# The Proteus Presentation System*

Susan L. Graham† Michael A. Harrison† Ethan V. Munson†

Computer Science Division, 571 Evans Hall
University of California, Berkeley, California 94720

## Abstract

Software development environments can increase user productivity by presenting information in more useful ways. This paper describes Proteus, the presentation system of Ensemble, a software development environment that supports a wide variety of language and document capabilities. Proteus provides a set of services which allow the appearance of software development documents, such as programs or design specifications, to be determined by formal specifications of style. Proteus is based on a generic model of presentation services and is intended for use with a wide variety of media.

## 1 Introduction

One of the key purposes of a software development environment should be to allow users to view the artifacts of the development process in a variety of insightful ways. Insight may be stimulated by one or more thoughtful presentations of a program's text. Alternatively, the results of program analyses may be the source of new insight. Whether these results need to be presented in textual or graphic form in order to be understood cannot be known in advance.

This realization, that presentation of information is a central task for a software development environment, is one of the motivations behind our design of Ensemble, an environment providing language-based services for integrated software and multi-media document development. This paper presents our current work on Proteus, the presentation system of Ensemble. Proteus provides a unified mechanism for specifying the presentation of formalized language documents such as programs or design specifications and natural language documents such as user manuals. This mechanism extends to the incorporation of additional media. Documents may be viewed interactively using many different styles at the same time, each defined by a different formal specification.

*This research has been sponsored in part by the Defense Advanced Research Projects Agency (DARPA) under Contract N00039-88-C-0292, monitored by Space and Naval Warfare Systems Command, and under Grant MDA972-92-J-1028. The content of the information in this paper does not necessarily reflect the position or the policy of the Government.

†Computer Science Division, University of California, Berkeley, CA, 94720. Email addresses: {graham, harrison, munson}@cs.berkeley.edu.

To appear in Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments, Tyson's Corner, VA, December 9–11 1992.

## 2 Background

### 2.1 Ensemble

The software development process involves the creation of many different kinds of documents. Some are written in formalized languages such as programming and specification languages. Others, such as requirements documents and user manuals, are primarily natural language documents. Research in software development environments has typically focused on supporting formalized language documents, emphasizing tools to create formally correct content. The production of natural language documents has relied on other tools, such as page layout systems, word processors, and batch formatters.

The Ensemble project is attempting to bridge the gap between formalized language tools and natural language tools by creating a new type of environment providing integrated, interactive support for the development of both software and multi-media documents. There are several reasons that it makes sense to join document preparation services and advanced support for formalized languages in an integrated system.

**Interoperability:** Software developers must create, browse, and maintain both formalized language and natural language documents. An integrated system can more easily provide seamless support for all the artifacts of the software process.

**Structure:** It is easy to think of natural language documents as having little structure, particularly in comparison to computer programs. In reality, natural language documents have substantial structure such as paragraphs, sections, and captions. However, unlike formalized language documents, this structure cannot be found by analysis of the document's content. Instead, the structure of these documents must be specified explicitly by their authors. The process of defining a natural language document's structure is little different from the top-down editing techniques used in syntax-directed editors. The net result is that both classes of documents can make good use of environment services for managing structure.[1]

**Incremental Consistency:** A key problem for interactive systems of any kind is the incremental maintenance of consistency among a variety of representations. Both software development environments and document processing systems have had to address this issue. An integrated system can provide both domains with a generic solution to the problem.

**Document Services:** The same services that present attractive natural language documents can be used to make formalized language documents easier to read and understand.

[1] Structured documents are an important area of current document processing research. The original ideas were introduced in the Scribe system [20] and have been elaborated in work by Quint and Vatton [19] and Furuta [6]. Expressing document structure has been the major concern in document interchange research [7, 9]. In Ensemble, all documents have structure, though the structure of some is so simple as to make them appear unstructured.

**Language Services:** Many services needed to support the creation, analysis, and maintenance of formalized language documents are also required when managing structured natural language documents. Examples of shared services are structure specification (via grammars), support for top-down editing, and incremental maintenance of attribute consistency.

**Multi-media:** Since software objects have a number of graphical analogs, it makes sense to include multi-media capabilities in a software development environment. Multi-media services could be used to manage the presentation of graphical abstractions such as Gantt charts and to attach digitized audio annotations to documents.

The Ensemble system is an integrated, interactive environment for software and multi-media document development. It is the direct descendant of the Pan language-based editing/browsing system [3] and the VORTEX multiple-representation document editor/formatter [5]. Among its features are integrated analysis tools for formalized languages (lexical analysis, parsing, semantic analysis) and direct-manipulation editing of multi-media documents.

The user of Ensemble will be able to browse and edit a wide variety of software and document objects. Furthermore, the user will be able to simultaneously view and edit multiple representations of the same object. For instance, a programmer might see both the source code and call graph representations of his program. Changes to the source code which affect the call graph could appear immediately in the call graph representation. The programmer could use the call graph representation to move around in the source code window.

A key reason that Ensemble can support multiple-representation editing is that the appearance of a document is specified separately from the specification of its structure. Thus, a single object can be presented in radically different ways simply by choosing a different appearance specification.

Finally, Ensemble supports compound documents. That is, a particular Ensemble document can be constructed from components with differing language structure and presentation styles. This eases the construction of documents like interactive user manuals which mix executable program material with multi-media explanations.

## 2.2 Program Presentation

The work of Baecker and Marcus [2] on the presentation of C programs illustrates the range of output styles we want to support in Ensemble. The code fragment in Figure 1 is an example. To produce output like this requires support for

- multiple, proportionally-spaced fonts of various sizes,
- fine-grained adjustment of spacing,
- non-textual typographic effects like rules and shading,
- elision of portions of the program text (such as block delimiters),
- rearrangement of the sequence of material (placing in-line comments in the left margin, for example).

It might be argued that that this kind of fine-tuning of the presentation of programs is more an exercise in aesthetics than in some essential aspect of software development. However, Oman and Cook [14] demonstrated that careful use of typographic effects can strongly influence how well programmers, either novice or expert, understand a program.

Effective presentation of programs has long been a concern in computer science. The Algol 60 report [16] included the notion of a *publication language* that was separate from the *reference language*

```
Encode phone number as a vector of digits, without
punctuation. Returns number of digits in phone
number or FALSE to indicate failure.

static bool
getpn(str)
______________________________________________
      char                              *str;
______________________________________________
      int                               i = 0;

      while (*str != '\0')
            if (i >= PNMAX)
                  return FALSE;

            Set pn to the digits ignoring spaces and dashes

            if (*str != ' ' && *str != '-')
                  if ('0' <= *str && *str <= '9')
                        pn[i++] = *str – '0';
                  else
                        return FALSE;
```

Figure 1: **A program presentation example.** Baecker and Marcus [2] present a careful exploration of design choices for C programs from the viewpoint of a graphic designer. This code fragment is taken from their work (p. 61).

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.

More recently, 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 *prettyprinter* takes as input a stream of characters and prints them with aesthetically appropriate indentations and line breaks [15].

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 [12] provides a good example of this research while many other such works are cited by Baecker and Marcus [2, p. 18]. However, our interest is primarily in mechanism rather than policy.

Early attempts at automatic pretty-printing were made by the Lisp community. The culmination of this process is Waters's XP [23], 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 [15] 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 [17] modified Oppen's algorithm to be more incremental and enriched the set of text stream markers. Mikelsons [13] developed a different approach based on nested boxes which appears to be language-independent, but his implementation is not. Knuth and Plass [11] 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 [22] 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 [21], which uses unparsing directives in an attribute grammar. Pugh and Sinofsky's work on algorithms was intended for use with the Synthesizer Generator. PPML [8], 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 all shares a common flaw; there is an implicit 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.

Grif, an editor for structured natural language documents [19], has significantly influenced our work on Proteus. Each Grif document belongs to a *document class*. The possible structures of the members of a document class are defined formally by a *structure schema*, similar to a context-free grammar. The appearance of a Grif document is also defined formally, by a *presentation schema*. Like PPML specifications, Grif's presentation schemas are separate entities from their corresponding grammar. Where Grif moves ahead of PPML is in providing control over a wide range of typographic features, including font choice, fine-grained layout, elision, and cross-referencing. There may be many presentation schemas for a document class, each defining a different appearance style. A particular Grif document may be displayed and edited in several different styles simultaneously, with the changes made in one view being reflected in all other views. However, Grif has no support for editing formal languages. It cannot lexically analyze or parse a program. Thus, while Grif can be used to produce sophisticated renderings of programs, their structure must be explicitly defined by the user, rather than deduced by analysis of a text stream. Furthermore, Grif is largely limited to the textual medium and cannot be extended to support new media.

The Chiron user interface development system [10, 24] is another influence on our work. A central concept in Chiron is that of the *artist*, which is an agent responsible for managing the appearance of an abstract data type (ADT). The Chiron system provides a server that renders user interface objects like buttons and scroll bars and that multiplexes user input to its various artist clients. Furthermore, 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 will require extension to support high-quality multi-media documents [4].

# 3 A Model of Presentation

Proteus is based on a generic model of the presentation process. This model has two parts, an architecture and a set of presentation services.

In discussing this model of presentation, the term *medium* (plural *media*) will be used frequently. Intuitively, a medium is a mode of communication or artistic expression [1]. Common examples of media include text, graphics, audio, animation, and video. In designing Proteus, we had to construct a more operational definition of the term.

> A *medium* is a set of object types, a set of dimensions in which such objects are laid out, and a set of parameterized formatting operations on those objects.

This definition easily distinguishes between the obvious examples of media. However, it can also make much finer distinctions. For instance, our definition implies that a graphics system which can fill objects with color gradients provides a different medium than a system which does not support color gradients.

## 3.1 Architecture

The presentation process involves three types of agents: presentation systems, formatters, and renderers.

- A *presentation system* is a medium-independent, device-independent service that maps the rules in formal style specifications to the actual content of particular documents. The formal style specifications used by a presentation system are called *presentation schemas*.
- A *formatter* is a medium-specific, device-independent agent that lays out documents in a virtual space. Some of the parameters of its layout process are obtained by querying the presentation system for values specified in a presentation schema. The presentation schema used with a particular document is chosen by the user. Once the document is laid out in the virtual space, the formatter may display or print it by sending output commands to a renderer.
- A *renderer* is a medium-specific, device-specific agent that takes objects laid out in a virtual space and renders them on a particular device.

It may seem unusual that we reserve the name "presentation system" for a single architectural component when formatters and renderers are clearly also involved in the process of presenting information to the user. We do so because the presentation system is a generic module whose only purpose is to support the management of presentation. In contrast, a formatter can be discussed in the context of its medium (e.g. a text formatter or a graph formatter), while a renderer can be discussed with reference to its medium or device (e.g. an X11 graphics renderer).

The flow of information between presentation systems, formatters, and renderers is shown in Figure 2. The presentation system accepts two major classes of input, documents and presentation schemas. A document is a tree-structured object with typed nodes. The possible structures of the document are defined by a structure schema. As stated before, a presentation schema is a formal specification of the appearance of the document.

Based on this input, the presentation system provides two forms of information to the formatter. First, it produces a new version of the document, called the *presentation tree*. As a general rule, the presentation tree contains all of the nodes in the original document

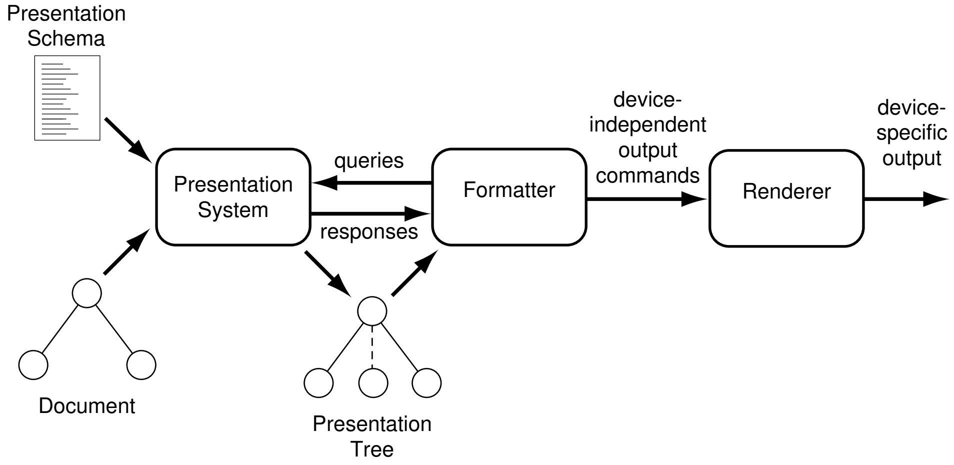


Figure 2: **Overall presentation architecture.** This figure shows the flow of information in the presentation architecture.

tree.[2] It also contains additional structure and content generated in accordance with *tree elaboration rules* in the presentation schema. Secondly, the presentation system stands ready to answer queries from the formatter about the values of presentation attributes for particular nodes of the presentation tree.

The formatter is the controlling element in the presentation process. It receives requests from other parts of the editing system to format and display documents. Given such a request, the formatter queries the presentation system for presentation attribute values and based on the answers to those queries lays out the document in a virtual space. It then displays or prints the document by issuing output commands to a renderer, which renders the document on a particular device.

An important additional service provided by the formatter is to help other parts of the program handle pointer-based user interaction. The formatter helps by mapping locations or regions on devices (such as CRT displays) to nodes in the document tree. To provide this service, the formatter will almost certainly use information from the presentation system (about attributes which determine size and position of objects) and from the renderer (about how device coordinates map into coordinates on the virtual space). Our presentation architecture assumes that all other aspects of user interaction are handled by other components of the editing system.

An important complication arises in the interaction between the formatter and the presentation system. Because the presentation system is medium-independent, there are certain aspects of a particular document's presentation that it cannot know. The most common example is that only the formatter can determine the actual size of primitive objects (e.g. characters, paragraphs, polygons). In order for the presentation system to support specifications based on this information, it must make callbacks to the formatter to obtain the correct values.

An environment supporting multiple media needs one presentation system, several formatters and several renderers. A single presentation system is sufficient because the services provided by it do not depend on either the medium being presented or the device being used for presentation. In contrast, we envision having several different formatters, each embodying the concepts and methods of a different medium (e.g. text, graphics, or animation). Finally, an environment must have at least one renderer for every device to which it sends output, though it may have more than one for some devices. In our model, devices are abstract entities such as the X11 window system or a PostScript printer rather than hardware devices. Though abstract, these kinds of devices still differ considerably in their interfaces and rendering models. A renderer encapsulates these differences.

[2]It is possible for the presentation system to support elision by excluding certain nodes in the document tree from the presentation tree. We are exploring this and other approaches to elision.

## 3.2 Presentation Services

Our model of presentation is based on a set of four medium-independent presentation services: attribute propagation, box layout, tree elaboration, and interface functions. These services are provided by the presentation system and are reflected in the commands available in the languages in which presentation schemas are written.

### 3.2.1 Attribute propagation

*Attribute propagation* is the most important of these services. Each medium has a set of attributes whose values determine the appearance of an object. These attributes are the parameters of the medium's formatting operations. For text documents, these attributes might include

- font family,
- type size,
- type style,
- a boolean determining whether to line break,
- a boolean determining whether lines are justified, and
- the amount of whitespace between lines.

Most of the time, the values of these attributes are not specified explicitly for every node. Rather, they are propagated through the document tree by constraints on the values of other nodes. A common example in text documents is the choice of font family (e.g. Times or Helvetica), for which a value is typically specified at the root of the document tree and then inherited by all other nodes. Inheritance is a particular kind of constraint in which the value of a node is constrained to be the same as that of its parent.

#### 3.2.2 Box layout

The *box layout* service provides a minimal mechanism for laying out elements of the document. It is based, in part, on our assumption that each medium has one or more dimensions (e.g. text has two). Every object in a particular medium has four attributes for each dimension: extent, minimum, maximum, and center. It is important to note that there are only two degrees of freedom among these four attributes. Typically, these two degrees of freedom are characterized as being extent and position (which subsumes minimum, maximum, and center). This choice is intuitively reasonable, but arbitrary.

The dimensional attributes are similar to other attributes. Their values may be set explicitly or by constraint on attribute values of other nodes. The key difference between them and other attributes is the existence of the implicit constraints that result from having four attributes describing a system with two degrees of freedom.

The layout service is called "box layout" because it assumes that the objects being laid out are shaped like $n$-dimensional boxes. This is much simpler than laying out general polyhedra. In particular, $n$-dimensional boxes have unique values for the four attributes of each dimension, while polyhedra do not.

#### 3.2.3 Tree elaboration

The *tree elaboration* service augments the document tree with material specified in the presentation schema. For instance, the horizontal lines in Figure 1 would be produced by tree elaboration. The presentation system automatically adds these nodes to the document tree in response to certain commands in the presentation schema. The presentation tree is the result of all the additions to the document tree. Because the formatter sees only the presentation tree, it is not aware of which material is added by elaboration and which is not.

#### 3.2.4 Interface functions

If the presentation engine were just a passive repository of information responding to queries by the formatter, certain complex formatting operations would be very difficult. A sensible program documentation style might have page headings that show the name of the function or module being documented. However, the presentation system has no way of knowing how the content of the document tree is broken into pages; that is the job of the formatter. Meanwhile, the formatter cannot know what material should appear in the page header; that information is the domain of the presentation system.

The solution is to have the formatter designer create *interface functions* and register them with the presentation system. An interface function is a callback which allows the presentation system to query the formatter for useful information. The page header problem can be solved by an interface function which returns the first leaf node falling on the current page. This function can be used in presentation schema rules which use tree operations to ascend the tree and find the function or module name.

# 4 Proteus

Proteus is the presentation system of Ensemble. Its role in the overall presentation architecture of Ensemble is to take a document and a presentation schema, produce a presentation tree and respond to queries from a formatter about the values of presentation attributes for particular nodes of the presentation tree. More importantly, Proteus must provide this service for several different combinations of document and presentation schema at a time. Furthermore, Proteus must be able interact with multiple formatters simultaneously.

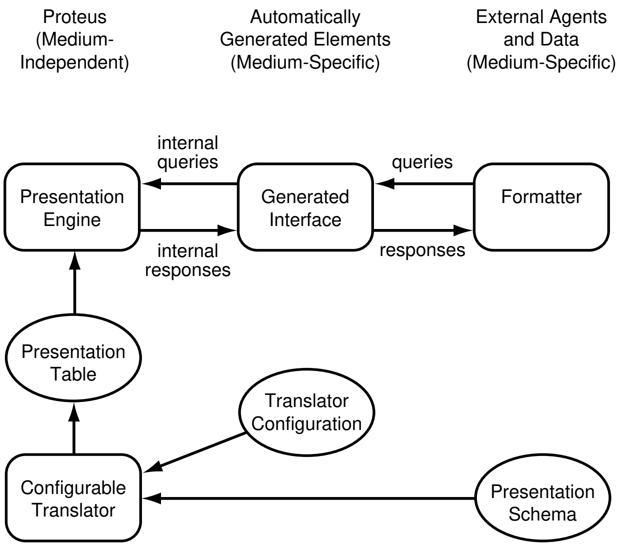


Figure 3: **The internal architecture of Proteus.** Boxes represent agents, while ovals represent data. The arrows indicate the flow of information.

## 4.1 The Architecture of Proteus

Proteus has an interesting internal architecture that allows it to adapt to the needs of different formatters. This architecture is shown graphically in Figure 3. The underlying goal of this architecture is to create a clear separation between the medium-independent and medium-specific aspects of presentation. This separation allows Proteus to mold its single set of generic presentation services to the requirements of media with very different demands.

The left side of Figure 3 shows the two major components of Proteus itself, the *presentation engine* and the *configurable translator* for presentation schemas. These two components are completely generic. The presentation engine is the runtime agent responsible for attribute propagation, box layout, and tree elaboration. To provide these services, it relies on *presentation tables* which are created by the configurable translator from presentation schemas.

Proteus adapts to the needs of different media via a *custom interface* and a *translator configuration* (seen in the middle of the figure). They are best thought of as part of the formatter, rather than as part of Proteus, because each medium's formatter has its own versions of these two elements. The custom interface maps medium-specific queries from the formatter into the generic internal forms used by the presentation system. It performs the inverse transformation on the responses of the presentation system. It is type-safe and completely hides the internals of the presentation engine from the person implementing the formatter. The translator configuration contains all the medium-specific information needed to parse the presentation schemas for a particular medium. It provides the following information to the configurable translator:

- For each dimension used by the medium, the names of its extent and position attributes and the names of its minimum, maximum, and middle positions.
- For some dimensions, the units which are available for use. For example, the time dimension uses hours, minutes, and seconds instead of distance units like inch or centimeter. In such cases, one type of unit is identified as canonical and conversion ratios between it and the other units must be supplied.
- The name and type of each non-dimensional attribute.

- For each interface function, the name of the function and the types that it maps between.

The result is that each medium has its own unique presentation schema language, but that language shares a common syntax and set of presentation concepts with every other presentation schema language. Both of these medium-specific elements are designed to be generated automatically from a specification. They are currently generated manually.

## 4.2 Presentation Schema Languages

The appearance of a program or other Ensemble document is determined primarily by a presentation schema, which specifies a set of formatting rules for the document's class. Each presentation schema is written in a *presentation schema language*.

Unlike earlier systems, Proteus does not have a single presentation schema language supporting all media. Rather, it supports a family of presentation languages, there being a separate one for use with each medium. These languages have in common their syntax and the core presentation concepts described in Section 3. They differ in the attributes they support, the number of dimensions in which layout is performed, and the interface functions provided.

Having a separate language for each medium leaves each language uncluttered by concepts native to other media. The text language doesn't need to support layout in the $Z$ or time dimensions, while the language for three-dimensional animation need not support the `Justify` attribute. In fact, the languages for different media can give different names to the same dimension if doing so better suits the conventions of the media. Furthermore, the addition of a new medium to Ensemble does not invalidate any existing schemas for other media.

The key disadvantage to having multiple presentation schema languages is that it is not possible to write a single presentation schema for a multi-media document. Instead, each subcomponent of the document that is to be presented in a foreign medium must have a separate schema. Whether this is an important limitation remains to be investigated.

To make this discussion more concrete, a sample presentation schema for documents of the `SimpleDoc` class is shown in Figure 4. Documents of this class are composed of a title, which is text, and a body which is a sequence of paragraphs, each of which is text. The presentation schema is written in the presentation schema language for the textual medium.

Presentation schemas have four parts, a preamble, a defaults section, a boxes section, and a rules section. The preamble gives the name of the presentation and the name of its corresponding structure schema. The schema in Figure 4 states that the presentation is named `SimplePres` and it is for use with the `SimpleDoc` structure schema.

The defaults section defines rules for computing attribute values that are to be used when no rule is given in the rules section. Most default rules define an attribute by constraint on some other node. In the `SimplePres` example, the default rule for the `Height` attribute constrains it to be the height of the node's children. Attributes can also be given literal default values, as with the `Break` attribute, for which the default value is `No`.

The boxes section declares the material that is to be added to the document in creating the presentation tree. The `SimplePres` schema only defines one box, the `TitleLabel`. The `Content` attribute is used to specify the actual material to be generated (in this case, the string "Title:") and can only be used in box declarations.

The rules section defines rules for each node type in the document class. Only those rules that deviate from the default rules need to be defined. In the example, the rules for the `SimpleDoc` node set up values that are inherited by most of the other nodes. The remaining nodes in this simple document generally need only adjust

```
PRESENTATION SimplePres
      FOR SimpleDoc;

DEFAULT
   BEGIN
   Width = Parent . Width;
   Height = AllChildren . Height;
   VertPos: Top = LeftSib . Bottom;
   HorizPos: Left = Parent . Left;
   Justify = Parent . Justify;
   Break= No;
   FontFamily = Parent . FontFamily;
   Bold = No;
   Italic = No;
   Size = Parent . Size;
   LineSpacing = Parent . LineSpacing;
   Indent = 12 pt;
   Visible = Yes;
   END;

BOXES
   TitleLabel :
      BEGIN
      Content = Text("Title:");
      Size = 16 pt;
      END;
RULES
   SimpleDoc :
      BEGIN
      Width = 432 pt;
      HorizPos : Left = Parent . Left + 72 pt;
      VertPos : Top = Parent . Top + 36 pt;
      FontFamily = "Times";
      LineSpacing = 1.2;
      Size = 11 pt;
      Justify = No;
      END;
   Title :
      BEGIN
      CreateBefore(TitleLabel);
      HorizPos : Left = LeftSib . Right + 12 pt;
      Bold = Yes ;
      Size = 16 pt;
      Indent = 0 pt;
      END;
   Body :
      BEGIN
      VertPos : Top = LeftSib . Bottom + 10 pt;
      END;
   Paragraph :
      BEGIN
      VertPos : Top = LeftSib . Bottom + 5 pt;
      Break = Yes;
      END;
END
```

Figure 4: Prototype presentation schema for the SimpleDoc document class.

their position. An important exception is the `Title` node, which generates its label and places it in the presentation tree as its left sibling using the `CreateBefore` elaboration operator. Note that while the label must be generated by another node, it has its own independent set of attribute rules.

While the gist of this prototype language should be clear from the example, a number of issues should be explained in more detail.

- Attribute values may be boolean, string or numeric (floating-point). The boolean literals are `Yes` and `No`. String literals are enclosed in double quotes.

- In order to define an attribute's value by constraint, another node in the tree must be identified. This is done with *tree operators*, of which the prototype supports three: `Parent`, `LeftSib`, and `RightSib`. They can be composed to access values at non-adjacent nodes, as in

  ```
  LeftSib(Parent)
  ```

  which specifies an "aunt".

- The `AllChildren` term is also a tree operator, but it is special and can only be used with layout attributes. It instructs the presentation engine to scan all children of the node to find the smallest minimum position and the largest maximum position among them. The requested attribute is then computed from those two values. For instance, the value of `AllChildren.Height` would be computed by finding the highest `Top` and the lowest `Bottom` among the node's children and taking their difference.

- The position attributes (`VertPos` and `HorizPos`) subsume the minimum, maximum, and middle of their respective dimensions. This approach gives the language simple semantics and facilitates a simple implementation of the presentation engine. We are interested in finding a more general approach to dimensional attributes.

- Generated nodes are created with the four tree elaboration operators. Siblings are generated using the `CreateBefore` and `CreateAfter` operators. The `Create` and `CreateLast` operators generate first and last children, respectively. Placement of the generated node in the tree is used to manipulate how attribute constraints are resolved.

- Certain kinds of rules can fail. For instance, constraints based on the left sibling fail for nodes which are the leftmost child. If a normal rule fails, the corresponding default rule is used instead. If a default rule fails, there is an implicit default rule that the value of the node's parent is used. To prevent the implicit default rule from failing, the root of the document tree must have a valid rule (normal or default) for every attribute.

### 4.3 Using Proteus for Program Presentation

Proteus is already capable of producing interesting program presentations and is still being enhanced. As an example, we show (in Figures 5 and 6) screen dumps of two different presentations of a C switch statement. The first presentation uses a single font and has a fairly conventional line-oriented layout. The second presentation uses a table-like layout and has a number of font changes. Our goal is not to advocate either of these presentations, per se, but rather to demonstrate the range of output possible with just a simple presentation system based on our model.

It is important to note that the presentation in Figure 6 is very difficult to reproduce in other systems which alter the text stream

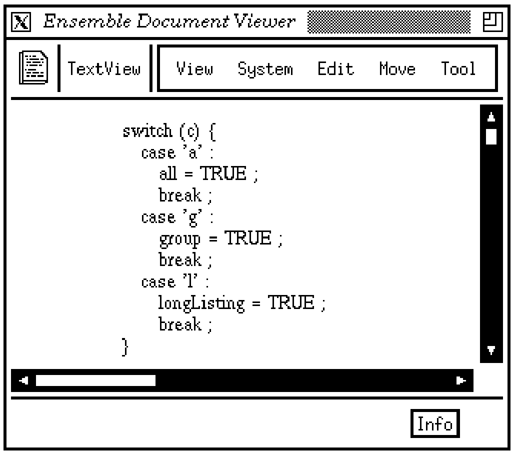


Figure 5: **Simple presentation of a switch statement.**

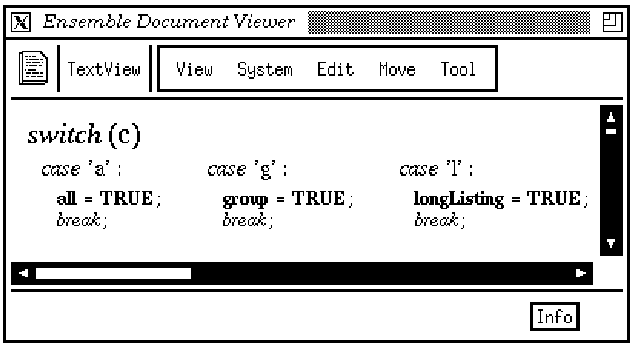


Figure 6: **Complex presentation of a switch statement.** This example shows both coarse-grained and fine-grained presentation operations.

in order to achieve layout effects. In such systems, it would be necessary to rearrange the sequence of tokens to get elements from different cases on the same line. Doing so would render the text stream syntactically invalid, which would probably be unacceptable.

For these examples the structure of the switch statement was defined by the following grammar fragment (in extended BNF).

```
switch: sw_header lbrace case_list rbrace ;
sw_header: sw_key expr ;
case_list: case_block+ ;
case_block: case_key value colon stmts ;
stmts: stmt* ;
```

The schema which produced the simpler presentation in Figure 5 primarily required specifying the location of line-breaks. The default position rules were

```
VertPos: Top = Parent . Top;
HorizPos: Left = LeftSib . Right + 4 pt;
```

The line-breaks were created by giving the `case_list`, `case_block`, `stmts`, and `stmt` nodes explicit vertical and horizontal position rules:

```
VertPos: Top = LeftSib . Bottom ;
HorizPos: Left = Parent . Left ;
```

All other attribute values were obtained via default rules or by being set at the root of the document and inherited.

The more complex tabular presentation in Figure 6 has a slightly more complex presentation schema. The `case_list` is placed below the header with the rules

```
VertPos: Top = LeftSib . Bottom + 5 pt;
HorizPos: Left = Parent . Left + 10 pt;
```

Then, each `case_block` is placed horizontally by the rule

```
HorizPos: Left = LeftSib . Right + 30 pt;
```

This places most `case_blocks` thirty points to the right of their predecessors. The exception is the first `case_block`, which has no left sibling. Thus, both its explicit and default rules for the `HorizPos` attribute fail, forcing Proteus to find a value by the implicit default rule (inheritance from the parent). As a result, the left edge of the first `case_block` is aligned with the left edge of the `case_list`. Similar manipulations were used to indent the statement list of each case and to force line-breaks between statements.

In addition to the rather coarse-grained layout manipulations already described, Figure 6 shows a number of more fine-grained changes from the presentation in Figure 5. Keywords were italicized by giving the `Italic` attribute the `Yes` value. Identifiers appear in bold-face because they have a `Yes` value for the `Bold` attribute. More interestingly, the braces surrounding the `case_list` were elided because their `Visible` attribute had a value of `No`. The most fine-grained change was applied to the semicolon terminating each statement, which has only two points of white space preceding it instead of the normal four points. This makes the semicolon seem more closely attached to rest of the statement.

# 5 Discussion

So far, we have presented the model on which Proteus is based and described Proteus's architecture and prototype implementation. In this section, we attempt to compare Proteus with other approaches.

Attribute propagation is such a key service of Proteus that there is an obvious comparison with attribute grammar systems. Attribute grammar systems typically focus on the problem of computing and maintaining static semantics of programming languages. Since programming languages generally have only a single correct semantic interpretation, there is no separation between the grammar specification and the specification of attributes. In contrast, Proteus's presentation schemas are kept separate from the specification of structure in order to support an arbitrary number of presentations. Another interesting contrast is that presentation attributes seem to be primarily inherited, rather than synthesized. It remains to be seen whether this difference has an impact on the choice of implementation. This being said, we plan to explore the use of a general attribute propagation system in Proteus.

Because rules for attribute propagation in Proteus are expressed as constraints on the values of other nodes in the document, another obvious comparison arises with general-purpose constraint maintenance systems. In point of fact, Proteus *is* a constraint system (just as an attribute grammar system is one, also). Some of our earlier research explored using a general-purpose constraint system as the core of the presentation system. This work led us to believe that the performance of the general purpose solution is inadequate for a presentation system.

Proteus bears an architectural resemblance to Chiron [10, 24]. Both systems have a centralized service that provides presentation services to clients. Both systems help clients adapt to their model by generating interfaces for them. The interface between Chiron and its artists seems to be quite similar to that between Proteus and the formatters of Ensemble. However, the two systems have substantial differences. Under Chiron, a new presentation of a class of objects is created by building a new artist, which is a program module issuing procedural commands. Under Proteus, a new presentation is created by the addition of a human-readable declarative specification. The Chiron server is a device-specific agent that renders user-interface objects and multiplexes user input to artists, which are device-independent and medium-specific. Thus, under our model of presentation, the Chiron server is a renderer while the artists are formatters. Perhaps Proteus could be used in conjunction with Chiron to create a user interface system configurable via presentation schemas.

The system most closely related to Proteus is Grif [19]. The underlying model of Grif's presentation system is very similar to that put forth here and the syntax of Proteus's presentation languages is based on that of P, the presentation language of Grif [18]. However, Grif's presentation model is closely intertwined with the textual medium. It has a single presentation language and there is no sign that new media can be added to the system by straightforward extension. Extensibility is a key contribution of Proteus. Because Proteus is based on a generic model of presentation and because its implementation is likewise generic, Proteus can be extended to support new formatters relatively easily. Furthermore, because the presentation schema languages for the various media share a large common base, users should find moving from one language to another relatively simple.

# 6 Conclusion

The presentation of information is a key task for a software development environment. An environment must manage and analyze many different objects. A large subset of these objects must be presented to the user from time to time in order to enhance the user's understanding. While other research has not ignored presentation, it has generally linked its presentation services to a particular internal representation such as the text stream or the concrete parse tree.

Our work on Proteus has been motivated by a desire to construct presentation services that are generic and independent of a particular internal representation. In order to achieve this goal, we have based

Proteus on a clear and rational model of presentation that is also generic. Furthermore, because Proteus allows the presentation of an object to be controlled by an external specification, it places control of the appearance of the object in the hands of the user, rather than the implementors of the environment.

We have shown that Proteus can be used to produce both traditional and innovative presentations of formalized language text. We anticipate that Proteus can be used to produce interesting and informative presentations of other types of objects such as call graphs and runtime data structures.

Our research on Proteus is by no means complete. Continuing work on Proteus will allow us to explore the best implementation strategies for presentation systems. Other important areas for research include examination of approaches to incremental constraint maintenance and exploration of how different media interact in complex documents.

# References


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

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

[3] Robert A Ballance, Susan L. Graham, and Michael L. Van De Vanter. The Pan language-based editing system. *ACM Transactions on Software Engineering and Methodology*, 1(1):96–127, January 1992.

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

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

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

[7] 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.

[8] INRIA: Centaur Project, Sophia-Antipolis, France. *The PPML Manual.* For Version 1.1 of Centaur. Undated.

[9] International Standards Organization. *Office Document Architecture*, 1986. Draft International Standard 8813.

[10] Rudolf K. Keller, Mary Cameron, Richard N. Taylor, and Dennis B. Troup. User interface development and software environments: The Chiron-1 system. pages 208–218, Austin, TX, May 1991.

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

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

[13] 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.

[14] Paul Oman and Curtis R. Cook. Typographic style is more than cosmetic. *Communications of the ACM*, 33(5):506–520, May 1990.

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

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

[17] 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.

[18] Vincent Quint. Les langages de Grif. Distributed with the program, July 1988.

[19] 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.

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

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

[22] 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.

[23] 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.*

[24] 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.