, October 1990.

[11] Mary Cameron. *Chiron-1 Guide to User Interface Building*, May 1991. Arcadia Document UCI-91-04.

[12] Pehong Chen, April 1988. Electronic mail message to VORTEX Project members.

[13] Pehong Chen. *A Multiple Representation Paradigm for Document Development.* PhD dissertation, Computer Science Division, University of California, Berkeley, California, 1988.

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

[15] L. Peter Deutsch, April 1994. Personal electronic mail communication. Used by permission.

[16] Encyclopedia Brittanica, 1964. Proteus.

[17] Richard Fairley. *Software Engineering Concepts.* Series in Software Engineering and Technology. McGraw-Hill, New York, 1985.

[18] Charles Donald Farnum. *Pattern-Based Languages for Prototyping of Compiler Optimizers.* PhD dissertation, Computer Science Division, University of California, Berkeley, California, December 1990. Available as Technical Report No. UCB/CSD 90/608.

[19] Frame, Inc., San Jose, CA. *Framemaker 4: Using Framemaker*, September 1993.

[20] FrameBuilder: A guide to managing document-based information. Frame Inc. White Paper, February 1993.

[21] R. Furuta. An object-based taxonomy for abstract structure in document models. *The Computer Journal*, 32(6):494–504, 1989.

[22] 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.

[23] Roy Goldman. Variable shape specification in Proteus. Ensemble Project internal memorandum, November 1993.

[24] Grif: Composition and editing of SGML documents. White Paper from Grif S.A., Immeuble "Le Florestan," 2, boulevard Vauban, B.P. 266, St. Quentin en Yvelines, 78053 Cedex, France, March 1994.

[25] Douglas R. Grundman. *Graph Transformations and Program Flow Analysis.* PhD dissertation, Computer Science Division, University of California, Berkeley, California, December 1990. Available as Technical Report No. UCB/CSD 90/620.

[26] Rei Hamakawa and Jun Rekimoto. Object composition and playback models for handling multimedia data. In *ACM Multimedia 93 Proceedings*, pages 273–282. ACM Press, August 1993.

[27] Michael A. Harrison and Derluen Pan. IncTEX: An incremental document processing system. UCB/CSD 91/614, Computer Science Division, University of California, Berkeley, California 94720, April 1991.

[28] Allan Heydon and Greg Nelson. The Juno-2 constraint-based drawing editor. Technical report, Digital Systems Research Center, Palo Alto, California, July 1994.

[29] Allan Heydon and Greg Nelson. Juno-2 reference manual. Technical report, Digital Systems Research Center, Palo Alto, California, July 1994.

[30] Scott E. Hudson. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM Transactions on Programming Languages and Systems*, 13(3):315–341, July 1991.

[31] Scott R. Hudson and Roger King. Semantic feedback in the Higgens UIMS. *IEEE Transactions on Software Engineering*, 14(8):1188–1206, August 1988.

[32] INRIA: Centaur Project, Sophia-Antipolis, France. *The PPML Manual*, February 1994. For Version 1.3 of Centaur. Available by ftp from babar.inria.fr in directory `pub/croap/bertot`.

[33] Ian Jacobs and Laurence Rideau-Gallot. A Centaur tutorial. Technical Report 140, INRIA, July 1992.

[34] Paul Walton Purdom Jr. and Cynthia A. Brown. Parsing extended $LR(k)$ grammars. *Acta Informatica*, 15:115–127, 1981.

[35] Rudolf K. Keller, Mary Cameron, Richard N. Taylor, and Dennis B. Troup. User interface development and software environments: The Chiron-1 system. In *Thirteenth International Conference on Software Engineering*, pages 208–218, Austin, TX, May 1991.

[36] Donald Knuth. *Literate Programming*, volume 27 of *CSLI Lecture Notes*. Center for the Study of Language and Information, Stanford, CA, 1992.

[37] Donald E. Knuth. The WEB system for structured documentation, version 2.3. Technical Report STAN-CS-83-980, Computer Science Department, Stanford University, Stanford, California, September 1983.

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

[39] 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.

[40] Donald E. Knuth and Michael F. Plass. Breaking paragraphs into lines. *Software—Practice & Experience*, 11(11):1119–1184, November 1982.

[41] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, pages 26–49, August/September 1988.

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

[43] Pierre L'Ecuyer. Formal formatting rules for Pascal programs. *The Journal of Systems and Software*, 7:311–322, 1987.

[44] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *Computer*, 22(2):8–22, February 1989.

[45] Macromedia, Inc. *Action! 1.0*, 1993.

[46] Macromind, Inc. *Macromind Director: User's Guide*, 1993.

[47] Microsoft Corporation, Redmond, Washington. *Microsoft Word 5.0*, 1992.

[48] Microsoft Corporation, Redmond, WA. *Object Linking & Embedding, Version 2.0, Programmer's Reference*, 1992. Pre-release version.

[49] Microsoft Corporation, Redmond, Washington. *Microsoft PowerPoint: User's Guide*, 1993.

[50] Martin Mikelsons. Prettyprinting in an interactive programming environment. Technical Report RC 8756, IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY, March 1981.

[51] Kannan Muthukkaruppan. SPINE, a synthesizer for practical incremental evaluators. Master's thesis, University of California, Berkeley, CA, May 1994.

[52] Brad A. Myers, Dario A. Giuse, Roger B. Dannenburg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, pages 71–85, November 1990.

[53] Open Software Foundation. *OSF/Motif Programmer's Guide*, version 1.1 edition, 1991.

[54] OpenDoc: Shaping tomorrow's software. White paper available by anonymous FTP from `cil.org` in directory `pub/opendoc-interest`, March 1994.

[55] Derek C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, October 1980.

[56] 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.

[57] John K. Ousterhout. An X11 toolkit based on the Tcl language. In *Proceedings of the Winter 1991 USENIX Conference*, pages 105–115, 1991.

[58] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[59] John L. Pasalis. Realize: An interactive graphical data structure presentation system. Master's thesis, Computer Science Division, University of California, Berkeley, December 1992.

[60] Randy Pausch, Nathaniel R. Young II, and Robert DeLine. SUIT: The pascal of user interface toolkits. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 117–126, Hilton Head, SC, November 1991. ACM Press.

[61] Peter Naur et. al. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6(1):1–17, 1963.

[62] G. Pfaff, editor. *User Interface Management Systems*, Berlin, 1985. Springer-Verlag.

[63] Procase Corporation, San Jose, California. *SMARTsystem Tutorial*, release 2.0 edition, April 1993.

[64] William W. Pugh and Steven J. Sinofsky. A new language-independent prettyprinting algorithm. Technical Report TR 87-808, Dept. of Computer Science, Cornell University, Ithaca, NY, January 1987.

[65] Vincent Quint. The languages of Grif. Available by anonymous ftp from ftp.imag.fr in directory `/pub/OPERA/doc`, December 1993. Translated by Ethan V. Munson.

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

[67] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: a System for Constructing Language-Based Editor*. Springer-Verlag, 1988.

[68] Lisa F. Rubin. Syntax-directed pretty printing—a first step towards a syntax-directed editor. *IEEE Transactions on Software Engineering*, SE-9(2):119–127, March 1983.

[69] Page Smith. *The Rise of Industrial America*. Penguin Books, 1990.

[70] Richard Stallman. *GNU Emacs Manual*. Free Software Foundation, Cambridge, Massachusetts, fourth edition, February 1986. Describes version 17.

[71] Ivan Sutherland. *Sketchpad: A Man-Machine Graphical Communication System*. PhD dissertation, Massachusetts Institute of Technology, Cambridge, Massachusetts, January 1963.

[72] Michael L. Van De Vanter, Susan L. Graham, and Robert A. Ballance. Coherent user interfaces for language-based editing systems. *International Journal of Man-Machine Studies*, 37:431–466, 1992.

[73] Harald Vogt, Doaitse Swierstra, and Mattijs Kuiper. *Efficient incremental evaluation of higher-order attribute grammars*, volume 528 of *Lecture Notes in Computer Science*, pages 231–242. Springer-Verlag, 1991.

[74] Richard C. Waters. XP: A Common Lisp pretty printing system. A.I. Memo 1102a, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, August 1989. Also appears in edited form as Chapter 27 of *Common Lisp: The Language, second ed.*

[75] Webster's Ninth New Collegiate Dictionary. Merriam-Webster Inc., Springfield, Massachusetts, 1987.

[76] Michal Young, Richard N. Taylor, and Dennis B. Troup. Software environment architectures and user interface facilities. *IEEE Transactions on Software Engineering*, 14(6):697–708, June 1988.

Appendix A

A Program Pretty-Printing Example

This appendix presents the parsing grammar and a presentation schema for a toy programming language called "`Simple`". The `Simple` language is intended primarily for testing the lexical and syntactic analysis of Ensemble. It has a C-like syntax and is simple enough to be understood quickly, but complex enough to make a good example of the use of Proteus for program pretty-printing.

A.1 The Grammar of `Simple`

This section contains the grammar for the `Simple` language. A `Simple` program is composed of a sequence of variable and function declarations. All variables in `Simple` are non-negative integers, so only their names need to be declared. The normal set of arithmetic operators is supported for them. A function declarations has a header that declares the function's parameters and a body composed of a sequence of declarations followed by a sequence of statements. There are four types of statements: assignments, function calls, return statements, and if statements.

The grammar of `Simple` is presented below in the format of the `bison` parser generator. It should be noted that there are several tokens that may appear in a `Simple` document that are not mentioned explicitly in any production. The `$`, `error`, and `illegal` tokens can be generated by the parser. The `$` token is used to mark the end of the text stream. The `error` and `illegal` tokens are used to mark subtrees that do not match the parsing specification. The `WS` and `COMMENT` tokens hold the text of whitespace and comments, respectively. They are generated by the lexical analyzer, but are ignored by the parser. However, they are kept in the program tree so that they may appear when the program is presented.

```
/* Simple - a simple integer language */

%{
%}
```

```
/* Implicit system tokens */
/* $, error, illegal. */

/* Single-char tokens */
%token SEMI COLON COMMA LPAREN RPAREN LBRACE RBRACE EQ

/* Variable-content tokens */
%token WS COMMENT IDENT NUM HI_OP LO_OP

/* Keywords */
%token RETURN VAR FUNCTION KEY_IF KEY_ELSE

%%

PROGRAM:       DECL_LIST STMT_LIST

DECL_LIST:      /* empty */
        |       DECL_LIST VAR_DECL SEMI
        |       DECL_LIST FUNCTION_DECL
        ;
VAR_DECL:       VAR IDENT_LIST

FUNCTION_DECL:  FUNCTION IDENT ARG_LIST BLOCK
        ;
ARG_LIST:       LPAREN RPAREN
        |       LPAREN IDENT_LIST RPAREN
        ;
IDENT_LIST:     IDENT
        |       IDENT_LIST COMMA IDENT
        ;
BLOCK:          LBRACE DECL_LIST STMT_LIST RBRACE
        ;
STMT_LIST:      /* empty */
        |       STMT_LIST STMT
        ;
STMT:           ASSIGN SEMI
        |       FUNCALL SEMI
        |       RETURN_STMT SEMI
        |       IF_STMT
        ;
ASSIGN:         IDENT EQ EXPR
        ;

EXPR:           PRODUCT
```

```
        |       EXPR LO_OP PRODUCT
        ;
PRODUCT:        SIMPLE_EXPR
        |       PRODUCT HI_OP SIMPLE_EXPR
        ;
SIMPLE_EXPR:    IDENT
        |       NUM
        |       FUNCALL
        |       LPAREN EXPR RPAREN
        ;
FUNCALL:        IDENT LPAREN RPAREN
        |       IDENT LPAREN CALL_ARGS RPAREN
        ;
CALL_ARGS:      EXPR
        |       CALL_ARGS COMMA EXPR
        ;
RETURN_STMT:    RETURN EXPR
        ;
IF_STMT:        IF_BRANCH
        |       IF_BRANCH ELSE_BRANCH
        ;
IF_BRANCH:      IF_HEADER BLOCK
        ;
IF_HEADER:      KEY_IF LPAREN EXPR RPAREN
        ;
ELSE_BRANCH:    KEY_ELSE BLOCK
        |       KEY_ELSE IF_HEADER BLOCK ELSE_BRANCH
        ;
%%
```

A.2 Presenting the `Simple` Language

To illustrate how Proteus specifications can be used to control the presentation of programs in `Simple`, this section uses the classic example of a factorial function. The raw ASCII text for this example is:

```
function factorial(x) {
   if (x) { return x * factorial(x - 1); }
   else { return 1; }
}
```

A presentation of this function is shown in Figure A.1. This presentation follows one of the common line-breaking conventions for C, which places the left brace marking the start of each block on its own line. The placement of the left braces is determined by the rules affecting two node types, `BLOCK` and `LBRACE`. `BLOCK` nodes represent the entire sequence of

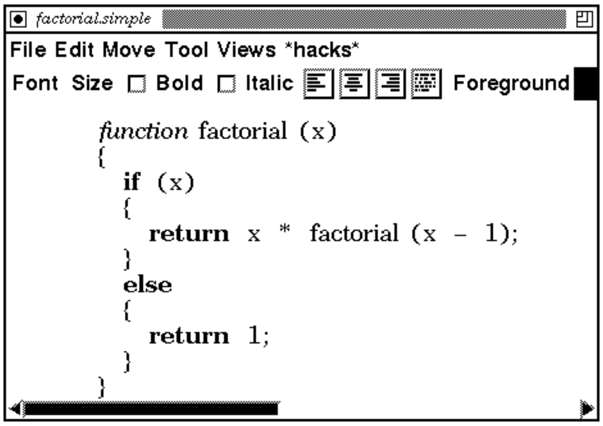


Figure A.1: A presentation of the factorial example for the `Simple` language. In this presentation, the left brace starting each block is placed alone on a new line.

declarations and statements that appear within a matching pair of braces. The `LBRACE` node is a leaf node (or token) that holds the left brace character that starts each block. In this presentation, the position of the left brace is determined entirely by default rules, because it has no rules of its own. Furthermore, because the `LBRACE` node is the first child of the `BLOCK` node and has no left sibling, its default rule

```
VertPos: Top = LeftSib . Top;
```

fails and the implicit default rule (i.e. simple inheritance is used). The explicit default rule for horizontal position

```
HorizPos: Left = Parent . Left;
```

also specifies simple inheritance. So, the `LBRACE` node's position is precisely that specified for its parent the `BLOCK` node.

The `BLOCK` node's vertical position is determined by the rule:

```
VertPos: Top = LeftSib . Bottom;
```

This aligns the top of the block with the bottom of its predecessor, which is either the argument list of the function header, the if header, or the else keyword. The `BLOCK` node's left edge is set to be the same as its parent by the default rule.

The complete presentation schema is:

```
MEDIUM text;

PRESENTATION simple FOR simple;

DEFAULT
   BEGIN
   Width = AllChildren . Width;
   Height = AllChildren . Height;
```

```
   VertPos: Top = LeftSib . Top;
   HorizPos: Left = Parent . Left;
   Justify = Parent . Justify;
   FontFamily = Parent . FontFamily;
   Bold = Parent . Bold;
   Italic = Parent . Italic;
   Size = Parent . Size;
   LineSpacing = Parent . LineSpacing;
   Indent = 0;
   RightMargin = Parent . RightMargin;
   Visible = Yes;
   END;

RULES
// In programs, the root node is 'ultra_root', to make sure that
// a comment at the head of the program is visible in the tree.
   ULTRA_ROOT:
        BEGIN
        Width = 432;
        HorizPos: Left = 72;
        VertPos: Top = 10;
        FontFamily = "new century schoolbook";
        LineSpacing = 1.2;
        Bold = No;
        Italic = No;
        Size = 18;
        Justify = LeftJustify;
        RightMargin = 1000;
        END;
   PROGRAM:
        BEGIN
        VertPos: Top = LeftSib . Bottom;
        END;
   BLOCK:
        BEGIN
        VertPos: Top = LeftSib . Bottom;
        END;
   FUNCTION:
        BEGIN
        Italic = Yes;
        END;
   DECL_LIST :
        BEGIN
        VertPos: Top = LeftSib . Bottom;
        IF (TypeOf(Parent) == "BLOCK") THEN
            HorizPos: Left = LeftSib . Left + 20;
        ENDIF
        END;
   FUNCTION_DECL :
        BEGIN
```

```
     IF (TypeOf(LeftSib) == "DECL_LIST") THEN
         VertPos: Top = LeftSib . Bottom;
     ENDIF
     END;
VAR_DECL :
     BEGIN
     IF (TypeOf(LeftSib) == "DECL_LIST") THEN
         VertPos: Top = LeftSib . Bottom;
     ENDIF
     END;
STMT_LIST :
     BEGIN
     VertPos: Top = LeftSib . Bottom;
     IF (TypeOf(Parent) == "BLOCK") THEN
         HorizPos: Left = Parent . Left + 20;
     ENDIF
     END;
STMT :
     BEGIN
     IF (TypeOf(LeftSib) == "STMT_LIST") THEN
         VertPos: Top = LeftSib . Bottom;
     ENDIF
     END;
ARG_LIST:
     BEGIN
     HorizPos: Left = LeftSib . Right + 10;
     END;
IDENT_LIST:
     BEGIN
     IF (TypeOf(Parent) != "IDENT_LIST") THEN
         IF (TypeOf(LeftSib) == "LPAREN") THEN
             HorizPos: Left = LeftSib . Right + 2;
         ELSE
             HorizPos: Left = LeftSib . Right + 10;
         ENDIF
     ENDIF
     END;
IDENT:
     BEGIN
     IF (TypeOf(Parent) == "FUNCTION_DECL") THEN
         HorizPos: Left = LeftSib . Right + 8;
     ELSE
         HorizPos: Left = LeftSib . Right + 2;
     ENDIF
     END;
VAR:
     BEGIN
     Italic = Yes;
     END;
KEY_IF:
```

```
     BEGIN
     Bold = Yes;
     END;
KEY_ELSE:
     BEGIN
     Bold = Yes;
     END;
RETURN:
     BEGIN
     Bold = Yes;
     END;
EQ :
     BEGIN
     HorizPos: Left = LeftSib . Right + 4;
     END;
HI_OP :
     BEGIN
     HorizPos: Left = LeftSib . Right + 4;
     END;
LO_OP :
     BEGIN
     HorizPos: Left = LeftSib . Right + 4;
     END;
NUM :
     BEGIN
     HorizPos: Left = LeftSib . Right + 4;
     END;
SEMI :
     BEGIN
     HorizPos: Left = LastLeaf(LeftSib) . Right + 2;
     VertPos: Bottom = LastLeaf(LeftSib) . Bottom;
     END;
COMMA :
     BEGIN
     HorizPos: Left = LeftSib . Right + 2;
     END;
LPAREN:
     BEGIN
     IF ((TypeOf(Parent) == "FUNCALL") ||
         (TypeOf(Parent) == "IF_HEADER")) THEN
         HorizPos: Left = LeftSib . Right + 10;
     ENDIF
     END;
RPAREN:
     BEGIN
     HorizPos: Left = LeftSib . Right + 2;
     END;
RBRACE:
     BEGIN
     VertPos: Top = LeftSib . Bottom;
```

```
     END;
EXPR:
     BEGIN
     IF (TypeOf(LeftSib) == "LPAREN") THEN
         HorizPos: Left = LeftSib . Right + 2;
     ELSE
         HorizPos: Left = LeftSib . Right + 10;
     ENDIF
     END;
SIMPLE_EXPR:
     BEGIN
     HorizPos: Left = LeftSib . Right + 10;
     END;
PRODUCT:
     BEGIN
     HorizPos: Left = LeftSib . Right + 10;
     END;
HI_OP:
     BEGIN
     HorizPos: Left = LeftSib . Right + 10;
     END;
LO_OP:
     BEGIN
     HorizPos: Left = LeftSib . Right + 10;
     END;
CALL_ARGS:
     BEGIN
     HorizPos: Left = LeftSib . Right + 2;
     END;
IF_HEADER:
     BEGIN
     HorizPos: Left = LeftSib . Right + 10;
     END;
ELSE_BRANCH:
     BEGIN
     VertPos: Top = LeftSib . Bottom;
     END;
WS:
     BEGIN
     IF (TypeOf(LeftSib) == "COMMENT" ||
         TypeOf(LeftSib) == "DOC_COMMENT") THEN
         VertPos: Top = LeftSib . Bottom;
     ELSE
         HorizPos: Left = LeftSib . Right;
     ENDIF
     END;
COMMENT:
     BEGIN
     HorizPos: Left = LeftSib . Right + 10;
     FontFamily = "helvetica";
```

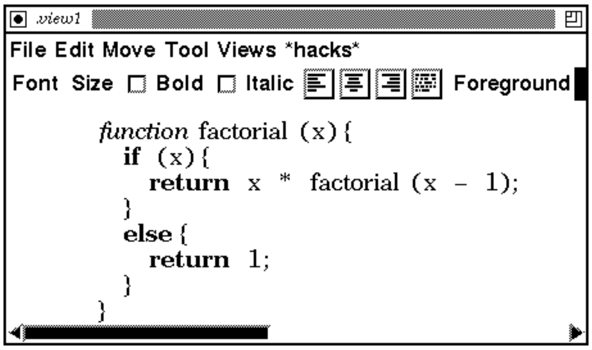


Figure A.2: An alternate presentation of the factorial example in which the left brace beginning each block is placed at the end of the line containing the preceding material.

```
        END;
   DOC_COMMENT:
        BEGIN
        HorizPos: Left = LeftSib . Right + 10;
        END;
END
```

An alternate presentation of the same program can be seen in Figure A.2. This presentation uses the other common format for left braces, in which the left brace is placed on the same line as the preceding material. To produce this presentation only two changes were made in the presentation schema. First, the vertical position rule for the `BLOCK` node was removed. This makes the default rule apply, which aligns the top of the `BLOCK` node with the top of its left sibling. Since the LBRACE node inherits its vertical position from the `BLOCK` node, this change effectively moves the left brace up to the same line as the preceding material. Secondly, the `LBRACE` node is given an explicit horizontal position rule:

```
HorizPos: Left = LeftSib(Parent) . Right + 2 ;
```

This places the left edge of the brace two points to the right of the preceding element. (The material preceding the left brace is always part of the left sibling of its parent.)