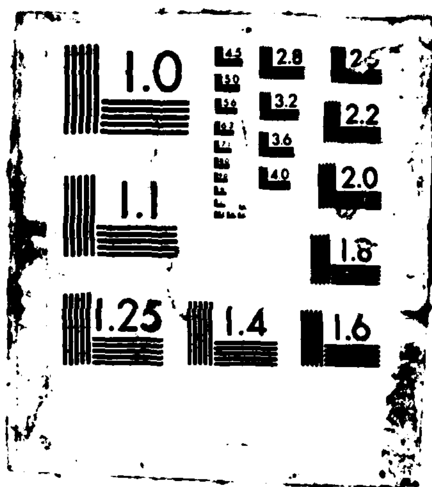
AD-A178 433 ADA (TRADE NAME) FOUNDATION TECHNOLOGY VOLUME 2 SOFTWARE REQUIREMENTS FOR..(U) INSTITUTE FOR DEFENSE ANALYSES ALEXANDRIA VA T KACZMAREK ET AL. DEC 86 1/1

UNCLASSIFIED IDA-P-1893-VOL-2 IDA/HQ-86-30819 F/G 9/2 NL



AD-E500 844
Copy 16 of 38 copies

AD-A178 433

IDA PAPER P-1893

Ada* FOUNDATION TECHNOLOGY

Volume II: Software Requirements for WIS Command Language Prototypes

Thomas Kaczmarek, *Task Force Chairman*
Joseph D. Hrycyszyn, *IDA Task Force Manager*
Philip Hayes
Kenneth Holmes
Robert Jacob
Jon Meads
Bradley Myers
John Salasin, *Program Manager*

DTIC ELECTE MAR 3 1 1987 S D E

December 1986

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

This document has been approved for public release and sale; its distribution is unlimited.

DTIC FILE COPY

87 3 30 061

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 86-30819

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.

Approved for public release, distribution unlimited.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE AD-A178 433

Field	Entry
1a REPORT SECURITY CLASSIFICATION	UNCLASSIFIED
1b. RESTRICTIVE MARKINGS	
2a SECURITY CLASSIFICATION AUTHORITY	
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
3 DISTRIBUTION/AVAILABILITY OF REPORT	Approved for public release; distribution unlimited.
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	P-1893 - Volume II
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.	WORK UNIT ACCESSION NO.
		T-W5-206	

11 TITLE (Include Security Classification)
Ada™ Foundation Technology: Volume II - Software Requirements for WIS Command Language Prototypes

12 PERSONAL AUTHOR(S)
T. Kaczmarek, J. Hrycyszyn, P. Hayes, K. Holmes, R. Jacob, J. Meads, B. Myers

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Final	FROM ______ TO ______	1986 December	90

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), command languages, automatic data processing (ADP), local area network (LAN), command, control, and communication (C3), Ada programming language, CAIS, design tools

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

The World Wide Military Command and Control System (WWMCCS) Information System (WIS) Joint Program Management Office (JPMO) requested that the Institute for Defense Analyses (IDA) develop specifications for prototype tools to support the design, implementation, and use of comand languages. This report contains Software Requirements Specifications (SRS's) for such tools.

The specifications were developed by members of a Task Force established by IDA and the IDA staff members listed on the title page of this report. The specifications were prepared over the period of August 1984-January 1986. Due to time constraints and funding limitations, not all of the specifications have been completed to the desired level of detail.

Specifications were developed for the following eight projects: 1) CAIS Form On Page Terminal (Section 4.1); 2) CAIS Page On Bitmap Terminal (with proposed extensions to the Common Ada Programming Support Environment (APSE) Interface Set (CAIS) for bitmap terminals) (Section 4.2); 3) Network Based User Interface Management System (UIMS) (Section 4.3); 4) String Based Ada Command Language (SBACL) (Section 4.4); 5) Video Based Ada Command Language (VBACL) (Section 4.5); 6) Interaction Object UIMS (Section 4.6); 7) History and Logging (Section 4.7), and 8) Style Guide (Section 4.8).

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
■ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	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

19. This volume is the second 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; software design, description, and analysis tools; text processing; database management system; operating systems; planning and optimization tools; graphics; and network protocols.

IDA PAPER P-1893

Ada™ FOUNDATION TECHNOLOGY

Volume II: Software Requirements for WIS Command Language Prototypes

Thomas Kaczmarek, *Task Force Chairman*
Joseph D. Hrycyszyn, *IDA Task Force Manager*
Philip Hayes
Kenneth Holmes
Robert Jacob
Jon Meads
Bradley Myers
John Salasin, *Program Manager*

December 1986

A1

DTIC COPY INSPECTED 6

IDA

INSTITUTE FOR DEFENSE ANALYSES

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

TABLE OF CONTENTS

1.0	INTRODUCTION	1
2.0	SCOPE	1
3.0	BACKGROUND	1
3.1	WIS Foundation Technology Program	1
3.1.1	WWMCCS	1
3.1.2	WIS	1
3.1.3	Foundation Software Studies	2
3.2	Command Languages	3
3.2.1	Command Language Tools Requirements	3
3.2.2	Factors and Issues	4
3.2.2.1	Issues Affecting the Specification of Command Language Tools	4
3.2.2.2	Application Issues Affecting Command Language Tools	6
3.2.3	Reference Model	7
3.2.3.1	Overview	8
3.2.3.2	Transactions	8
3.2.3.3	Help Management	10
3.2.3.4	Interaction Objects	10
3.3	Terms and Abbreviations	11
3.4	References	11
4.0	COMMAND LANGUAGE SOFTWARE REQUIREMENTS SPECIFICATIONS	13
4.1	CAIS Form On Page Terminal Software Requirements Specification	13
4.1.1	Introduction	13
4.1.1.1	Purpose	13
4.1.1.2	Scope	13
4.1.2	General Description	13
4.1.2.1	Product Perspective	13
4.1.2.2	Product Functions	14
4.1.2.3	User Characteristics	17
4.1.2.4	General Constraints	17
4.1.3	Specific Requirements	17
4.1.3.1	Functional Requirements	17
4.1.3.1.1	Local Editing Keys	17
4.1.3.1.2	CAIS FORM_TERMINAL Emulation	18
4.1.3.1.3	Procedural Interface to Editing Key Stroke Sequence Definition	19
4.1.3.1.4	Logging Facilities	19
4.1.3.2	External Interface	20
4.1.3.2.1	User Interface	20
4.1.3.2.2	Hardware Interface	20
4.1.3.3	Performance Requirements	21
4.1.3.4	Design Constraints	21
4.1.3.5	Attributes	21

TABLE OF CONTENTS (Continued)

4.2 CAIS PAGE ON BITMAP TERMINAL Software Requirements Specification 22
4.2.1 Introduction 22
4.2.1.1 Purpose 22
4.2.1.2 Scope 22
4.2.2 General Description 22
4.2.2.1 Product Perspective 22
4.2.2.2 Product Functions 23
4.2.2.3 User Characteristics 23
4.2.2.4 General Constraints 24
4.2.3 Specific Requirements 24
4.2.3.1 Functional Requirements 24
4.2.3.1.1 Initialization and Termination 24
4.2.3.1.2 CAIS PAGE_TERMINAL EMULATION 25
4.2.3.1.3 Virtual Function Key Support 25
4.2.3.2 External Interface 26
4.2.3.2.1 User Interface 26
4.2.3.2.2 Hardware Interface 26
4.2.3.3 Performance Requirements 26
4.2.3.4 Design Constraints 26
4.2.3.5 Attributes 27
4.2.4 Proposed CAIS Extension for Bitmap Terminal 27
4.3 Network Based User Interface Management System (UIMS) (NBU) Software Requirements Specification 30
4.3.1 Introduction 30
4.3.1.1 Purpose 30
4.3.1.2 Scope 30
4.3.1.3 Definitions 31
4.3.2 General Description 31
4.3.2.1 Product Perspective 31
4.3.2.2 Product Functions 32
4.3.2.3 User Characteristics 33
4.3.2.4 General Constraints 33
4.3.3 Specific Requirements 33
4.3.3.1 Functional Requirements 33
4.3.3.1.1 Description Language 33
4.3.3.1.1.1 General Principles 33
4.3.3.1.1.2 Specific Characteristics of the Language 34
4.3.3.1.2 Dialogue Description Editor 35
4.3.3.1.3 Dialogue Specification Translator 36
4.3.3.1.4 Documentation Extractor 36
4.3.3.1.5 Executor 37
4.3.3.1.6 Debugger 37
4.3.3.1.7 Generic Input and Output Tokens 38
4.3.3.1.8 Other Support Functions 39
4.3.3.1.9 Front End Module 39
4.3.3.2.3 Software Interfaces 40
4.3.3.3 Performance Requirements 40

TABLE OF CONTENTS (Continued)

4.4 String Based Ada Command Language (SBACL) Software Requirements Specification 41
4.4.1 Introduction 41
4.4.1.1 Purpose 41
4.4.1.2 Scope 41
4.4.1.3 Definitions 41
4.4.2 General Description 42
4.4.2.1 Functions 43
4.4.2.2 User Characteristics 43
4.4.2.3 General Constraints 43
4.4.3 Specific Requirements 44
4.4.3.1 Command Language 44
4.4.3.2 Command Language Interpreter 45
4.5 Video Based Ada Command Language (VBACL) Software Requirements Specification 48
4.5.1 Introduction 48
4.5.1.1 Purpose 48
4.5.1.2 Scope 48
4.5.2 General Description 48
4.5.2.1 Product Perspectives 48
4.5.2.2 Product Functions 48
4.5.2.3 User Characteristics 49
4.5.2.4 General Constraints 49
4.5.3 Specific Requirements 49
4.5.3.1 Function Requirements 49
4.5.3.2 Performance Requirements 50
4.5.3.3 Design Constraints 50
4.5.3.4 Attributes 50
4.6 Interaction Object UIMS (IOU) 51
4.6.1 Introduction 51
4.6.1.1 Purpose 51
4.6.1.2 Scope 51
4.6.2 General Description 51
4.6.2.1 User Interface and Interaction Objects 51
4.6.2.2 Input Handler and Interaction Object 51
4.6.2.3 Individual Interaction Object Event Handlers and Overall User Interface 52
4.6.2.4 Tokens in Interaction Objects 52
4.6.2.5 Schema for Interaction Objects 53
4.6.3 Interaction Object Example 53
4.6.3.1 Example Without Inheritance 53
4.6.3.1.1 Discussion of Example 55
4.6.3.1.2 Expression of Alternate Behaviors 55
4.6.3.2 Example With Inheritance 56
4.7 History and Logging Software Requirements Specification 59
4.7.1 Introduction 59
4.7.1.1 Purpose 59
4.7.1.2 Scope 59
4.7.1.3 Definitions 59
4.7.2 General Description 60
4.7.2.1 Functions 62

TABLE OF CONTENTS (Continued)

4.7.2.2	User Characteristics	62
4.7.2.3	General Constraints	62
4.7.3	Specific Requirements	62
4.7.3.1	Logging	62
4.7.3.1.1	Logging Function Inputs	63
4.7.3.1.2	Logging Function Processing	63
4.7.3.1.3	Logging Function Outputs	63
4.7.3.1.4	Performance Requirements of the Logger Function	64
4.7.3.1.5	Standards	64
4.7.3.1.6	Other Constraints	64
4.7.3.2	Logger Function Options Functional Requirements	64
4.7.3.2.1	Inputs	64
4.7.3.2.2	Processing	64
4.7.3.2.3	Outputs	68
4.8	Style Guide Software Requirements Specification	69
4.8.1	Introduction	69
4.8.1.1	Purpose	69
4.8.1.2	Scope	69
4.8.1.3	Definitions	69
4.8.1.4	Overview	70
4.8.2	General Description	71
4.8.2.1	Product Perspective	71
4.8.2.2	Product Functions	71
4.8.2.3	User Characteristics	72
4.8.2.4	General Constraints	72
4.8.2.5	Assumptions & Dependencies	72
4.8.3	Specific Requirements	72
4.8.3.1	Outline	72
4.8.3.2	Required Topics	72
4.8.3.2.1	System Model	72
4.8.3.2.2	Modes	72
4.8.3.2.3	System Names and Icons	72
4.8.3.2.4	Cursors/Prompts	73
4.8.3.2.5	Status	73
4.8.3.2.6	Feedback/Response Times	73
4.8.3.2.7	Text Manipulation	73
4.8.3.2.8	Menus	73
4.8.3.2.9	Forms	74
4.8.3.2.10	Help	74
4.8.3.2.11	Errors	74
4.8.3.3	Guideline and Design Rule Numbering	75
4.8.3.4	Design Rule Exceptions	75
4.8.3.5	Appendices	75
4.8.3.6	Index	75

LIST OF FIGURES

Figure		Page
1	Real FORM_TERMINAL	15
2	FORM_TERMINAL Emulation	16
3	Logger Function Interface Diagram	60

LIST OF TABLES

Table

I Logger Package Functions...65

1.0 INTRODUCTION

The World Wide Military Command and Control System (WWMCCS) Information System (WIS) Joint Program Management Office (JPMO) requested that the Institute for Defense Analyses (IDA) develop specifications for prototype tools to support the design, implementation, and use of command languages. This report contains Software Requirements Specifications (SRS's) for such tools.

2.0 SCOPE

The specifications were developed by members of a Task Force established by IDA and the IDA staff members listed on the title page of this report. The specifications were prepared over the period of August 1984-January 1986. Due to time constraints and funding limitations, not all of the specifications have been completed to the desired level of detail.

Specifications were developed for the following eight projects:

- CAIS Form On Page Terminal (Section 4.1)
- CAIS Page On Bitmap Terminal (with proposed extensions to the Common Ada Programming Support Environment (APSE) Interface Set (CAIS) for bitmap terminals) (Section 4.2)
- Network Based User Interface Management System (UIMS) (Section 4.3)
- String Based Ada Command Language (SBACL) (Section 4.4)
- Video Based Ada Command Language (VBACL) (Section 4.5)
- Interaction Object UIMS (Section 4.6)
- History And Logging (Section 4.7)
- Style Guide (Section 4.8)

3.0 BACKGROUND

3.1 WIS Foundation Technology Program

This section provides an overview of WWMCCS, WIS, the software foundation studies, the command language task force, and the command logging facility.

3.1.1 WWMCCS

WWMCCS is an arrangement of personnel, equipment, communications, facilities, and procedures employed in planning, directing, coordinating, and controlling the operational activities of U.S. Military forces.

3.1.2 WIS

WIS is responsible for the modernization of WWMCCS Automated Data Processing (ADP) system capabilities, including information reporting systems, procedures, databases and files, terminals and displays, communications, and ADP hardware and software. The WIS

environment is complex, consisting of many local area networks connected via long distance wide area networks. The networks will contain a wide variety of hardware and software and will continue to evolve over many years.

The main functional requirements for WIS can be briefly categorized into seven areas:

a. Threat identification and assessment functions to identify and describe threats to U.S. interests.

b. Resource allocation capabilities for national, theater and supporting levels.

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

d. Detailed planning capabilities to 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 to determine readiness, to direct mobilization, deployment and sustainment at the Joint Chiefs of Staff (JCS) and supported command levels, and to promulgate and report execution and operation orders.

f. Monitoring capabilities to provide information needed to relate political-military situations to:

   (1) National security objectives

   (2) Status of intelligence, operations, logistics, manpower and C3 situations

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

3.1.3 Foundation Software Studies

In order to support these high level objectives, the WIS system software must provide an efficient, extensible and reliable base. To develop such system software, several projects are prototyping foundation technologies for WIS using Ada. The purpose of these prototypes is to produce software components that:

a. Demonstrate the functionality required by WIS.

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

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

IDA is developing specifications for prototypes in the following foundation areas:

a. Command Languages

b. Software Design and Analysis Tools

c. Text Processing

d. Database Tools

e. Operating Systems

f. Planning and Optimization Tools

g. Graphics

h. Network Protocols

3.2 Command Languages

Command languages must provide for a wide variety of user/operator interfaces. The factors affecting any particular user/operator interface include:

a. The type of application (range of systems)

b. The range of users

c. The environment, particulary the terminal features and capabilities and the communications bandwidth between the terminal and the controlling processes

The range of systems includes, at one end, relatively simple data entry applications where user interaction is constrained to input of a specified type and where efforts are made to detect errors at the point of entry. At the other end are general purpose systems which provide the user with a large number of flexible options for interaction with the system, its objects, processes and resources.

The range of users is multidimensional. Currently identified dimensions are system knowledge, frequency of use, and authority. This is complicated by the fact that a highly authorized, frequent, and expert user of one system component may be an infrequent, inexperienced user with minimum authority on another component of the same system.

The range of hardware environments is also quite large. Although it is possible to limit the variety of environments with which command languages must be concerned, this may not be a practical approach. Besides cost considerations, simple terminals may be better suited to some applications than sophisticated terminals. On the other hand, some applications could not be implemented without the capabilities provided by advanced workstations. This is particularly true of graphics applications. Communication capability also affects the design of a command language.

3.2.1 Command Language Tools Requirements

The command language tools should be capable of supporting command languages for the anticipated spectrum of systems, users, and environments. The tools should also separate the user interface from the application program.

There are two types of tools required for the development of command languages: external tools and internal tools. External tools, which exist separately from the command languages, are needed to support the design, development, and implementation of command languages. For the most part, these tools will be those commonly employed in the development of software of all sorts.

Internal tools will be needed specifically for the development of command languages. It is these tools that are of concern in this effort. These tools will constitute a collection of components for use in the construction or implementation of command languages.

Command language tools must cover the full range of systems. They will be needed to provide support for the specification and enforcement of data entry constraints. Tools for interaction with general purpose systems can be quite complex because of the large number and variety of objects as well as numerous actions and options possible. The tools may be further complicated by the need for semantic support of the user interface by the application, for instance, an "UNDO" capability to reverse the last action.

Command language tools should be capable of supporting command languages which are sensitive to the user's current profile relative to the system component being used. They need to support a variety of levels of user help, inquiry, and guidance as well as being capable of promoting interface transparency.

Command language tools must be provided so that the full range of environments can be supported. These tools should, whenever possible, free the application from environmental constraints.

3.2.2 Factors and Issues

Many factors that affect both the specification of tools for command language development as well as application issues were reviewed and discussed in the process of developing specifications. Both groups are further discussed.

3.2.2.1 Issues Affecting the Specification of Command Language Tools

The following issues were considered in the specification of command language tools. The tools specified for WIS must provide sufficient flexibility and power to support any particular resolution of these issues.

a. Comparison of Text, Form, and Iconic Command Languages

   Similarities and differences between text string command languages, form-based command languages, and graphical, iconic, and direct manipulation command languages were considered to determine if there is a mapping among these that can be used to provide a specification independent of style.

b. Menu Selection versus Command Specification

   Benefits and constraints of presentation and selection (menu systems) versus command recall and construction (traditional command languages) were considered to determine the situations where one style would be preferred over the other style.

c. Command Structure

   Various command structures were considered to determine if there is a naturally preferred command structure (for text, form-based, and iconic command languages).

Command structures were considered with regard to specification of command parameter defaults to determine methods for deciding what default values should be used.

d. Command Reconfirmation

Factors that affect the requirements for reconfirmation of a command were considered to determine how command reconfirmation is best managed for various cases.

e. Emergency Usage

The distractions and competing perceptual input that are likely to interfere with command language usage during emergency situations were investigated to determine what measures should be taken in the development of command languages and command language support tools to minimize these factors.

f. Minimal Terminal

User terminals were investigated to determine what minimum features should be required. This includes providing emulation to increase the flexibility of terminals.

g. Application Interface

The effort studied how application programs should interface with a UIMS and the command language interpreter/manager, as well as with command language support tools. This is to determine what underlying semantic support will be required for the user interface (e.g., UNDO) and how such support should be provided.

h. Design Principles

Areas of human factor concerns related to command languages were identified to determine which design principles should be followed when developing command language constructs or capabilities dealing with those areas.

i. System Model

The importance of developing an overall system model that gives the user a conceptual guide to the overall system was discussed. This includes, where possible, a model for subsystems. These should be natural extensions of the overall system model and should maintain compatibility with each other.

The Software Requirement Specification (SRS) should provide guidelines for development of compatible models for use with future sub-systems. These guidelines should be based upon cognitive principles related to human understanding, recall, and memory requirements.

j. Reference Model

The importance of a reference model that shows the logical relation of the components supporting command languages, both within a UIMS and in

relation to other system components as well as application programs was noted. Further discussion of the reference model is in Section 3.2.3.

k. Uniform Style Guide

The importance of a Style Guide to specify uniform screen layout, forms, status display and response, and other formats to be used for consistent command language interaction was noted. The Guide must identify cases and specify usage for when different versions of these items should be used. The Guide should also specify a command method for command line continuation.

l. Standard Names and Icons

The importance of standardizing names for common system commands, facilities, resources, objects, processes, devices, etc was noted. There should also be standard icons for system objects in iconic command languages.

m. Security Management

Areas of access and privilege requiring secure protection were identified.

Various levels of system security such as system access, file protection, and privileged functions were identified and classified.

NOTE: A complete study of the security concerns would also include system requirements for insuring against the violation of security including recording and trace capatilities.

n. Standard/Established Software Models

Specifications were developed to adhere to standards or established software models where appropriate and suitable.

3.2.2.2 Application Issues Affecting Command Language Tools

Application issues were assessed in terms of the following:

a. The use of friendly features such as delaying responses for uniform pacing of interaction, busy and percent completed indicators

b. Automatic invocation of a process with selection of an object

c. Use of keyword versus positional specification of command parameters and options

d. Use and distinction of meta-characters used for command line construction (e.g., command editing) and immediate control commands (e.g., abort current foreground process--- Control/C)

e. Specification of control sequences (e.g., font change) and representation of non-printing characters within a text string

f. Specification of command completion

g. Specification, inquiry, and access to distributed, remote, and local resources

h. Use of conceptual workspaces with user created and pre-defined system objects and functions, both accessible and capable of evaluation from within a command

i. Access to system status, event prompted status display, and constant status

j. Device presentation, specification, and access

k. Representation and specification of data set type (source code, command file, data, etc.)

l. Multiple ownership of data sets; multiple logical locations (e.g., belong to more than one directory) of data sets

3.2.3 Reference Model

A User Interface Management System (UIMS)j provides the basic support and framework for user interaction. In the past user interaction has been handled primarily through text-oriented command languages, consisting of command verbs and optional objective nouns and adverbial or adjectival modifiers. Most command languages have been line-oriented. Several page-oriented languages using forms and menus for command specification have also been developed. More recently, highly interactive graphical (screen-oriented) command languages have been introduced. A reference model for a UIMS must be capable of supporting line-oriented, page-oriented, and screen-oriented command languages.

In addition, the reference model should support the following objectives:

a. Separable: A UIMS should allow the application's user interface to be logically separated from the application's processing and analytical routines.

b. Implementable: A UIMS should interface to the application's processing and analytical routines in a straightforward manner.

c. Portable: A UIMS should allow access to the application's processing and analytical routines accross a variety of workstations.

d. Modifiable: A UIMS should allow changes to occur to the user interface without affecting the application's processing and analytical routines.

e. Consistent: A UIMS should support a consistent user interface across a variety of applications.

f. Capable: A UIMS should support advanced user interface capabilities such as user profiles, history and logging, script/macro definition and help.

A major factor affecting user interaction is response time. This is particularly the case for graphical command languages which often require fairly complex activities to occur in real time. For this reason, it is expected that the major part (if not all) of the WIS UIMS will be located within the user's workstation and remote from the application's processing and analytical routines. This requires the UIMS to support downloading of application-defined interface specifications.

In addition, since the user may be accessing several independent applications simulataneously, the UIMS must be independent of any particular application and should provide support for data transfer between independent applications.

3.2.3.1 Overview

The major components of a UIMS are the UIMS Controller, User Profile, Help Library, Interaction Objects, and Application Interface. Interactions between an application and the UIMS are considered to be a series of transactions. A transaction has two parts: A request for "Service" and a reply which at a minimum indicates the interest the recipient has in providing the requested service. In additon to transactions, an application and the UIMS may communicate with status. Status is simply the provision of gratuitous information. The originator of status receives no reply from the recipient (although acknowledgement of receipt may be provided) and is not necessarily aware of any use the recipient may make of status.

Interaction objects are the primary constituents of the user interface. An interaction object has the responsibility of presenting prompts to the user, accepting and managing user input including editing, and replying to the application with the requested information. Interaction objects may also be used to allow the user to initiate requests for service from the application. The Application Interface is the collection of information used to customize interaction objects. Interaction objects may call upon and interface to other system resources such as visual objects, an image generation package, and the window system.

The UIMS Controller is a high level counterpart to the window system. While the window system manages screen real estate and determines which window to direct input, the UIMS Controller acts somewhat like a traffic cop and ensures that output from a given application is directed to the proper window, and that input directed to a given window is provided to the proper interaction object, and that output from interaction objects are directed to the proper application or "parent" interaction object. The UIMS Controller is also responsible for fulfilling requests to initiate and terminate interaction objects. In many ways, the UIMS Controller may be considered to be the "root interaction object" which provides the primary user interface to the overall distributed system.

The User Profile contains that information which is specific to the user and may be inspected by any of the active objects. The Help Library contains special interaction objects which are called by other interaction objects whenever the user requests help.

3.2.3.2 Transactions

A transaction may be user-originated or application-originated. From the recipients point of view, transactions are initiated as events. A prompt appears on the screen or the application is notified of the occurrence of an event. Event notification to an application must be a strictly defined data set of a universally standard type. Three items of information should be provided.

a. Event Type: Allows the application to use an appropriately typed record to receive the event description.

b. Event Source: Identifies the originator of the transaction.

c. Event Priority: Allows the originator to clamor for attention.

Having received notice of an event, the application can request the event description from the event queue and take appropriate action. The event description is a record containing the information needed to perform the requested "service", which may be anything from a request for information to a request for analysis or modification to the application's model.

A number of different replies may occur. The performance of the service itself may provide the user with a sufficient response. However, it will be necessary to provide a more specific reply to the direct originator [an interaction object] to complete the transaction. Possible replies include the following.

a. Request Initiated and Underway

b. Request Completed

c. Request Not Possible

d. Event Description Not Understood

e. Event Description Incomplete

f. Not Interested in Servicing the Request

A null reply or time out is a possibility which the originator of the transaction should also consider.

The following is an example of how an application-originated transaction would proceed.

a. The application requests input of a given type by activating an interaction object. (Very likely this would be accomplished as a result of a transaction with the UIMS Controller requesting the initiation of the interaction object.)

b. The interaction object then looks at the User Profile to determine if there are any user-specific factors which will affect the interaction. These may range from determining the prompt to be used to providing user specified defaults.

c. The interaction object issues the prompt to the user. (The prompt, among other things, may be the display of a visual object or the modification of an existing visual object.) It may also initiate subordinate interaction objects.

d. The interaction object then samples appropriate logical input devices and inspects queued events and may act upon the input received by:

(1) Initiating a visual object

(2) Terminating a visual object

(3) Modifying a visual object

(4) Initiating a subordinate interaction object

(5) Enabling an independent interaction object

(6) Canceling a subordinate interaction object

(7) Modifying or setting a record item value

(8) Aborting itself (not interested)

(9) Completing itself or some other appropriate action.

e. When completion or abortion occurs, the interaction object performs any necessary cleanup and returns its reply to the application. A user-originated transaction would occur as follows.

(1) The application enables [event type] interaction objects which will "listen" for user input.

(2) The user performs an action which obtains the "attention" of an interaction object.

(3) If the initiating action by itself is not sufficient to complete the transaction request, the interaction object acts on the input and continues to accept input until the transactions request is completed.

(4) On completion of the transaction request, the interaction object notifies the application of the event, stores the event description in the appropriate queue and waits for a reply.

(5) On receipt of a reply or when a time out occurs, the interaction object performs any final actions necessary (which should include notifying the user of the reply) and either resets itself for the next transaction request or terminates.

3.2.3.3 Help Management

Help is provided through a set of help interaction objects which are available to any other interaction object. A help interaction object may be available to more than one interaction object and have subordinate help interaction objects (which is one way to provide levels of help).

When an interaction object receives a request for help, it initiates the appropriate help interaction object. The choice of which one of those available to it may be determined by any appropriate means including the current state of the interaction.

3.2.3.4 Interaction Objects

This UIMS model is heavily dependent upon the concept of interaction objects. Interaction objects are relatively high-level objects embodying semantic capabilities which determine its behavior. Also it is necessary that interaction objects may be downloaded to a remote device or workstation.

Interaction objects would be created by a request to the UIMS Controller. A procedure which would be a necessary part of the interaction object's package would link the application interface information, if any, to the created interactive object. The UIMS Controller would be responsible for insuring that links to basic interaction objects such as the text editor and to other system resources such as a clock are provided. Deletion would occur simply by detaching such links and freeing up the memory.

3.3 Terms and Abbreviations

ADP	Automated Data Processing
APSE	Ada Programming Support Environment
ASCII	American Standard Code for Information Interchange
BNF	Backus Naur Form
CAIS	Common APSE Interface Set
COSCL	Common Operation System Command Language
CODASYL	Conference on Data System Languages
C3	Command, Control, and Communications
EPROM	Erasable PROM
EEPROM	Electronic EPROM
FLAVORS	Object Oriented Programming System for LISP (See LISPMACHINE MANUAL)
GKS	Graphical Kernel System
IEEE	Institute of Electronic and Electrical Engineers, Inc.
IO	Interaction Object
IOU	INTERACTION_OBJECT_UIMS
IPC	Interprocess Communications
JCS	Joint Chiefs of Staff
JOD	Journal of Development, Draft 0.5
MIL-STD-CAIS	Military Standard Common APSE Interface Set
MIXIN	Term used in FLAVORS. Mixes or combines two objects (See LISPMACHINE MANUAL)
MS	Milliseconds
OS	Operating System
OSCRL	Operating Systems Command and Response Language
PDL	Program Design Language
PROM	Programmable Read Only Memory
RAM	Random Access Memory
SBACL	String Based Ada Command Language
SRS	Software Requirements Specification
UI	User Interface
UIMS	User Interface Management System
VBACL	Video Based Ada Command Language
WIS	WWMCCS Information System
WWMCCS	World Wide Military Command and Control System

3.4 References

[188C] MIL-STD-188C Table 1, *American Standard Code for Information Interchange.*

[1815A] U.S. Department of Defense, *Ada Programming Language,* ANSI/MIL-STD-1815A, February 1983.

[CAIS 85] U.S. Department of Defense, *Military Standard Common APSE Interface Set*, MIL-STD-CAIS, January 1985 (Not Approved)

[COSCL 84] *CODASYL - Common Operating Systems Command Language Journal of Development, Version 3.0,* September 1984.

[Fole 83] Foley, J.D. and A. van Dam, *Foundations of Interactive Computer Graphics*, Addison-Wesley, Reading, MA, 1983.

[Fole 1982] Foley J.D. and A. van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA, 1982.

[Gree] M. Green, *The University of Alberta User Interface Management System Design Principles*, Dept. of Computing Science, University of Alberta.

[Hart 1986] H.R. Hartson and D.H. Johnson, "Dialogue Management: New Concepts in Human-Computer Interface Development," *Computing Surveys* (1986), in press.

[Jaco 1983] R.J.K. Jacob, "Using Formal Specifications in the Design of a Human-Computer Interface," *Communication of the ACM* 26 (1983): 259-264.

[Jaco 1985a] R.J.K. Jacob, "An Executable Specification Technique for Describing Human-Computer Interaction," in *Advances in Human-Computer Interaction*, ed. H.R. Hartson, Ablex Publishing Co., Norwood, N.J., 1985.

[Jaco 1985b] R.J.K. Jacob, "A State Transition Diagram Language for Visual Programming," *IEEE Computer* 18, 8 (1985): 51-59.

[Jaco 1986] R.J.K. Jacob, "Direct Manipulation in the Intelligent User Interface," in *Intelligent Interfaces: Theory, Research, and Design*, ed. M.H. Chignell, North-Holland, Amsterdam, 1986, in press.

[Smit UP] S. Smith, *Standards versus Guidelines For Designing User Interface Software*, The MITRE Corporation, Bedford, MA (unpublished).

[Yunt 1985] T. Yunten and H.R. Hartson, "A Supervisory Methodology and Notation (SUPERMAN) for Human-Computer System Development," in *Advances in Human-Computer Interaction*, ed. H.R. Hartson, Ablex Publishing Co., Norwood, N.J., 1985.

4.0 COMAND LANGUAGE SOFTWARE REQUIREMENTS SPECIFICATIONS

4.1 CAIS Form on Page Terminal Software Requirements Specification

4.1.1 Introduction

4.1.1.1 Purpose

The purpose of this section is to describe requirements for a software product to emulate the operation of a form terminal on a page terminal, as defined by [CAIS 85].

4.1.1.2 Scope

The product to be specified will be known as the CAIS_FORM_ON_PAGE_TERMINAL package. This package should provide the full functionality of a CAIS-defined FORM_TERMINAL on a CAIS-defined PAGE_TERMINAL. The package will provide a default set of key strokes that emulate the local editing functions of a form terminal. It will allow the user to modify this set of definitions both under manual and program control, and support an unalterable reset sequence that allows the terminal to be returned to the default configuration. Furthermore, the package will support a quoting mechanism that instructs the emulation software to pass editing key stroke sequences to the underlying application.

This software will be used in a variety of application contexts where data entry and review are critical. The goal of the software package is to permit software written for a form terminal to be accessible to a user with a page terminal. This will allow the CAIS package FORM_TERMINAL to be used as a common development tool usable by all applications. Since a declarative specification of screen layout has been written for the FORM_TERMINAL in Ada, this declarative tool will also be usable for a page-terminal, once the emulation specified in this document is available.

4.1.2 General Description

The product and its requirements are derived from a number of factors, which will be presented in this section of the requirements specification.

4.1.2.1 Product Perspective

A FORM_TERMINAL (e.g., an IBM 307x) supports local editing keys that allow the user to modify the data presented in forms. Examples of the functions these keys invoke are clear field, switch to insert mode, switch to overstrike mode, move to next field, and clear form. When these special keys are struck on a FORM_TERMINAL, the application and the host system have no knowledge of them. Only the final results of the editing is passed through to the application in a block transmission mode when the user strikes an enter key.

A FORM_TERMINAL emulator defines key stroke sequences that invoke the local editing functions on the host, but with no knowledge of the application. The emulator provides a layer of software that intercepts special key stroke sequences, interprets them, and performs the appropriate manipulation on the PAGE_TERMINAL without the application knowing of the interaction. The FORM_TERMINAL emulator thus allows software written for a FORM_TERMINAL to be used on a PAGE_TERMINAL without modification.

From the perspective of application software written for use with a form terminal, calls are made on various functions and procedures in the package FORM_TERMINAL. This

results in forms being presented to the user of the software. As previously mentioned, the user can review the data transmitted, add to it or edit it, and when satisfied with the result, send the revised information back to the application software. The data entry and editing process is handled locally by the terminal and the application software has no knowledge of any of the intermediate results. Figure 1 depicts a typical flow of control.

The emulation software described in this document places another layer of software between the application and the user. The user instructs the system to use the emulator package instead of the FORM_TERMINAL package. The application software calls the same functions and procedures but they are handled by the software in the CAIS_FORM_ON_PAGE_TERMINAL emulator package, which in turn calls software in the PAGE_TERMINAL package. No modifications to the application software is needed to use the emulation package. As previously mentioned, the key strokes designed to perform local editing are intercepted by emulation package and used to update the display on the page terminal. When the user enters the special key stroke sequence to send the revised information to the application, the emulation software constructs a response that is identical to that of the page terminal and transmits it back to the application. Figure 2 presents the flow of control using the emulation software.

4.1.2.2 Product Functions

This package provides the functional capability of a CAIS-defined FORM_TERMINAL for a CAIS-defined PAGE_TERMINAL. The FORM_TERMINAL emulator package must support a user modifiable emulation of FORM_TERMINAL local editing keys. The emulation must also support a quoting mechanism so that key strokes normally used for local editing are transmitted through to the application. A definition of a minimal set of local editing functions and a set of recommended optional functions must be included.

Default settings for the editing key sequences should be specifiable. The definition of the editing keys should be modifiable by the user and by program control. A specific unalterable sequence of key strokes for returning to the default settings is required.

A query key stroke sequence must be supported that will display to the user the current key sequences used for local editing. After reviewing the displav of key bindings produced by the query request, the user should be able to return to the form display prior to the query with no loss of information. How the solution will lead to an emulation with acceptable response times will also be shown.

In order to provide the emulation capabilities the package must support all of the type, subtypes, functions and procedures of the FORM_TERMINAL package. These can be found in the [CAIS 85].

In addition, the responder to this request will be responsible for defining additional types, subtypes, functions and procedures as is necessary to provide the required functional capabilities. In particular, data types and functions to support the following functions and procedures are required:

a. A reset procedure, which when called tells the emulation package to use the default settings for local editing keys.

b. A query function, which when called returns the current definition of the local editing keys.

c. A manual procedure to establish a set of local editing keys.

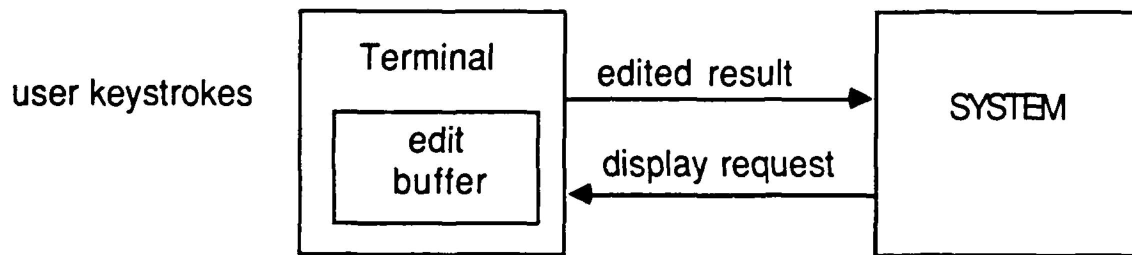


Figure 1. Real FORM_TERMINAL

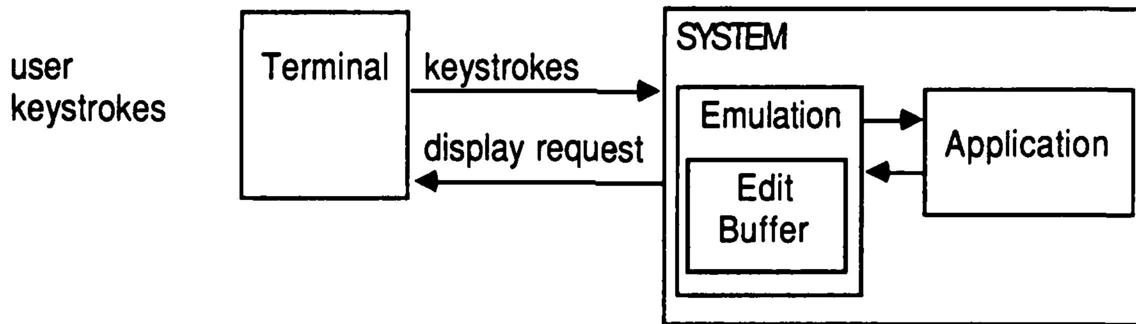


Figure 2. FORM_TERMINAL EMULATION

d. Procedures to establish a set of local editing keys under program control.

e. A pair of procedures to commence and terminate a logging of key strokes handled by the emulation package.

The purpose for these functions and procedures is to allow software to be built that uses the underlying application software without a user having to actually interact with it.

4.1.2.3 User Characteristics

This emulation software will be used by a wide range of users. Data entry clerks, software development professionals, military commanders, and office workers are just a few of the user communities that are anticipated.

The skill and training levels of the users will very widely. It can be assumed that all users will understand the operation of a page terminal and a form terminal and that they will have received some instruction in the use of a form terminal. Specific instruction in the use of the emulation package can be assumed, but minimization of such instruction is a requirement of the emulation package.

4.1.2.4 General Constraints

The software sought must correspond to government regulations concerning the use of Ada as established in [1815A]; all software must be written in standard Ada.

The software must also comply with the most current revision of the [CAIS 85] at the time of submittal. Comply means that the emulation package will support all of the types, subtypes, functions, and procedures in the package, FORM_TERMINAL. It also means that all input and output to and from the terminal will be through the use of the types, subtypes, functions, and procedures specified in the CAIS-defined package, PAGE_TERMINAL.

4.1.3 Specific Requirements

The software package must comply with the functional requirements specified in the following sections.

4.1.3.1 Functional Requirements

The functional requirements consist of support for local editing keys, the emulation of the CAIS FORM_TERMINAL, a procedural equivalent for keyboard entry of editing key stroke sequences, and input/output logging facility.

4.1.3.1.1 Local Editing Keys

The first requirement for the software product is support for the definition and use of key stroke sequences that emulate the local editing keys of a form terminal.

Introduction

This is the basic functional capability that the users of computer systems will see. Users who have access to page terminals will be able to interact with software written for form

terminals. This will provide a uniform interaction paradigm, that of electronic form filling, for the most prevalent classes of terminals, the form terminal and the page terminal.

Inputs

The sources of inputs for this requirement are key strokes supplied by the user. These will be made accessible to the emulation package through the capabilities in the CAIS-defined PAGE_TERMINAL package. The characters available are any of those defined by the ASCII standard character set. It can be assumed that the page terminal in use has the standard character keyboard.

Processing

Input key stroke sequences fall into the following categories and require the processing indicated.

a. An unalterable reset sequence, which when identified by the emulation software will result in setting the table of local editing keys to a predefined sequence.

b. A capability to package keystroke sequences into a local editing function callable by a sequence of function-keys. The local editing function can alter the keyboard semantics dynamically. Local editing functions can either be created interactively and saved, or loaded from a file.

c. A quoting sequence which, when identified, will pass the characters following it through to the application without further interpretation by the emulation software.

d. A query sequence which, when identified will display the current table of local editing keys.

e. Local editing key stroke sequences which, when identified perform the editing operations on the displayed information.

f. Normal key strokes which are buffered and passed to the application software in a manner prescribed by the CAIS package, FORM_TERMINAL.

The set of local editing functions to be supported by the supplier of the package should be broken down into a minimal set and a set of recommended extensions. A recommendation should be included regarding the interaction between the user and the system that follows from the recognition of a keystroke sequence in any of the categories above.

Outputs

The outputs of the package will be identical to those of a FORM_TERMINAL as specified in the [CAIS 85].

4.1.3.1.2 CAIS FORM_TERMINAL Emulation

This requirement is derived from the definition of the FORM_TERMINAL package in [CAIS 85]. The official definition of the package is however to be found in [CAIS 85].

Introduction

Satisfaction of this requirement will permit software that was designed to be used with a form terminal to be used with a page terminal without any modifications. This means application software will require no modification to work with either class of terminal.

Inputs

The inputs for this requirement are as specified in MIL-STD-CAIS document.

Processing

The processing is as specified in the [CAIS 85].

Outputs

The outputs of this requirement are as specified in [CAIS 85].

4.1.3.1.3 Procedural Interface to Editing Key Stroke Sequence Definition

Procedures defining key stoke sequences for local editing functions provide a simple mechanism for initializing configurations for users.

Introduction

This requirement consists of a procedure or procedures to set the key stroke sequences that are emulation sensitive and to reset the software to the default settings. It should also be useful in terms of providing the save and restore capabilities mentioned in Section 3.1.1.

Inputs

Definitions should be included of data types and subtypes necessary to describe key stroke sequences and their interpretations. These data types will serve as the inputs to the functional definition of key stroke sequences.

The procedure to reset the configuration of the key stroke sequences is an exception in that it has no input requirement.

Processing

When a key stroke definition is specified, the software will respond by modifying the key stroke definition table to provide the correct interpretation of key stroke sequences.

Outputs

Output will be resetting of keystroke definition table, and the results will be observable by query as indicated in Section 4.1.3.1.1.3, d.

4.1.3.1.4 Logging Facilities

A facility to record each key stroke is part of the required software.

Introduction

This requirement results from a desire to provide a facility for analyzing user interaction, providing audit functions, and debugging capabilities. It consists of three capabilities.

a. The capability to initiate or continue logging

b. The capability to suspend logging

c. The capability to terminate logging

Logging of input and output must be supported and each should be under independent control with regards to initiation, suspension, resumption and termination. Despite independent control, the log file should be shared and marked as to whether the characters are inputs or outputs to the terminal.

Inputs

The inputs to the logging software will be a record of characters presented or entered using the data types defined for characters in the PAGE_TERMINAL package of the CAIS.

Processing

The emulation software, which must handle all characters typed by the user or sent to the terminal, will record all input or output characters (as appropriate depending on whether one or both are activated). A request to initiate logging will open the required file and start placing the input or output characters into that file. Suspension of the facility will maintain the position in the file and stop logging until a resume or terminate command is given. A resume command will cause characters following to be recorded at the position of suspension. Termination of both input and output logging will close the file.

Outputs

The outputs of the software to support this requirement are file nodes as defined by [CAIS 85].

4.1.3.2 External Interface

The external interface to this package consists of a user interface and a hardware interface.

4.1.3.2.1 User Interface

The software proposed must include the design of a user interface that simplifies operation. Section 4.1.3.3 will outline the performance requirements of the user interface.

4.1.3.2.2 Hardware Interface

The software should be designed to work with a wide variety of commercially available page terminals. Standard ASCII characters and keyboards can be assumed to be the minimum hardware configuration of the terminal. Communications speeds will vary from tens of characters per second to hundreds of characters per second.

4.1.3.3 Performance Requirements

Flexibility, trainability, and responsiveness are key characteristics upon which the user interface will be evaluated. Flexibility refers to the simplicity of the procedure to alter defined local editing key stroke sequences and suitability of the design for a range of keyboards (i.e., ranging from only the standard keys to those with programmable function keys and cursor movement key pads.) Trainability refers to the amount of time it would take a user to understand the operation of the software. Responsiveness refers to the time it would take the software package to respond to the editing requests of the user. The responsiveness of the system over both low speed and high speed connections should be taken into account. Considerations about the number of key strokes and the ease of entry are also of concern under the topic of responsiveness. Proposed solutions must indicate how each of these user interface issues are addressed.

4.1.3.4 Design Constraints

The software will be implemented in Ada and consistent with [CAIS 85], as indicated throughout this document.

4.1.3.5 Attributes

The software must be machine and terminal independent relying on the use of character and keyboard standards as well as [1815A] and [CAIS 85].

4.2 CAIS PAGE ON BITMAP TERMINAL Software Requirements Specification

4.2.1 Introduction

4.2.1.1 Purpose

The purpose of this section is to describe requirements for a software product to emulate the operation of a PAGE_TERMINAL, as defined by [CAIS 85], on a bitmap terminal. The bitmap terminal will heavily depend on the functionality provided by a graphics Window Manager. The software product will be used as part of the WIS.

4.2.1.2 Scope

The product to be developed will be known as the CAIS_PAGE_ON_BITMAP_TERMINAL package. This package should provide the full functionality of a CAIS-defined PAGE_TERMINAL on a bitmap terminal. When combined with the previously defined CAIS_FORM_ON_PAGE_TERMINAL package, the result should include the ability to emulate both form and PAGE_TERMINALs on a bitmap terminal.

The package must provide a multi-window capability. Each window should be treated as a separate terminal. The package must include support software that allows the user to initiate the process of assigning the terminal emulator to a logical device. The initialization process should be designed to have minimal dependencies on functionality not defined by [CAIS 85].

This software will be used in a variety of application contexts where data entry and review are critical. The goal of the software package is to permit software written for a PAGE_TERMINAL, including form terminal emulation software, to be accessible to a user with a bitmap terminal. This will allow the CAIS package PAGE_TERMINAL to be used as a common development tool usable by all applications.

4.2.2 General Description

The product and its requirements are derived from a number of factors, which will be presented in this section of the requirements specification.

4.2.2.1 Product Perspective

A bitmap terminal provides multiple windows that the user can shape and place on the screen of a high resolution display. The terminal also includes a pointing device that allows the user to focus the attention of the system on some location on the screen.

A PAGE_TERMINAL is the most common form of operator terminal available for most data processing tasks. Most software developed today is intended for use with such a device. To make this software usable on a bitmap terminal, an emulation package is required so that calls to CAIS software to drive the PAGE_TERMINAL perform the corresponding correct operations on the bitmap terminal.

A PAGE_TERMINAL emulator must support certain parameters that define characteristics of the terminal being emulated. The size of the terminal to emulate in terms of rows and columns is one of these parameters. The position of the emulation window on the display screen is another. The graphics renditions to support is also a parameter. The final parameter is the number of virtual function keys to support and the labels for them.

While the emulation package must support the functions that are physically present on the keyboard of the bitmap terminal, it must also support virtual function keys which can be displayed graphically. Graphics support consists of displaying a menu on the bitmap screen each element of which represents a single function key. These function keys may be "struck" by selecting them with the pointing device.

From the perspective of application software written for use with a PAGE_TERMINAL, calls are made on various functions and procedures in the CAIS package PAGE_TERMINAL.

The emulation software described in this document places another layer of software between the application and the user. The user instructs the system to use the emulator package instead of the PAGE_TERMINAL package. The application software calls the same functions and procedures but they are handled by the software in the CAIS_PAGE_ON_BITMAP_TERMINAL package, which in turn calls software in the BITMAP_TERMINAL package. No modifications to the application software is needed to use the emulation package.

4.2.2.2 Product Functions

This package provides the functional capability of a CAIS-defined PAGE_TERMINAL for a bitmap terminal.

The emulation package must support an initialization process that directs the environment to treat the emulation as a logical device in the system. The emulation must also support parametric settings of the capabilities of the PAGE_TERMINAL being emulated. These include graphic rendition support, size, and virtual function key support. User defined defaults for these parameters should be supported. How the solution will lead to an emulation with acceptable response times must be shown as well.

In order to provide the emulation capabilities the package must support all of the types, subtypes, functions and procedures of the PAGE_TERMINAL package. These can be found in [CAIS 85].

In addition, definitions must be generated for additional types, subtypes, functions and procedures as is necessary to provide the required functional capabilities. In particular, data types and functions to support the following functions and procedures are required:

a. An initialization procedure to create the emulation window and inform the system about the existence of the new logical device

b. Support for virtual function keys displayed as menus

c. Parametric setting of emulation parameters

d. A termination procedure to remove the window from the display and to instruct the APSE that the logical device associated with the window is no longer available

4.2.2.3 User Characteristics

This emulation software will be used by a wide range of users. Data entry clerks, software development professionals, military commanders, and office workers are just a few of the user communities that are anticipated.

The skill and training levels of the users will vary widely. It can be assumed that all users will understand the operation of a PAGE_TERMINAL and a bitmap terminal and that they will have received some instruction in the use of both. Specific instruction in the use of the emulation package can be assumed, but minimization of such instruction is a requirement of the emulation package.

4.2.2.4 General Constraints

The software sought must correspond to government regulations concerning the use of [1815A]; all software must be written in standard Ada.

The software must also comply with the most current revision of [CAIS 85] at the time of submittal and the definition of the bitmap terminal package as defined in this report. Comply means that the emulation package will support all of the types, subtypes, functions, and procedures in the package PAGE_TERMINAL. It also means that all input and output to and from the terminal will be through the use of the types, subtypes, functions, and procedures specified in a BITMAP_TERMINAL package.

4.2.3 Specific Requirements

The software package must comply with the functional requirements specified in the following sections.

4.2.3.1 Functional Requirements

The functional requirements consist of support for initialization and termination of emulation, the actual emulation of the PAGE_TERMINAL, and input/output logging facility.

4.2.3.1.1 Initialization and Termination

The first requirement for the software product is support for the initialization and termination of the emulation session.

Introduction

These are the basic procedures used to create and destroy terminal emulator windows. Parameterization of the initiation will allow the user to determine the size of the emulation window, the graphic rendition support, and the number of virtual function keys to support graphically.

Additionally, the placement of the window on the display and the input terminal file and output terminal file must be specified during initialization. Termination of the input or output file may be achieved separately.

Inputs

The sources of inputs for this requirement are parameters supplied by the user in an Ada function call. The data types associated with these parameters must be those defined in [CAIS 85] document. An additional data type must be defined to support the labeling of virtual function keys displayed on the bitmap terminal.

Processing

On initialization the emulation software will create a window of proper size at the specified location with input and output terminal files properly assigned.

The termination procedure will allow separate termination of input and output terminal capabilities for an emulation window. Termination of both input and output will result in removal of the emulation window from the display.

Outputs

The outputs of these procedures will be acknowledgement of successful completion. Exception conditions will include not enough space to place a window of the required size at the required position and errors resulting from the assignment of logical devices. Other exception conditions as needed can be developed as part of the proposed effort.

4.2.3.1.2 CAIS PAGE_TERMINAL EMULATION

This requirement is derived from the definition of the PAGE_TERMINAL package in [CAIS 85].

Introduction

Satisfaction of this requirement will permit software that was designed to be used with a PAGE_TERMINAL to be used with a bitmap terminal without any modifications. This means application software will require no modification to work with either class of terminal.

Inputs

The inputs for this requirement are as specified in [CAIS 85].

Processing

The processing is as specified in [CAIS 85].

Outputs

The outputs of this requirement are as specified in [CAIS 85].

4.2.3.1.3 Virtual Function Key Support

The bitmap terminal's capabilities allow the emulation of function keys. Virtual function keys can be displayed on the bitmap terminal for selection by the user.

Introduction

This software will display menus of virtual function keys that the user may select with the pointing device. Selection of one of these menu items will result in behavior that emulates the striking of a function key on a PAGE_TERMINAL.

Inputs

The input to this support is user performed selection gestures. The graphic pointing device of the bitmap terminal is used to focus the attention of the system on one of the menu items. A pointing device selection gesture effectively strikes the corresponding function key.

Processing

When a menu item is selected, the software buffers the proper function key specification for transmittal to application software. The format and structure of the function key descriptions can be found in [CAIS 85] as part of the description of a PAGE_TERMINAL.

Outputs

This support has outputs that match the function key support described for the PAGE_TERMINAL package.

4.2.3.2 External Interface

The external interface to this package consists of a user interface and a hardware interface.

4.2.3.2.1 User Interface

The software proposed must include the design of a user interface that simplifies operation. Section 4.2.3.3 will outline the performance requirements of the user interface.

4.2.3.2.2 Hardware Interface

The software should be designed to work with a wide variety of commercially available bitmap terminals. Standard ASCII characters and keyboards can be assumed to be the minimum hardware configuration of the terminal. Communications speeds will vary from tens of characters per second to hundreds of characters per second. Pointing devices will typically be mice with at least one button, but others ought to be accounted for. The resolution of the display will range from several hundred to several thousand pixels in each dimension.

4.2.3.3 Performance Requirements

Flexibility, trainability, and responsiveness are key characteristics upon which the user interface will be evaluated. Flexibility refers to the simplicity of the procedure to use the emulation software in a variety of ways. Trainability refers to the amount of time it would take a user to understand the operation of the software. Responsiveness refers to the time it would take the software package to respond to input, output, and initialization and termination requests. The designer should take into account the responsiveness of the system over both low-speed and high-speed connections. Proposed solutions must indicate how each of these user interface issues are addressed.

4.2.3.4 Design Constraints

The software will be implemented in Ada and compliant with [CAIS 85].

4.2.3.5 Attributes

The software must be machine and terminal independent relying on the use of character and keyboard standards as well as [1815A] and [CAIS 85].

4.2.4 Proposed CAIS Extension For Bitmap Terminal

The CAIS does not specify many of the functions required to suport bitmapped terminals. This section provides a preliminary description of a standard interface.

In general, the set of functions described in this section provide sufficient primitives to implement the ideas of Section 4.2.3.1. Additional functionality can be built up on top of this set since everything is not included at this level. This level is essentially a device driver for a terminal which knows about windows.

One global, mouse-tracking cursor is provided but no support is provided for individual window cursors. Since the package supports windows, it is appropriate for it to provide the cursor that transcends window boundaries. The local, within-window cursors can be provided by higher-level software packages using PUT_RASTER to place their images in the windows.

Each window has several ways in which it can be presented on the screen: conventionally, as an icon, or in various other forms such as a full-screen format, reduced format, color versus monochrome, etc. Each VIEW remembers data sent to it, so that you can switch from one VIEW of a WINDOW to another and have them remember what was written to them. Where applicable, the procedures allow the user to specify on which VIEW of the WINDOW you are operating.

```
type POSITION_TYPE is CAIS.IO_DEFINITIONS.POSITION_TYPE;
     -- We will express positions in terms of coordinates in virtual
     -- bitmap units, which need not be the same as those of
     -- any real screen

type FILE_TYPE is CAIS.IO_DEFINITIONS.FILE_TYPE;

type BITMAP_TYPE is BITMAP_PACKAGE.BITMAP_TYPE;
     --imported from a yet-to-be-written package

type SCREEN_REGION_TYPE is record
    TOPLEFT: POSITION_TYPE;
    BOTRIGHT: POSITION_TYPE;
end record;

type WINDOW_TYPE is record
     null; -- whose contents are private to this package
end record;

type VIEW_TYPE is (NORMAL, ICON, FULL, SMALL, LARGE, etc...);

type INPUT_EVENT_TYPE is record
    null;
     -- information to identify an input event
     -- keyboard key or other event name
     -- time stamp
end record;
```

```
function TERMINAL_SIZE(TERMINAL: in FILE_TYPE)
    return POSITION_TYPE;
     -- tells the size in raster coordinates of the whole screen

function MAXIMUM_FUNCTION_KEY(TERMINAL: in FILE_TYPE)
    return NATURAL;
     -- may also want functions to tell if have mouse
     -- and how many buttons it has

function CREATE(
    TERMINAL: in FILE_TYPE;
    SCREEN_REGION: in SCREEN_REGION_TYPE)
    return WINDOW_TYPE;
     -- creates a window on TERMINAL and returns a handle that can be used
     -- to refer to the window subsequently
     -- also does a SET_VIEW(this window, NORMAL)
     -- also does SET_REGION for all views of this window to
        SCREEN_REGION
     -- Note that the returned WINDOW_TYPE value is unique across terminals,
     -- so subsequent functions that refer only to the WINDOW
     -- implicitly identify the TERMINAL

procedure DESTROY(WINDOW: in WINDOW_TYPE);

procedure PUT_GRAPHICS(
    WINDOW: in WINDOW_TYPE;
    VIEW: in VIEW_TYPE := NORMAL;
    POSITION: in POSITION_TYPE;
     Thing to write: in);
     -- Writes graphic object into this VIEW of this WINDOW
     -- this will be refined into point, line, circle, etc
     -- in coordination with graphics task force.

procedure PUT_RASTER(
    WINDOW: in WINDOW_TYPE;
    VIEW: in VIEW_TYPE := NORMAL;
    POSITION: in POSITION_TYPE;
    RASTEROP: in RASTEROP_TYPE;
    BITMAP: in BITMAP_TYPE);
     -- BITMAP will be clipped to fit inside the WINDOW
     -- In addition, we may want to add parameters for:
     -- a specific region of the window
     -- (BITMAP would then be clipped to that region)
     -- an X,Y offset into BITMAP
     -- repetition of BITMAP to fill a given screen area
     -- which is larger than BITMAP itself (to draw textures)
     -- a predefined null BITMAP for painting areas fixed colors
     -- or inverting, etc.

procedure PUT_TEXT(
    WINDOW: in WINDOW_TYPE;
    VIEW: in VIEW_TYPE := NORMAL;
    POSITION: in POSITION_TYPE;
    TEXT: in STRING;
     FONT: in Font descriptor?;
     ANGLE: in Orientation angle?);

procedure MOVE_CURSOR(
    WINDOW: in WINDOW_TYPE;
    VIEW: in VIEW_TYPE := NORMAL;
    POSITION: in POSITION_TYPE);
     -- moves the global (mouse) cursor to POSITION
     -- measured with respect to this WINDOW's VIEW's coordinates
```

```
procedure SET_CURSOR(
    WINDOW: in WINDOW_TYPE;
    VIEW: in VIEW_TYPE := NORMAL;
    SHAPE: in BITMAP_TYPE);
    -- Makes the mouse cursor this SHAPE
    -- whenever the cursor is in this WINDOW
    -- and WINDOW is being displayed with this VIEW

procedure SET_REGION(
    WINDOW: in WINDOW_TYPE;
    VIEW: in VIEW_TYPE := NORMAL;
    SCREEN_REGION: in SCREEN_REGION_TYPE);
    -- moves VIEW of WINDOW to SCREEN_REGION

function GET_REGION(
    WINDOW: in WINDOW_TYPE;
    VIEW: in VIEW_TYPE := NORMAL);
    return SCREEN_REGION_TYPE;

procedure TOP(WINDOW: in WINDOW_TYPE);
     -- pops overlapping window to top of screen
     -- affects whatever view is being displayed

procedure BOTTOM(WINDOW: in WINDOW_TYPE);
     -- Note may in the future want more sophisticated
     -- manipulations of ordering than just top and bottom

procedure SET_VIEW(
    WINDOW: in WINDOW_TYPE;
    VIEW: in VIEW_TYPE);
    -- sets WINDOW so that it is displayed according to VIEW

function GET_VIEW(WINDOW: in WINDOW_TYPE)
    return VIEW_TYPE;
    -- tells current VIEW being displayed for this WINDOW

procedure ACTIVATE_INPUT(
    WINDOW: in WINDOW_TYPE;
    KIND: in Description of class of input := ALL);
    -- makes WINDOW the recipient of all input events in KIND
    -- applies to all VIEWs of WINDOW

function GET(WINDOW: in WINDOW_TYPE)
    return INPUT_EVENT_TYPE;
    -- return the next input event of which WINDOW is recipient
```

4.3 Network Based User Interface Management System (UIMS) (NBU) Software Requirements Specification

4.3.1 Introduction

4.3.1.1 Purpose

The most important component for a successful WIS is the design and implementation of a robust UIMS. The UIMS modules/packages develop a common (across systems) interface between users and system facilities/capabilities. These modules perform the following:

a. Develop and use application specific command languages employing multiple options for program control and data input.

b. Employ Ada statements and package calls.

c. Create and execute libraries of Ada packages.

d. Provide error message, help, logging, and error recovery based on the application context.

The UIMS is a set of tools, guides, and building block packages to assist in developing WIS user interfaces. The design goal is to provide basic reusable packages that may be customized for specific applications while reducing development time and encouraging consistency.

The goal of this project is to produce a UIMS based on a state transition diagram specification language. The *UIMS will* comprise *a set of tools for defining the grammar* of the interaction between a system and its user. They will all be based on a common description language for describing user interfaces, which will be developed as part of this project. That language is in turn based on state transition diagrams or augmented transition networks. Dialogue descriptions written in this language will be interpreted or compiled to provide an interaction environment or run-time UIMS that implements the described user interface.

UIMS tools must be capable of supporting the definition and implementation of a string based, video based and interaction based Ada command language (Sections 4.4, 4.5 and 4.6).

4.3.1.2 Scope

The UIMS and its description language are oriented principally toward single-thread dialogue command languages. The tools developed in this project shall support the development of specifications for such languages and the execution of such languages, including runtime support for execution.

The code for the tools themselves shall be developed using an approved Ada Program Design Language (PDL) and shall include internal and user documentation, design notes, testing requirements, and sample test drivers, any special services required, interfaces to other WIS packages as needed (e.g., database, operating system). In particular, the WIS database package shall be used for all data base requirements within this project.

4.3.1.3 Definitions

Application program author: Writes the internal routines that perform application specific tasks, in contrast to the dialogue author, who designs the externally-visible user interface.

Command language: Syntax of commands entered by user and responses generated by system.

Description language: Notation for describing the syntax of a command language.

Dialogue author: Designs the syntax of a user interface, in contrast to the application program author, who designs the semantics.

End user: WIS system user who uses the system to perform military applications, in contrast to the application program author and the dialogue author, who develop the system.

Lexical level: Describes how input and output tokens are actually formed from primitive hardware operations

Semantic level: Describes the functions performed by the system. This corresponds to a description of the functional requirements of the system, but does not address how the user will invoke the functions.

Syntactic level: Describes the sequences of inputs and outputs necessary to invoke the functions described in the semantic level.

User interface management system (UIMS): A distinct software component that conducts all interactions with the user. It is separate from the application program that performs the underlying task.

4.3.2 General Description

4.3.2.1 Product Perspective

This project shall develop a set of Ada programs. The programs shall operate as self contained, programming tools, each able to stand alone. They shall also be designed to be compatible with one another, so that they can be used together when desired.

The language and tools developed in this project shall support the concept of a UIMS. A UIMS is a distinct software component that conducts all interactions with the user. It is separate from the application program that performs the underlying task. It is analogous to a database management system in that it separates a function used by many applications and moves it to a shared subsystem. It removes the problem of programming the user interface from each individual application and permits some of the effort of designing tools for man-machine interaction to be amortized over many applications and shared by them. It also encourages the design of consistent user interfaces to different systems, since they share the user interface component. Conversely, it permits "dialogue independence," where changes can be made to the dialogue design without affecting the application code. [Hart 1986]

Where it is necessary to determine whether a particular function belongs in the UIMS or the application code, the following guidelines shall be observed:

The split of functions into User Interface (UI) and Application is a division of the program into software modules, that is, assignments of programming tasks to people, i.e., who writes which module and who maintains it. On which processor a function is ultimately executed or possibly downloaded is not of concern in making this split. In order to apply the philosophy that underlies UIMS, it is necessary to divide the overall design of a system into two pieces that can be programmed separately, by different kinds of people. The UI is designed by a knowledgeable user interface designer, possibly a cognitive psychologist or human factors engineer with limited programming skill. The Application is designed by an expert computer programmer who knows about system design, architecture, efficiency considerations, good coding, etc., but not necessarily about good human-computer dialogues.

This, then, provides a specific criterion for deciding what code goes into the UI versus the Application. The split is determined by what kind of person should be responsible for each coding task. It is important that every detail that relates to the design of the user dialogue be in the hands of the programmer of the UI module (the dialogue designer). He must control what happens and when it happens so that he can design a good, consistent user interface and not have unexpected events be presented to the user. Conversely, anything to do with semantics, particularly any code that knows about application data types, must be exclusively in the hands of the application programmer.

4.3.2.2 Product Functions

The tools developed in this project shall provide the following functions:

a. Interactive graphical editing of the state transition diagram-based dialogue description language. A dialogue designer shall be able to enter, modify, and examine a complete dialogue description, including links to diagrams called as subroutines.

b. Translation between the description language and alternative forms (e.g., to or from graphical representations of the diagrams).

c. Extracting information for user help messages and for portions of user manual directly from the dialogue description.

d. Accepting a description of the syntax of the command language, expressed in the description language, and acting as an interpreter for that command language.

e. Interactive debugging of a dialogue description, during execution of that description.

f. Creation and use of generic input and output tokens.

g. Other support functions for input and output, as needed to be called from the dialogue description language to provide low level functions.

h. Parsing the dialogue description language into a form suitable for use in building new tools that use this language.

4.3.2.3 User Characteristics

The systems developed shall support and differentiate three individual roles. The first is the application program author, who writes the internal routines that perform application specific tasks. Next, the dialogue author designs the syntax of the user interface (but not the semantics). He will be the principal user of the language and tools produced by this effort. Finally, the end user uses the combined system to achieve desired results.[Yunt 1985]

4.3.2.4 General Constraints

All programs shall be written in Ada.

The WIS database package shall be used for all data base requirements within this project. The Graphics Kernel System (GKS) shall be used for all graphics requirements. Where needed, operating system services shall be invoked via calls to functionality as provided in [CAIS 85].

4.3.3 Specific Requirements

4.3.3.1 Functional Requirements

4.3.3.1.1 Description Language

A specific language for describing command languages shall be developed, using state transition diagrams and following the concepts outlined in this section.

While the project itself is oriented principally toward single-thread dialogue command languages, the dialogue description language shall be designed with the possibility of extensions to cover multi-thread and direct manipulation, object-oriented interfaces in mind. An approach for achieving this goal is provided in [Jaco 1986].

4.3.3.1.1.1 General Principles

The principal purpose of the dialogue description language is to describe a dialogue explicitly from the user's point of view, as contrasted with the programmer's. In designing this language, therefore, the user's view shall be stressed at all times. For this reason, a language based on state transition diagrams is preferred, as it makes the user visible time sequence of a dialogue explicit. Time sequence is an important aspect of the surface structure of an interactive system as seen by a user. Specifications based on state transition diagrams are particularly suitable for describing interactive human-computer interfaces because they represent time sequences explicitly, in contrast to BNF, in particular, where sequence is implicit. They are useful even for many purportedly "modeless" user interfaces. The state transition model is also helpful in depicting a user's mental model of an interactive computer system.

The conceptual model underlying the language should similarly be as simple as possible, and should be based on simple constructs. Again, a language based on state transition diagrams is preferred for this purpose to one based on constructs such as context free grammars or Petri nets. The use of constructs that are understood chiefly by programmers rather than users should generally be avoided, even though they might make the language more compact.

The user interface to the description language tools shall be graphical if possible. A textual representation of the language should of course be available, but the primary interaction ought to be through a graphic editor. If possible, the tools should also supply an interactive graphic debugging facility.

The description language shall possess the following additional properties:

a. The description of a user interface should be easy to understand. In particular, it must be easier to understand (and take less effort to produce) than the software that implements the user interface.

b. The specification should be precise. It should leave no doubt as to the behavior of the system for each possible input.

c. It should be easy to check for consistency.

d. The specification technique should be powerful enough to express nontrivial system behavior with a minimum of complexity.

e. It should separate what the system does (function) from how it does it (implementation). The technique should make it possible to describe the behavior of a user interface, without constraining the way in which it will be implemented.

f. It should be possible to construct a prototype of the system directly from the specification of the user interface.

g. The structure of the specification should be closely related to the user's mental model of the system itself. That is, its principal constructs should represent concepts that will be meaningful to users (such as answering a message or examining a file), rather than internal constructs required by the specification language.

4.3.3.1.1.2 Specific Characteristics of the Language

The dialogue description language shall follow the notation and principles given in [Jaco 1985a].

A state transition diagram consists of a set of states and transitions. The start state and end state of a diagram are distinguished from the remaining states. Each transition between two states may be labeled with one of the following:

a. An input token

b. An output token

c. A nonterminal diagram

d. A function call representing a semantic action

e. A call to semantic function that returns a Boolean condition

Tbe Boolean condition must be true for this transition to be taken. Each nonterminal symbol is defined in a separate state diagram, which may be called like a subroutine from a

transition and must be traversed from start to end to complete that transition. If the called nonterminal has more than one exit state, the transition that calls it will also have several end states, one for each of the exits in that nonterminal. Each diagram has a name, which is the symbol by which it may be called as a nonterminal from another diagram.

The description language must support a clear decomposition of the user interface description into semantic, syntactic, and lexical components [Fole 1982]. A specific notation suitable for each of the three levels of the description shall be provided. The semantic level describes the functions performed by the system. This corresponds to a description of the functional requirements of the system, but does not address how the user will invoke the functions. The syntactic level describes the sequences of inputs and outputs necessary to invoke the functions described. The lexical level determines how the inputs and outputs are actually formed from primitive hardware operations.

The language shall also support a modular decomposition of the design of a user interface. Nonterminal diagrams, which can be called like subroutines, will aid in this decomposition. The language shall also support a process of stepwise refinement of the syntactic specification that leads from an informal specification to a formal, executable one.

The language shall provide a means of specifying generic input and output tokens. These will be described by the dialogue designer in the dialogue description language once, and then instantiated as needed with specific parameters supplied. They will be used as a convenient way of describing input or output operations which are largely similar.

In addition, certain operations will be found to pervade many states such as help keys, abort keys, and input line editing operations. The descriptive language shall permit the definition of "kernel" or "mixin" states in order to make these operations easier to specify. A kernel state is a named collection of state transitions leading out of a state. One or more kernel states can then be associated with sets of actual states in the dialogue description without the need to repeat their full specifications. An actual state can combine the behaviors of several such kernel states in a manner similar to the inheritance of "flavors" in LISP. The kernel state mechanism shall be designed to help a dialogue designer ensure agreement with the WIS Command Language Style Guide.

Extensions to the dialogue description language needed to cover multi-thread and direct manipulation, and object-oriented dialogues shall be investigated. In particular, the use of diagrams called as co-routines rather than subroutines as a means for describing such dialogues in the present language shall be studied.[Jaco 1986]

4.3.3.1.2 Dialogue Description Editor

Introduction

This tool shall provide an interactive graphical editor for state transition diagrams. It shall allow a dialogue designer to enter, modify, and examine a complete dialogue description, including links to diagrams called as subroutines.

Inputs

Inputs are interactive editing commands, applied to a displayed state transition diagram.

Processing

The system shall accept editing commands, update, and redisplay the displayed diagrams, and translate and store the diagrams in an internal representation common to all the tools built in this project.

Outputs

Outputs include an interactive display of state transition diagram to user and finished state transition diagram translated to internal representation.

4.3.3.1.3 Dialogue Specification Translator

Introduction

This tool shall translate between the description language and alternative forms (e.g., to or from graphical representations of the diagrams).

Inputs

Inputs are state transition diagrams in graphical, text, or internal representation.

Processing

Processing translates a state transition diagram in any of the three input representations into any other of those representations.

Outputs

Outputs are state transition diagrams in graphical, text, or internal representation.

4.3.3.1.4 Documentation Extractor

Introduction

The documentation extrator extracts information for user help messages and for portions of user manual directly from the dialogue description.

Inputs

Inputs are state transition diagrams in graphical, text, or internal representation.

Processing

Processing is finding, extracting, and formatting data for user help messages, user manual, and other user documentation.

Outputs

Outputs include user help messages, user manual, and other user documentation.

4.3.3.1.5 Executor

Introduction

The executor accepts a description of the syntax of a command language, expressed in the description language, and acts as an interpreter for that command language.

Inputs

Inputs include state transition diagrams description of a command language in graphical, text, or internal representation.

Processing

Processing acts as an interpreter for any given command language, as specified in the state diagram description language input.

Semantic operations (application level tasks) shall be performed by making procedure calls in Ada or Ada Command Language from the dialogue description to a separate program which implements the underlying functional capabilities of the system. For example, for an operating system command language, the semantic operations would be provided by calls to CAIS functions. Given an appropriate set of application programs, the Executor can thus be used as a complete run-time UIMS.

This tool shall support rapid prototyping of new command languages. It must therefore be possible to make small changes or additions to a command language and have the new language available rapidly, either by means of rapid recompilation or a facility for separate compilation or direct interpretation of the description language without compilation.

Outputs

The output is a running system that exhibits the user interface specified in the input.

4.3.3.1.6 Debugger

Introduction

This tool shall allow interactive execution of a dialogue description, as in the Executor. In addition, it shall interface to the Dialogue Description Editor, so that a user can track the progress of the dialogue through its description, much like a source-level debugger for a programming language.

Inputs

Inputs include state transition diagrams in graphical, text, or internal representation.

Processing

The debugger shall permit the user to edit a diagram and resume executing the dialogue with the changed diagram. It shall also allow the user to set breakpoints and to change the state of the executing dialogue.

Outputs

The output is a running system that interactively executes a dialogue description, as in the Executor, and permits its user to track the progress of the dialogue via its graphical representation.

4.3.3.1.7 Generic Input and Output Tokens

Introduction

The syntax portion of the dialogue description is written in terms of tokens. Each token is then defined in terms of the actual hardware input or output operations it comprises. A dialogue designer can provide these definitions separately for each token in the dialogue. However, many tokens will be quite similar to one another, except for some parameters. Hence a facility for defining and using generic tokens shall be provided.

The library of tokens shall include the following:

a. Simple input tokens, such as function keys.

b. Simple output tokens, such as prompts in particular locations.

c. A command name or other word entry, where any unique prefix is sufficient to enter a word, and additional (strictly unnecessary) matching characters are accepted.

d. Command or word completion, from a specified vocabulary.

e. Spelling correction, with respect to a given vocabulary.

f. Input token for a line-edited line of input. Parameters to this generic token will include line length, location and type of echo.

g. Help or explanation facility

h. Tenex/Tops20 style prompting and word completion

Inputs

Inputs include commands to create, modify, and use specific and generic tokens.

Processing

This facility accepts definitions of input and output tokens in terms of actual hardware input or output operations they comprise. It allows the creation of specific tokens and also the creation and use of generic tokens. A generic token can be instantiated with different values assigned to its parameters, yielding a variety of specific tokens.

It also includes a library of useful generic tokens. These take the form of token specifications in which certain characteristics of the token are described by parameters which can be filled in when the generic token is instantiated. The library of generic tokens shall make it easy to implement input and output services in a uniform way. It shall also provide a mechanism for ensuring agreement with the style guide at the token or lexical level.

The generic token facility and library shall be designed for and usable as a tool to promote the development and use of a consistent interaction style among groups of WIS user interface designers. It shall not prescribe such a style, but shall facilitate the uniform use of whatever style each group of designers may choose.

Outputs

Outputs include specific tokens and generic tokens usable in dialogue specifications.

4.3.3.1.8 Other Support Functions

Introduction

Other support functions for input and output, which can be called from the dialogue description language to provide needed low-level functions conveniently.

Inputs

Inputs include primitive user input actions, as appropriate to each of the individual functions.

Processing

Facilities shall be provided for common low-level input and output operations, including the following:

a. Definition and naming of logical screen regions or windows

b. Input echoing, color changes, and the like

c. Cursor position tracking and shape changes

d. Interpretation of cursor position in terms of structure of windows and screen objects (rather than absolute screen positions)

e. A graphical editor for entering screen and window layouts by manipulating examples, rather than by typing in coordinates and descriptions

Outputs

Outputs include primitive output operations as needed for each individual function and indications of acceptance of user inputs.

4.3.3.1.9 Front End Module

Introduction

A front end module, which parses the dialogue description language, should be suitable for use in building new tools that use this language. In addition the module itself shall be usable in the development of new tools to meet future needs.

Inputs

Inputs include state transition diagrams in graphical, text, or internal representation.

Processing

Processing is when a module or package (as contrasted with a stand-alone program) that accepts state transition diagrams in graphical, text, or internal form, checks and processes the data, and then makes it available in a form suitable for use by client programs.

Outputs

Outputs include data extracted from the state transition diagrams input in graphical, text, or internal form.

4.3.3.2.3 Software Interfaces

All tools developed shall share the common front end module described in Section 4.3.3.1.9. In addition, the interface to that module shall be designed so that users can build new tools which incorporate the same front end module.

All application-specific or semantic operations shall be called from the Executor as procedures in Ada or Ada Command Language.

The WIS database package shall be used for all database requirements within this project. Specifically, the libraries of generic tokens as discussed in Section 3.1.7 shall be stored in such a database. The internal representation of the state transition diagrams may consist of a database of state transitions.

The GKS shall be used for all graphics requirements. All operating system services shall be invoked via calls to CAIS.

4.3.3.3 Performance Requirements

Response time of all tools for processing new dialogue descriptions shall be appropriate for rapid prototyping. That is, it shall be easy to develop command languages, make changes to them, and experiment with alternatives without long waits for tool processing time. Response time of the Executor itself while executing a dialogue shall be adequate for a convincing demonstration of a proposed user command language. That is, it shall appear to the user to respond at least as fast as the proposed system being prototyped would be expected to respond. In particular, the system shall take no longer than 10 MS to make a single primitive state transition (not involving a sub-diagram call) in a user interface state transition diagram. The time for making a transition involving a sub-diagram call shall be no more than 10 MS plus the time taken for each of the primitive transitions made by the sub-diagram.

4.4 String Based Ada Command Language (SBACL) Software Requirements Specification

4.4.1 Introduction

4.4.1.1 Purpose

An important WIS component is the design and implementation of a robust UIMS. The string based command language is one component of this system. It provides a line oriented language for interface with the WIS operating system facilities. It also verifies and validates the network based UIMS definition tools for building user interfaces.

This command language is to work on the CAIS-defined page terminal and shall be modelled after the COSCL language definition [COSCL 84].

The document specifies certain constraints that the language and interpreter must meet. It does not itself define the command language. This project shall define a string based command language to the WIS Operating System (WIS OS). The language shall be known as the WIS String Based Ada Command Language (SBACL). The project shall design and implement an interpreter for this command language using the Network Based UIMS (NBU) definition tool. The goal of this project is both to provide a string based command language interface to WIS OS and to validate NBU and its associated tools.

4.4.1.2 Scope

This project shall develop a set of Ada packages. The packages shall be developed using a WIS-approved Ada Program Design Language (PDL) and shall include the following.

a. Internal and user documentation

b. Design notes

c. Testing requirements and sample test drivers

d. Any special services required

e. All other WIS package interfaces (e.g. database, operating system)

4.4.1.3 Definitions

This section contains descriptions for words that may have multiple meanings or different meanings in other contexts.

Application:	Any package, program, routine or sub-system which interacts with a user through the Command Language Facilities. This will typically be a set of procedures and an Interaction Object.
Default:	A value or action supplied by the software when no other action is explicitly identified or specified.
Input:	Information provided to a package, program, routine or sub-system from external sources.

Interaction Object: A package under UIMS control that defines and implements the user interface for a given application.

Operator: An interactive user of a system, usually one with a limited knowledge of the internal structure or processes of the system.

Output: Information generated by a package, program, routine or subsystem for external destination(s).

Prompt: Output indicating that user input is expected.

Special: These are the characters with ASCII codes less than <sp> (20H), and <del> (7FH).

4.4.2 General Description

The string based command language is a particular implementation of a UIMS. It will provide user interaction with the operating system and other programs. The language/interpreter shall be designed to allow for extensibility.

The command language shall meet the following general constraints.

a. The language shall be string based. That is, all statements in the language shall be expressible as strings of ASCII characters. In particular, languages involving graphical interaction such as menu selection are not acceptable.

b. All statements in the language shall conform to Ada syntax. However, it is not required that the entire Ada language be part of the command language.

c. The language shall cover all objects and operations of WIS OS that are intended to be referred to or invoked by WIS OS users.

d. The language shall be modelled, in terms of its operations and their parameters, on the COSCL. However, this requirement may be overridden when it conflicts with the requirement to cover all WIS OS objects and operations.

The interpreter for SBACL shall satisfy the following general constraints.

a. The interpreter shall accept input and generate output on a CAIS page terminal. That is, it shall use for its input and output the standard operations defined by CAIS for page terminals.

b. The interpreter shall translate its commands into the objects and operations supported by WIS OS and execute its commands by invoking the appropriate WIS OS functions.

c. The interpreter shall be implemented using the facilities provided by the Network Based UIMS (NBU). That is, the fine details of its user interaction and the way this interaction maps into WIS OS objects and operations shall be specified through NBU transition nets. To the extent that the NBU network construction and manipulation tools are available, they shall be used in the development of the appropriate nets.

d. The interpreter shall provide as much "user-friendly" support as possible for command entry and feedback. In particular, the interpreter shall at least provide:

   (1) On demand completion of command names, keywords, and other syntactic constants of the language.

   (2) On demand completion of file names and other dynamically determinable syntactic elements.

   (3) On demand on-line assistance for what commands are available, what their arguments are, what are the syntactic and semantic constraints on elements of the language, and such other information as may be helpful to a user performing command entry, and is practical to provide.

   (4) Informative, focused, and polite error messages in appropriate circumstances, including lists of alternatives when unambiguous completion is not possible (or on demand).

   Examples of interactions that satisfy the above requirements are shown in Section 4.4.3.2.6.

e. The interpreter shall use computational resources that involve less than one second of elapsed time for its role in the processing of a single command, i.e., not counting user type-in time, and not counting the time required by WIS OS to execute the command. This time requirement shall be met for the "normal" load situation of whatever computing environment it is implemented in. Higher loads on time sharing systems may cause performance to be worse.

4.4.2.1 Functions

The primary function of the string based Ada command language (SBACL) is to provide the user interface for the WIS OS facilities. The statements to the language are straightforward, but the interaction is not.

The SBACL shall provide prompting, command completion, help, noise words, and limited searching facilities for command completion tasks. It also provides error messages that help the operator in correcting the problem.

4.4.2.2 User Characteristics

The users of this package will be system designers, application programmers, and end users.

4.4.2.3 General Constraints

The primary constraint is that this package should have minimal loading impact on the final product performance. It shall be optimized toward speed. This package should also "look" and "feel" like the other UIMS packages in terms of style, I/O conventions, naming conventions, and documentation.

The package shall conform to WIS standards for Ada PDL and documentation. The package shall be written in Ada.

4.4.3 Specific Requirements

This section details the requirements of the command language package for the User Interface Management System. The "package" is a command language and an interpreter for the language. The interpreter interacts with the user with the commands provided.

4.4.3.1 Command Language

The command language shall be string based and shall not use menu selection or an icon type interface. The statements of the language shall conform to Ada syntax for procedure or function calls.

The comman language shall cover all objects and operations of the WIS operating system available to users.

The command language shall be modelled in terms of operation and parameters on the COSCL. It will include the COSCL commands except in cases where a WIS OS command conflicts, it will then use the WIS OS command.

Inputs

The input for the command language is the WIS OS commands, the COSCL model commands, and [1815A] for command syntax.

The NBU shall provide the structure for defining the language for use by the language interpreter.

Processing

The command syntax shall be processed through the NBU package to produce the dynamic part of the command language interpreter.

This processing consists of the following steps:

a. Select the commands to be used in the language from WIS OS and COSCL.

b. Modify the command syntax to conform to valid Ada calling sequences and target non-WIS OS commands to WIS OS commands or procedures that call possibly a set of WIS OS commands.

c. Rewrite the commands in terms of the NBU syntax.

d. Compile the "code" into the language interpreter using the NBU system tools.

Outputs

The output of the command language phase of this task is a validated (through the network based UIMS) command language consisting of WIS OS and COSCL commands.

Performance Requirements

None.

Standards

All command language statements shall be valid Ada syntax. All commands shall be derived from the WIS OS or COSCL.

Other Constraints

This task is expected to be a prototype for other command language implementations. It should be well organized, documented, and "easy" to follow.

4.4.3.2 Command Language Interpreter

The command language interpreter shall accept input and generate output on a CAIS page terminal. It shall use standard operations for this terminal. It shall also provide user friendly support on this terminal.

The interpreter shall be an "executive" to the WIS OS and shall execute its commands by invoking the appropriate WIS OS functions.

Inputs

The inputs to the interpreter are the WIS network based UIMS NBU output for command syntax and organization, and characters from the user.

All ASCII characters shall be recognized and acted upon, including all special characters.

Processing

The interpreter shall process the input text into WIS OS commands and execute these with calls to WIS OS system utilities.

The interpreter shall also provide user feedback and help to ensure a smooth and consistent operation. It shall provide as part of this:

a. On demand completion of command names, keywords, and other syntactic constants of the language.

b. On demand completion of file names and other dynamically determinable syntactic elements.

c. On demand on-line assistance for what commands are available, what their arguments are, what are the syntactic and semantic constraints on elements of the language, and such other information as may be helpful to a user performing command entry, and is practical to provide.

d. Informative, focused, and polite error messages in appropriate circumstances, including lists of alternatives when unambiguous completion is not possible (or on demand).

Outputs

The output of the command interpreter shall be text to the user and calls to the WIS OS utilities. It shall also be able to display information about commands available and parameters required for commands (i.e., information about itself).

Performance Requirements

The command language interpreter shall require less than one second of elapsed time for its role in processing a single command. This time does not count user type-in time or the time required by WIS OS. This shall be true for all commands except for information commands about the command language interpreter itself.

Standards

All package names and visible data items shall comply with WIS standards. All internal names and data items shall use a consistent and non-conflicting standard, preferably the WIS standard if possible.

Other Constraints

This package is the users primary view into WIS OS and the WIS applications in general. It should be robust and "impossible" to break, but provide as friendly an environment as possible with the given commands.

This package will set the example for acceptable WIS application interfaces and should not burden the user with extra work or memory but should also not limit the range of user actions.

An example of command language interaction shows how command completion and noise words can be used to simplify the command syntax for the user. The user input will be in upper case except for special characters, which will be indicated by <"special character">. Text typed by the system will be lower case, and explanatory text will be enclosed in brackets.

[The user types the following]

```
CO<esc>                [want to do the COPY command]

COpy(from_file => "    [system command completion]

[User adds file name]
COpy(from_file => "MY_FILE_WITH_A<esc>

[System completes to the following]
COpy(from_file => "MY_FILE_WITH_A_long_name.ada",to_file => "

[The user adds]
COpy(from_file => "MY_FILE_WITH_A_long_name.ada",to_file =>
"MY_COPY.ADA<esc>

[System completes to the following]
COpy(from_file => "MY_FILE_WITH_A_long_name.ada",to_file =>
"MY_COPY.ADA"
```

4.5 Video Based Ada Command Language (VBACL) Software Requirements Specification

4.5.1 Introduction

4.5.1.1 Purpose

The purpose of this requirements specification is to convey the necessary requirements to build a prototype interactive dialogue system. This system gives the user the flexibility to solve his problems by displaying the semantic information collectively, in a form, and allows the option of requesting more elaborate information on chosen or available commands through a menu system. The intended audience of this document shall be command language user interface designers with a good knowledge of human factors engineering skills who have developed, built, and implemented dialogue systems for super-mini or large main frame computers.

4.5.1.2 Scope

The software product to be produced shall be called the Video Based Ada Command Language (VBACL) and will be a menu-selected, parameter-driven, form-based method of using the operating system. Selection on menus will present either forms or other menus which allow the user to execute commands based on the displayed default parameter or user specified parameters. Among the values to be derived is that the command, its options, and its defaults along with other relevant information are simultaneously displayed producing a synergetic effect. Disposition of the displayed command is either through direct execution or through integration with other commands to form a macro. An active window shall be a part of each display to allow direct interaction with the system. The model to be implemented is that developed by the CODASYL and is known as [COSCL 84}. Since the bit mapped requirements for Video_COSCL are deferred, the package will provide only forms and menu oriented, screen driven method of controlling the operating system.

4.5.2 General Description

4.5.2.1 Product Perspective

This software should allow the user to interact with the system and invoke operating system utilities using forms and menus. Commands should be executed singly or in a sequenced group such as from a command file. A user shall be able to create or edit forms and menus and should be able to navigate through the system using forms and menu management. Since Video_COSCL is based on COSCL, and the String Based Ada Command Language (SBACL) is based on COSCL as a model, a one-to-one relationship should be maintained between SBACL and VBACL in order to allow switching between these two command languges.

4.5.2.2 Product Functions

The software shall have a forms and menu management style which should allow the storing and retrieving of forms and menus which are either created or already in existence. There should be a beginner and expert mode of operation, and a help facility which has access to the system manual for VBACL.

For menus, a choice among alternatives should select another menu or advance to perform a task. There should be help text for each menu item. Return completion codes may be implemented as part of the menu system which would return the user to designated locations.

4.5.2.3 User Characteristics

The software will be used by a wide range of users such as data entry clerks, software developers, military personnel, and office workers. Ths skill and training levels of the users will vary widely, however, it is assumed that the fundamental concepts of page and form terminal operations are within the grasp of users, and that a brief on line tutorial should be sufficient to allow the user to work with the software.

4.5.2.4 General Constraints

The software must correspond to government regulations concerning the use of Ada [1815A]. The software must comply with the most current version of [CAIS 85] at the time of submittal.

4.5.3 Specific Requirements

4.5.3.1 Function Requirements

The following functions are defined in the current version of Video-COSCL

a. Menu Editor

b. Method of Specification

   (1) A forms definition language to describe the syntax of forms.

   (2) A restrained form of English to describe the semantics in algorithmic form.

   (3) A data modeling method to describe the characteristics of object-oriented Video_COSCL.

c. Definition of Forms

d. Hierarchy of Menus and Flow Control

e. Selection Forms

f. Command Forms

g. Object Forms

h. Menu Creation Facility

Not all the functions in Video_COSCL have been defined, and the following is a list of functions which still have to be determined.

a. Help

b. Display

c. Activators

d. History File

e. Procedure Creation Facility

f. Bit Map Graphics

4.5.3.2 Peformance Requirements

The users of a menu should wait no longer than 1.0 second for the execution of a menu selection. In the case of the creation of a new form, the semantics of the new form should be interpreted, linked, and available for execution within 1.5 seconds after submission.

4.5.3.3 Design Constraints

The software will be implemented using [1815A] and the most current versions of [CAIS 85] and Video_COSCL.

4.5.3.4 Attributes

The software must be machine and terminal independent, relying on the use of character and keyboard standards.

4.6 Interaction Object UIMS (IOU)

4.6.1 Introduction

4.6.1.1 Purpose

This section outlines the use of "interaction objects" to provide a direct style of user interface (one made of an interacting collection of active and/or responsive objects) rather than a single-thread command language dialogue with the user. A dialogue is described as a collection of individual, possibly mutually interacting, Interaction Objects.

The Network Based UIMS (Section 4.3) must be capable of supporting the definition and implementation of an IOU.

4.6.1.2 Scope

To describe a direct manipulation user interface, this section will:

a. Define interaction objects.

b. Specify their behavior (input or event handlers).

c. Provide an example of how interaction objects with inheritance can be used to specify an overall coordinated user interface.

Note that while there will be a hierarchy of interaction objects, this document will concentrate only one subset of them, those with associated input handlers or dialogue specifications. Also, a complete specification is not provided, since Volume VIII, Graphics Prototypes, describes interaction objects in more detail.

4.6.2 General Description

4.6.2.1 User Interface and Interaction Objects

An interaction object is the smallest unit with which the user conducts a meaningful, step-by-step dialogue; that is, one that has continuity or syntax. It can be viewed as the smallest unit in the user interface that has a state that is remembered when the dialogue associated with it is interrupted and resumed.

In this respect, it is like a window, but in a direct manipulation user interface, it is generally smaller---a screen button, a single type-in field on a form, a command line window. It can also be viewed as the largest unit of the user interface over which we want input events serialized into a single stream rather than divided up and farmed out to separate objects. Thus, an interaction object that has an input handler is a locus of maintained state and of input serialization.

4.6.2.2 Input Handler and Interaction Object

Each individual object has a single-thread dialogue, with all inputs serialized and with a remembered state whenever the individual dialogue is interrupted by that of another interaction object. Thus, a single-thread state diagram is the appropriate representation for the dialogue associated with an interaction object. The input handler for each individual interaction object is specified as a conventional state transition diagram.

4.6.2.3 Individual Interaction Object Event Handlers and Overall User Interface

The individual interaction object event handlers must be combined into an "outer loop" or overall user interface. This could be described with still another master state diagram, but, since the user sees the structure of the user interface as a collection of many semi-independent objects, that is not a particularly perspicuous description. Instead, a built-in executive is defined that embodies the basic structure of direct manipulation dialogue and has the ability to make co-routine calls between individual state diagrams. The executive simply collects all of the input handlers (in the subset of interaction objects that have input handlers) and executes them as a collection of co-routines, assigning input events to them and arbitrating among them as they proceed.

To do this, a co-routine call mechanism must be defined for activating state diagrams. This means that whenever a diagram is interrupted by a co-routine call to another diagram, the state in the interrupted diagram is remembered. Whenever a diagram is resumed by co-routine call, it will begin executing at the state from which it was last interrupted.

The overall process is that the state diagram of one of the interaction objects will be active at any time. As it runs, it reaches each state, examines the next input event, and takes the appropriate transition from that state. It proceeds in this way until it reaches a state from which no transition matches the input.

The executive then takes over, suspending the current diagram but remembering its state for later resumption. The executive examines the diagrams associated with all the other interaction objects and looks at their current (i.e., last suspended from) states to see which of them can accept the current input. It then resumes (with a co-routine call) whichever diagram has a transition to accept that input. Typically there will be only one such diagram. (In fact, since entering and exiting screen regions will be important input tokens in the direct manipulation user interface, this is easy to arrange when the individual dialogues are associated with areas on the screen.)

Thus, each locus of dialogue is clearly and appropriately described as a sequential state diagram, which can be suspended and resumed, but always retains state. The overall user interface is defined by the behavior of the executive itself, rather than inappropriately as a large, highly regular state transition diagram.

4.6.2.4 Tokens in Interaction Objects

To complete the user interface specification, a collection of low-level inputs and outputs is needed which can be invoked by the state diagrams. These will correspond to tokens. Examples for input are button clicks (both down and, where supported, up), locator entering or exiting regions, and keyboard characters; for output, they include highlighting or de-highlighting regions, displaying or erasing graphical objects, and rubber band or other continuous "dragging" feedback.

These tokens can be associated with transitions in the state diagrams. The internal details of these low-level input and output operations will be specified separately and may be given as additional state diagrams, Ada code, a simple new language for this purpose, or some combination. In practice, the use of inheritance discussed below will make tokens easy to describe in any of these notations.

4.6.2.5 Schema for Interaction Objects

An interaction object will behave like a flavors object. It comprises a collection of variables, methods, and other impedimenta, all of which are subject to inheritance. Specifically, it will contain the following elements:

a. FROM

   A list of of other interaction objects from which this one inherits elements, with ordering rules similar to those for flavors.

b. IVARS

   Instance variables for this object.

c. METHODS

   Procedures pertinent to this object. In addition, each interaction object must supply (possibly by inheriting a default one) certain standard procedures, such as Draw, Init, and Destroy.

d. TOKENS

   Definitions of each of the input and output tokens used in the syntax diagram for this interaction object.

e. SYNTAX

   The input handler for this interaction object, expressed as a state transition diagram to be called as a co-routine.

f. SUBS

   Additional state diagrams, called as subroutines by the SYNTAX diagram above.

g. STATES

   A list of "mixin" or "kernel" states, which are used to define standard sets of behaviors, such as sensitivity to abort or help keys, and which can be applied to states in the above diagrams, so that such descriptions do not have to be repeated.

4.6.3 Interaction Object Example

4.6.3.1 Example Without Inheritance

An example of an interaction object follows. In this first example, inheritance is not used; everything is specified explicitly for this object. Following the example is an explanation of how inheritance will help.

```
INTERACTION_OBJECT    MessageFileDisplayButton  is

-- This is a single action button.  When the mouse cursor enters it,
-- the button is highlighted; when it exits, the button is dehighlighted.
-- If you click on the left mouse button while the cursor is inside the
-- button, it performs a "message file display" operation (whose details
-- are given in the application semantics, not here).  Here we assume a
-- mouse that only reports down clicks.

IVARS:

Position :=     { 10, 20, 5, 8 };     --i.e., coordinates of screen rectangle

METHODS:

Draw            { CREATE_TEXT_BUTTON("Display"); }

TOKENS:

iENTER           { --locator moves inside Position-- }

iEXIT            { --locator moves outside Position-- }

iLEFTDN          { --mouse left button goes down-- }

oHIGHLIGHT       { --invert video of rectangle given by Position-- }

oDEHIGHLIGHT  { --same as oHIGHLIGHT-- }

-- Note:  names of input tokens begin with 'i' and output tokens begin
-- with 'o'

SYNTAX:

+st:           iENTER ->highlight

highlight:     oHIGHLIGHT ->click

+click:        iLEFTDN ->doit
+click:        iEXIT ->dehighlight

doit:           ->done act: DisplayMf(inbox);

+done:          iEXIT ->dehighlight

dehighlight:   oDEHIGHLIGHT ->st
;

-- Note:  This and subsequent diagrams are given as text in a fairly
-- standard format, i.e., a list of transitions of the form

--  startState:      SYMBOL ->endState

-- where SYMBOL can be an input or output token, the name of another
-- diagram, or empty.  Actions can also be associated with transitions.

-- For documentation purposes, states from which it is possible for
-- the dialogue to be suspended are marked with a '+'.

-- DisplayMf is a procedure defined in the application (semantics)
```

```
-- For this example, assume that the protocol between the UIMS and the
-- application is synchronous procedure calling.  We will later
-- generalize to one-way messages.  Then, the present example would be
-- handled as two state transitions, one that sends a message requesting
-- DisplayMf(inbox) followed by one that waits for the reply message.

end INTERACTION_OBJECT;
```

4.6.3.1.1 Discussion of Example

Interaction object dialogue with the executive:

When the locator enters the screen area for Message File Display Button, the local token iENTER is generated. Since no other interaction object will have state transitions that accept it, the diagram for this object will be called as a co-routine by the executive. This diagram will take over, accept the input, highlight the button, then wait for more input in state "click."

If the next input is the button press, this object performs its action. If the next input is the locator exiting this region, this object dehighlights itself and returns to its start state. There it waits only for an iENTER and ignores other inputs. (In particular a iLEFT button click will no longer be accepted by this object but would probably be accepted by some other object.)

If the next input in state "click" is anything else (for example, a keyboard key), another diagram that has a transition that can accept that input will be called by the executive. As soon as another input that this button diagram can accept occurs, the button diagram will be resumed in the same state (click).

4.6.3.1.2 Expression of Alternate Behaviors

The reason why the above discpription appears complex is that there are several possible plausible alternative behaviors for the handling of sequences of clicks and mouse motions in a screen button. There are other ways in which the exiting and dehighlighting could be handled. Or the button could highlight when the mouse is depressed and perform the action when it is released. The user interface designer can indicate exactly which sequence he needs. It is not sufficient to have him describe the user interface imprecisely and leave the details up to the coder. Nor is it sufficient for the builder of the UIMS code to code in a standard version of a screen button and prevent the individual user interface designer from overriding it. Given that the user interface designer must provide this detail, state transition diagrams appear to be a reasonable notation for doing so. The mechanism for inheriting from standard library interaction objects, with the ability to override them, will relieve most of this burden. This mechanism is discussed in further detail in the following sections.

As another example, consider an interaction object that provides a type-in field in a form. There are a number of ways in which this could be implemented, and this notation will let the designer indicate the one he wants. For example, it might accept characters while the mouse is pointing to the field, but not at any other time, much as the screen button above accepts a button press only when the mouse is pointing at it. Another alternative (actually modeled on the one used by SUN) is to have a mouse click designate the current typein field. Once designated, that continues to receive all keyboard input until the mouse is clicked on another typein field (i.e., it gets keyboard input even when the mouse has moved away from it). The latter would be described as a diagram that stays in the same state even after the locator exits and simply continues to accept keyboard input until interrupted by another diagram, which then grabs subsequent keyboard input.

4.6.3.2 Example with Inheritance

One problem with the notation used in Section 4.6.3.1 is that these interaction object descriptions are going to become numerous and repetitive. The solution is flavors-style inheritance of the parts of an interaction object. Specifically, an interaction object inherits all of the IVARS, METHODS, TOKENS, STATES, and SUBS of its parents and adds them to any that the object itself declares. If the object declares an IVAR, METHOD, TOKEN, STATE, and SUB of the same name, it overrides the inherited one. In turn, all of an object's own and inherited IVARS, METHODS, TOKENS, STATES, and SUBS are inherited by its children. For the present, the entire SYNTAX diagram is inherited and may be overloaded as a unit, and presently, one cannot overload parts of it selectively, but this may change.

The following illustrates building some generic interaction objects, and we then rebuild the previous example using them. First, a generic screen interaction item is defined which provides some useful behaviors that can be inherited by more specific objects. Like a "mixin" flavor, it is not expected that this object will be instantiated by itself, but will contribute tokens, methods, etc. to other specific objects by inheritance.

```
INTERACTION_OBJECT GenericItem is

TOKENS:

iENTER          { --locator moves inside Position-- }

iEXIT           { --locator moves outside Position-- }

oHIGHLIGHT      { --invert video of rectangle given by Position-- }

oDEHIGHLIGHT  { --same as oHIGHLIGHT-- }

iLEFTDN         { --mouse left button goes down-- }

-- Note that Position is a local variable that will have to be
-- provided by any instantiations that use this mixin.

-- Note no SYNTAX section

SUBS:

-- This section defines some useful subdiagrams others may want to call

enterclick -> (in, out)

st:         iENTER ->highlight

highlight:      oHIGHLIGHT ->click

+click:         iLEFTDN ->in
+click:          exitdehigh ->out
;

exitdehigh  ->ret

st:         iEXIT ->dehighlight

dehighlight:   oDEHIGHLIGHT ->ret
;
```

```
end INTERACTION_OBJECT;
```

Next, we define a generic screen button, similar to the specific one defined earlier, but useful for other applications. Again this object is a mixin, not expected to be instantiated by itself. It inherits the primitives provided by GenericItem above.

```
INTERACTION_OBJECT GenericButton is

FROM:          GenericItem

METHODS:

Draw           { CREATE_TEXT_BUTTON(Legend); }

-- again, Legend is a local variable that somebody will provide

SYNTAX:

+st:           enterclick ->(doit,st)

doit:          ->done act: DoTheAction();

+done:         exitdehigh ->st
;

-- This diagram inherited and used definitions of enterclick and
-- exitdehigh, could overload them in this object if we wanted to.
-- DoTheAction will be a locally-defined METHOD.

end INTERACTION_OBJECT;
```

The UIMS will provide a library of such interaction objects, so the designer of a particular user interface can call upon them easily. If the standard behaviors provided in the library are sufficient for his needs, the user interface will be very easy to put together. Moreover, if he wants to determine the precise behaviors of the library interaction objects, they are available in the dialogue specifications, rather than hard-coded in the UIMS. If he needs more specialized dialogue handlers, he can write or modify them using the same notation.

Given the two generic interaction objects defined above, the original MessageFile-DisplayButton can now be defined much more compactly by inheriting the aspects that are common to all screen buttons and defining only those specific to this particular button. The interaction object below does exactly the same thing as the MessageFileDisplayButton above, but implements it using the generic objects defined. Note that outside of the few specific details given below, everything else is inherited from the generics, including the SYNTAX diagram itself. And note that if this particular screen button is to be different from the standard ones, any or all parts of it could be overloaded.

```
INTERACTION_OBJECT MessageFileDisplayButton is

FROM:          GenericButton

IVARS:

Legend :=       "Display";
Position :=    { 10, 20, 5, 8 };
File :=        inbox;

METHODS:

DoTheAction     { DisplayMf(File); }
```

```
end INTERACTION_OBJECT;
```

It is clear now that, with a reasonable library of generic interaction objects, it need not be cumbersome to describe a direct manipulation interface with this notation.

4.7 History and Logging Software Requirements Specification

4.7.1 Introduction

4.7.1.1 Purpose

The logging package is a plug-in component of the UIMS that records the interaction between the user and the computer. It will be designed to assist the user in recalling frequently used command sequences and in analyzing user interaction.

4.7.1.2 Scope

This document takes into consideration that the command logging provides a facility for the developers of computer human interfaces for recording commands in the user/computer dialog. Command Logging also provides a method for storing notes of special conditions from application programs and/or comments from the user, a view or window to the logged data, and retrieval of logged commands.

This project shall develop a set of Ada packages. The packages shall be developed using a WIS-approved Ada Program Design Language (PDL) and shall include the following:

a. Internal and user documentation

b. Design notes

c. Testing requirements and sample test drivers

d. Any special services required

e. All other WIS package interfaces (e.g. database, operating system)

4.7.1.3 Definitions

This section contains descriptions for words that may have multiple meanings or different meanings in other contexts.

Application	Any package, program, routine or sub-system which interacts with a user through the Command Language Facilities. This will typically be a set of procedures and an Interaction Object.
Checkpoint	The explicit writing to a permanent media of items in temporary media. Example: RAM to disk.
Default	A value or action supplied by the software when no other action is explicitly identified or specified.
Input	Information provided to a package, program, routine or sub-system from external sources.
Interaction Object	A package under UIMS control that defines and implements the user interface for a given application.
Operator	An interactive user of a system, usually one with a limited knowledge of the internal structure or processes of the system

Output — Information generated by a package, program, routine or sub-system for external destination(s).

Prompt — Output indicating that user input is expected.

Special — These are the characters with ASCII codes less characters than <sp> (20H), and <del> (7FH).

4.7.2 General Description

The logging facility lies "between" the UIMS and the application and records data going one or both ways. It may receive data directly from the application to include in the log (eg. status messages, context switch headers, etc.). Figure 1 shows the User, UIMS, logger and application program relation. The application IO and logger IO modules are the Interaction Objects provided to the UIMS for handling the user interface.

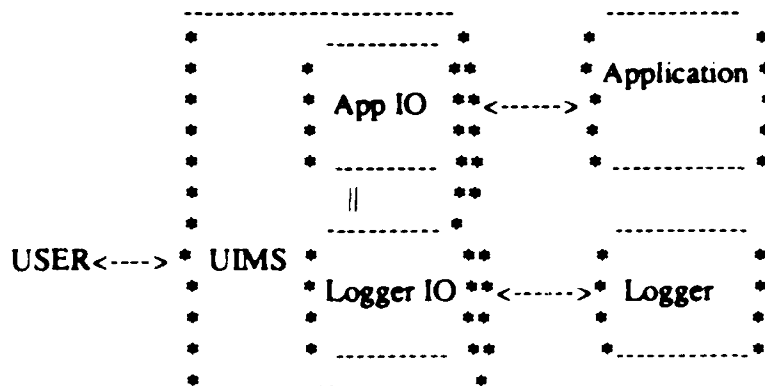


Figure 3. Logger Function Interface Diagram

The logger IO is the UIMS Interaction Object associated with the logger package. If active, it provides the user with a scrolling real-time view of the log, and implements any user available log operations. This may include starting and terminating the logging function. The system designers determine what functions the user will have with the log (if any) and define how the logger IO is to be instantiated. It is possible that the user will have no knowledge of logging in a particular application environment. The logging package shall be designed to be modular in structure so that it may satisfy any or all of the following goals without the overhead of full functionality for all applications and/or phases of an application life cycle.

The logging information may be used for the following:

a. Evaluating the user interface design

   In this case the user interface designer reads the log and the log information shows how the end user is performing. It may include information about how commands were specified, how many low level typing errors were made and other complete commands, etc.)

b. Showing the users what they have done.

This requires more high-level descriptions and they shall be in end-user readable form. This will typically be much longer than the machine readable form that might be appropriate for the interface designer.

c. Grading the end user.

This would be used by the end-user's superiors or instructors to evaluate the user, and as a guide for seeing whether they need more training. The requirements for this log are similar to "b" but may require additional timing and error information.

d. Forming command files.

Logs may be used for combining a group of commands together to form a file to perform commands without user intervention. For this, the logs shall contain only correct, final commands in normal text. "Noise words" shall be considered in how they may effect rerunning the file with the WIS command language interfaces.

e. Supporting UNDO, REDO, or REPLAY.

Some logs (usually called "transcripts") are used to support UNDO or REDO of previous commands (much like a command file but more immediate). Some systems (eg. text editors) allow REPLAYing the entire session as a way of recovering from crashes. This type of log would need to include application specified data to enable these operations.

f. Keeping track of actions and events.

The system users and/or implementors can find out what was going on when a system "fails" or crashes. For this, the log does not need low level lexical data, but will need some additional semantic information about process activity, default arguments of the applications, current context, and application supplied environment data. This log might be in a tightly encoded form to conserve space. It would not normally be used by end-users.

Command logs for the above six types of information are not required to be of the same format, although a minimum number of formats is desirable.

Graphical commands may also appear in logs. It may be appropriate to save coordinates, invent or require names for objects or provide some description of the effected action.

The command logger will have to log command languages that are command line based, form based, and graphically based. This may require variants on a primary logging mechanism, or totally separate packages.

The logger shall (optionally) keep a set of statistics on the logged data. This provides a set of summary data for user view and/or post session analysis.

The logger shall be able to record data on a variety of output devices. This may include output only devices (paper) or remote storage devices.

The application environment may be distributed with the logging device remote and possibly slow. To provide a log window, some local storage shall be reserved and used by the logger for its log window.

4.7.2.1 Functions

To support the expected variety of logging applications, the logging package must provide several functions. The functions, as the package itself, are plug-in features for the application developer. Table I provides a list of functions and their description.

Some of the listed functions may not be independent. Those cases shall be highlighted in the package documentation. All functions shall have default positions or actions if not specified.

4.7.2.2 User Characteristics

The users of this package will be system designers, application programmers, and possibly, the end user.

4.7.2.3 General Constraints

The primary constraint is that this package should have minimal impact on the final product performance. That is, it should be optimized toward speed. This package should also "look" and "feel" like the other UIMS packages in terms of style, I/O conventions, naming conventions, documentation, and usability. The package shall conform to WIS standards for Ada PDL and documentation.

There shall be the following media drivers for a variety of output types:

a. Write only (paper)

b. Write once, read many (optical disks, PROM)

c. Read/Write (Disk, bubble, tape, EEPROM)

The specific output device will then be configured by the system designer. The package shall be written in Ada.

4.7.3 Specific Requirements

This section details the requirements of the logging package for the command language task force UIMS. The "package" is a set of packages providing a set of functions to provide logging facilities for a system design. Applications will use logging facilities in the system if available, and the UIMS will be responsible for handling log queries/messages to both active and inactive/null loggers.

4.7.3.1 Logging

The logging function is fundamental to the logger package. Its primary function provides a human and/or machine readable transcript of a specific user-application session. The log may include user and application generated comments. Various other functions shown in Table I control the log format, content, and user visibility.

The basic constraint on the logging function is that it does not degrade system performance more than 2% as measured by average response time to user input keystrokes, except for log initiation, which may take longer with user notification.

The logger Interaction Object, if visible to the user, provides a window into part of the log with scrolling, retrieving, initiation, and termination functions.

4.7.3.1.1 Logging Function Inputs

The input data for the logging function are characters and commands from the UIMS and character responses and comments from the application package. The logger package is between the UIMS and the application. Some data may be explicitly targeted to the logger package (application or user comments, time of day).

The "unit of measure" of the items input to the logger is characters in ASCII. Included with each block of data may be a data descriptor (command, comment) and a process ID for optional log inclusion.

The quantity and timing of data will be application dependent so parts of this package must be tailorable to specific cases. Normally data will be at human speed and quantity but there may be cases where the rate is 1 to 2 orders of magnitude higher for a given system.

The range of valid input shall be the ASCII character set as defined in [188C] and includes all characters from 00 Hex to 7F Hex.

The logger IO shall accept commands to start, stop, read and extract the log. Deleting the log may or may not be allowed.

4.7.3.1.2 Logging Function Processing

The logger function shall normally look at the input data, decide if it has to record it, send it either to the output process or the null device, and either log it or not. It will optionally record statistics and add time stamps to the recorded output.

Errors in input data, or device specification will cause error messages in the log if possible. As a final fall back position (and test mode), the logging package will go in to transparent mode and do nothing to the data through it.

Errors or abnormal conditions in the logger will terminate the log unless a specific recovery action has been defined at initiation. In the case of system errors, the logger function shall continue to attempt to log with each new entry and shall record its number of attempts and failures. This may happen if the logger output device is remote and/or stops responding to logger output requests.

The logger IO shall format data for user view, retrieve log records, initiate and terminate the logging process, provide statistics and error messages to the user, and scroll the users view of the log up or down.

4.7.3.1.3 Logging Function Outputs

The output of the logging function is a log of the user/application session recorded in a design selected format on a permanent media. The logger outputs (thru-puts) data to the connected processes.

Illegal character values (in the context of the specified recording format) may be counted in an illegal character count for log end display or they may be optionally rejected. They may be noted in the log with some identifier (eg **) indicating an illegal character. A string of illegals may be identified in some compressed format (e.g., **27** for 27 illegals in a row).

The logger function is under the control of the UIMS and is part of the UIMS. It inputs and outputs through the UIMS.

4.7.3.1.4 Performance Requirements of the Logger Function

The logger function, when integrated, is part of the UIMS. Its performance parallels that of the UIMS in terms of terminals, applications, and users.

4.7.3.1.5 Standards

All package names and visible data items shall comply with WIS standards. All internal names and data items shall use a consistent and non-conflicting standard, preferably the same if possible.

4.7.3.1.6 Other Constraints

This package is expected to be used on a variety of machines in a wide range of projects. It must be maintainable, reusable, and transportable. Non-transferable components shall be separated and clearly identified as to function and implementation to aid in portability.

4.7.3.2 Logger Function Options Functional Requirements

The remaining functions of the logger package, as shown in Table I, modify the log format and its content and tailor the logger Interaction Object. This section discusses these modifiers as a group with specific identification in the processing sections.

The general intent of these functions is to customize the log for a given system. There should be at least two ways to change a functions effectivity. First should be a set of compile time "switches" to set the default action to the required mode. Second is a communication between the UIMS and the logger to set or select a given mode of operation. The logger should return status after completion of the set up procedures.

4.7.3.2.1 Inputs

These inputs are the same inputs as for the logger.

4.7.3.2.2 Processing

The following sections describe the requirements for the processing supported by the various functions supplied with the logging package.

"All Characters"

The all character option sends a message to the UIMS to inform it of all character logging. The UIMS then sends characters to the logger.

Table I. Logger Package Functions

Function	Description
All Characters	Records all characters entered, including command line editing characters and graphic input command "characters".
Checkpoint	Sets minimal checkpoint criteria. Defaults to a command line.
Context Save	Saves terminal/workstation context at log initiation.
Delete	Automatically deletes an old log file. Default is to create a new generation or append to an existing log.
Identify	Mark each entry with an identifier as to originator (user, UIMS, application id).
Logging	Enables or disables the logging function. Disable forces a checkpointing.
Max Internal Log Memory	Sets the upper bound on memory that the logger may use for its internal (local) log contents.
Max Log Length	Set a maximum number of entries allowed in a log
Name	Requests a log file name from the user whenever logging is initiated.
Return Commands	Enables a command returning mechanism
Scroll Down	Enables the down scrolling capability of the logger IO.
Scroll Up	Enables the up scrolling capability of the logger IO.
Statistics	Turns on the statistics gathering system in the logger package. Records characters received, transmitted, and logged. Records illegal characters received and unsuccessful log write attempts. The logger prints this information in the log at log termination.
Time Stamp	Appends each log line with a time of occurrence (system time) stamp. (eg. ddmonyr hh:mm:ss)
UIMS Only	Records only user commands, not application returns.

This mode of operation would normally be used in a debugging environment or in a situation of evaluating operator or command language performance. This would provide much more detail than usually required in a log file.

Checkpoint

This function sets a limit on the minimally acceptable checkpoint criteria. The minimum shall be a single log line but may be increased to some other reasonable (within the application context) limit. Between checkpoints, log entries may be stored in volatile memory and then written to non-volatile memory at checkpoint or log shutdown time.

When a large checkpoint interval is in force, the system (user, UIMS, application) shall be able to request an explicit checkpoint. The number of log entries in volatile memory may be much greater than the checkpoint criteria, as determined by Max Internal Log Memory.

Context Save

This function causes calls to the UIMS and possibly the operating system to find the current context of operation. This may include OS version, UIMS version, logger version, application version, user ID, current window, terminal type and characteristics, and any other data required to replicate the context at some later time.

This function would normally be initiated immediately after log initiation (default) and additionally on application, UIMS, or user request.

Delete

This function controls the update of old log files. It may delete them, append to an old one, or create a new generation without deleting the old. This will be log device dependent but all functions shall work on all output devices. That is, if the output device were paper then all operations would map to append. The translation shall be in the output device interface, not in the delete function.

Identify

This function forces a marker with each log entry identifying source. This may be user, UIMS, or one of many applications. Log entries generated by the logger (eg. statistics) need not be marked. Default will normally be no identify.

Max Internal Log Memory

This parameter specifies the amount of volatile memory that the Logger may use for fast window response. It is an upper bound that will be system specific.

Max Log Length

This parameter sets a maximum number of entries allowed in a log. Additional parameters should tell what action to take on log overflow (start new generation, wrap around, etc.)

Name

This function requests that the UIMS query the user for a file specification to use for logging. Default is to use a pre-defined file name.

Return Commands

This function enables a command returning mechanism. This should support returning command N (from the top position), returning prior, next, first, and last. This would be used for script generation and possibly by the scrolling mechanism.

Scroll Down

This function enables the down scrolling capability in the logger IO.

Scroll Up

This function enables the up scrolling capability in the logger IO.

Statistics

This function records transaction statistics in the logging function. This should include all information usable for post interaction analysis and may include:

a. Total Characters

    (1) UIMS ---> Application

    (2) UIMS ---> Logger

    (3) Application ---> UIMS

    (4) Application ---> Logger

b. Total log lines written

c. Total log write failures

d. Total illegal characters encountered

e. Min, max and average response time (Application to UIMS command).

f. Errors

It should also include any other items that may help in analyzing or improving the interaction environment.

Time Stamp

This function adds a time stamp to each log line. The stamp should be in a fixed format in the first n columns of the log so subsequent processes (on the log) can easily handle or strip the data.

UIMS Only

This function determines what parts of the interaction will be logged. This may be all, application only, UIMS only, or other combinations. The default should be UIMS only.

The UIMS optionally records the commands entered by the user, and additionally, comments by the user and by the application program.

4.7.3.2.3. Outputs

These functions modify the content and form of the log. The logger function handles the exception and log writing tasks.

4.8 Style Guide Software Requirements Specification

4.8.1 Introduction

4.8.1.1 Purpose

This project shall result in the development of a Style Guide to be used as a manual for specification, design and implementation of user interfaces for the WIS-Ada Foundation Technology programs. The Style Guide for User Interface Development shall perform the following:

a. Provide guidance to the developers of system and application programs on the design of interactive use interface

b. Encourage consistent user interfaces appropriate to the type of application, level of user, and available terminal resources.

c. Specify appropriate user interface design rules.

4.8.1.2 Scope

There will be two aspects to this manual. One will be the provision of guidelines, including explanation and examples, which will describe concepts, goals, features, and system behavior appropriate to given interface design goals. The other aspect of this manual will be a set of design rules specifying interface presentation format and layout, the use of interaction objects, and other appropriate factors affecting the user interface such as response time considerations and feedback mechanisms. Conditions when deviation from established Design Rules would be appropriate will also be stated and a process for review of such will be outlined.

4.8.1.3 Definitions

Application: Any package, program, routine or sub-system which interacts with a user through the Command Language Facilities.

Design Rule: A binding specification particular to the WIS-Ada Foundation Technology Program development efforts.

Entry: In regard to forms, the data which has been entered into a field.

Feedback: A response provided to a user by a system as an acknowledgement or echo of some action taken by the user.

Field: In regard to forms, that portion of a form which a user may enter data into.

Form: A screen presentation used to guide and specify a collection of user input, usually in the form of text, as a single data set.

Guideline: A non-binding principle which provides general guidance to designers and implementers.

Icon: A symbol representing an object which can be invoked or manipulated by a user.

Input: Information provided to a package, program, routine or sub-system from external sources.

Input Token: A collection of values constituting a record of a given type provided to a package, program, routine or sub-system from an external source.

Interaction Object: A software package which supports interaction with the user for the purpose of obtaining specific input tokens.

Menu: A presentation of options which may be selected by the user.

Mode: An environmental setting for a system or application which may affect (a) the nature of an action, (b) the availability of options, and/or (c) access to objects.

Operator: An interactive user of a system, usually one with a limited knowledge of the internal structure or processes of the system.

Output: Information generated by a package, program, routine or sub-system for external destination(s).

Output Token: A collection of values constituting a record of a given type generated by a package, program, routine or sub-system for external destination(s).

Presentation: The visual image presented to a user as a representation of a object or in conjunction with an interaction task.

Prompt: Output from an application indicating that user input is expected.

Response Time: The time taken by a system to begin presenting output in response to some action by the user.

Standard: An official specification or set of specifications developed external to the WIS-Ada Foundation Technologies Program.

User: A person who uses a system or application.

User Interaction: A process or technique by which a user interactively communicates with an application.

User Interface: The user interactions, presentations, and feedback which constitute the full set of communication capability between a user and an application.

Visual Object: An object whose visible representation may be manipulated as a single output token.

Workstation: The input and output devices available to the user in the user's work area through which the user communicates with an application.

4.8.1.4 Overview

The Style Guide shall provide the developers with generic descriptions of the requirements for user interfaces associated with various types of application tasks such as analysis, command/control, information retrieval, etc. and with a description of the requirements for

user interfaces serving different types of users according to their skill and experience levels.

In addition, this document shall define specific design rules which will be followed in the construction of the various application and system user interfaces for the purpose of enforcing consistency across the various user interfaces.

4.8.2 General Description

4.8.2.1 Product Perspective

This specification requires familiarity with the following:

a. The major system components (particularly those providing system functions which are to be accessed directly by the user).

b. Tools and support provided for the user interface.

c. System-provided interaction objects and visual objects.

d. System communications facilities and hardware, particularly the terminals and workstations used in the implementation of this effort.

The design rules specified as part of this effort will be appropriate to both the hardware and software components of the system and will reference interaction objects and visual objects as needed.

4.8.2.2 Product Functions

The Style Guide will perform the following:

a. Define interaction styles and the situations and conditions appropriate to their use.

b. List and discuss design guidelines to be considered in the development of user interfaces.

c. Classify interaction objects and functional support tools for user interaction and illustrate their appropriate use.

d. Classify and differentiate the various terminals and terminal resources; list those interface concerns particular to the type of terminal resources (NOTE: If a standard workstation is to be employed throughout the system, this may not be required.)

e. Specify design rules for presentation of output to and acceptance of input from the user of WIS-Ada Foundation Technology programs

f. Provide basic screen layout formats for various presentation items

g. Provide standard names and icons for referencing standard commands, facilities, and objects

h. Specify those conditions for which deviation from established Design Rules would be appropriate.

4.8.2.3 User Characteristics

It should be assumed that the users of the Style Guide will be knowledgeable software engineers but will have only a rudimentary understanding of the human factor considerations affecting user interface design.

4.8.2.4 General Constraints

Not applicable.

4.8.2.5 Assumptions & Dependencies

Completion of design rule specification will be dependent upon the specification and performance criteria for the major system components. Test applications will need to be constructed using the design rules specifications to determine their validity and suitability. These test applications may be implemented using software which models overall system performance and functionality.

4.8.3 Specific Requirements

4.8.3.1 Outline

The Style Guide shall be an extensive document containing information relevent to user interface development in addition to the material specified here as required for inclusion.

4.8.3.2 Required Topics

4.8.3.2.1 System Model

A generic system model shall be defined and used to explain and illustrate the concepts of the overall system to the user. Since subsystems may be defined which have models independent of the system model, guidelines shall be specified as to how subsystem models should relate to the overall system model.

4.8.3.2.2 Modes

Guidelines shall be established to indicate when modes may be useful and how they should be managed, including default modes, mode status presentation, and mode changes. The guidelines shall also explain the relationship of modes to the applicable model and when the use of modes is inappropriate. Design rules defining the use and management of modes appropriate to common standardized functionality are encouraged.

4.8.3.2.3 System Names and Icons

Design rules shall be established which provide standardized names and icons for standard system functions, utilities and objects, and for functions or capabilities common to system procedures and applications. The standardized names and icons shall be used in the construction of user interfaces and in all user manuals.

Guidelines shall be provided to

a. Indicate when an alternative to the standardized name may be used.

b. Assist the designer in selecting names and in the design of icons for functions, utilities, and objects which have not been provided with standard names or standard icons.

4.8.3.2.4 Cursors/Prompts

Design rules shall be established that define:

a. The shape of cursors to be used for certain functions, such as the system cursor and for text manipulation.

b. Prompts to be used in conjunction with common system functions and utilities.

Guidelines should be provided which describe

a. The significant factors a designer should consider in the construction of a new cursor. (These guidelines should also explain when and how a cursor may be used as a prompt.)

b. The purpose of prompts, how various interactive objects may be used as prompts, and the type of prompts to be used according to the situation.

4.8.3.2.5 Status

Guidelines shall be provided describing those situations for which status should be presented to a user automatically, constantly, and/or on request. Techniques for providing implied status should also be discussed. Design rules shall be established to specify those factors which will be common to all explicit displays of status, particularly those related to the presentation of status.

4.8.3.2.6 Feedback/Response Times

Guidelines shall be provided explaining user requirements for feedback and the levels of feedback which are needed for user comfort. Design rules shall be established which will establish the minimum requirements for response times to user actions.

4.8.3.2.7 Text Manipulation

Design rules shall be established which will define the user interface as related to the entry and manipulation (editing, deleting, copying, etc.) of text.

4.8.3.2.8 Menus

Guidelines shall be provided which discuss the appropriate use of menus including the factors which affect the beneficial use of menus.

Design rules shall be established which will define:

a. The methods which will be used to present menus and the number of menu levels which will be acceptable.

b. How current status will be reflected in menu selections for both inclusive and exclusive option selection.

c. How menu selection, including "no selection", should occur

4.8.3.2.9 Forms

Guidelines shall be provided which discuss the appropriate use of forms including the factors which affect the beneficial use of forms.

Design rules shall be established:

a. Which will define the methods which will be used to present forms and, in particular, standard explanatory text needed to identify the application and the form, itself.

b. For presenting options (both currently available options and possible future options), default entries, and unfilled fields.

c. For identifying common restricted fields including numeric only and alpha only fields.

d. Which will allow the user to distinguish between fields which require an entry and those for which an entry is optional.

e. Which will define how fields will be selected, how movement from one field to another shall occur, and how data within a field may be manipulated.

f. Which will define a common method for the user to indicate completion of a form. (These Design Rules shall specify how a user will be notified of required fields which have been left empty and of fields which have been incorrectly filled.)

4.8.3.2.10 Help

Guidelines shall be provided which discuss user requirements for help and assistance. These guidelines should also address the different needs for tutorial help and the help needed to guide an otherwise knowledgeable user.

Design rules shall be established

a. Which will define how a user will access help and indicate the level of help required.

b. Defining how help will be presented

4.8.3.2.11 Errors

Guidelines shall be provided which will discuss the types of errors which may occur as part of the user interface and how they may be prevented or reduced. Particular emphasis

should be placed on the prevention of catastrophic errors and the methods to be employed for command confirmation.

Design rules shall be established:

a. Which will define how errors shall be reported and presented to the user.

b. Defining how a user should be notified that certain actions are not permissible. Levels of notification should be defined.

4.8.3.3 Guideline and Design Rule Numbering

Guidelines and design rules shall be concisely stated as declarative statements. A numbering system shall be established for easy identification and locating of guidelines. A separate and distinguishable (though possibly related) numbering system shall be established for easy identification and locating of design rules.

4.8.3.4 Design Rule Exceptions

Guidelines shall be provided describing those conditions which may exist which would make following of a design rule inappropriate. A review process shall be outlined which may be followed to obtain approval for a design rule exception.

4.8.3.5 Appendices

At least one appendix will be provided as a part of the Style Guide which will list the design rules in numeric order with cross references to related Guidelines and other related design rules.

4.8.3.6 Index

An index shall be provided which will allow easy location of guidelines and design rules applicable to a given subject.

Distribution List for IDA Paper P-1893

Sponsor

Maj. Terry Courtwright WIS Joint Program Management Office 7798 Old Springfield Road McLean, VA 22102	5 copies
Maj. Sue Swift Room 3E187 The Pentagon Washington, D.C. 20301-3040	5 copies

Other

Col. Joe Greene STARS Joint Program Office 1211 Fern St., Room C107 Arlington, VA 22202	1 copy
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2 copies

CSED Review Panel

Dr. Dan Alpert, Director Center for Advanced Study University of Illinois 912 W. Illinois Street Urbana, Illinois 61801	1 copy
Dr. Barry W. Boehm TRW Defense Systems Group MS 2-2304 One Space Park Redondo Beach, CA 90278	1 copy
Dr Ruth Davis The Pymatuning Group, Inc. 2000 N. 15th Street, Suite 707 Arlington, VA 22201	1 copy
Dr. Larry E. Druffel Software Engineering Institute Shadyside Place 580 South Aiken Ave. Pittsburgh, PA 15231	1 copy

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
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. Robert Knapper, 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

END

4-87

DTIC