IDA PAPER P-1893

Ada* FOUNDATION TECHNOLOGY

Volume IV: Software Requirements for WIS Text Processing Prototypes

Alan Shaw, *Task Force Chairman*
Murray Berkowitz, *IDA Task Force Manager*
Bill Brykczynski
Christopher Fraser
Edgar Irons
Marvin Zelkowitz
John Salasin, *Program Manager*

December 1986

Prepared for
Office of the Under Secretary of Defense for Research and Engineering

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311

*Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

IDA Log No. HQ 86-30821

The work reported in this document was conducted under contract MDA 903 84 C 0031 for the Department of Defense. The publication of this IDA Paper does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that agency.

This paper has been reviewed by IDA to assure that it meets high standards of thoroughness, objectivity, and sound analytical methodology and that the conclusions stem from the methodology.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

AD-A178435

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) P-1893 - Volume IV	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Institute for Defense Analyses 6b OFFICE SYMBOL IDA	7a NAME OF MONITORING ORGANIZATION
6c ADDRESS (City, State, and Zip Code) 1801 N. Beauregard St. Alexandria, VA 22311	7b ADDRESS (City, State, and Zip Code)
8a NAME OF FUNDING/SPONSORING ORGANIZATION WIS Joint Program Management Office 8b OFFICE SYMBOL (if applicable) WIS/JPMO	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA 903 84 C 0031
8c ADDRESS (City, State, and Zip Code) 7798 Old Springfield Road McLean, VA 22102	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. T-W5-206 / WORK UNIT ACCESSION NO.
11 TITLE (Include Security Classification) Ada™ Foundation Technology: Volume IV - Software Requirements for WIS Text Processing Prototypes	
12 PERSONAL AUTHOR(S) A. Shaw, M. Berkowitz, B. Brykczynski, C. Fraser, E. Irons, M. Zelkowitz	
13a TYPE OF REPORT Final 13b TIME COVERED FROM _____ TO _____	14 DATE OF REPORT (Year, Month, Day) 1986 December 15 PAGE COUNT 46
16 SUPPLEMENTARY NOTATION	

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

World Wide Military Command and Control System (WWMCCS), WWMCCS Information System (WIS), automatic data processing (ADP), Ada programming language, CAIS, text processing, word processing, text editor

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This document describes general requirements for the WIS Text Processing System prototype project. The objective for this effort is the development of a complete Ada based "document management" system with capabilities for, but not limited to, word processing, providing output with multiple type fonts, and user-oriented "help" messages tailored to the operations being performed and user expertise. In general, critical design issues include the following: a) The provision for compatibility with multiple subsystems that may or may not have been completely defined and/or developed; b) The ability to use a variety of input/output devices; c) The capability of using textual syntax/semantics to provide assistance in document preparation; d) The ability to provide user-oriented "help" based on analyses of operations being performed (e.g., code being executed) and user history/expertise.

Two specifications have been generated and support the stated objective of developing software in Ada to support WIS communications functionality in the 1990's: a) Specification for Structure Generator Editor for Documents, Programs and Other Structured Data; b) Specification for a Writer's Workbench.

This work is the result of the identification of the functionality requirements and research of the technology base.

This volume is the fourth of a nine-volume set describing projects which are planned for prototype foundation technologies for WIS using the Ada programming language. The other volumes include command language; software design, description and analysis tools, database management system, operating systems; planning and optimization tools; graphics; and network protocols.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ■ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code) / 22c OFFICE SYMBOL

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

IDA PAPER P-1893

Ada™ FOUNDATION TECHNOLOGY

Volume IV: Software Requirements for WIS Text Processing Prototypes

Alan Shaw, *Task Force Chairman*
Murray Berkowitz, *IDA Task Force Manager*
Bill Brykczynski
Christopher Fraser
Edgar Irons
Marvin Zelkowitz
John Salasin, *Program Manager*

December 1986

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 84 C 0031
Task T-W5-206

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

1.0 INTRODUCTION

1.1 Purpose

The World Wide Military Command and Control System (WWMCCS) is an arrangement of personnel, equipment (including automatic data processing (ADP) equipment and software), communications, facilities, and procedures employed in planning, directing, coordinating, and controlling the operational activities of U.S. Military forces.

The WWMCCS Information System (WIS) is responsible for the modernization of WWMCCS ADP system capabilities, including information reporting systems, procedures, databases and files, terminals and displays, communications (or communications interfaces), and ADP hardware and software. The WIS environment is a complex one, consisting of many local area networks connected via long distance networks. The networks will contain a wide variety or hardware and software and will continue to evolve over many years.

The main functional requirements for WIS are presented in [JACK 84]. Briefly, the functional requirements have been categorized into seven areas:

a. Threat identification and assessment functions involve identifying and describing threats to U.S. interests.

b. Resource allocation capabilities must be provided at the national, theater, and supporting levels.

c. Aggregate planning capabilities must provide improved capabilities for developing suitable and feasible courses of action based on aggregated or summary information.

d. Detailed planning capabilities must provide improved methods for designating specific units and associated sustainment requirements in operating plans and for detailing the sustainment requirements in supporting plans.

e. Capabilities must must be provided to determine readiness, for directing mobilization, deployment and sustainment at the Joint Chiefs of Staff (JCS) and supported command levels and for promulgating and reporting execution and operation orders.

f. Monitoring capabilities must provide information needed to relate political-military situations to national security objectives,and to the status of intelligence, operations, logistics, manpower, and C3 situations.

g. Simulation and analysis capabilities must include improved versions of deterministic models that are comparable to those contained in the WWMCCS.

In order to support these high level objectives, the WIS system software must provide an efficient, extensible, and reliable base upon which to build this functionality. To develop such system software, several projects are planned for prototype foundation technologies for WIS using the Ada programming language. The purpose for developing these prototypes is to produce software components that:

a. Demonstrate the functionality required by WIS.

b. Use the programming language Ada to provide maximum portability, reliability, and maintainability consistent with efficient operation.

c. Demonstrate consistency with current and "in-progress" software standards.

Foundation areas in which prc types will be developed include:

a. Command Languages

b. Software Design Description and Analysis Tools

c. Text Processing

d. Database Management Systems

e. Operating Systems

f. Planning and Optimization Tools (Computational and Analytic Segment (CAS) Smart Advisor for Planning and Execution Decisions (WISSAPED))

g. Graphics

h. Network Protocols

1.2 Scope

This document describes general requirements for the WIS Text Processing System prototype project. The objective for this effort is the development of a complete Ada-based "document management" system with capabilities for, but not limited to, word processing, providing output with multiple type fonts, and user-oriented "help" messages tailored to the operations being performed and user expertise. In general, critical design issues include the following:

a. The provision for compatibility with multiple subsystems that may or may not have been completely defined and/or developed

b. The ability to use a variety of input/output devices

c. The capability of using textual syntax/semantics to provide assistance in document preparation

d. The ability to provide user-oriented "help" based on analyses of operations being performed (e.g., code being executed) and user history/expertise

Two specifications have been generated and support the stated objective of developing software in Ada to support WIS communications functionality in the 1990's:

a. Specification for a Structure Generator Editor for Documents, Programs and Other Structured Data

b. Specification for a Writer's Workbench

This work is the result of the identification of the functionality requirements and research of the technology base.

The Structure Generator Editor for Documents, Programs and Other Structured Data specification describes a project to design, develop, and implement a prototype system for automatically generating general and special-purpose editors across a variety of computers, input devices, and hard/soft copy output devices.

The Writer's Workbench specifies Ada packages for common tools necessary for a text processing system to provide a writer with complete document preparation capability. This includes tools for spelling error detection/correction, style analysis, indexing, bibliography database, on-line dictionaries, and glossaries.

Efforts were initiated to develop specifications for a text editor/formatter system with the following capabilities:

a. High speed and efficient editing of large files.

b. File operations such as copying, moving or searching large blocks of text optimized to perform at nearly the speed achievable by the underlying operating system.

c. Defining character fonts arbitrarily and of arbitrary size from a pixel to larger than a screenful.

d. WYSIWYG ("What you see is what you get") formatting, so that the text stays in format as you type, including proportionally spaced text using characters of different widths.

e. Providing calculator and spreadsheet functions on data in the text.

f. Providing the opportunity to define forms whose fields may be selected from a list of alternatives, or constrained in arbitrary ways by an interpreted Ada program.

g. Allowing interpretive execution of Ada programs mixed with editing operations.

Upon further investigation it was determined that such capabilities exist with current commercially available technology and products such as the Slater Tower (Estes Park, Colorado) product "SPROUTS". This obviated the need for development of a specification of such a component as it could easily be purchased/licensed from the vendor and converted to Ada. General requirements for such a text editor are, however, included in this report.

In this document the basic design, structure and interfaces of the WIS Text Processing System are provided. Many implementation details are left to the implementor; however, it is intended that the text processing system will be designed and implemented in Ada.

The following documents form part of this document to the extent specified herein.

U.S. Department of Defense. Reference Manual for the Ada Language: ANSI/MIL-STD-1815A, January 1983.

U.S. Department of Defense. Common APSE (Ada Programming Support Environment) Interface Set (CAIS). Proposed MIL-STD-CAIS edition, KAPSE Interface Team (KIT), 1985.

U.S. Department of Defense. Joint Staff Officer's Guide 1984. (AFSC Pub 1).

Proposed MIL-STD Document Interchange Format (DIF).

1.3 Terms and Abbreviations

ADP	Automatic Data Processing
APSE	Ada Programming Support Environment
CAIS	Common APSE Interface Set
dag	Directed acrylic graph
DIF	Document Interchange Format
GKS	Graphical Kernel System
I/O	Input/Output
ISO	International Standards Organization
JCS	Joint Chiefs of Staff
LAN	Local Area Network
MS	Milliseconds
SQL	Structured Query Language
TCP/IP	Transmission Control Protocol/Internet Protocol
WIS	WWMCCS Information System
WWB	Writer's Workbench
WWMCCS	World Wide Military Command and Control System
WYSIWYG	What You See Is What You Get

1.4 References

[BAHL 85] Bahlke, R. and G. Snelting, "The PSG-Programming System Generator," in *Proceedings ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, published as *SIGPLAN Notices* 20, 7 (July 1985): 28-33.

[BIGG 84] Biggerstaff, T., D. Mack Endres, and I. Forman, "TABLE: Object-Oriented Editing of Complex Structures." *Proceedings of the 7th International Conference on Software Engineering* (March 1984). IEEE Computer Society Press, 1984, pp. 334-345.

[FRAS 80] Fraser, Christopher W., "A Generalized Text Editor," *Communications of the ACM* 23,3 (March 1980): 154-158.

[FRAS 81a] Fraser, Christopher W., "Syntax-Directed Editing of General Data Structures," *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation.* Published as SIGPLAN Notices 16, 6 (June 1981): 17-21. The proceedings of the conference containing this paper are also available as *SIGOA Newsletter* 2, 1&2, Spring/Summer 1981.

[FRAS 82] Fraser, Christopher W., "A Programmable Text Editor," *Software--Practice and Experience*, 12, 3 (March 1982): 241-250.

[FRAS 81b] Fraser, Christopher W., and A.A. Lopez, "Editing Data Structures," *ACM Transactions on Programming Languages and Systems* 3, 2 (April 1981): 115-125.

[FURU 82] Furuta, Richard, Jeffrey Scofield, and Alan Shaw, "Document Formatting Systems: Survey, Concepts, and Issues," *Computing Surveys* 14, 3 (September 1982): 417- 472.

[JACK 84] Jackson, B. and Salasin, J., *Preliminary Requirements for the Army WWMCCS Information System (AWIS)*, WP-84W00035, Mitre Corporation, February 1984.

[KIMU 84] Kimura, Gary D., *A Structure Editor and Model for Abstract Document Objects*. Ph.D. thesis, Department of Computer Science, University of Washington, July 1984. Also issued as Technical Report 84-07-04.

[KIMU 83] Kimura, Gary D., and Alan C. Shaw, *The Structure of Abstract Document Objects*. Technical Report 83-09-02, Department of Computer Science, University of Washington, September 1983. Also in: *Proceedings of the ACM-SIGOA Conference on Office Information Systems* (1-2 June 1984). ACM, New York, 1984, pp. 161-169.

[MEDI 82] Medina-Mora, Raul, *Syntax-Directed Editing: Towards Integrated Programming Environments*, Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, March 1982.

[MEYR 82] Meyrowitz, Norman, and Andries van Dam, "Interactive Editing Systems: Parts I and II," *Computing Surveys* 14, 3 (September 1982): 321-415.

[NIEV 82] Nievergelt, J., G. Coray, G.D. Nicoud, and A.C. Shaw (editors), *Document Preparation Systems*, North-Holland, 1982.

[NOTK 84] Notkin, David S., *Interactive Structure-Oriented Computing*, Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, February 1984. Also issued as Technical Report CMU-CS-84-103.

[NOTK 85] Notkin, David S., "The GANDALF Project," *The Journal of Systems and Software* 5, 2 (May 1985): 91-105.

[REIS 84] Reiss, Steven P., "PECAN: Program Development Systems that Support Multiple Views," in *Proceedings of the 7th International Conference on Software Engineering*, March 1984. IEEE Computer Society Press, 1984, pp. 324-333.

[SCOF 85] Scofield, Jeffrey A., *Editing as a Paradigm for User Interaction*, Ph.D. thesis, Department of Computer Science, University of Washington, August 1985.

[SMIT 82] Smith, David Canfield, Charges Irby, Ralph Kimball, and Bill Verplank, "Designing the Star User Interface," *Byte* 7, 4 (April 1982): 242-282,.

[TEIT 81] Teitelbaum, Tim and Thomas Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM* 24, 9 (September 1981): 563-573. Also published as Cornell CS Tech Report TR80-421.

2.0 A STRUCTURE EDITOR GENERATOR FOR DOCUMENTS, PROGRAMS, AND OTHER STRUCTURED DATA

2.1 Introduction

The purpose of this project is to produce a prototype for a generator of structure editors. The generator system should provide the necessary facilities for constructing editors; for example, the following:

a. The preparation of documents, particularly the standard military forms used for transmitting orders, intelligence, plans, and other communications.

b. The construction of Ada programs.

c. The creation of a variety of other types of structured data, for example, spreadsheets, electronic mailboxes, and file directories.

The system is to be driven by grammatical descriptions of the object classes together with associated Ada action routines which govern the output display. The command language part of the user interface may be fixed in the initial version, but the design should permit a more flexible syntax-directed version that could easily accommodate different interaction styles.

This document briefly discusses several recent research and development efforts that demonstrate the feasibility of such a generalized approach. Section 2.3 specifies the applications, i.e., the objects that the editor should handle, in more detail. The range of host hardware and software environments are then presented; in summary, these are high quality, high performance workstations, and modern Ada-based graphics and database packages, respectively. The user interface is described in Section 2.5, including input commands for editing data and structure, control commands, and output forms. Section 2.6 outlines the requirements and possibilities for object grammars and action routines.

2.2 Background

Traditionally, editors edit text. However, the editing paradigm has also been used extensively as the way to interact with many other kinds of objects. Program editors edit abstract syntax trees for computer programs (e.g., [TEIT 81]). Word processing software edits documents and formats them in real-time [FURU 82; MEYR 82; NIEV 82]; such documents may contain different types of textual objects, as well as tables, mathematical formulae, and other special symbols. Electronic mail systems edit electronic mailboxes. Standard text editors often edit structured data encoded as linear text; for example, calendars and accounting files are often represented as text files. Window managers edit properties of windows, such as their size, position, and priority. Debuggers edit memory images, and sub-editors within them edit tables of breakpoints. Various utilities edit file system directories, for example, to display directories, rename files, or remove files. Some operating system command interpreters permit the re-use and modification of past commands by maintaining and editing command scripts. Applications packages often edit diverse structures; for example, graphics systems may edit complex linked structures representing images and spreadsheet managers, such as Visicalc, edit financial models.

All of these systems require a similar core set of facilities, for example, to insert, delete, move, and copy objects. These editing similarities over such a wide range of objects and structures have led to the recent development of general syntax-directed (structure) editors. Thus, there are structure editors for programs [BAHL 85, MEDI 82, REIS 84], structure

editors for documents [KIMU 84, KIMU 83], and more general editors that work with any object at the user interface [FRAS 80, 81a, 82; FRAS 81b; NOTK 85; SCOF 85]. Typically, these editors may prompt for input according to grammars defining the objects of interest, may parse some input when complete syntactical prompting is not practical or desired, and display structures in realtime through a formatting (or "unparsing") process. This project is concerned with the automated construction of such editors.

2.3 Applications of the Editor

Initially, the editor system must be able to handle standard military documents used for the formal communications of orders and plans, military electronic mail, spreadsheets, and Ada programs. Emphasis should be on the editing and viewing of electronic data, with the option of producing reasonable quality hardcopy versions. High-quality typography is not a first-order concern.

Formats for military standard documents are defined in Appendix I of AFSC PUB 1 (Forms 1 through 8). For electronic mail, the principal need is to edit and view messages prepared according to the military joint message form (DD Form 173).

As a structure editor for the preparation of Ada programs, the editor system must provide for the syntax-directed preparation of Ada compilation units and subunits. These include packages, generics, subprograms, and tasks.

Fairly simple spreadsheets, say at the complexity level of Multiplan, should also be constructable. Finally, it should not be difficult to extend or modify the system so that the other objects and applications mentioned in the last section can be processed also.

2.4 The Host Environments

It is assumed that the generator and resulting editors will run under modern hardware and software host environments, and that their features will not be compromised by past and obsolete technologies. In particular, in the hardware area, it will be written for advanced input and output systems, powerful personal workstations, and high bandwidth local area networks (LAN's). Input includes an efficient positioning and selection device such as a mouse. Screen output should be a high precision raster display, refreshable in realtime from a user workstation. A laser printer, or equivalent, is assumed for hardcopy output. Printer, filing, database, and mailing services are provided across the LAN.

In addition to an Ada compiler and run-time system, the software environment has various front- and back-end facilities that the editor may use. The editor should assume the availability of a standard graphics package, such as Graphical Kernel System (GKS), and should use device-independent logical or virtual IO for accessing and manipulating text, line, and general raster images. Similarly, there is a window package that permits the definition and management of virtual screens; this permits several simultaneous and differing views of objects being edited, as well as convenient space for dialogs, menus, and other interface elements. Eventually, a window manager may be yet another application of the editor generator.

A back-end need and assumption for the editor is a filing and/or database system that allows the storage and retrieval of object and object classes. For example, the description (grammar and action routines) for the military document class "Planning Directive" would be stored and accessible through this system. There is also a mail system that presents higher-level send and receive operations, that the user can access through the editor. (The mail system itself is not part of this project.) The management, enforcement, and

maintenance of security classifications would be implemented through these host systems (mail, filing, database,...), but the editor must be cognizant that certain operations may be disallowed because of security violations.

As a general rule, standard interfaces and tools are to be adopted and assumed. In particular, the CAIS standard is to be employed for Ada programs, filing tasks, and operating system functions. SQL is the adopted standard language for database access. TCP/IP and ISO are the networking standards. As implied above, graphics is done through GKS.

2.5 The User Interface

2.5.1 Views

The editor displays and allows editing of at least three different views of the structure that is being edited: the "what you see is what you get" (WYSIWYG) view, a linear textual view, and a structural view.

The WYSIWYG view displays objects in a fashion that mimics their formatted or pretty-printed hardcopy images as much as possible. This view is the default and normal interface. A prominent example of a WYSIWYG interface is the Xerox Star workstation [SMIT 82] or the Apple Macintosh.

The structural view displays the object's decomposition using graphics or text, whichever best suits the object at hand. Graphic images might use boxes and arrows to display structure. Textual images might use indentation or parenthesization to display structure. The default WYSIWG view suggested in Section 2.6.1 uses indentation and thus presents structure at the same time.

The linear textual view represents the data file for the object. This view might be a prefix encoding of the structure, although if the WYSIWYG view includes enough keywords to permit reconstruction of the object, then a full prefix encoding may not be required.

2.5.2 The Interaction Language

The user commands fall into three classes. Viewing commands provide the means for traveling through and reading a document, program, message or other component. The second class, the data and structure editing operations, create and modify objects and their structures. The third class is a miscellaneous category and contains various important initialization, filing, and control functions.

For viewing, the editor should implement user instructions for scrolling through objects in a linear fashion and for traversing and displaying elements in a structured manner. In the latter case, there should be commands for viewing the parent(s), children, and siblings for a given object. It should be possible to define a viewing area implicitly as the result of executing a search command that searches for a particular object in the structure. The window manager should also make available a number of user commands related to viewing; typically, these permit interactive manipulation of window sizes, position, and priority (for overlapped windows).

In addition to conventional text editing commands, there should also be structure-editing operations to perform such functions as insert, delete, create, move, copy, share (attach an object to more than one parent in the structure, thus producing a directed acrylic graph

(dag), and change (rename or reclassify) an object of a given class. The next subsection elaborates on the structure-editing commands.

The editor should provide a minimal undo feature that backtracks one command by essentially executing the inverse of the last command entered. Ideally, the undo should extend backwards more than one step right to the beginning of a session, though the user interface must take care that the user understands exactly how much editing is being undone.

Miscellaneous commands include those commands to store objects in a resident database or file, to produce hardcopy versions, and to specify the type of view. Another command important for user convenience is a help operation that displays system documentation.

Regardless of command type, objects are selected (e.g., for editing) by naming them or by indicating their geometric extent on the display screen. The latter indication can be done explicitly by pointing or implicitly by directing the editor to move up or down in the structure surrounding a specified screen cursor position.

The editor should allow the user to enter any command by typing text, striking a function key, or selecting from a menu. A fixed command set for each of these three styles should be defined for the structure-independent commands (e.g., undo, delete, write). These fixed sets should be augmented with the appropriate structure-dependent commands (e.g., node creation), which are automatically inferred from the structure description (see the next). This feature should be provided by driving the command interpreter from a grammar, so that different styles of command entry may be used dynamically. In the long run, services, such as command completion and help, can be generated automatically from these grammars.

The command language should have a common form for all views and all objects, and it should follow the WIS standard for command language where it applies. The command interpreter should be isolated to simplify future modifications.

2.5.3 Editing Objects Structurally and Textually

The user interface is syntax directed; that is, object class grammars drive and prompt user editing. The structure specification should have associated concrete templates that are presented to the user to guide editing. However, because different users prefer different styles, and may want structure editing for only the larger objects, if at all, it should also be possible to selectively interact non-structurally via linear text. The editor distinguishes structural and textual commands by using distinct but consistent command sets for each.

For example, suppose the program grammar contained the following rule (Ada "if" statement):

```
if_statement ::=
        'if' condition 'then' statements
        { 'elseif' condition 'then' statements }
          [ 'else' statements ] 'endif'
```

The user may enter such statements structurally, perhaps using a ".if" command that displays a template for a general if_statement and implicitly directs the user to fill in each of the required and desired components. The user may also enter such statements by typing a

conventional text string; this string must, however, be parsed according to the rule to produce the appropriate internal structure, check for "syntax" errors, and allow formatted display.

The set of structure-editing commands like ".if" is automatically generated from the object grammar. This automatic inference might be implemented by concatenating the non-terminal symbol naming the rule with some prefix like "." to distinguish structure-editing commands from the text-editing commands.

This procedure would yield ".if_statement" for the rule above. Such commands could then be shortened by deleting any trailing characters that are not needed to distinguish this command from the other structure-editing commands. This procedure might yield a simple ".if" or even ".i" for the rule above.

Mixed structure and text editing is implemented by parsing and "unparsing" as necessary. When the user enters a text-editing command, the editor identifies the smallest substructure that covers the point of interest, unparses it by calling its action routine (See Section VI) and allows it to be edited as text. The resulting text is reparsed and reattached to the main structure as soon as the user enters a command that does not apply to the text (such as a command for editing structure or traveling to another part of the object).

The ability to mix structure and text editing implies that the text must contain enough information to recover structure; i.e.it must be parsable. All parsing shall be driven by the grammar, perhaps using a simple recursive-descent or LL(1) parser. Programs are easily parsed, but pure textual documents and mail objects will require user-specified tags to identify structural elements; examples of such tags may be TROFF -ms macro commands or Scribe environments [NIEV 82].

2.5.4 Editing Object Definitions

New object classes should be specified via the editor and their definitions stored in a library. Commands for retrieving and editing object instances and class definitions are also clearly necessary. The grammars defining classes may be created and edited by supplying the editor with a grammar for grammars, such as the one below:

```
grammar ::=  { rule action }

rule ::= *identifier "::=" expr

expr ::= term { "|" term }

term ::= factor { factor }

factor ::= ["*" | "%"] *identifier | literal
        | "(" expr ")" | "[" expr "]" | "{" expr "}"

literal ::= """ { *character } """
```

The asterisk (*) preceding an identifier is used to indicate a user-entered terminal or token, while the percentage sign (%) denotes a system-generated terminal. The actions associated with each rule are written in Ada, so they should be entered with the Ada grammar driving the editor that offers program editing. See Section VI for more detail on the actions.

The facility to change structure specifications shall allow the editor to impose different structures on the same data and thus incidentally unify some previously distinct editing styles. For example, the grammar

```
file ::= { printable | separator }
```

parses a file as a string of printable characters and line separators, which are treated symmetrically. It thus treats the entire file as a single string. In contrast, the grammar

```
file ::= { line }

line ::= { printable} separator
```

parses a file into a list of lines, where each line is a string of printables followed by a separator. It thus specifies a line editor. Thus the editor should be able to offer both editing styles.

2.6 Specification of Object Classes

An object class is defined by a set of rules (a grammar) with associated action routines. The rules give the syntactical or structural possibilities for members of the class, while the action routines generate the displayed version of the objects.

The syntax rules are written in a context free grammar form, similar to the notation used to express Ada syntax rules. Section 2.5.3 presented an example of an if_statement rule using this format and Section 2.5.4 gave a complete description of the possible forms for grammar rules (in the grammar for grammars). Further examples are given in Sections 2.6.1, 2.6.2, and 2.6.3.

The action routines play a role analogous to semantic action routines in compiler technology. The primary function of these routines, which are to be written in Ada, is to produce WYSIWYG display screen output; it should also be possible to direct the output to a hard copy device. Thus the user is normally interacting with objects whose screen images are the result of executing action routines. The action routines can also be used for other purposes such as defining templates and maintaining tabular constraints in spreadsheets. Default actions are automatically provided when action rules are not specified.

The next three subsections describe how grammars with rules and actions may be written to define structure editors for the object classes of interest. First, a useful set of default "actions" and templates are outlined. We then show how explicit action routines can be used to compute templates and WYSIWG views, overriding the defaults. Examples in military documents, Ada programs, and spreadsheets illustrate the ideas.

2.6.1 Default Actions and Prompting

If action rules are not given explicitly with the syntax rules, the system will provide defaults for:

(a) Prompting the user with templates in a goal-directed manner

(b) Viewing the partially specified document

Each unelaborated or unspecified syntactic unit is represented by default to the user by pretty-printing its character string description in the corresponding grammar rule; this template is treated as an atomic selectable unit. For example, alternative units (x1 | x2 | ... | xn) could be displayed on the same line, when possible, while sequenced units (x1 x2 ... xn)| could be displayed on successive lines. An unelaborated unit must be distinguished from an elaborated one, say, by using reverse video.

Example:

Suppose a grammar contains the rule (production):

```
document_element ::= text_block {text_block | list} | table
```

and the document_element unit has been selected. The screen could then display the string:

```
"text_block {text_block | list} | table"
```

with two selectable units, denoted by "text_block {text_block | list}" and "table", respectively. If the first of the two is selected, the "table" alternative could be erased from the screen and the user presented with a sequence of two selectable units: "text_block" followed by "{text_block | list}" on the next output line.

When units are partially or fully specified, a reasonable default is to display the character string defining each unit in a linear fashion, with a separate line for each element and an indentation tab for each level down the syntax tree. Unspecified portions are interleaved with the elaborated units and shown on separate lines according to the conventions above. For repeated items generated from the form {x}, the following default display could be employed:

If x1 x2 ... xn has been generated from {x} and each xi is an x unit, then the screen will have the appearance:

```
{x}
x1
{x}
x2
{x}
 .
 .
 .
{x}
xn
{x}
```

with the {x} tagged as unspecified, say by reverse video.

Example.

Using the first alternative of the document_element rule defined above, the following elaborated units consisting of a text_block followed by a list of three elements, followed by a text_block are possible:

This document has three parts:

1. header
2. body
3. references

Other organizations are possible.

The default display would be:

This document has three parts:

{text_block | list}

1. header
2. body
3. references

{text_block | list}

Other organizations are possible.

{text_block | list}

There should be an option for the user to turn off the display of elements that are unspecified but not required. These are the units given by the forms {x} and [x]. This option permits the final WYSIWYG view. The first part of the last example shows the view with the "{text_block | list}" turned-off.

2.6.2 Using Action Routines to Define Views of Unelaborated Objects

Action routines, written in Ada, can be used to override the defaults. For an unelaborated rule identified by the non-terminal symbol e, the display of the selectable units on the right-hand side of the rule (i.e., the template for the possible components of e) can be defined by such a routine.

The routine will be an Ada procedure of the form:

```
procedure DISPLAY_UNELABORATED_UNIT(UNIT :In UNIT_REFERENCE);
```

where **UNIT** is of enunmeration type **UNIT_REFERENCE**, and may be, for example, IF_STATEMENT, or DOCUMENT_ELEMENT

where DU stands for "Display Unspecified or Unelaborated unit" and e identifies the syntactic unit. Examples are DU_document_element and DU_if_statement.

A current position for a "pen" on the display surface is available to DU and other action routines. This position, denoted by **LOCATION**, is updated by the primitive and default display routines, and can be manipulated in order to translate geometric data such as

should be available, where rectangle gives the coordinates of a rectangular box surrounding the unit unit that is to be made selectable. For example, make_selectable(A(3)(1), BOX3) would attach the unit G (A(3)(1) in the above example) to the displayed box BOX3.

Examples:

1. Given again the rule for document_element defined above in Section 2.6.1, suppose that the template for the "table" alternative appears below that for "text_block {text_block | list}" and that the template for the "list" unit is to be shown indented one tab stop further than a text_block. An action routine to accomplish this could consist of the DU procedure:

```
DISPLAY_UNELABORATED_UNIT(DOCUMENT_ELEMENT);

-- Display templates for each unit.

DISPLAY_DEFAULT_UNELABORATED_UNIT
(DOCUMENT_ELEMENT (1)(1), LOCATION); -- first text_block.

DISPLAY_DEFAULT_UNELABORATED_UNIT(DOCUMENT_ELEMENT(1)(2)(1),
LOCATION);      --      second text_block.

LOCATION.X := LOCATION.X + TAB; -- tab contains indent value.

DISPLAY_DEFAULT_UNELABORATED_UNIT(DOCUMENT_ELEMENT(1)(2)(2),
LOCATION); -- indented list template

LOCATION.X := LOCATION.X - TAB; -- return to old margin.

DISPLAY_DEFAULT_UNELABORATED_UNIT(DOCUMENT_ELEMENT(2),
LOCATION);
-- table
```

2. Let a right-hand side of a non-terminal x be three units in sequence x1 x2 x3. Suppose that the unelaborated view is to be x1 and x2 in adjacent columns with x3 centered below. A DU action routine program to produce this format is

```
DISPLAY_UNELABORATED_UNIT(X)

PREVIOUS_Y_VALUE := LOCATION.Y; -- save previous y value.

DISPLAY_DEFAULT_UNELABORATED_UNIT(X(1),LOCATION);   -- display x1.

LOCATION.Y := PREVIOUS_Y_VALUE; -- stay on same line as x1.

LOCATION.X := LOCATION.X + COLUMNX; -- move to next column for x2;

DISPLAY_DEFAULT_UNELABORATED_UNIT(X(2),LOCATION);   -- display X2;

LOCATION.X := (LOCATION.X_COLUMNX2);

DISPLAY_DEFAULT_UNELABORATED_UNIT(X(3),LOCATION);   -- display x3.
```

3. A grammar rule that is intended to prompt a user to set the time may be of the form:

```
time ::= hour minutes
```

If it is desired to set the time in analog fashion, then the time template could be an analog clock-face with hour and minute hands defined by the action routine program:

```
DISPLAY_UNELABORATED_UNIT(TIME);

FACE(CENTER, RADIUS); -- routine to draw a clock face.

HEAD.X := CENTER.X;

HEAD.Y := CENTER.Y + RADIUS - DELTA;

-- Draw an hours arrow pointing at 12 o'clock.

ARROW(CENTER, HEAD); -- draw arrow.

HEAD.X  := X + RADIUS - DELTA/2;

HEAD.Y := CENTER.Y;

-- Draw a minutes arrow at 15 minute mark.

ARROW(CENTER, HEAD);

-- Define rect1 and rect2 as rectangles surrounding the

-- hour and minutes arrows, respectively (not shown).

-- Now make these rectangles selectable so that a user

-- can subsequently define the time by selecting and

-- moving the hands of the clock.

MAKE_SELECTABLE(TIME(1),   RECT1);

MAKE_SELECTABLE(TIME(2),   RECT2);
```

4. The first rule of a grammar for spread sheets may be:

```
spread__sheet ::= [header] {row | column} |
```

Assume that the initial "template" presented to the user is a grid defining possible entries and that the {row | column} templates are two cursors, one pointing at a potential column and the other pointing at a potential row. A row (column) cursor will be some icon, for example an outline of a pointing "finger", at the left of (above) the potential row (column). Initially, this will be row 1 and column 1. Then, the action routine program for DU might be:

```
DISPLAY UNELABORATED_UNIT(SPREAD_SHEET);

-- Assume grid is initialized (not shown);

DDU(SPREAD_SHEET(1),   LOCATION); --  template  for  optional header.

OLDX := LOCATION.X;

OLDY := LOCATION.Y;

LOCATION.X := 0;  -- row cursor is to the left of grid.
```

```
ROW_CURSOR(LOCATION); -- draw cursor for row.

-- Assume rect_row is a box surrounding the row cursor.

MAKE_SELECTABLE(SPREAD_SHEET(2)(1), RECT_ROW);

LOCATION.Y := OLDY;

LOCATION.X :=OLDX;

COLUMN_CURSOR(LOCATION);  -- draw cursor above column.

-- Assume rect_col surrounds column cursor.

    MAKE_SELECTABLE(SPREAD_SHEET(2)(2), RECT_COL);
```

2.6.3 Action Routines for Viewing and Elaborating Objects

This section is concerned primarily with the application of action routines for viewing elaborated objects. A second purpose is their use in maintaining constraints.

Using a convention similar to that employed for an unelaborated unit, we will reference the elaborated units of a grammar production with a structured array notation: $e(i) denotes the ith first level elaborated unit of the rule named e (having left-hand side e), $e(i)(j) references elaborated element j within element i of e, and so on. This will be particularly useful for accessing elements of a repeated unit; for example, the elements of b in the elaborated rule x ::= a {b} would be referenced by $x(2)(i), i = 1 to size($x(2)), where size(e) returns the number of units in the list e.

A procedure DISPLAY(t, LOCATION) is assumed that displays a terminal token t, such as a user_defined terminal, starting at location LOCATION on the screen; LOCATION is updated to point to the beginning of the next "line".

For displaying elaborated objects, the programmer must write a routine execute(e) corresponding to each production e. Otherwise, a default execute procedure is used. The default just calls execute repetitively for each component in turn. For example, the default execute for the rule A ::= B C | D E {F} is:

```
case unit(A(1)) is when BC = >

when BC = >

      EXECUTE(A(1)(1)); -- execute for B.

      EXECUTE(A(1)(2)); -- execute for C.

when DE = >

      EXECUTE(A(1)(1)); -- execute for D.

      EXECUTE(A(1)(2)); -- execute for E.

      N := SIZE(A(1)(3)); -- n = # of F's.

      for I in 1 .. N loop

            EXECUTE(A(1)(3)(I)); -- execute Ith F.

      end loop;
```

```
end case;
```

The last major procedure, called instantiate, is used to initialize and update data structures and views when a particular syntactic unit is instantiated, i.e. selected for elaboration by a user. The execute routine described above assumes that all units have previously been instantiated. A suitable default is also defined.

2.7 Example Applications

Examples from the three principal application areas are used to illustrate the desired techniques. First, a military message document is specified. Next, we give a small example showing part of a possible system for preparing Ada programs. The last section develops some ideas for a spreadsheet description.

2.7.1 Definition of a Military Document

A simple form of military document might contain the elements and structure given by the following (incomplete) grammar of eight rules:

1. mil_doc ::= security_classification issuer date_time subject body
2. security_classification ::= 'Security Classification = '('Unclassified' | 'Confidential' | 'Secret' | 'Top Secret')
3. issuer ::= issuing_hq place_of_issue
4. date_time ::= zone hour day month year
5. subject ::= *string
6. body ::= paragraph {paragraph}
7. paragraph ::= *string {*string | elist}
8. elist ::= %number *string {%number *string}

Each paragraph in the body consists of a text string entered by the user (*string) followed by a sequence of text strings and/or elists. An elist, for "enumerated list", is a sequence of numbered items, each of which is a system-generated number (%number) followed by a user-defined text string.

Action rules for computing the values and views from elaborated units in Rule 8 could be the following segment:

```
ELIST(1) := "1.  "; -- first %number in rule 8.

DISPLAY(ELIST(1), LOCATION); -- display " 1. "

LOCATION.X := LOCATION.X + 3; -- move to the right past "1."

LOCATION.Y := LOCATION.Y - LINE_SPACE; -- move back to same line.

DISPLAY(ELIST(2), LOCATION); -- display first *string.

N := SIZE(ELIST(3)); -- n =    of items in ().
```

```
for I in 2 .. N LOOP
    -- generate next number.
    ELIST(3)(I)(1) := MAKESTRING(I) & ".";  -- generate next number
    DISPLAY(ELIST(3)(I)(1), LOCATION);
  LOCATION.X := LOCATION.X + 3;
  LOCATION.Y := LOCATION.Y -- LINE_SPACE;
    DISPLAY(ELIST (3)(I)(2), LOCATION); -- display Ith string.
end loop;
```

The view of the mil_doc grammar above could have been produced by this program, since the grammar rules are in the form of an enumerated list. The above routine also works correctly if the user_defined terminals, denoted by *string, can be designated as shared. A particularly simple example of sharing might occur when entering the mil_doc grammar, where it may be convenient to share the five instances of "*string" and two instances of "%number".

Action rules could also be written to display other units in locations different than their default ones. For example, one could write an action routine to center the subject unit (Rule 5). Similarly, a program analogous to that in Example 2 of Section 2.6.2 might be employed to display the issuer and date_time units in adjacent columns rather than vertically; the action routines below for "executing" Rule 1 include this format:

```
EXECUTE(MIL_DOC(1)); -- security classification
OLDY := LOCATION.Y; -- save y for date_time.
EXECUTE(MIL_DOC(2)); -- issuer
LOCATION.Y := OLDY; -- restore y value.
LOCATION.X := LOCATION.X + COLUMNX; -- move right to adjacent column.
EXECUTE(MIL_DOC(3)); -- date_time
LOCATION.X := LOCATION.X - COLUMNX; -- move back.
EXECUTE(MIL_DOC(4));
EXECUTE(MIL_DOC(5));
```

2.7.2 Ada Programs

The default WYSIWG view may not be the standard pretty-printed display expected for a computer program. For example, the expected default view for English text is not the same as that for programs.

Consider an Ada loop statement, given by the rule:

```
loop_statement ::= ['while' condition | 'for' for_spec]
                   'loop' statements 'end loop;'
```

An example of a fully elaborated loop statement containing a while iteration scheme is:

```
while COST < LIMIT loop
        COST := COST + PRICE(N);
        N := N + 1;
end loop;                          -- (1)
```

Using the default suggested in Section 2.6.1, the above statement would appear:

```
while
COST < LIMIT
loop
COST := COST + PRICE(N);
N := N + 1;
end loop;                          -- (2)
```

To obtain the more standard pretty-printed view which was shown first (labeled by (1)), the following execute action routine could be provided for the loop_statement:

```
OLDX := LOCATION.X; -- save current indentation.
case UNIT(LOOP_STATEMENT(1)) is
        when WHILE_SCHEME =>
                DISPLAY("WHILE", LOCATION);
                EXECUTE(LOOP_STATEMENT(1)(2));
        when  FOR_SCHEME =>
                DISPLAY('for', LOCATION);
                EXECUTE(LOOP_STATEMENT(1)(2));
        when others => NULL; -- just a plain loop.
end case;
DISPLAY("LOOP", LOCATION);
LOCATION.X := OLDX + INDENT; -- indent for statements.
EXECUTE(LOOP_STATEMENT(3)); -- show the statements.
LOCATION.X := OLDX; -- move   ck to old indentation.
DISPLAY("END LOOP", LOCATION);
LOCATION.X := OLDX;
LOCATION.Y := LOCATION.Y + NEW_LINE;
-- updated LOCATION for next statement
```

2.7.3 Spread Sheet

Consider a simple spread sheet with a title and an "arbitrary" number of rows and columns - arbitrary up to the size of the display surface. Each entry can hold either a user-defined token, such as a number of a text string, or the result of executing a user-defined formula that may refer to the values of other entries. To make things concrete, assume that the formula is given by an Ada program segment that can access an array containing the table entries:

```
SS:   array(1..nr, 1..nc) of entry;
```

where nr is the number of rows currently elaborated and nc gives the number of columns.

The following grammar specifies this class of spread sheets. It is assumed that a given entry is shared by both a row and a column; i.e., entry SS(i,j) is shared by row i and column j.

1. spread_sheet ::= [title] {row | column}
2. row ::= entry {entry}
3. column ::= entry {entry}
4. entry ::= *string | *number | *formula
5. title ::= *string

More elaborate rules could be given. For example, the rows and columns could be numbered explicitly by including a system-generated %number with each row and column. Because the entries defined in Rules 2 and 3 are shared, the syntactical structure of a fully elaborated table will be a directed acyclic graph (dag) with spread_sheet at the root, row and column designators (and possible title) at level 1, and shared entries (producing the dag) at the next level. A shared entry will also be connected to its associated SS element.

The execute action routine to produce a WYSIWYG view corresponding to the first grammar rule is:

```
R := 0 ;
EXECUTE(SPREAD_SHEET(1));   -- assumed a no-op for undefined title.
N := SIZE(SPREAD_SHEET(2)); -- n = #of rows and columns.
for I in 1 .. N loop
    if EQUAL(SPREAD_SHEET(2)(I), SPREAD_SHEET(2)(1)) then
            -- ...2(I) is a row
            R := R + 1;
            (EXECUTE(SPREAD_SHEET(2)(I)); -- execute the row rule
   end if;
end loop;
```

This just displays the title, if present, and then the table in row order starting from the first or top row.

Associated with Rule 2 may be the following actions which draw a particular row r. This is the action program invoked by execute ($spread_sheet(2)(i)) in the above program for Rule 1.

```
N := SIZE(ROW(2));

-- Draw a horizontal grid line (code not shown) to

-- accommodate n + 1 entries.

-- Draw vertical line demarking first box (not shown).

C : = 1;

EXECUTE(ROW(1)); -- show first entry.

-- Draw vertical line ending first box (not shown).

for I in 1 .. N loop

        C := C + 1;

        EXECUTE(ROW(2)(I)); -- show next entry.

        -- Draw closing vertical line (code not shown).

end loop;
```

For the complete view, an execute routine is not necessary for Rule 3 since we are moving through the structure on a row by row basis. However, when a new row is instantiated, action routines are necessary, as described below. Rule 4 has the corresponding execute actions:

```
  if EQUAL(ENTRY(1), ENTRY(3)) then -- it's a formula.

        SS(R,C) := EVALAUTE(ENTRY(1));

    else

        SS(R,C) := ENTRY(1);

    end if;

DISPLAY(SS(R,C), LOCATION);
```

Here, the evaluate function is an interpreter that evaluates the formula specified in the entry at row r and column c. r and c are passed down from the row program above.

For this class of spreadsheets, we want to include the ability to insert a new row or new column anywhere in the currently defined table, i.e. between any two rows or columns. Adopt the convention that a selection of an unelaborated row (column, respectively) at row i (column i), where i <= nr+1 (i <= nc+1), will cause a new row i (column i) to be instantiated and old rows i through nr (columns i through nc) to be shifted one row down (one column right) to i+1 through nr+1 (i+1 through nc+1). This facility can be provided by an 'instantiate' action routine for row (column) in Rule 2 (Rule 3).

Consider the column case (Rule 3). After determining the selected column, say i, each old column from i through nc is shifted right by reassigning its entries to the next column and adjusting the SS array. (A column's entries are obtained by referencing column(j), j = 1 to nr.) The shared row entries (row(j), j = i to nc for each row) are also adjusted right since a row entry previously shared by the kth column is now shared by the (k + 1)st column when k >= i. Finally, the new column is initialized with unelaborated entries; these entries are also attached to their corresponding rows so that they are shared correctly.

3.0 GENERAL REQUIREMENTS FOR A TEXT EDITOR/FORMATTER

3.1 Introduction

The state of the art in text processing should be able to produce an integrated document containing text, pictures, tables, formulae, and calculations in the "What-you-see-is-what-you-get" (WYSIWYG) style. The purpose of this section is to define the following characteristics and objectives of a modern text editor/formatter, including:

a. Word processing characteristics

b. WYSIWYG document production

c. Performance considerations

d. Other characteristics, such as universality, compatibility, etc.

3.2 Word Processing Characteristics

The objective for word processing is to provide typing and editing capability which meets the expectations of common commercial systems. The package provides for formatting information, for example about placement of tabs and margins, in the text so that several formats may be used in and recorded with the document. Display modes are provided in such a way that formatting information may be seen and edited with text editing commands, or hidden so that the document displayed on the screen looks exactly as it would look when printed. The package is operated with function keys. Menus are provided in some situations to assist in operating the program, though most operations, especially frequently used ones are available without using menus.

3.2.1 The Idiom

It is nearly impossible to draw a picture of a face with function keys, and it is awkward to move a few words of text by mousing to highlight them, then using a menu to cut and past. Most consultants in office products believe a combination of function keys and pointing devices is useful in the integrated system. Simple manipulation of text without pictures is probably better done with function keys.

Above all else, one must keep uppermost in the mind that while it is useful to do a calculation every now and then, and while one does want to include pictures in documents, most of the work done at a computer terminal is typing and simple editing. A text processing system must never, never, compromise the ease and speed with which these basic operations are accomplished.

3.2.2 Cursor Motion

The cursor may be moved arbitrarily in the text, and anything typed will be placed in the text at the cursor location unless prohibited by write protection of the data or field definitions in a form. When the cursor reaches the extreme of the screen, the text will scroll up, down, left, or right to allow the cursor to move to material located off the screen. When typing, the text will scroll right, left, or down appropriately so the characters being typed are placed in the correct position in the document.

3.2.3 Scrolling

The input device, whether it be a mouse, or function keys, will provide capability to scroll the text up, down, left, or right by full or partial screens. It will also provide the ability to see a specified line or page of the text.

3.2.4 Formatting

Format lines may be inserted into the text specifying tabs and margins for the following text, including standard and decimal tab stops, and the ability to have the text lines flush left, right, centered, or justified both left and right. Text is formatted as it is typed, so at least the material on the screen is correctly formatted at all times. Margins may be set so that the material may be wider than the screen. Proportional spacing is shown directly on the screen including the use of fonts in which the characters may be of different widths. Text wraps by words from one line to the next as material is typed, inserted or deleted. Page breaks may be inserted or deleted or recalculated with a function key. Page breaks may be manually inserted at a specified spot, and regions of text may be marked so that a page break will not occur in the region.

3.2.5 Editing

Material may be deleted, copied or moved with cut and pasting operations. Such operations are, in general, initiated by highlighting a region of text to be deleted, moved, or copied, then using a function key to effect the operation. Regions highlighted may be any rectangular region on the text, including full lines. The entire document may be so highlighted to move, copy or delete the whole text. Data is moved from one place to another by first cutting or copying it. These operations place the data on a "clipboard" capable of holding a number of such items. They may subsequently be placed in the same document or into another with a paste operation. Operations to delete characters, or backspace over a character are provided. Typing may be either insertive or overstriking.

3.2.6 Search and Replace Operations

The ability to search for a specified phrase, and to replace phrases found by searching with another phrase are provided. The search may match patterns exactly or with "wild card characters" to match simple patterns in the text.

3.2.7 Calculating Capability

A capability will be provided to call for the evaluation of formulas like those used in operating a calculator. Such formulas may appear free form in the text.

3.2.8 Forms

Forms may be defined identifying data fields into which data may be typed or calculated in a controlled way. Forms in common use, such as Federal tax forms can be specified. Some field values may be calculated from data in other fields, and the data entered into fields may be constrained arbitrarily by a program. The field definitions may be shown so the form itself may be changed by editing it, or may be hidden when the form is in use. Forms may be arbitrarily long, may be wider or longer than the screen.

3.2.9 Document Interchange Format Capability

An important aspect of a text formatter will be the ability to conform to the proposed MIL-STD Document Interchange Format (DIF). DIF will allow documents to be transmitted among different word processing computer systems, retaining the original indents columns, paragraphs etc. All ASCII characters will be transferred. However, word wraparound and embedded blank lines may not transfer identically.

Conformance to the DIF will allow rapid transfer of documents from one word processor to another. This will save the time and work of rekeying these documents.

3.3 WYSIWYG Document Production

The working document should be an image of the final document. Many kinds of dynamic formatting, such as keeping paragraphs in a specified format even as they are being edited are becoming typical in modern products. More sophisticated kinds of formatting such as page break placement, sophisticated footnote placement, etc. are more typically done in a post process. It is acceptable to delay some computationally intensive operations until a command is issued to bring the document back into format, but the principle should be that the working document is as close an approximation to the final document as possible at all times.

3.3.1 The Integrated Document

The most innovative systems today are including all aspects of common office computing, word processing, graphics, calculating in spreadsheets, access to databases in one integrated processing product. The ideal is to be able to include objects of a variety of types in a "compound document" which not only displays the image to be finally printed, but carries along in an appropriate way the data used in creating the image.

The compound document is fundamentally a text in the direct image of what is to be printed finally, but includes pictures which might have been drawn on the computer or scanned by a digitizer, tables of numbers produced from underlying formulae in a spreadsheet style, and graphs generated from such tables. The ideal is that all of these constituents remain "live" so that one can alter the numbers in a table and as a consequence of that action have other numbers in the document or graphs derived from them change as a consequence of the editing operations.

The requirements for the text processing system are the capability:

a. To manipulate pictures and graphics objects in a fashion tightly integrated with text.

b. To provide a way to store arbitrarily complex objects, like the formulae which generated a table, in conjunction with the text for the table.

c. To activate processing system like spreadsheets which are associated with the data.

A good test of the compound document is to be able to mail it through an electronic mail system, and have the recipient be able to revise figures in a spreadsheet component of the document and activate the spreadsheet calculator to carry out consequent changes in the document.

3.3.2 Extended Formatting

The package provided formatting features found in common batch formatting packages used for preparing typeset material. We use the UNIX Troff package as a model for which features are included, but not for how they are included. The package provides all these features in a WYSIWYG fashion so that the document is presented to the user and edited appearing in this final format. Some global formatting operations, such as page break or footnote placement may be deferred until called for with a function key, but it is always possible to put the document into its final print form on the screen with minimal delay.

The package has the following capabilities:

a. Fonts and graphics: The package allows a character set which can be defined by users giving the picture for a character by designating its pixels.

   Such characters may be arbitrary in size and shape, as small as a pixel or larger than the screen. Picture objects such as common business graphics or photographs obtained through a digitizer may be designated as characters, and may then be manipulated as any character, in particular typed, cut, and pasted arbitrarily. Any text characters may be made bold or italic, underlined, or raised or lowered to superscript or subscript positions.

b. Layout: Positioning of material may be done at the pixel level, generally in the style and with the same capabilities as found in modern typesetting machines.

c. Formatting: Automatic section numbering in a variety of styles, page headings commonly found in books and publications are provided. Footnotes are positioned properly when page breaks are calculated. Material may be placed in a number of columns, mixed in with pictures or other displays arbitrarily placed.

3.3.3 Dynamic Constituents

A more innovative aspect of a compound document is one which provides for inclusion of capabilities which could not be printed, like moving pictures and voice. A system which Wang has used for voice is to be able to record voice by turning on a "recorder". As you speak into the recorder, the voice is recorded digitally, and the indication of the recording appears in your document as a special character placed in the document as though you had typed it, one character for 1 second of speech. Having recorded for a while, you may then play back a segment of speech by placing the cursor on one of the characters indicating speech, and press the "play" key. The cursor moves along the characters representing speech as you hear the recorded voice.

The extension of this system is to be able to manipulate the characters representing speech as any ordinary character, using cut and past operations to move speech data from one place to another, mailing the speech with the document containing the characters, etc.

An extension of such a speech system is possible with laser disk technology to provide similar representation for frames of TV images, so one could record, edit, and mail scenes captured with a TV camera.

3.4 Performance Requirements

The most important single aspect of a good editor/formatter is performance. The heart of a editor/formatter is viewing and typing text, and these operations should offer quality performance, equal to optimized disk and display operations. The systems should expect to operate on files of several hundred pages. Some potentially time consuming operations and challenging execution times include:

Typing Speed: A good typist can type about 50ms/character. "Monkey typing", typing characters as fast as possible with no content can be done at 25ms/character. Any typing machine should keep up with the expert typist, ideally with the monkey.

Displaying the Next Page: Computer users can easily scan material at 100 ms/page, and in systems offering this performance frequently do. Wang uses a figure of about 200ms as the maximum time to see the next screenful (usually about 200 characters) in a word processor.

Opening a File: Opening a file should not take more time than the time it takes for an optimized read of the file. This time would be approximately 10 seconds.

Searching a File: Searching a file should not take more than reading a file. This time would be approximately 10 seconds.

Closing an Altered File: Closing an altered file should take the same amount of time as copying the file. This time is approximately 20 seconds.

Cut or Paste the Entire File: Cutting or pasting an entire file should not take longer than copying the entire file. This would be approximately 20 seconds.

Operations which depend on file system performance should seek to equal optimized operation of the file system. The above figures for file operations on a 100 page, 500,000 byte file can be attained on the IBM PC/XT under MS/DOS using the hard disk. It takes 10 seconds to read such a file and 20 seconds to copy it.

Performance of this kind is achievable and expected in the world of word processors and personal computers. What are some of the implications for a text processor?

One implication is that a terminal operating at 9600, or even 19,200 baud on a time-shared computer will not do the job satisfactorily. A bandwidth of 20,000 characters/per second or 200,000 baud is required to display 2,000 characters in 100 ms. A time sharing system (for example UNIX on a VAX) is vary hard pressed to keep up with computation of this demand in any event. A PC like the IBM PC or better as the display device is probably a requirement, and it should either store its data locally or have at least a 500,000 baud path to the data.

A second implication is that graphics operations required for the kind of text processing we recommend must be done with great care. It takes 53ms to simply copy the 16,000 bytes of data required for the graphics display on the IBM PC. Soft character generation can be done at speeds approaching our recommendations, but must be carefully optimized.

4.0 WRITER'S WORKBENCH

4.1 Introduction

This section represents an initial list of tools that should be made available under the general term "Writer's Workbench" (WWB). It is assumed that such tools will be used to produce electronically generated information in many formats, including program source code, reports, formatted text (as in electronic mail), tables, pictures, graphs, etc.

Users need a consistent interface in order to effectively use such a set of tools. Since the primary purpose of the Writer's Workbench is to produce and modify documents, the primary interface to this set will be a text editor. Ideally the editor will provide the total environment for the production of these documents. However, costs, current technology and schedules means that this will probably not be the case. The user will interface with a text editor, but the editor will interface with an entire host of tools. This interface with other tools will be hidden from the user and a single interface will be presented.

4.2 Scope of Requirements

In this section, the scope of the WWB will be outlined. The underlying Ada environment will be explained, as well as the relationship to other text processing tools. This section will also list those features that are NOT part of the WWB, although are needed by users of the WWB.

4.2.1 Environment

This section presents a list of features that a writer will need, and gives a preliminary specification of how the editor environment will interface with the tool. Probably the most significant design decision is the information flow between the editor and the tool. The tool can process the entire document or to process a single object (e.g., word, picture, paragraph, etc.). The former means that the tool is an "off-line" type of process to be invoked when the document is completed, while the latter means that the tools is an "extension" of the basic editing function, and that the user is expecting results immediately.

A primary driving force in these decisions is the design of the basic editor. For the WIS-Ada Foundation Technology Program - Text Processing Area, the decision was made to input most textual information via a "What You See Is What You Get" (WYSIWYG) text editor with an underlying character-oriented editor to process streams of text. A companion section, the General Requirements for a Text Editor/Formatter describes the specifications for this editor in greater detail. This report describes the additional features needed to provide a full document preparation environment.

All tools in the WWB are to be written in Ada and run under the CAIS environment. There is an important need for consistency of the user interface among the various tools. Since the editor is WYSIWYG with a relatively high resolution raster display to exhibit various type fonts, the associated tools need the same input/output characteristics. For this reason, the CAIS terminal packages SCROLL_TERMINAL, PAGE_TERMINAL and FORM_TERMINAL are not applicable. It is assumed that I/O for all tools, even those with simple textual I/O requirements, will use the same graphical interface - e.g., GKS and window manager - as the text editor. Most of the tools will need to open a window for display purposes and to present information using the same fonts and formats as the editor.

Most tools will also need to read source document files, which will have embedded formatting commands. A common Ada package for accessing this text should be provided.

The amount of information that the editor needs to know about each specific tool should be minimal. For example, many of the tools return a "picture" that is simply to be inserted into the source document by the editor. The editor may move this picture to another place in the document, but it is not the responsibility of the editor to understand the structure of the "picture". The underlying tool must be called for that.

In addition, several of the tools require information from the editor. In order to simplify such information, the editor will have a feature for enclosing segments of a document (e.g., word, sentence, picture, paragraph) and sending this to another tool. With this generalized interface, tool development can proceed independently of editor development.

4.2.2 Omissions

Because the computer screen always displays information as it will appear on the printed page, there does not need to be a separate formatting language and tool (as NROFF in UNIX) which is manipulated by the user. However, files may contain such internal information in order to be able to display such information on the screen. In addition, the WYSIWYG structure imposes other constraints on the tools which will be described in the following section.

The specifications for a Structure Editor and the Text Editor are described in the preceeding sections. This report addresses the additional tools not handled by these two editors.

This specification also ignored tools specifically needed by the program designer and implementor. While it is fully expected that programmers will use the WWB both for program development (via the editor) and documentation (via the full WWB features), tools specifically for program source code production are beyond the scope of this specification. The class of tools not included here include:

1) Compiler and linker: Tools like Ada compilers, linkers, APSEs, etc. are not included.

2) Source configuration control: Tools to manage source code, dependency relationships, version control are not included. However, WWB tools will include some features that are needed since documents have similar problems.

3) Source code analyzers: The set of source code tools like statement analyzers, complexity analyzers, path analyzers, programming standards checkers, runtime path monitors, and symbolic debuggers are outside of the scope of this report.

4.3 Writer's Workbench Tools

In this section, the set of tools making up the WWB will be described. Additional information, such as whether the tool is provided by the editor, is stand-alone, or is part of a larger environment, is given.

4.3.1 Text Input

This is the primary function of the Text Editor, which is the main interface with the user at a terminal. This is described by a companion requirements definition.

4.3.2 Formatter

Since the Text Editor is WYSIWYG, formatting concerns are minimal. There does not need to be a separate formatting language like NROFF of UNIX. A corollary to this is the need for a sufficiently high bit-mapped display to simulate graphics and various type fonts that will appear on the printed version of the document.

4.3.3 Equation Input

There is a need to input mathematical and other scientific notation. It is assumed that the Text (or Structure) Editor will handle this eventually, but initially a separate tool will be used. Information will be passed to the tool on an equation basis and will return the formatted text to the editor. When called by the editor, the tool will open a window and use the same menu system as the editor. The tool will interact with the user to build the equation, and when completed, will pass the equation back to the calling editor for insertion into the document.

It is assumed that the editor will be able to move the equation around in the document; however, if it is necessary to modify the equation, then the Equation tool must be invoked. To the editor, the equation appears as a "formatted picture" and it does not need to understand its syntax. The tool, however, must pass information using the same notation as the editor uses for displaying information.

It is expected that this tool will appear as a "popup window" like on the Macintosh computer. The user will hit an "equation" menu command, a window will open, the equation will be built, a "finished" menu button is hit, and the popup window will disappear with the equation text now appearing at the appropriate place in the source document. It is expected that most tools will operate in this manner, thus hiding the differences between the editor and the associated tools.

4.3.4 Table Input

There is a need to format information in tabular form. The Text or Structure Editor should handle this and no additional tool is needed.

4.3.5 Fonts

It is important to process documents in a variety of type fonts. The Text Editor should handle this. It is important that all tools that display textual information use the same fonts so that the "sameness" of the tools is preserved.

4.3.6 Graphics

It is necessary to include diagrams, graphs and other forms of pictures. While it should be integral to the Text Editor, it is initially assumed that a separate tool will be used to create pictures. The editor will invoke a tool that will communicate with the user, who will build the picture. Upon completion, the tool will return the finished picture to the editor for insertion into the document.

The operation of this tool is similar to the Equation tool. A separate popup window will appear, and when completed, the tool will return a segment of the document that can be inserted directly by the editor. It is assumed that the editor can move this completed diagram, but does not have the knowledge to edit or modify it. If that is needed, then the Graphics tool needs to be invoked.

4.3.7 Spelling Checker

An important tool is one which checks the spelling of the words in the document. This is a fairly common tool in use today, and is best implemented as a call to the tool to check the entire document.

Input and output for the tool will be the source document file. The tool has to know how to read the text and ignore formatting information embedded in the file.

The tool requires at least two input dictionaries:

a. A standard dictionary of English words useful for any document.

b. A separate applications dictionary. This second dictionary includes the special names, acronyms, and terms specific to a single application area.

There is also an optional third dictionary, terms specific to this document. While this third dictionary can be merged with the second-application specific dictionary, a separate third dictionary allows for each document to have its own set of individual objects. These dictionaries are effectively merged by the Spelling checker as it looks for misspellings.

The Spelling checker may operate in one of two ways:

a. A file of misspelled words is created. As with other tools, the misspelled words appears in a separate popup window on the screen. In this mode, the file can be edited after the source document is checked, and the words either fixed in the document or else added to one of the dictionaries as a new legal term.

b. The document is scrolled in the window, and the misspelled words are successively highlighted on the screen. At each highlighted word, the user can tell the spelling checker to either add the word to one of the dictionaries or else correct the spelling.

In either case, the Spelling checker knows many of the rules for English spelling, and can determine other forms of words in the dictionary (present, past, future tense, plurals, participles, etc.). When it displays misspellings, it should also display likely candidates for the correct term from its dictionaries.

4.3.8 Dictionary

An on-line dictionary is useful for text creation. A writer can invoke the tool with a word and receive its definition. This can also be used as a spelling checker as words are inserted. Input will be a word of text, and output will be a small popup window with the definition.

4.3.9 Thesaurus

This is a companion to the Spelling checker and Dictionary. The editor passes a word to this tool and receives a list of synonyms in return (again in its own popup window).

4.3.10 Index

The creation of indices will be handled by a tool that is called with the word to be indexed. Since the document can be updated, which changes the page numbering, for simplicity it is assumed that the index tool is again called after the document is completed.

When the user wants to add a word to the index, the term will be highlighted by the editor. This term will then be passed to a separate tool for inclusion into the index. If the document has been modified, the index printing routine will recompute the index. Keeping track of creation times like in the UNIX program make can handle this.

4.3.11 Table of Contents

The production of tables of contents, figures, appendices, etc. can be handled by a tool similar to the indexing tool. Each new title is appended to the table, and after the document is completed, the tool is again called to process the page numbers.

4.3.12 Fog Index

A tool of growing importance is one which checks the grammar and style of the document. Concepts like the ability of understanding the text (e.g., the complexity of English sentences, active versus passive voice, the use of acronyms, the vocabulary level, etc.) have been grouped under the general concept of a "fog" index. It is useful to process all documents for these features. This tool, similar to the UNIX command "diction" accepts a source file as input and produces a series of tables describing the language level that has been used.

A companion tool accepts a part of a document and produces statistics about it. For example, a writer can pass a single sentence or paragraph to the tool, and in a popup window, an immediate commentary about the sentence can be produced. This can enable the writer to get immediate feedback on sentence structure and to change the sentence if necessary.

4.3.13 File Conversion

While this specification proposes an environment for document preparation, it is equally important to be able to process documents produced on other systems. It is expected that the files created by these tools will contain information needed to format and display the information, e.g., font sizes, types, graphical information, etc. Thus tools are needed to read "foreign" documents and convert them to the internal editor format that is to be developed.

It is unlikely that a single tool can be developed to handle all such foreign documents, however, a single tool can be developed to handle many of them. Any document consisting of ASCII (or another character code) characters with a relatively formal syntax (e.g., a NROFF file) can be described via a context free grammar. A tool can then be built which reads such a grammar description and a given source file and converts it to the WWB format needed by the WWB set of tools.

It is expected that additional tools might be needed to read specific formats into WWB format, e.g., specific editors, spreadsheets, etc.

4.3.14 Printing

There is the need to process the documents and turn them into hardcopy on paper via impact printers, laser printers or film. Languages like Impress already are used to describe such processing. There is a need for a tool to convert the internal WWB format into these standard formats for printing the document.

4.3.15 Comparator

There is a need for a tool to compare successive versions of a document for changes. Since the information needs to be presented to the user in the same WYSIWYG format, a simple file comparator is insufficient. This tool has three basic functions as it compares two different versions of a document:

a. Generates the set of differences between the two documents as a set of changes displayed in a window (and saved in a file),

b. Prepares a "script" which can be input to the editor which will convert one document into the other.

c. Prepares a "merged" document consisting of the old and new text.

The old deleted text will appear in one type font (e.g., italics or 'change bars' in the margins), the new inserted text will appear in another font (e.g., bold), while the text that is unchanged between documents appears in normal type.

Function (a) is useful for determining what has changed between different documents and function (b) is useful for backup purposes by saving only the new document and a supposedly short script that can be used to "undo" the latest changes. Function (c) is useful for producing revised user guides or other text to indicate to the user what has changed recently. This third feature eliminates the document preparer from manually keeping track of such tedious changes.

4.3.16 Encryption

There is a need, especially in a military environment, to encrypt information. It is assumed that encryption is a feature of the underlying WIS-Ada file system and does not need to be directly addressed here.

4.3.17 Database

There is a need to maintain an underlying database of documents. Again, this is a feature of the underlying WIS-Ada file system and does not need to be addressed here. It is assumed that such a database includes aspects of configuration control such as: time and date of creation, alternative versions, and dependency relationships. Features present in a simple version control system like make under UNIX are assumed to be present.

4.3.18 Command Language

There is a need to interface with a command language and the editor needs to build pop up menus on the screen for the user. This will require coordination with two other task forces. The graphics task force has been developing the low-level primitives needed to build such menus on a screen, while the command language task force is working on primitives for pop up menus. The specification of such windows should be included as part of these specifications, but is not directly addressed here.

Distribution List for IDA Paper P-1893

Sponsor

Maj. Terry Courtwright — 5 copies
WIS Joint Program Management Office
7798 Old Springfield Road
McLean, VA 22102

Maj. Sue Swift — 5 copies
Room 3E187
The Pentagon
Washington, D.C. 20301-3040

Other

Col. Joe Greene — 1 copy
STARS Joint Program Office
1211 Fern St., Room C107
Arlington, VA 22202

Defense Technical Information Center — 2 copies
Cameron Station
Alexandria, VA 22314

CSED Review Panel

Dr. Dan Alpert, Director — 1 copy
Center for Advanced Study
University of Illinois
912 W. Illinois Street
Urbana, Illinois 61801

Dr. Barry W. Boehm — 1 copy
TRW Defense Systems Group
MS 2-2304
One Space Park
Redondo Beach, CA 90278

Dr. Ruth Davis — 1 copy
The Pymatuning Group, Inc.
2000 N. 15th Street, Suite 707
Arlington, VA 22201

Dr. Larry E. Druffel — 1 copy
Software Engineering Institute
Shadyside Place
580 South Aiken Ave.
Pittsburgh, PA 15231

Dr. C.E. Hutchinson, Dean — 1 copy
Thayer School of Engineering
Dartmouth College
Hanover, NH 03755

Mr. A.J. Jordano — 1 copy
Manager, Systems & Software
Engineering Headquarters
Federal Systems Division
6600 Rockledge Dr.
Bethesda, MD 20817

Mr. Robert K. Lehto — 1 copy
Mainstay
302 Mill St.
Occoquan, VA 22125

Mr. Oliver Selfridge — 1 copy
45 Percy Road
Lexington, MA 02173

IDA

General W.Y. Smith, HQ	1 copy
Mr. Seymour Deitchman, HQ	1 copy
Mr. Robin Pirie, HQ	1 copy
Ms. Karen H. Weber, HQ	1 copy
Dr. Jack Kramer, CSED	1 copy
Dr. Robert I. Winner, CSED	1 copy
Dr. John Salasin, CSED	1 copy
Mr. Mike Bloom, CSED	1 copy
Ms. Deborah Heystek, CSED	1 copy
Mr. Michael Kappel, CSED	1 copy
Mr. Clyde Roby, CSED	1 copy
Mr. Bill Brykczynski, CSED	1 copy
Ms. Katydean Price, CSED	2 copies
IDA Control & Distribution Vault	3 copies