Prototype for a Document-Preparation Environment

Richard L. Broussard
B.S., Oregon State University, 1980

A thesis submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Masters of Science
in
Computer Science & Engineering

December 5, 1986

The thesis "Prototype for a Document-Preparation Environment" by Richard L. Broussard has been examined and approved by the following Examination Committee:

Richard Hamlet, Thesis Research Advisor
Professor
Department of Computer Science and Engineering

Robert Babb
Associate Professor
Department of Computer Science and Engineering

Richard Kieburtz
Professor and Chairman
Department of Computer Science and Engineering

Daryl Madura
Floating Point Systems, Inc.

Acknowledgments

I wish to thank my Program committee for taking the time to read and comment on the thesis, and Dr. Hamlet, the committee chairman, for answering questions and proofing the thesis so many times. Thanks is also due to my company, Floating Point Systems, Inc., and especially my managers Daryl Madura and Toby Skinner for being very flexible with the hours I was required to work. And, of course, much thanks is due to Linda, my wife, for being patient and supporting me during my many long evenings of study.

Table of Contents

List of Figures

List of Tables

1. Introduction

A document-preparation environment is a specialized editor for creating and manipulating documents. The document is formatted on a terminal screen as it is being created, and is also checked for many types of errors, ranging from spelling through structural integrity to style. Documents are modeled by some structure, which the user can traverse and manipulate in a variety of ways not available with most current editors.

This thesis both develops ideas and provides a small prototype based on the document-preparation environment suggested by Hamlet [Haml86a]. Three major reasons for prototyping a large software system are: to discover, but not necessarily to solve, major problems; to demonstrate feasibility; and to provide building blocks for the larger system; all without requiring a large time investment. The prototype presented here accomplishes these goals.

An editor is presented which models documents via a graph structure. The editor is divided into three processes: Input, Editor and Display. The Input process reads characters from an input device and sends messages to the Editor. The Editor manipulates the graph and/or associated structures appropriately and sends messages to the Display describing the modifications. The Display determines how best to show the modifications and updates the screen. To avoid confusion, the names Input, Editor and Display will always be capitalized when referring to one of the concurrent processes, and lower case when referring to some generic function.

Since this is a prototype, the user commands implemented are those that demonstrate a particular feature, such as graph manipulation or display. User interface considerations are left for future development. The user is restricted to inserting objects and cursor movement. Legal objects for insertion are characters, paragraphs and text blocks. Legal cursor movements are one character left or right or one line up or down. The format of commands is shown in Appendix A. The command line syntax for running the editor is described in Appendix B.

When the prototype is invoked, it clears the screen and waits for keyboard input. At this point the user must create a structure, either a new block of text or a new paragraph, by typing the appropriate command. The Input process sends an appropriate message to the Editor process, which updates the graph and sends appropriate messages to the Display process, which updates the screen. The cursor is moved to the point on the screen where the structure's text will reside on the screen.

After the structure is created the user can either fill the structure with text by typing characters, or can create a new structure by typing an appropriate command. In either case the Input process sends a message to the Editor, the graph is updated, messages are sent to the Display, and the screen is updated. Several structures can be created and filled with text to form a document.

The position of the cursor on the screen can be changed by one of the cursor movement commands. The cursor will only move to positions on the screen where the user can modify text, and not to positions automatically updated by editor formatting, such as paragraph indenting or vertical spacing after a paragraph title.

When the document has been created the quit command will cause the prototype to cease execution and control to be returned to the operating system.

2. Related Work

Document Environment Characteristics

A study by Shneiderman [Shne83a] suggests the following characteristics of a good document-preparation environment:

(1) show only objects of interest,

(2) manipulate objects rapidly, and

(3) replace complex language syntax with direct manipulation,

where an object is a paragraph, section, chapter, etc.

Previous and Current Work

Document preparation environments can be traced to two major beginnings, text-editors and text-formatters. Both editors and formatters began as simple programs in which little thought was given to ease of use, and which were specific to a task. Eventually they developed into programs that were completely general, meaning that editors could edit any text file and formatters could do almost any type of formatting desired; however, they were not easy to use. Editors and formatters then began to merge, and, in order to provide more intelligence, became more specific again. Currently being researched are entire environments devoted to document-preparation, which are editors that automatically format documents, allow the user to directly manipulate paragraphs, sections, chapters, tables, figures, equations, etc., along with word processing tools such as spelling and style checkers.

Editors

Editors of one sort or another have existed since programs were first entered into a computer. At first editing was a process performed by toggling in a new program, or retyping cards in a card deck, either process being both laborious and manual.

With the advent of card images stored in a computer system, there came a need for a program, called a batch editor, that could manipulate the images. These editors were a step forward from the manual process since the editor provided global search and replace functions. However, the display was to a line printer, and thus manipulation was far from direct.

Teletypes brought about line editors which edited single lines of a file. Although early line editors could only manipulate card images, later editors allowed any line length, thus abstracting the display from the file contents. Text manipulation was thus more direct than previous editors; however, the only object available to the user was a line.

Display screens set the stage for screen editors, which displayed the contents of a file on the screen, for example, visually Vi or vi [Joy80a]. A user could perform operations, such as character insertion, at an active position on the screen, called the "cursor position", resulting a in modified file. These editors were general enough to manipulate any text file. but still only allowed users to manipulate low level objects, such as characters and blocks of text.

This progression from batch to line to screen editors shows the development of Shneiderman's first two characteristics for a good document-preparation environment, namely showing and manipulating only objects of interest. Meyrowitz [Meyr82a] presents a more detailed discussion of editor history.

Extensible editors, such as EMACS [Stal81a], can be tailored to a specific application. Files are included with an extensible editor which define the interface, and these files can themselves be modified by the editor. These editors are very general but have problems when applied to document-preparation, since they do not record and use the structure of the document, thus allowing the user to enter arbitrary text, including errors, while preventing direct manipulation of document objects. Also, these basic editors have no provision for formatting the document.

Formatters

Text formatters have been available since the advent of line printers, although the first formatters were very rudimentary, having no automatic features such as numbering sections or tables, or filling in references. Output of these early formatters went to line printers, which also lacked many capabilities.

Probably the first formatters of any note were RUNOFF and FORMAT [Furu86a], both of which had some rudimentary object support, such as the ability to number sections and format paragraphs. The control over the output format was still very limited, partially due to the formatter and partially due to the output device.

Structured formatters such as TROFF [Ossa76a] with its preprocessors such as Tbl [Lesk79a], were then devised which format many objects automatically. Furthermore, with the advent of devices such as computer-driven typesetters, control of output was greatly improved. TROFF, for instance, can handle preprocessor output to draw pictures and format equations, or tables, and has macros for lists, automatically numbering sections and figures, as well as intelligently placing objects on a page.

The progression from unstructured formatters to structured formatters shows development of object manipulation needed in a good document-formatting environment. Furuta [Furu82a] gives a detailed history of text-formatters.

Current formatters are very general, yet, like editors, are deficient when applied to document-preparation. Most of them require formatting commands to be included in the text, so the file that is edited when modifying a document is hardly readable. Also, the level of control is at such a low level that the user must be an expert to use the system. The TROFF system is an example of a general formatter with document-preparation deficiencies.

Syntax Directed Editors

Syntax-directed editors can be viewed as a combination of screen editor and formatter, which impose editing constraints from a context free grammar, as described by Morris [Morr81a]. Usually these editors are applied to programs in a particular language, although sds [Fras81a] edits any data structure described by a hierarchical grammar, and BIOSTATION [Nana86a] edits biological genetic codes. These editors are easily modified by changing their grammar-description file, and in this way are similar to the extensible editors. Because they are specific to a particular problem, they are in some sense higher-level editors than general purpose editors.

Another advantage enjoyed by syntax directed editors is that pretty printing, or formatting, can be done directly, with neither special formatting commands inserted in the file nor a special parse computation. The editor "knows" the structure and objects making up the file, and allows them to be manipulated directly by the user, and formatted correctly by the editor. Thus editor development has continued towards manipulating objects that the user is interested in.

All syntax directed editors face a basic problem: modification of a syntactically legal program that into another syntactically legal program may require going through stages where the program is not legal. Z [Wood81a] solves the problem by adding some syntax directed features to a standard screen editor, allowing the user to compile the program without leaving the editor, and to read the compiler error output also within the editor. Syned [Horg84a] and SUPPORT [Zelk84a] allow the user to enter a program either by using standard syntax-directed techniques, meaning the user is restricted to "legal" manipulations, or by using standard text editor techniques, meaning that manipulations are unrestricted, after which the changed text is reparsed, or some combination of the two. Thus arbitrary text can be entered, and is syntactically checked later.

Document Editors

Document editors attempt to apply techniques from syntax directed editors to structures that are not as rigid as programming languages, i.e., textual documents. With these editors, the user generally gets immediate feedback as to how the document looks, and can manipulate the document's objects directly. None of these editors are complete document environments—they all lack breadth—but each shows techniques that could be used to develop such an environment.

An early editor of this type was JANUS [Cham81a]. JANUS represents an intermediate step between a general purpose editor with a batch formatter and a true document editor. It runs on a work-station with two screens, and requires the user to place formatting commands in a file, similar to the general editor/batch formatter combination, but displays the formatted text on the second screen.

The next step towards a document editor is found in PEN [Alle81a]. PEN is mostly aimed at editing mathematical documents, but can be used for other purposes as well. Documents have a hierarchical model, where nodes are "chapters," "sections," "paragraphs," etc. Nodes and their attributes are defined through a template, which can be user modified. Both formatting and modification attributes are associated with each node, determining how information in the node is to be formatted and how the node can be modified. For instance, if a chapter can only have sections and paragraphs, then trying to add a chapter to a chapter would not be allowed. One major drawback of PEN is that the templates are written in a difficult language, requiring modification by an expert.

Kimura's paper [Kimu84a] has a good abstract model for documents along with a prototype that uses the model to modify documents. Many of the following editors are based on this model. Kimura models documents as a hierarchy of objects which can be combined to make other objects. Each object contains a name, which is something similar to a type; data, the nonstructural information associated with it; and composition, a list of other objects that make up the object. If the name is an atomic object then the composition is empty. Thus a section would be represented by an object where the name is the string "Section", the data is the section title, and the composition is the list of paragraphs and sections making up the section. Nonhierarchical structures, cross references, for example, are represented by explicit threads through the data structure.

Other editors that use this model are those by Cowan [Cowa86a], Coray [Cora86a] and Quint [Quin86a]. Each of these editors takes as input more than just the user keyboard entries; each also reads some sort of structural definition, called the syntactic specification, and a formatting definition, called the semantics specification. Text from the keyboard is formatted and displayed as it is typed. All three of these editors also recognize that there are parts of a document that cannot be represented by a simple hierarchy, and thus provide explicit linking threads. The major pitfall of these systems is that they are hard to modify. Each requires an expert user to set up the syntactic and semantic files, since both use a rather obscure description language, although Coray [Cora86a] does provide for user definable parameters in these two descriptions.

Grif [Quin86a], Star [Furu82a,Meyr82a] and MacWrite [John84a], are examples of What-You-See-Is-What-You-Get editors, meaning that the text is displayed on the terminal exactly as it would be printed. These editors require a graphics screen to display the document (because of tables, figures, and special characters), which is both an advantage and a disadvantage. Since the user can edit an exact representation of what will be printed, the user is apt to make fewer errors, thus the advantage. However, graphics screens are not universally available, and hence the disadvantage.

Furuta [Furu86a] presents a What-You-See-Is-Almost-What-You-Get, meaning that the text displayed on a screen is fairly close to what will be printed except for differences which will occur where the graphics capabilities of the printer output device exceed that of the terminal. Thus, enough information is displayed on the terminal screen to manipulate the document, without requiring a graphics display as does Grif. The editor is basically hierarchically structured, but has provisions for unstructured levels, which aids in storing structures that are not strictly hierarchical, such as tables, where the entries have multiple parents.

The W editor [King86a] is the most complete document editor of any described here. It also uses a hierarchy to model the document, but can handle text, mathematical formulae, and graphics. The syntactic and semantic structure can be modified by the editor without requiring the skill of an expert user. Instead of using description languages, the descriptions are represented within the editor as structured documents, where changing the structure changes the description.

One major downfall of Kimura's document model and the editors based on it is its hierarchical nature. Some views of a document are indeed hierarchical, the table of contents, for instance, while other views are not, for example, the cross references and bibliography. Kimura gets around these by introducing explicit threads through the document tree. Another, more general solution would be to modify the document model to be a graph; references and bibliographies could then use the same structure as the table of contents hierarchy.

Document Systems

A document system is one that stores relations between documents, and allows one to follow, and perhaps modify, these relations. They are of interest both because they use general graphs to store the arbitrary relations, and because they present more of an environment than an editor.

CONCEPT BROWSER [Cord86a] allows a user to interactively create and move through a set of dynamic "documents." These "documents" could be as simple as single objects, such as a paragraph, section, or chapter, or could be entire papers. Thus this system could be used to create a single document, modeled as a graph, or many papers related by bibliography or subject. Relations are stored as arcs of a graph, where nodes are the documents. The complication of graph traversal, however, is side-stepped by the fact that only one node and its associated arcs are displayed on the screen at any one time. Traversal decisions are left to the user, who must specify which arc to follow, and thus this browser is awkward to use in creating a document.

TEXTNET [Trig83a] is a system similar to CONCEPT BROWSER. In TEXTNET, relations between scientific documents are stored via a general graph. Provision is made for making critiques of a paper, and allowing the author to critique the critiques, ad infinitum. The similarity to CONCEPT BROWSER is that the documents can be single objects, and thus the system can be used to create new documents. It is superior in that it can automatically traverse hierarchical subgraphs; however, its fixed set of relational types make it overly restrictive.

Constraint Editors

Constraint editors are editors written using a constraint language. ThingLab [Born81a] is a constraint editor biased towards manipulation of graphics objects. The user edits objects onto a screen, constrains the objects to certain properties, and can then manipulate the objects. The editor ensures that the constraints are always met.

These editors are still in their infancy and have not yet been extensively researched. Their relation to document editing is as yet unclear. One can imagine constraining nodes of the document graph to places on a screen and having the screen be "automatically" updated on succeeding changes to the node. It is a matter of research to see if, and how, this could be accomplished.

Document Environments

A variety of Document environments have recently been proposed. The work described in this thesis is an example of a prototype document environment.

Walker [Walk81a] discusses ideas for a document environment based on a hierarchical document model. The structural restrictions and automatic formatting of a document editor are combined with other tools to form an environment. Many tools are suggested which are grouped into three categories: writing aids, such as spelling and style checkers, structure editing aids, such as deleting or inserting an object, and document management aids, such as cross references and indexing. The environment consists of a top-level document editor which invokes these tools as needed. The many tools available make this a document environment, although it has problems similar to document editors since it is based on a hierarchical model.

Another type of document environment is suggested by Hamlet [Haml86a] who presents a document environment combining a document editor with a document system. Also included would be other tools: a spelling checker, style writing aids, etc. The environment would consist of several editors, of which this thesis prototypes the Input editor. This Input editor would function much like a document editor, incorporating localized writing aids such as spelling checks, where the document is modeled as a graph rather than a hierarchy.

3. Prototype Description

3.1. Specifications

This thesis prototypes the Input editor described in [Haml86a]. The specifications are to

(1) use a graph rather than a hierarchal structure to model documents.

(2) have simple, consistent data structures in order to reduce complexity, and keep the number of manipulative functions to a minimum.

(3) be small, so only a few different document objects are legal: a paragraph and a text block; and only one user function is available, namely insert.

(4) use coding techniques that will make it expandable, since one use of a prototype is to have something to modify and expand at a later time.

3.2. Coding Techniques

Since the editor described in this thesis is a prototype, it should be both easy to modify and should provide building blocks easily extracted from the prototype environment. Two coding techniques used to provide these features are data abstraction and concurrent processes. There are also internal debugging aids.

3.2.1. Data Abstraction

Data abstraction [Parn72a, Gutt78a] is a technique whereby the representation of a data structure is abstracted from most of the code. A data structure is represented by both values and a set of operations performed on them. The conventional technique for coding data structures is to explicitly declare the memory configuration for the structure, but to leave the conceptual operations implicit in the code. However, if all conceptual operations are explicit functions, and the only manipulation of the data structure is by these functions, then the method used to store the data structure can change completely and require changing only the functions.

One inherent difficulty with data abstraction is that performance may be decreased, since a function invocation must be used to reference an item of a structure, as opposed to the usual in-line reference. However, flexibility is more important than performance for a prototype, so data abstraction techniques were used extensively.

3.2.2. Concurrent Processes

Dividing the overall editor into concurrent processes is a prototyping technique with these benefits: very clear interfaces between the high level functions, ease of future modifications or additions, and a more disciplined coding environment. This technique is also used in Coray's document editor [Cora86a] and King's W editor [King86a], the three processes of this editor being similar to those of King.

Each function is clearly delineated, being a separate process; each takes a very limited set of inputs and generates a very limited set of outputs. Further, errors in one function do not propagate to other functions.

Another reason for coding with concurrent processes is the ease of modifying the editor to use a very different input device, like a terminal with a mouse, or another output device, like a laser printer. A new Input process, Display process, or Editor process for that matter can be written without having to rewrite any of the other processes, as long as the new process preserves the interfaces.

The coding environment is more disciplined with concurrent processes than with subroutines because each process has a separate state, as opposed to subroutines which share states. Since all communication between processes must be done by means of messages, the information exchanged tends to be much smaller. Subroutine state sharing tends to lead to exchanging, perhaps implicitly, large quantities of information. Also, there tend to be fewer assumptions made about how and when information is processed, resulting in cleaner interfaces; a concurrent system is viewed as less coupled than a subroutine system. Thus the interface between processes is usually cleaner than between subroutines.

There are drawbacks with using concurrent processes. Extra software must be written for message passing. Performance may be degraded since message passing is usually slower than parameter passing. Also, when using subroutines a large data structure can be shared, whereas with concurrent processes two (or more) processes have to each keep a copy of the structure. Hence processing may be duplicated between processes.

3.2.3. Debugging Aids

There are three debugging aids written into the editor: tracing, parameter validation and conditional compilation.

Each process has a set of global trace values defining the trace file and items to be traced at run-time. All trace messages are written to the trace file. Traceable items are:

(1) the call/return sequence of all functions

(2) the values of formal parameters for all functions

(3) the messages sent/received by/for the process, along with the time of day.

Trace values are selected by the user from the command line; see Appendix B for the manual page.

Where appropriate, formal parameters are validated and error messages written to the trace file if the parameters are invalid. Pointers are checked for NULL if they must be pointing to an object, integers are checked to make sure they fall within a possible range.

Conditional compilation is used with the trace statements in the low level routines to cut down on the amount of trace output.

4. Data Structures

4.1. Message Structure

There are six components of a message:

(1) the sending process,

(2) the intended recipient process,

(3) an identifier for the message,

(4) a reference identifier,

(5) a command,

(6) and parameter values associated with the command.

The first two components, the sending and receiving process, are used for message tracing while debugging. Furthermore, the receiving process component is used to ensure a message is not sent to the wrong process. If by some mishap a process does receive a message not intended for it, then an error is printed to the trace file, but the message is still processed.

The third and fourth components, the message and reference identifiers, are used to uniquely identify the message and to respond to a message, respectively. When a message is created it is given a unique identifier. If a message, call it 'A', is in direct response to another message, call it 'B', then the reference identifier of 'A' is the message identifier of 'B'.

The fifth and sixth components, the command and parameter values, tell what action is to be taken and give any extra information, respectively. For instance, if a command to the Editor is to insert a new character, then the parameter value would be the new character.

Functions needed to manipulate messages are:

(1) Make or remove a message

(2) Get a unique message identifier

(3) Get a component of a message

(4) Read a message

(5) Write a message

Messages are implemented as a data structure with the six fields shown in Table 4.1. To facilitate sending, receiving and tracing messages all fields are character strings. The *From* and *To* fields represent the names of processes, i.e., "Input",

Field Name	Description
From	Process sending message
To	Intended recipient process
Id	Unique message identifier
Reference Id	Reference identifier
Command	Action request
Text	Parameter values

Table 4.1: Message Fields

"Editor", or "Display". *Id* and *Reference Id* are time-date stamps, which are unique if there is a fine enough timing division. *Command* is the command, for example "InsertChar" for *insert character*. *Text* is a character string containing the parameter values.

It was decided early in the design of the prototype that messages should have identifiers, and that some messages would be in direct response to a query message, hence the *Id* and *Reference Id* fields. However, as the coding progressed it became apparent that there would be no query messages, so the identifier fields are never used to process a message. They are left in for future development.

To transfer a message the sending process concatenates all fields into one character string, adding a byte count at the beginning of the string and special characters at the field boundaries. The new string is written to the receiving process, which reads the message byte by byte, first decoding a byte count, then reading the rest of the characters. Transfer is complete when the receiving process has parsed the string into a message structure.

Figure 4.1 shows the function `GetMesId` that gets the identifier of a message. This function takes a pointer to a message structure and returns a character string pointer to the message's identifier. It demonstrates both the simplicity of the message functions, and the volume of code added because of the debugging techniques.

```
char *GetMesId (Mes)
   MESSAGE *Mes;                    /* Message to be read */
{
   /*  Local variables  */
   char    *Results;                /* Function return value */

#ifdef DEBUG
   if (TraceCall)
      fprintf (TraceFd, "GetMesId\n");
#endif

   Results = NULL;
   if (Mes == NULL) {
      fprintf (TraceFd, "Error GetMesId\n");
      fprintf (TraceFd, "     Message pointer is NULL\n");
   }
   else
      Results = Mes->Id;

#ifdef DEBUG
   if (TraceCall)
      fprintf (TraceFd, "GetMesId returning\n");
#endif

   return (Results);
}
```

Figure 4.1: Get Message Identifier Example

4.2. Graph Structure

The graph structure is based on Kimura's document model, modified from a hierarchy to a general graph. Since this is a prototype, one of the criteria for the graph is simplicity, meaning that it should comprise only a few node types. If every node of the graph has the same structure, then a few simple functions can perform all graph manipulations.

There are four components of each graph node:

(1) the document object represented by the node,

(2) data associated with the object,

(3) list of incoming arcs,

(4) list of outgoing arcs.

Each node represents an object of a document, e.g., a section, each of which has data associated with it, e.g., a title.

The conceptual operations for the graph are:

(1) Make or Remove a node

(2) Get or Change a component of a node

The abstract graph above is implemented as a data structure with the four fields shown in Table 4.2. The *Id* field is a unique integer identifier used to validate graph references. The *Type* field is a character string for the document object

Field Name	Description
Id	Unique Identifier
Type	Object
Data	Associated Data
Backward	List of Incoming Arcs
Forward	List of Outgoing Arcs

Table 4.2: Graph Node Information

represented by the node. The only objects for this prototype are "Head", "Paragraph" and "Text", where "Head" is an object that does not have any incoming arcs and is thus the first node in the graph, "Paragraph" represents a paragraph, and "Text" represents a block of text, constrained to only be part of a paragraph node. The *Data* field is a character string storing the objects associated data. For a type "Head" node there is presently no data. *Backward* and *Forward* fields are the incoming and outgoing arcs, respectively. They are pointers to ordered lists of arc nodes, where an arc node consists of a pointer to a graph node and a pointer to the next arc in the list, see Table 4.3. The ordering is necessary since the user must be able to determine whether a given paragraph precedes or succeeds another paragraph.

Graph traversal is a complex problem. It was decided early in the coding of this prototype not to pay much attention to graph traversal, but rather to develop message passing and display; hence graph traversal was ignored. The graph is assumed to be an n-ary tree where the issue of graph traversal appears in the code, since the commands available to the user can only be used to create a n-ary tree. Documentation has been provided where this simplification affects the code.

Field Name	Description
Next	Next Arc Node
Node	Associated Graph Node

Table 4.3: Arc Node Information

Figure 4.2 shows the graph structure of a document that has two paragraphs, named "Parag 1" and "Parag 2", and "Parag 1" has a block of text, "The cat and the dog".

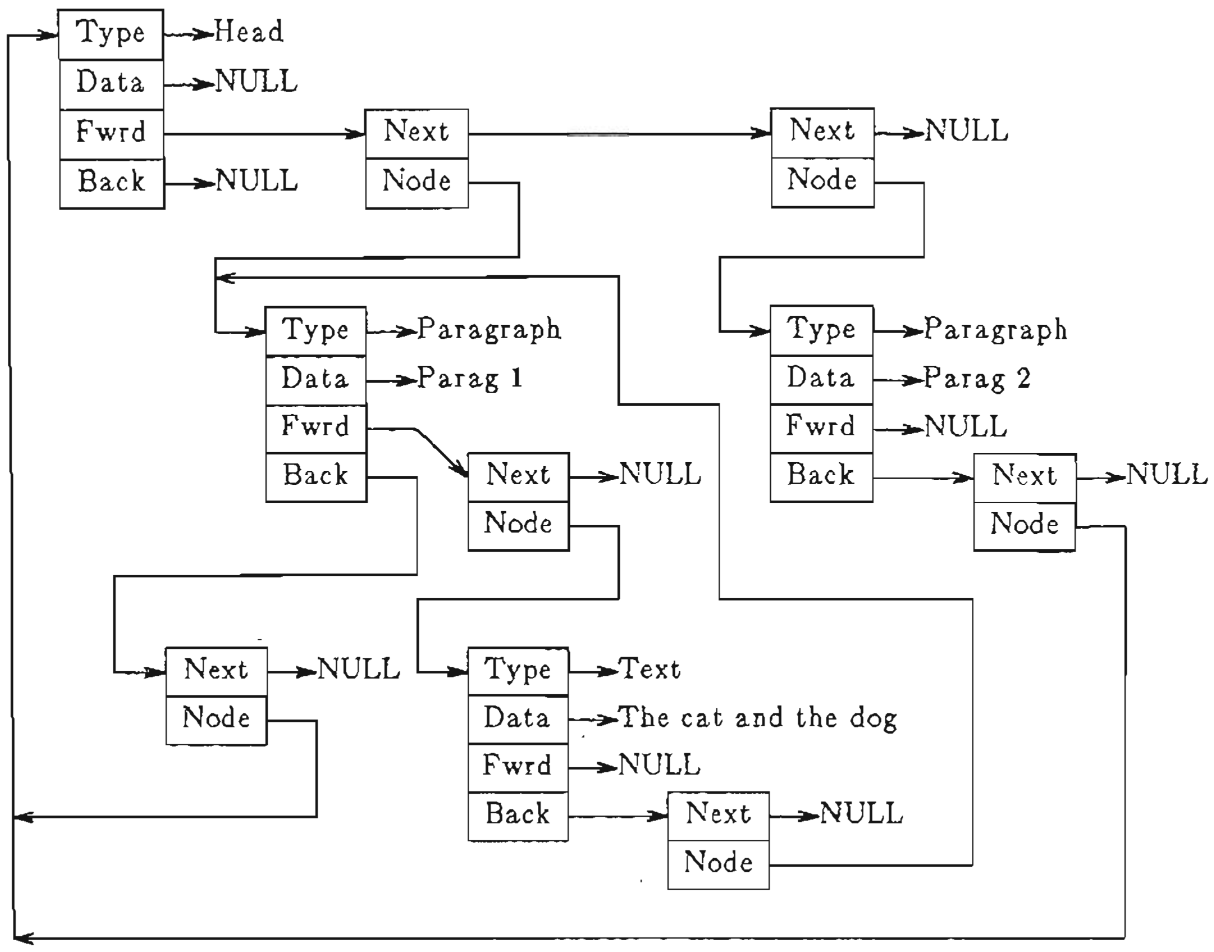

Figure 4.2: Graph Example

In order to validate graph node pointers, all node pointers external to the graph operations do not point directly to a graph node, but rather to a wrapper which both points to the graph node, and contains a copy of the graph node's Id. If the wrapper's Id and the corresponding graph node's Id are not equal, then the pointer is invalid. This technique is used to catch several types of internal errors when a graph operation is invoked:

(1) A pointer is passed to the operation that does not point to a wrapper, for example, it has been overwritten. In this case the validation is not foolproof, but there is a very low probability that the Id fields will match.

(2) The pointer points to a wrapper that points to a deleted graph node. In this case the Id fields will not match, since the operation that removes a graph node sets the Id field to an illegal Id.

Figure 4.3 shows the function `GetNdData` that gets a graph node's data. This function takes a graph node and returns the associated data if the node is valid. A NULL is returned if the node is invalid so that the caller can check for errors. This is not a foolproof error check, since the node's data field may be NULL. Note that double indirection is needed since the parameter is a pointer to a wrapper, which in turn points to the actual graph node.

Figure 4.4 shows the function `ChgNdData` that modifies the data of an existing graph node. This function takes a graph node and modifies its associated data field if the node is a valid graph node. This function returns a NULL if and only if the node pointer is invalid. The caller can use this feature to check for

```
char *GetNdData (Node)
   GRAPHNODE *Node;                    /*  Node to process   */
{
   /* Local variables */
   char *Results;                      /*  The value passed back  */

   /* External functions */
   int  PrtGphNd();                    /*  Print a graph node   */
   int  ValidateNd();                  /*  Validate a graph node  */

#ifdef DEBUG
   if (TraceCall)
      fprintf (TraceFd, "GetNdData\n");
   if (TraceParm) {
      fprintf (TraceFd, "GetNdData Node:\n");
      PrtGphNd (Node);
   };
#endif

   /*  First check the node for validity  */
   Results = NULL;
   if (!ValidateNd (Node)) {
      fprintf (TraceFd, "Error GetNdData\n");
      fprintf (TraceFd, "     Invalid node\n");
   }
   else {
      /*  Now get the data  */
      Results = Node->GNode->Data;
   };

#ifdef DEBUG
   if (TraceCall)
      fprintf (TraceFd, "GetNdData returning\n");
#endif
   return (Results);
}
```

Figure 4.3: Get Data From Graph Node

```
GRAPHNODE *ChgNdData (Node, Data)
   GRAPHNODE *Node;               /*  Node to change types   */
   char *Data;                    /*  Data to change it to   */
{
   /* Local variables */
   GRAPHNODE *Results;            /*  The value passed back   */

   /* External functions */
   int  PrtGphNd();               /*  Print a graph node   */
   int  ValidateNd();             /*  Validate a graph node   */

#ifdef DEBUG
   if (TraceCall)
      fprintf (TraceFd, "ChgNdData\n");
   if (TraceParm) {
      fprintf (TraceFd, "ChgNdData Node:\n");
      PrtGphNd (Node);
      fprintf (TraceFd, "ChgNdData Data: %s\n", Data);
   };
#endif

   /*  First check the node for validity  */
   Results = NULL;
   if (!ValidateNd (Node)) {
      fprintf (TraceFd, "Error ChgNdData\n");
      fprintf (TraceFd, "     Invalid node\n");
   }
   else {
      /*  Now make the change  */
      Results = Node;
      Node->GNode->Data = Data;
   };

#ifdef DEBUG
   if (TraceCall)
      fprintf (TraceFd, "ChgNdData returning\n");
#endif
   return (Results);
}
```

Figure 4.4: Change Data of a Graph Node

errors.

4.3. Virtual Screen Structure

Part of the design constraint of this prototype is to relegate all screen decisions to the Display process, and all graph information to the Editor process. As the graph is manipulated by the Editor process, it also must be displayed by the Display process on a terminal screen, with perhaps several different views on the screen at the same time. A view might be the entire graph, or a table of contents (the viewed graph to a particular level), or only nodes directly related to some given node (only its outgoing arcs). The two processes need some common ground for communication, having the properties of the Editor being able to ignore the details of the output device, and the Display not having to know anything about the graph. The concept of a Virtual Screen (VS) was created and implemented to facilitate information exchange between the Editor and Display processes.

A VS contains information about the graph's data, about formatting the graph's data and about the ordering of the graph's data. The VS is associated with both a view of the graph and a window on the screen, and thus there could be several simultaneous VS's. VS's are linked together to form a list, the ordering of the list determining the overlap of the windows. The Editor determines what information goes into, or should be taken out of, a VS by examining the graph, and sends that information to the Display, without knowing whether or not the information will appear on the screen. The Display determines what information from the VS to display on the screen, and how to display it based on the formatting information and the window on the screen, without knowing the graph structure. For the most part, the Editor and Display have the same representation for a VS. This VS concept works well at keeping the Editor and Display separate, but the two processes end up performing the same processing in many cases.

There are four components comprising a VS:

(1) a unique identifier,

(2) a list of Virtual Lines, (VLs),

(3) the preceding VS, and

(4) the succeeding VS.

There is one additional component used by the Display process:

(5) information to map the VS to the screen. This mapping information contains a window pointer, the location in the VS appearing in the upper left hand corner of the window, and the row and column size of the window.

The identifier is set by the Editor process, and is used to determine the active VS, i.e., the VS being modified. The VL list represents the data from the graph in a linear order. The preceding and succeeding VS components form the list of VS's. Mapping information is needed by the Display process to map the VS to a window on the screen.

Each VL represents one node of the graph, and is a structure with two components:

(1) data from the graph

(2) a list of physical line breaks and formatting information

Data from the graph is a direct graph reference for the Editor process, but it is a copy of the data in the Display process. Since one graph object may map to several lines on the screen, and each line may be formatted differently, line break and formatting information associated with the lines must be stored. (A text block for example, if occurring after a paragraph, will have the first line indented five spaces and succeeding lines along the left margin).

The VS operations common to the Editor and Display process are:

(1) Create a new VS,

(2) Get and Set the Id of a VS,

(3) Get the list of VL lines, and

(4) Link VS structures together.

In addition to these the Editor uses:

(5) Create a new Id for a VS and

(6) Modify a VS to reflect addition of new graph nodes.

The Display needs in addition:

(5) Get and Set the mapping information of a VS.

(6) Set the VL list for a VS.

Further, both processes need operations that modify information in a VL.

An abstract VS is implemented as a data structure with the five fields shown in Table 4.4. *Id* is the identifier for the VS, *Lines* is a pointer to the list of VLs, *Map* is a pointer to the screen mapping information, and *Next* and *Prev* form the doubly linked list. The `Map` field appears only in the Display's copy of the VS.

Figure 4.5 shows the Editor code that links a VS into an existing VS list as an example of the error checking in the VS operations. This function, LinkVS, prepends a VS element to an existing list of VS's. The VS list may be NULL, an empty list, or may be a pointer to the middle of a list. `LinkVS` begins by checking the input parameters for errors, but only those that would cause its abnormal termination, i.e., a NULL pointer reference to a VS. After error checking LinkVS does the link. A VS list is a doubly linked list of VS elements, and so this function does a simple insertion of an element at the front. Note that the VS list pointer may be NULL or it may be the first element in a list, both special cases.

Field Name	Description
Id	Unique VS identifier
Lines	VL list
Map	Screen mapping
Next	Succeeding VS
Prev	Preceding VS

Table 4.4: VS Fields

```
VSLIST *LinkVS (VS1, VS2)
   VSLIST *VS1;                   /*  The first VS node  */
   VSLIST *VS2;                   /*  The second VS node  */
{
   if (TraceCall)
      fprintf (TraceFd, "LinkVS\n");

   /*  Check the inputs  */
   if (VS1 == NULL) {
      fprintf (TraceFd, "Error LinkVS\n");
      fprintf (TraceFd, "      First VS pointer is NULL\n");
   }
   else {
      if (VS2 == NULL) {
         VS1->Next = NULL;
         VS1->Prev = NULL;
      }
      else {
         VS1->Next = VS2;
         VS1->Prev = VS2->Prev;
         if (VS2->Prev != NULL)
            VS2->Prev->Next = VS1;
         VS2->Prev = VS1;
      };
   };

   if (TraceCall)
      fprintf (TraceFd, "LinkVS returning\n");

   return (VS1);

}
```

Figure 4.5: Link Virtual Screens

This prototype does not fully implement multiple VS's. The user commands do not allow the creation or manipulation of windows on the screen. Only d uring initialization is a new VS created, and then the Display process opens the entire screen as one window.

4.4. Cursor Structure

A cursor is a complicated mechanism in this editor because of the three structures it needs to reference: VS's, the graph and the screen. Furthermore, if graph traversal was fully implemented, one node of the graph may occur in several VS structures, and several times on the screen. Thus multiple cursors are needed, although only one cursor is visible on the screen.

The cursor is slightly different between the Editor and Display processes. Components that are the same are:

(1) a VS,

(2) a VL,

(3) an offset into the graph data of the VL,

(4) a physical line in the VL, and

(5) an offset into the physical line.

The Editor process also uses:

(6) a graph node.

The Display process adds:

(6) VS row and column, and

(7) row and column in the screen window.

A cursor includes its associated VS, the VL in that VS, and the physical line in that VL, so that when characters are inserted the line break information can be updated. It further includes an offset into the graph node data so that when information is inserted the graph node data can be modified. Cursors in the Editor process must have an associated graph node so that the graph can be modified when document objects are inserted at the cursor location. Cursors in the Display process must be mapped to the screen, hence they need row and column components.

Operations needed for cursors also differ between the Editor and Display processes. Operations that are the same are:

(1) make and remove a new cursor, and

(2) get and set a component.

Additional Editor operations are:

(3) set the associated graph node.

(4) generate a list of cursor positions

Additional Display operations are:

(3) set the position of the visible (active) cursor,

(4) write a character to the screen at the active cursor position.

In the Editor process a cursor is stored as a data structure with the three fields shown in Table 4.5. Since this prototype allows the user to create only one VS, all cursors are assumed to reference this VS.

Field Name	Description
VL	Associated VL
Physical	Physical line in the *VL*
Offset	Offset into the physical line

Table 4.5: Editor Cursor Fields

In the Display process a cursor is stored as a data structure with the three fields shown in Table 4.6.

Field Name	Description
VS	Associated VS
VL	VL in the *VS*
Offset	Offset into the *VL's* graph data

Table 4.6: Display Cursor Fields

5. Processes

There are three processes, Input, Editor and Display, making up this prototype.

5.1. Input

The Input process reads characters from the input device, recognizes commands by means of a table driven Discrete Finite Automaton (DFA), and sends an appropriate command to the Editor process. See Appendix A for a list of commands recognized by the Input process. An error message is sent to the Editor if a character results in an unrecognized command.

5.2. Editor

The Editor process is responsible for responding to messages from the Input process, maintaining the graph and a copy of each VS, and sending messages to the Display process.

5.2.1. Messages

Table 5.1 lists legal values of the *Command* field of a message for the Editor process. The first nine commands correspond directly a sequence recognized by the Input process. *NewVS* is sent by the Input process during initialization to create the VS. The Input process sends *Message* when there is an unrecognized input sequence,

Command	Description
CursUp	Move the cursor up one line
CursDn	Move the cursor down one line
CursLf	Move the cursor left one character
CursRt	Move the cursor right one character
InsertChar	Insert a character at the current cursor position
NewLine	Create a new line at the current cursor position
NewChPar	Create a new paragraph as a child to the current structure
NewSbPar	Create a new paragraph as a sibling to the current structure
NewText	Create a new text block
NewVS	Create a new Virtual Screen
Message	Put an informative message on the screen
Quit	Cease execution

Table 5.1: Editor Process Commands

and *Quit* for either the quit or abort sequence.

5.2.2. Graph Usage

The Editor's primary purpose is to manipulate the graph structure described in section 4.2. As an example, Figure 5.1 is a projected design that uses graph operations to insert a new character at the current cursor location. The function `EdInsChar` inserts a character at the current cursor position only if the corresponding graph node can have a character added to its data field. The character is inserted in the graph node's data field, a message is sent to the Display process informing it of the change and the cursor position in updated. The Display process may or may not change the screen, depending on what part of the graph is currently

```
EdInsChar (Chr, ToDisplay)
   char Chr;
   int  ToDisplay;
{
   Get the cursor's graph node.
   if (the graph node can accept more characters) {
      Get the graph node's data.
      Get the cursor offset in the graph node's data.
      Insert Chr into the graph node's data at the offset.
      Write "InsertChar" message to ToDisplay.
      Change the graph node to have the new data.
      Update the cursor to just after the inserted character.
      Write "CursorOffset" message to ToDisplay.
   };
}
```

Figure 5.1: Graph Operations Example

being displayed.

5.2.3. VS Usage

Corresponding to every graph manipulation the Editor process must also modify its version of the VS and send appropriate messages to the Display process. As an example, Figure 5.2 shows the design for the Editor code which uses the VS operations to create and initialize a new VS. This function, EdNewVS, creats a new VS for the Editor process, links it in with the list of existing VS's, tells the Display process about the new VS and initializes the VS with data.

```
EdNewVS (ToDisplay)
   int ToDisplay;
{
   Create the new VS
   Link the new VS into the existing list of VS's
   Get an Id for the VS
   Set the VS to the new Id
   Write a "NewVS" message to ToDisplay
   Traverse the graph
      For each graph node not represented by a VL in the VS
         Insert a new VL into the VS
         Write a "NewVL" message to ToDisplay
   Set cursor to the VL that represents the "Head" graph node
   Write a "CursorList" message to ToDisplay
}
```

Figure 5.2: Editor VS Operations Example

5.3. Display

The Display process responds to messages sent by the Editor process, updating its VS and the screen accordingly.

5.3.1. Messages

Table 5.2 lists legal values of the *Command* field of a message for the Display process. The first five commands have little correspondence to an Editor command, which is to be expected since the Editor maps abstract manipulations to more concrete ones. The Editor process sends a *Message* command to write a message to the bottom of a VS window. *NewVS* is sent to create a new VS in the Display process. The *Quit* command causes the Display process to cease execution.

Command	Description
CursorList	Change the list of cursors
CursorOffset	Update the cursor offset
InsertChar	Insert a character at the current cursor position
NewVL	Create a new Virtual Line
ReplaceFmt	Replace the format of a Virtual Line
Message	Put an informative message on the screen
NewVS	Create a new Virtual Screen
Quit	Cease execution

Table 5.2: Display Process Commands

5.3.2. VS Usage

The primary purpose of the Display process is to manipulate the screen and its VS's. As an example, Figure 5.3 shows the design for the Display code which uses the VS operations to create and initialize a new VS. This function, DisNewVS,

```
DisNewVS (Id)
   char *Id;
{
   Create a new window on the screen
   Create a new mapping structure and set appropriate information
   Create the new VS
   Set new VS identifier to Id
   Set new VS mapping
   Empty out VL list for the VS
   Link the new VS with the current VS list
}
```

Figure 5.3: Display VS Operations Example

creats a new VS for the Display process, sets up mapping information to a window on the screen, and links it in with the list of existing VS's.

5.4. Example

Consider the process interaction when a user types the character Figure 5.4 shows a pictorial description of the interaction. The Input process reads this character and sees it as a legal pattern and sends the *InsChar* command, where 'x' is the text field of the message, to the Editor process.

The Editor process receives the message from the Input process and decodes it. The cursor position is calculated within the current VS and the associated graph node is found. If the graph node can have a character inserted then the character 'x' is inserted into the character string of the node's data, and a *InsChar* message is sent to the Display process informing it of the modification. The cursor position is updated and Display informed via a *CursOff* message. If the graph node cannot

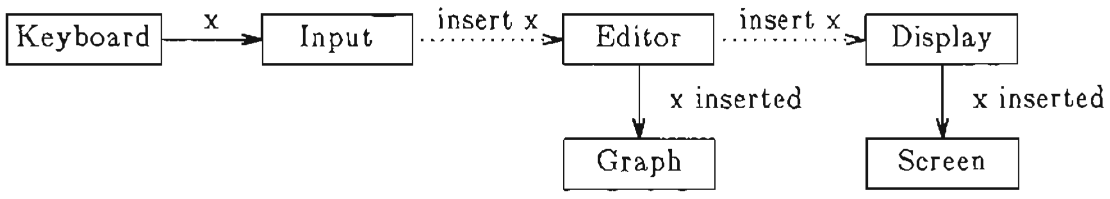

Figure 5.4: Insert Example

have a character inserted then an error is reported via a *Message* command to the Display process.

The Display process receives one of two messages from the Editor. If *Message* is received the text of the message is displayed on the screen. If *InsChar* is received then the character 'x' is picked out of the text part of the message and is inserted into the current VS at the current cursor location. The cursor position is examined, and if it lies within the VS's screen window then 'x' is also inserted on the screen. The Display process receives the succeeding *CursOff*, parses the new cursor coordinates from the text part of the message, and sets the cursor to the new coordinates.

6. Conclusion

This prototype demonstrates weaknesses and strengths of major design decisions as well as pointing out areas of future development.

It is questionable whether concurrent processes were an advantage in this prototype since they required the development of message passing software, slowed down execution, and complicated the mapping of graph to screen. The first two disadvantages were expected, and were expected to be offset by the more disciplined coding environment. However, mapping the graph to the screen required developing the VS, which greatly complicated implementation of the cursor. Furthermore, VS Editor processing had to be duplicated in the Display. VS and cursor processing is significant, accounting for almost two-thirds the total code. Perhaps a better approach would have been to use subroutines in this prototype, switching to concurrent processes in some future development when the code is better understood.

Data abstraction and the debugging aids greatly increased the amount of code. To implement data abstraction every reference to a data structure required a function invocation. The function would normally consist of a single line of code, but, because of the debugging aids, might require ten or twelve lines. However, data abstraction did provide clean interfaces and allow easy modification of data structures, and the debugging aids were useful when debugging the code. It seems that these techniques are useful, but not without their costs.

Graph traversal is a major problem that must to be solved before a robust system can be built. Traversal problems appeared whenever traversal had to stop and the position be saved for future traversal. Cursor position and graph display are two instances. If the document represented by the graph is to be displayed on the screen and printed on a printer in the traditional linear manner, then some linear ordering of the graph's nodes is required. Thus the graph must be traversed.

When traversing a cyclic graph each node traversed must somehow be marked, so that future traversal paths through the node can be halted before cyclic behavior begins. Explicit marking seems undesirable, since the graph may need to be traversed several times simultaneously. Displaying the document on the screen and searching for an occurrence of a string are examples of simultaneous traversals. Creating a list of nodes visited and storing this list as the current place in the graph works well until a node of the graph is deleted, requiring all lists to be recreated.

It could be better to revert to a hierarchal structure with labeled arcs. A hierarchal structure would eliminate the graph traversal problem, while labeled arcs would allow the document to be structured many different ways. The document-environment would have to keep track of which hierarchy was being traversed.

Along with pointing out problems, this prototype also shows some areas of future development, many of which could be taken in parallel. One area is to find a solution to the graph traversal problem stated above. A second path is to clean up the display and cursor problems.

Further, more interesting, expansion of the following editor capabilities could occur:

(1) deleting objects,

(2) retaining information on disk,

(3) to allow multiple simultaneous views the graph,

(4) supporting more objects, such as sections and chapters, which introduces problems with automatic numbering,

(5) improving the user interface.

References

Alle81a
Allen, T., Nix, R., and Perlis, A., PEN: A Hierarchical Document Editor, *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, **16**, 6 (June 1981), 74-81.

Born81a
Borning, A., The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory, *ACM Transactions on Programming Languages and Systems*, **3**, 4 (October 1981), 353-387.

Cham81a
Chamberlin, D., King, J., Slutz, D., Todd, S., and Wade, B., JANUS: An Interactive System for Document Composition, *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, **16**, 6 (June 1981), 82-91.

Cora86a
Coray, G., Ingold, R., and Vanoirbeek, C., Formatting Structure Documents: Batch versus Interactive?, pp. 154-170 in *Text Processing and Document Manipulation*, ed. J. C. van Vliet, Cambridge University Press, Cambridge, April 1986.

Cord86a
Corda, U. and Facchetti, G., CONCEPT BROWSER: a System for Interactive Creation of Dynamic Documentation, pp. 233-245 in *Text Processing and Document Manipulation*, ed. J. C. van Vliet, Cambridge University Press, Cambridge, April 1986.

Cowa86a
Cowan, D. D. and Smit, G. de V., Combining Interactive Document Editing with Batch Document Formatting, pp. 140-153 in *Text Processing and Document Manipulation*, ed. J. C. van Vliet, Cambridge University Press, Cambridge, April 1986.

Fras81a
Fraser, C., Syntax-Directed Editing of General Data Structures, *Proceedings of*

the ACM SIGPLAN SIGOA Symposium on Text Manipulation, **16**, 6 (June 1981), 17-21.

Furu82a
A Furuta, R., Scofield, J., and Shaw, A., Document Formatting Systems: Survey, Concepts, and Issues, *Computing Surveys,* **14**, 3 (September 1982), 418-472.

Furu86a
Furuta, R., An Integrated, but not Exact-Representation, Editor/Formatter, pp. 246-259 in *Text Processing and Document Manipulation,* ed. J. C. van Vliet, Cambridge University Press, Cambridge, April 1986.

Gutt78a
Guttag, J. V., Horowitz, E., and Musser, D. R., Abstract Data Types and Software Validation, *Communications of the ACM,* **21**, 12 (December 1978), 1048-1064.

Haml86a
Hamlet, R., A Disciplined Text-Formatting Environment, pp. 78-89 in *Text Processing and Document Manipulation,* ed. J. C. van Vliet, Cambridge University Press, Cambridge, April 1986.

Horg84a
Horgan, J. R. and Moore, D. J., Techniques for Improving Language-Based Editors, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* **19**, 5 (April 1984), 7-14.

John84a
Johnson, L., MacWrite, Apple Computer, Inc., Cupertino, California, 1984.

Joy80a
Joy, W., An Introduction to Display Editing with VI, University of California, Berkeley, September 1980.

Kimu84a
Kimura, A. G., A Structure Editor and Model for Abstract Document Objects, 84-07-04, Dept. of Computer Science, Univ. of Washington, July 1984.

King86a

King, P. R., An Overview of the W Document Preparation System, pp. 188-199 in *Text Processing and Document Manipulation*, ed. J. C. van Vliet, Cambridge University Press, Cambridge, April 1986.

Lesk79a

Lesk, M., Tbl - a program to format tables, Computer Science Technical Report, Bell Laboratories Murray Hill, January 1979.

Meyr82a

Meyrowitz, N. and Dam, A. van, Interactive editing systems: Parts I & II, *ACM Computing Surveys*, **14**, 3 (September 1982), 321-415.

Morr81a

Morris, J. and Schwartz, M., The Design of a Language-Directed Editor for Block Structured Languages, *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, **16**, 6 (June 1981), 28-33.

Nana86a

Nanard, M., Nanard, J., Sallantin, J., and Haiech, J., Semantic Guided Editing: A Case Study On Genetic Manipulations, pp. 90-106 in *Text Processing and Document Manipulation*, ed. J. C. van Vliet, Cambridge University Press, Cambridge, April 1986.

Ossa76a

Ossanna, J., NROFF/TROFF User's Manual, Computer Science Technical Report, Bell Laboratories, Murray Hill, October 1976.

Parn72a

Parnas, D. L., On the Criteria To Be Used in Decomposing Systems into Modules, *Communications of the ACM*, **15**, 12 (December 1972), 1053-1058.

Quin86a

Quint, V. and Vatton, I., Grif: An Interactive System for Structured Document Manipulation, pp. 200.2i13 in *Text Processing and Document Manipulation*, ed. J. C. van Vliet, Cambridge University Press, Cambridge, April 1986.

Shne83a

Shneiderman, B., Direct Manipulation: A Step Beyond Programming Languages, *Computer*, **16**, 8 (August 1983), 57-69.

Stal81a

Stallman, R., EMACS: The Extensible, Customizable Self-Documenting Display Editor, *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, **16**, 6 (June 1981), 147-156.

Trig83a

Trigg, R., A Network-Based Approach to Text Handling, Department of Computer Science, Univ. of Maryland, TR-1346, November 1983.

Walk81a

Walker, J., The Document Editor: A Support Environment for Preparing Technical Documents, *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, **16**, 6 (June 1981), 44-50.

Wood81a

Wood, S. R., Z-The 95% Program Editor, *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, **16**, 6 (June 1981), 1-7.

Zelk84a

Zelkowitz, M. V., A Small Contribution to Editing with a Syntax Directed Editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, **19**, 5 (April 1984), 1-6.

Appendix A

User Commands

Command	Description
<CNTL>Y	Abort the editor
<CR>	Insert a new line
<ESC>Q	Quit the editor
<ESC>NT	Start a new block of text
<ESC>NPC	Insert a new paragraph as a child to the current structure
<ESC>NPS	Insert a new paragraph as a sibling to the current structure
<ESC>[A	Move the cursor up one line
<ESC>[B	Move the cursor down one line
<ESC>[C	Move the cursor right one character
<ESC>[D	Move the cursor left one character

Table A.1: Prototype Commands

Appendix B
Manual Page

NAME

ge - graphical editor

SYNOPSIS

ge [option] [option] ...

DESCRIPTION

ge is a combination text-editor and text-formatter. It is intelligent, allowing the user to manipulate high level text items and formatting issues, and at the same time disassociates the user from the low level issues.

The following debug options are available:

-input=[suboption[,suboption]...]
these are debug traces for the Input process. All debug information is printed to the file "input.trace".

-editor=[suboption[,suboption]...]
these are debug traces for the Editor process. All debug information is printed to the file "editor.trace".

-display=[suboption,[suboption]...]
these are debug traces for the Display process. All debug information is printed to the file "display.trace".

Suboptions can be one of the following:

mes
print a diagnostic message for each message.

call
print a diagnostic when a function is called and when it returns.

parm
print all traceable formal parameters to functions.

FILES

input.trace
editor.trace
display.trace

BUGS

There are only a few structures supported (just a paragraph and block text).

There is no way to save information to disk.

Biographical Note

The author was born December 17, 1958, in Santa Monica, California. In 1960 he moved to Portland, Oregon and there attended various schools, graduating from Andrew Jackson High School in 1976. He then entered Washington State University in 1977, moved to Portland State University in 1978, and finally to Oregon State University in 1979, where he received his Bachelor of Science degree in August 1980.

In September 1980 the author began working for Floating Point Systems, Inc. as an Engineer. In June 1983 he began study at the Oregon Graduate Center where he completed the requirements for the degree Masters of Science in April 1987.

The author has been married four months to the former Linda Ranney. He is leaving the Graduate Center to pursue his career in engineering.