GRASP/Ada

Graphical Representations of Algorithms, Structures, and Processes for Ada

The Development of a Program Analysis Environment for Ada

Reverse Engineering Tools For Ada

Task 2, Phase 3 Final Report

Contract Number NASA-NCC8-14

Department of Computer Science and Engineering
Auburn University, AL 36849-5347

Contact: James H. Cross II, Ph.D.
Principal Investigator
(205) 844-4330

September 30, 1991

GRASP/Ada

Graphical Representations of Algorithms, Structures, and Processes for Ada

Reverse Engineering Tools For Ada

Task 2, Phase 3 Final Report
Contract Number NASA-NCC8-14

James H. Cross II, Ph.D.
Principal Investigator

September 30, 1991

Abstract: The GRASP/Ada project (*Graphical Representations of Algorithms, Structures, and Processes for Ada*) has successfully created and prototyped a new algorithmic level graphical representation for Ada software, the Control Structure Diagram (CSD). The primary impetus for creation of the CSD was to improve the comprehension efficiency of Ada software and thus improve reliability and reduce costs. The emphasis has been on the automatic generation of the CSD from Ada source code to support reverse engineering and maintenance. The CSD has the potential to replace traditional prettyprinted Ada source code. In Phase 1 of the GRASP/Ada project, the CSD graphical constructs were created and applied manually to several small Ada programs. A prototype (Version 1) was designed and implemented using FLEX and BISON running under VMS on a VAX 11-780. In Phase 2, the prototype was improved and ported to the Sun 4 platform under UNIX. A user interface was designed and partially implemented using the HP widget toolkit and X Windows. The prototype was applied successfully to numerous Ada programs ranging in size from several hundred to several thousand lines of source code. In Phase 3 of the project, the prototype was prepared for limited distribution (GRASP/Ada Version 3.0) to facilitate evaluation. The user interface was extensively reworked using the Athena widget toolkit and X Windows. The current prototype provides the capability for the user to generate CSD from Ada source code in a reverse engineering mode with a level of flexibility suitable for practical application.

ACKNOWLEDGEMENTS

We appreciate the assistance provided by NASA personnel, especially Mr. Keith Shackelford whose guidance has been of great value. Portions of this report were contributed by each of the members of the project team. The following is an alphabetical listing of the project team members.

Faculty Investigator:

Dr. James H. Cross II, Principal Investigator

Graduate Research Assistants:

Richard A. Davis
Charles H. May
Kelly I. Morrison
Timothy A. Plunkett
Narayana S. Rekapalli
Darren Tola

The following trademarks are referenced in the text of this report.

Ada is a trademark of the United States Government, Ada Joint Program Office.

Software through Pictures (StP), **Ada Development Environment (ADE)**, and **IDE** are trademarks of Interactive Development Environments.

PostScript is a trademark of Adobe Systems, Inc.

VAX and **VMS** are trademarks of Digital Equipment Corporation.

VERDIX and **VADS** are trademarks of Verdix Corporation.

UNIX is a trademark of AT&T.

TABLE OF CONTENTS

1.0 Introduction and Executive Summary 1
- 1.1 Phase 1 - The Control Structure Diagram For Ada 2
- 1.2 Phase 2 - The GRASP/Ada Prototype and User Interface 2
- 1.3 Phase 3 - CSD Generation Prototype and Preliminary Object Diagram Prototype 4

2.0 The System Model 6
- 2.1 GRASP/Ada System Data Flow 6
- 2.2 GRASP/Ada System Block Diagram 6

3.0 Control Structure Diagram Generator 12
- 3.1 Generating the CSD 12
- 3.2 Displaying the CSD - Screen and Printer 14
- 3.3 Displaying the CSD - Future Considerations 15
- 3.4 Incremental Changes to the CSD 16
- 3.5 Navigating Through Large CSDs - Alternatives 17
- 3.6 Internal Representation of the CSD - Alternatives 17
- 3.7 Additional CSD Constructs - Alternatives 19

4.0 User Interface 20
- 4.1 System Window 21
- 4.2 Source Window 21
- 4.3 Control Structure Diagram Window 24
- 4.4 Help Window 28

5.0 The GRASP Library 31

6.0 Object Diagram Generator 33
- 6.1 ODgen Symbol Set 33
- 6.2 Symbol Interconnections and Diagram Layout 37
- 6.3 GRASP/Ada ODgen Processing Alternatives 39
- 6.4 Displaying the OD - Screen and Printer 42
- 6.5 Incremental Changes to the OD 45
- 6.6 Internal Representation of the OD - Alternatives 47
- 6.7 Navigation Through Large ODs - Alternatives 48
- 6.8 Exploding/Imploding the OD 50
- 6.9 Generating a Set of ODs 51
- 6.10 Printing An Entire Set of ODs 51
- 6.11 Relating the CSD and OD - Alternatives 51
- 6.12 Index and Table of Contents Relating the CSDs and ODs 52
- 6.13 Design and Implementation of Preliminary ODgen Prototype 53

7.0 Future Requirements 65
- 7.1 Phase 1 - Generators and Editors for Visualizations 65
- 7.2 Phase 2 - Evaluation and Extension 67
- 7.3 Phase 3 - Evaluation and Integration with Commercial Systems 68

BIBLIOGRAPHY 70

APPENDICES 76

A. "Reverse Engineering"
by J. Cross, E. Chikofsky and C. May

B. "Control Structure Diagrams For Ada"
by J. Cross, S. Sheppard and H. Carlisle

C. Extended Examples

D. User Manual (MAN-Page)

LIST OF FIGURES

Figure 1. GRASP/Ada Overview 3

Figure 2. GRASP/Ada Context Level Data Flow Diagram 7

Figure 3. GRASP/Ada System Level Data Flow Diagram 8

Figure 4. GRASP/Ada System Block Diagram 7

Figure 5. Control Structure Diagram Constructs 13

Figure 6. GRASP/Ada System Window 22

Figure 7. GRASP/Ada Source Code Window 23

Figure 8. GRASP/Ada CSD Window 25

Figure 9. GRASP/Ada File Selection Window 26

Figure 10. GRASP/Ada Help Window - Rendezvous Construct 29

Figure 11. GRASP/Ada Help Window - Display All Constructs 30

Figure 12. The OOSD Notation Symbol Set 34

Figure 13. Typical ODgen User Interface Window 54

Figure 14. ODgen Source Code with CSD 56

Figure 15. ODgen All Diagram View 57

Figure 16. ODgen Multiple View 58

Figure 17. ODgen Development Approach 59

1.0 Introduction and Executive Summary

Computer professionals have long promoted the idea that graphical representations of software can be extremely useful as comprehension aids when used to supplement textual descriptions and specifications of software, especially for large complex systems. The general goal of this research has been the investigation, formulation and generation of *graphical representations of algorithms, structures, and processes for Ada* (GRASP/Ada). This task, in which we described and categorized various graphical representations that can be extracted or generated from source code, has focused on *reverse engineering*.

Reverse engineering normally includes the processing of source code to extract higher levels of abstraction for both data and processes. The primary motivation for reverse engineering is increased support for software reusability, verification, and software maintenance, all of which should be greatly facilitated by automatically generating a set of "formalized diagrams" to supplement the source code and other forms of existing documentation. For example, Selby [SEL85] found that code reading was the most cost effective method of detecting errors during the verification process when compared to functional testing and structural testing. And Standish [STA85] reported that program understanding may represent as much as 90% of the cost of maintenance. Hence, improved comprehension efficiency resulting from the integration of graphical notations and source code could have a significant impact on the overall cost of software production. The overall goal of the GRASP/Ada project is to provide the foundation for a CASE (computer-aided software engineering) environment in which reverse engineering and forward engineering (development) are tightly coupled. In this environment, the user may specify the software

in a graphically-oriented language and then automatically generate the corresponding Ada code [ADA83]. Alternatively, the user may specify the software in Ada or Ada/PDL and then automatically generate the graphical representations either dynamically as the code is entered or as a form of post-processing.

Figure 1 provides an overview to the three phases of the GRASP/Ada project. Ada source code or PDL is depicted as the starting point for application of the GRASP/Ada toolset. Each phase is briefly described below in the order that diagrams might be generated in a typical reverse engineering scenario.

1.1 Phase 1 - The Control Structure Diagram For Ada

Phase 1 concentrated on a survey of graphical notations for software and the development of a new algorithmic or PDL/code level diagram for Ada. Tentative graphical control constructs for the *Control Structure Diagram* (CSD) were created and initially prototyped in a VAX/VMS environment. This included the development of special diagramming fonts for both the screen and printer and the development of parser and scanner using UNIX based tools such as LEX and YACC. Appendix B provides a detailed description of the CSD and the rationale for its development. The final report for Phase 1 [CRO89] contains a complete description of all accomplishments of Phase 1.

1.2 Phase 2 - The GRASP/Ada Prototype and User Interface

During Phase 2, the prototype was extended and ported to a Sun/UNIX environment. The development of a user interface based on the X Window System represented a major part of the extension effort. Verdix Ada and the Verdix DIANA interface were acquired as potential commercial tools upon which to base the GRASP/Ada prototype. Architectural

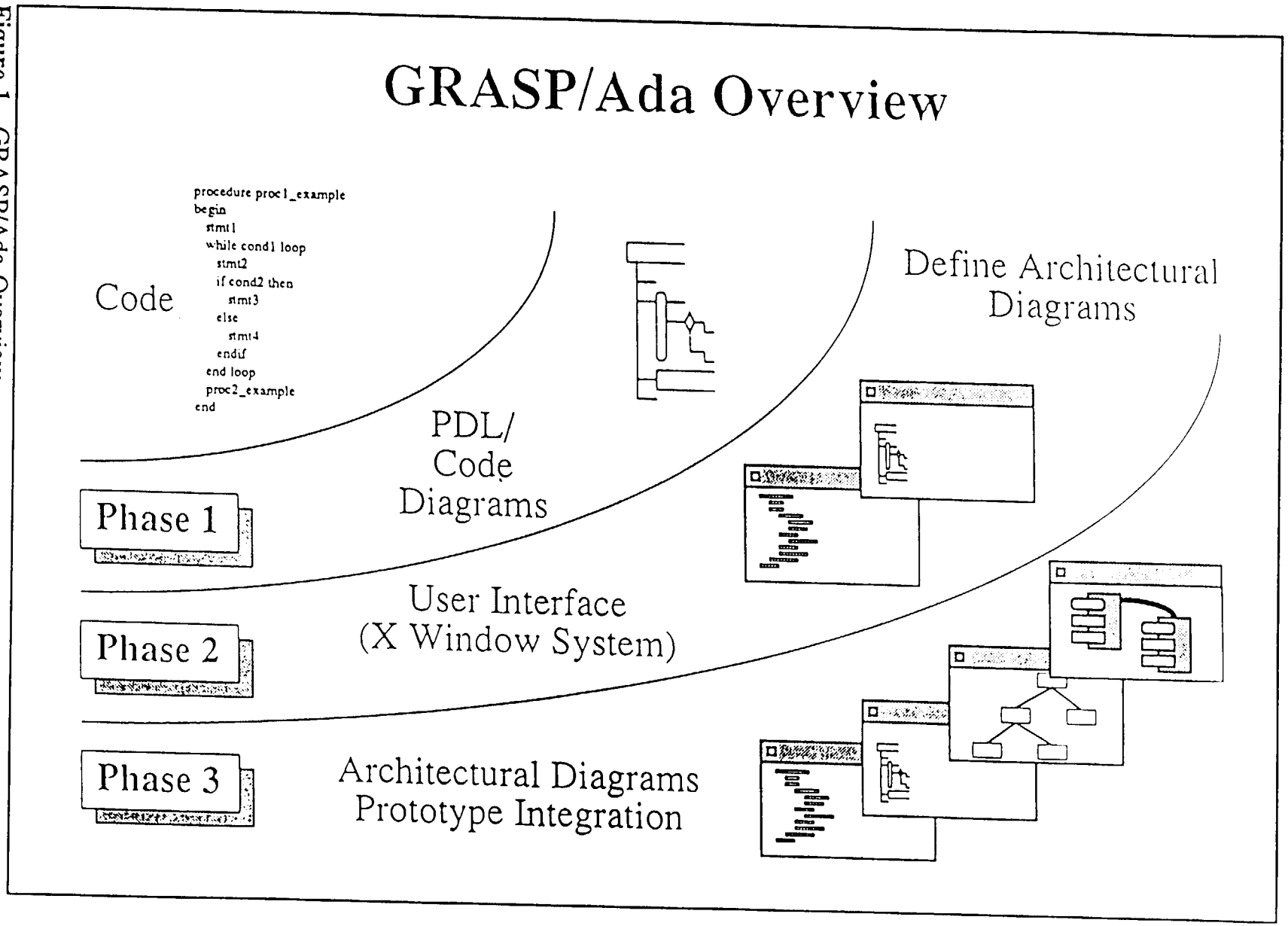


Figure 1. GRASP/Ada Overview

diagrams for Ada were surveyed and the OOSD notation [WAS89] was identified as having the best potential for accurately representing many of the varied architectural features of an Ada software system. Phase 2 also included the preliminary design and a separate exploratory prototype for an architectural CSD. The best aspects of architectural CSD are expected to be integrated into the fully operational GRASP/Ada prototype during a future phase of the project. The final report for Phase 2 [CRO90c] contains a complete description of the accomplishments of Phase 2.

1.3 Phase 3 - CSD Generation Prototype and Preliminary Object Diagram Prototype

Phase 3 has had two major thrusts: (1) completion and limited release of an operational GRASP/Ada prototype which generates CSDs and (2) the development of a preliminary prototype which generates object diagrams directly from Ada source code. Completion of the GRASP/Ada CSD prototype (CSDgen) included the addition of substantial functionality, via the User Interface, to make the prototype easier to use. CSDgen was installed and demonstrated on a Sun workstation at Marshall Space Flight Center, Alabama in September 1991. It is currently installed and in use in several software engineering courses at Auburn University. The latter, in particular, is providing information for evaluation prior to widespread release to teaching and research communities, business, and industry. To date, over 100 requests for information regarding GRASP/Ada have been received as a result of publications generated from this research. Responding to these requests is an important element of the ongoing evaluation and refinement of the GRASP/Ada reverse engineering system.

The development of a preliminary prototype for generating architectural object diagrams (ODgen) for Ada source/PDL has been an effort to determine feasibility rather than

to deliver an operational prototype as was the case with CSD generator above. The preliminary prototype has indicated that the development of the components to recover the information to be included in the diagram, although a major effort, is relatively straightforward. However, the research has also indicated that the major obstacle for automatic object diagram generation is the automatic layout of the diagrams in a human readable and/or aesthetically pleasing format. A user extensible rule base, which automates the diagram layout task, is expected to be formulated during future GRASP research. Interactive Development Environment's Software through Pictures (IDE/StP), which supports the OOSD notation in a forward engineering sense, has been identified as a candidate for a commercial CASE environment with which to integrate GRASP/Ada reverse engineering system.

The following sections of this report describe the overall GRASP/Ada system model, the control structure diagram generator, the user interface, the library, the object diagram generator, and future requirements. Appendix A contains paper entitled "Reverse Engineering" which has been accepted for publication in *Advances in Computing*. This paper, which was written during Phases 2 and 3, provides a taxonomy of reverse engineering and comprehensive review of the current literature. Appendix B contains a paper entitled "Control Structure Diagrams for Ada," published in *Journal of Pascal, Ada & Modula 2*. This paper, which was written during Phase 1, describes the overall rationale for the development of the CSD. Appendix C contains a CSD produced from Ada source code provided by Marshall Space Flight Center. Appendix D contains the MAN-page which describes each of its the current options.

2.0 The System Model

The general system model for the GRASP/Ada prototype is described in this section. The overall functionality of the system is briefly described from a data flow perspective and then each of the GRASP/Ada components is presented in the form of a system block diagram.

2.1 GRASP/Ada System Data Flow

Figures 2 describes the context and overall flow of information to and from the GRASP/Ada system. The primary input is Ada source code and GRASP commands and the primary outputs are control structure diagrams, object diagrams and library information. The Ada source code is assumed to be syntactically correct.

Figure 3 describes the major processes and overall flow of information within the GRASP/Ada system. Process 1 parses Ada source code and produces a parse tree, comments, symbolic and unit information. Process 2 produces CSD files as CSDgen action routines are called during the parse in Process 1. Process 3 produces object diagrams from symbolic and unit information. Processes 4 and 5 produce screen and printer images of control structure diagrams and object diagrams from intermediate files via the user interface when appropriate commands are received.

2.2 GRASP/Ada System Block Diagram

Figure 4 depicts the major system components hierarchically to illustrate the layers and component interfaces. The user interface (not shown in the system data flow diagram) was built using the X Window System and provides general control and coordination among

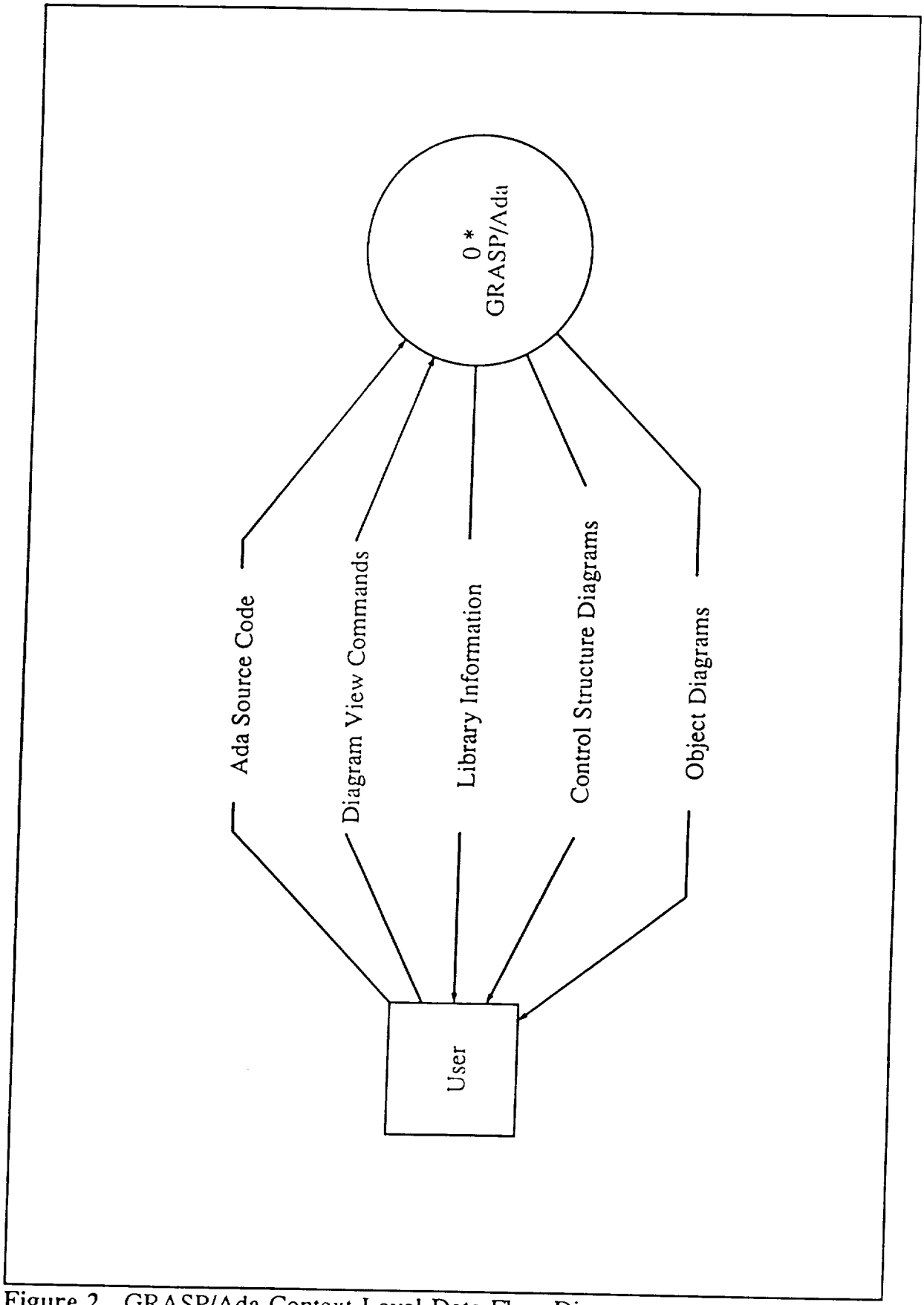


Figure 2. GRASP/Ada Context Level Data Flow Diagram

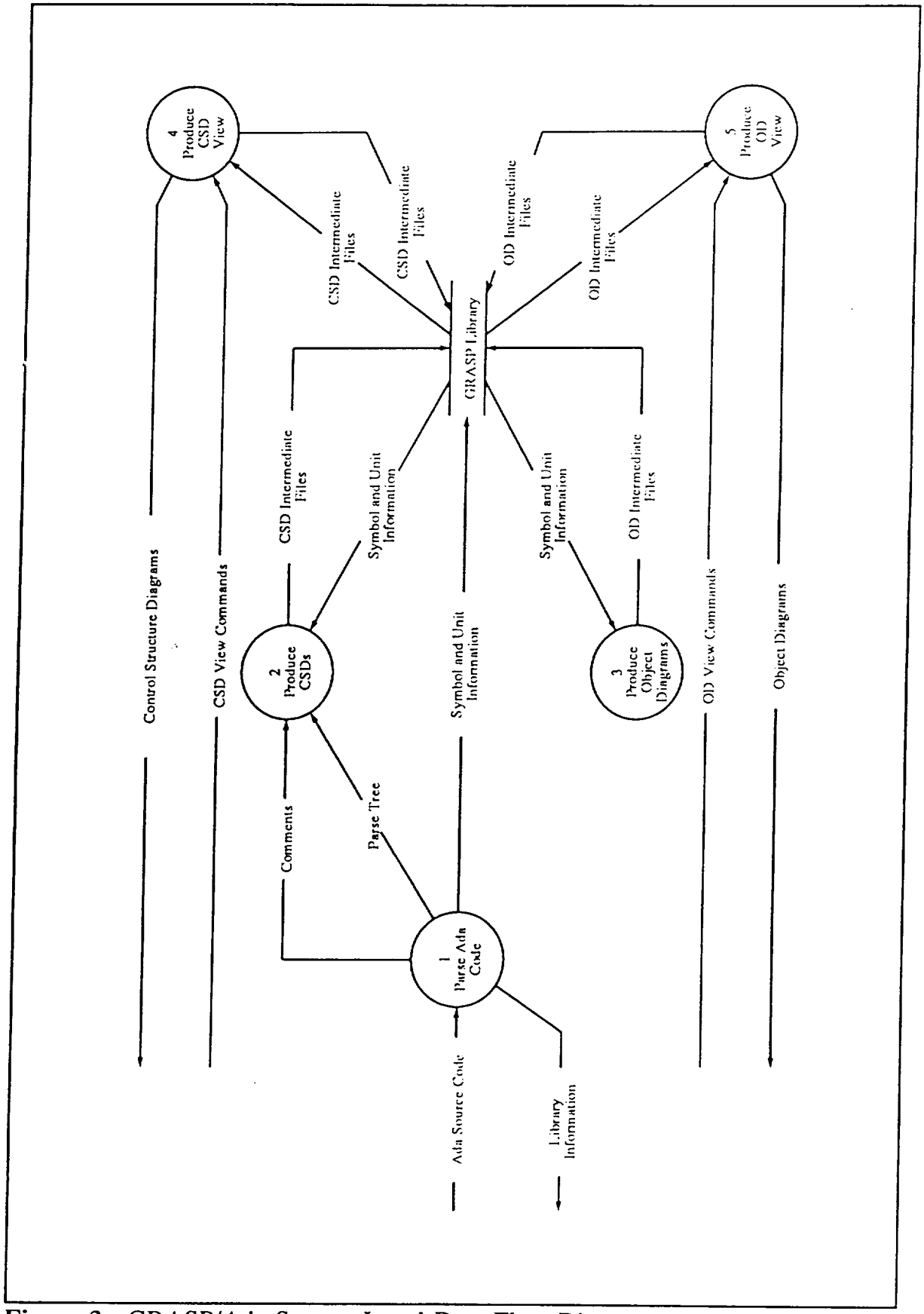


Figure 3. GRASP/Ada System Level Data Flow Diagram

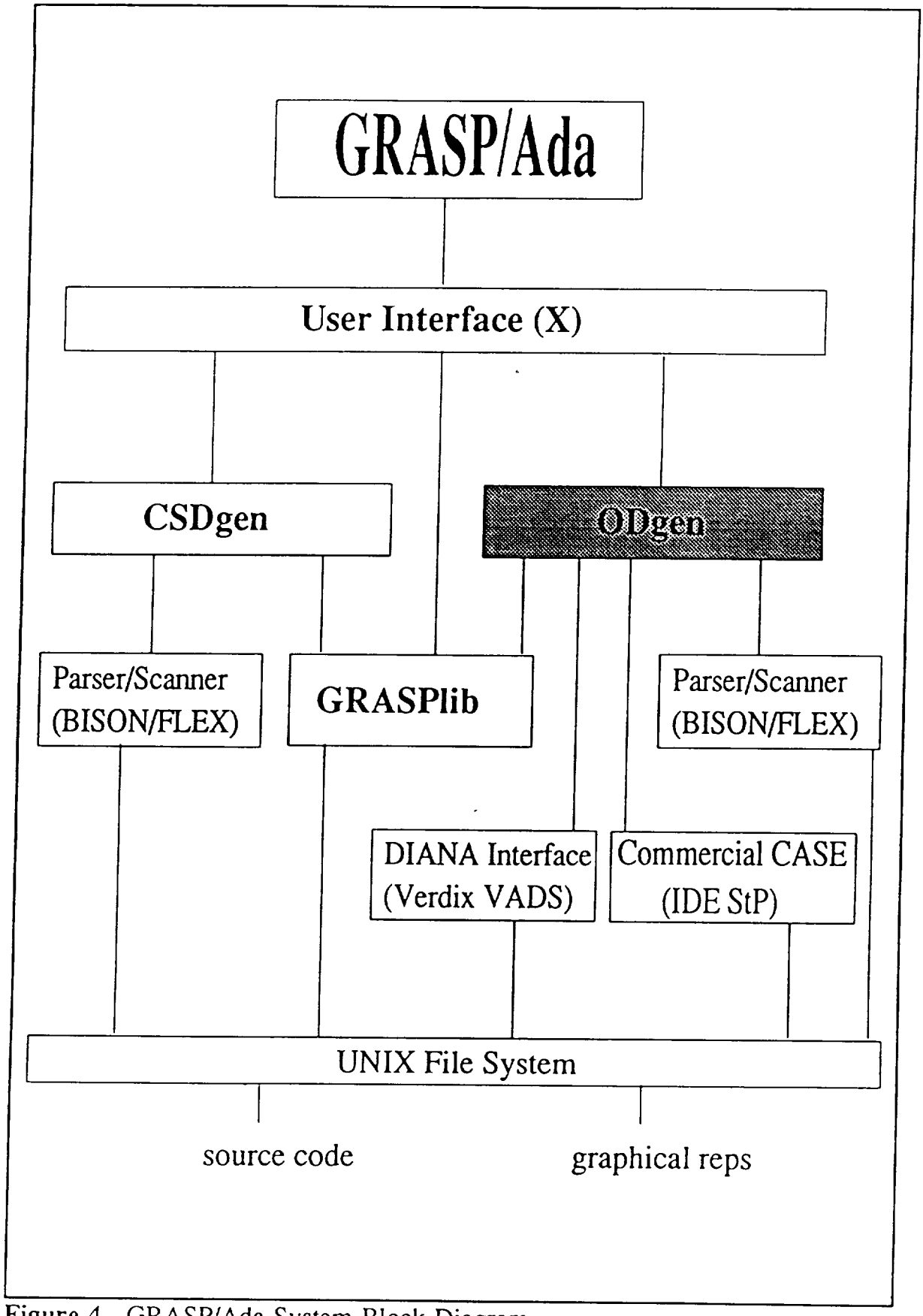


Figure 4. GRASP/Ada System Block Diagram

the other components.

The control structure diagram generator, **CSDgen**, has its own parser/scanner built using FLEX and BISON, successors of LEX and YACC. It also includes its own printer utilities. As such, CSDgen is a self-sufficient component which can be used from the user interface or the command line without the commercial components. When changes are made to the Ada source code or PDL, currently the user must modify the original source file rather than the generated CSD file. The entire file must be reparsed to produce an updated CSD. A CSD editor (**CSDedit**), which will provide for dynamic incremental modification of the CSD, is currently in the planning stages.

The object diagram generation component, **ODgen**, is in the analysis phase and has been implemented as a separate preliminary prototype. The shading indicates planned integration. The feasibility of automatic diagram layout remains under investigation. Beyond automatic diagram layout, several design alternatives have been identified. The major alternatives include the decision of whether to attempt to integrate GRASP/Ada directly with commercial components, namely (1) the Verdix Ada development system (VADS) and DIANA interface for extraction of diagram information and (2) IDE's Software through Pictures, Ada Development Environment (IDE/StP/ADE) for the display of the object diagrams. Each of these components are indicated in Figure 4.

The GRASP/Ada library component, **GRASPlib**, provides for coordination of all generated items with their associated source code. Its purpose is to facilitate navigation among the diagrams and the production of sets of diagrams. Both CSDgen and ODgen produce library entries as Ada source is processed. Currently, these consists of directories of UNIX files with identifying extensions.

In the following sections, the general functional requirements and prototype implementation are described for each of the major GRASP/Ada components: the control structure diagram generator, the user interface, and the object diagram generator.

3.0 Control Structure Diagram Generator

The GRASP/Ada control structure diagram generator (CSDgen) is described in this section. The rationale for the development of the CSD, which has been detailed in previous reports [CRO89, CRO90c], is summarized in Appendix B. Examples of the CSD are presented in conjunction with the User Interface in Section 4.0 and in Appendix C, Extended Examples. The UNIX MAN-page description of the options is contained in Appendix D.

3.1 Generating the CSD

The primary function of CSDgen is to produce a CSD for a corresponding Ada source file. The graphical constructs produced by CSDgen are summarized in Figure 5. CSDgen has its own parser/scanner constructed using LEX/YACC based software tools available with UNIX. Although a complete parse is done during CSD generation, CSDgen assumes the Ada source code has been previously compiled and thus is syntactically correct. Currently, little error recovery is attempted when a syntax error is encountered. The diagram is simply generated down to the point of the error. The current CSDgen prototype builds the diagram directly during the parse by inserting CSD graphics characters into a file along with text. To increase efficiency and improve extensibility, future versions of the CSDgen prototype will use a more abstract intermediate representation.

Since GRASP/Ada is expected to be used to process and analyze large existing Ada software systems consisting of perhaps hundreds of files, an option to generate all the CSDs at once is provided. Generating a set of CSDs is facilitated by entering **.a* or some other

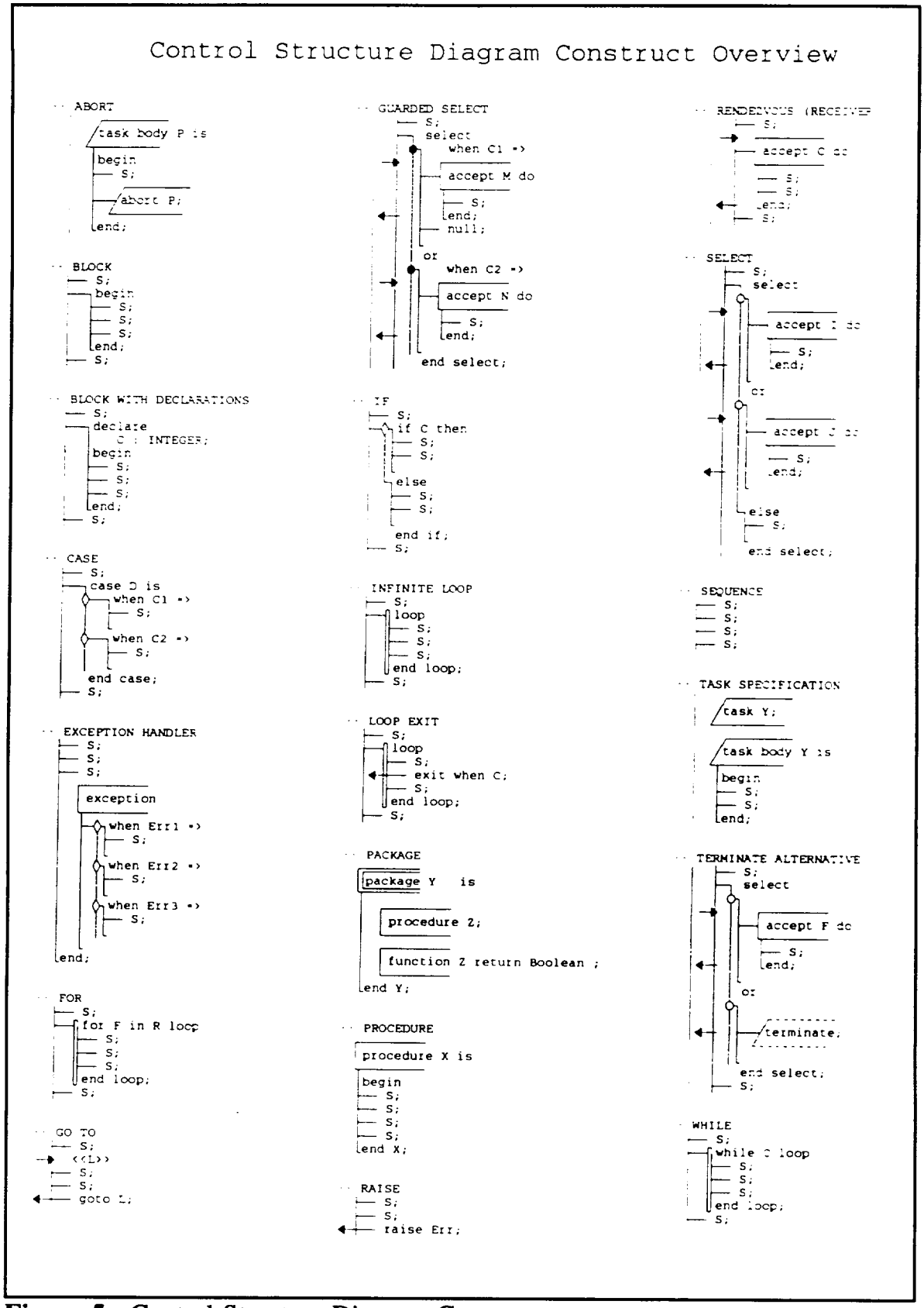


Figure 5. Control Structure Diagram Constructs

wildcard with a conventional source file extension, for the file name. A CSD generation **summary window** displays the progress of the generation by listing each file as it is being processed and any resulting error messages. The summary concludes with number of files processed and the number of errors encountered. The default for each CSD file name is the source file name with **.csd** appended. If an error is encountered, an extension of **.err** is used. As the CSDs are generated, the GRASP library is updated, which currently consists of populating a specified directory with file images of the CSDs. Generating a set of CSDs can be considered a user interface requirement rather than strictly a CSD generator requirement.

3.2 Displaying the CSD - Screen and Printer

Basic display capabilities to the screen and printer were implemented during Phase 2. Screen display is facilitated by sending the CSD file to a CSD window opened under an X Window manager. Printing is accomplished by converting the CSD file to a PostScript file and then sending it to a printer. Moving to a more abstract intermediate representation in future versions will necessitate the development of a new set of display routines which will be X Window System based. However, these new routines will increase the flexibility and capability of CSDgen, thus making it more immediately useful to the research community.

CSD Screen Fonts. The default CSD screen font is a bitmap 14 point Courier to which the CSD graphic characters have been added. The font was defined as a bitmap distribution font (BDF) then converted to SNF format required by the X Window System. Four additional screen fonts ranging from 5 to 18 point are user selectable.

CSD Printer Fonts. CSD Printer fonts were initially developed for the HP LaserJet series. These were then implemented as PostScript type 3 fonts and all subsequent font

development has been directed towards the PostScript font. The PostScript font provides the most flexibility since its size is user selectable from 1 to 300 points.

3.3 Displaying the CSD - Future Considerations

Layout/Spacing. The general layout of the CSD is highly structured by design. However, the user should have control over such attributes as horizontal and vertical spacing and the optional use of some diagramming symbols. In the current Version 3 CSDgen prototype, horizontal and vertical spacing are not user selectable. They are a part of the CSD file generation and are defaulted to single spacing with 80 characters per line. In order to change these, e.g., from single to double spacing, the CSD file would have to be regenerated. In future versions of the prototype, these options will be handled by the new display routines and, as such, can be modified dynamically without regenerating the CSD file.

Vertical spacing options will include single, double, and triple spacing (default is single). Margins will be roughly controlled by the character line length selected, either 80 or 132 characters per line (default is 80). Indentation of the CSD constructs has been a constant three blank characters. Support for variable margins and indentation is being investigated in conjunction with the new display routines. In addition, several display options involving CSD graphical constructs are under consideration. For example, the boxes drawn around procedure and task entry calls may be optionally suppressed to make the diagram more compact.

Collapsing the CSD. The CSD window should provide the user with the capability to collapse the CSD based on all control constructs as well as complete diagram entities (e.g., procedures, functions, tasks and packages). This capability directly combines the ideas of chunking with control flow which are major aids to comprehension of software. An

architectural CSD (ArchCSD) [DAV90] can be facilitated by collapsing the CSD based on procedure, function, and task entry calls, and the control constructs that directly affect these calls. The initial ArchCSD prototype was completely separate from CSDgen and required complete regeneration of the ArchCSD file for each option. In future versions of the prototype, the ArchCSD will be generated by the display routines from the single intermediate representation of the CSD.

Color. Although general color options such as background and foreground may be selected via the X Windows system, color options within a specific diagram were only briefly investigated for both the screen and printer. It was decided that these will not be pursued in the Version 3 prototype.

Printing An Entire Set of CSDs. Printing an entire set of CSDs in an organized and efficient manner is an important capability when considering the typically large size of Ada software systems. A book format is under consideration which would include a table of contents and/or index. In the event GRASP/Ada is integrated with IDE/StP/ADE, the StP Document Preparation System could possibly be utilized for this function.

3.4 Incremental Changes to the CSD

In the present prototype, there is no capability for editing or incrementally modifying the CSD. The source code is modified using a text editor and then the CSD is regenerated. While this has been sufficient for early prototyping, especially for small programs, editing capabilities are desirable in an operational setting. An editor has been proposed and is briefly discussed in Section 7.0 Future Requirements.

3.5 Navigating Through Large CSDs - Alternatives

Index (or Table of Contents). An index, similar to that presented in the Xman application provided with the X Window System for viewing manual pages, is used to navigate among a system of CSDs. The user clicks on the index entry and the corresponding CSD is displayed. The index entries would be created as CSDs are generated and stored in the GRASP/Ada library. Entries in the library are to include procedures, functions, tasks, task entries, and packages. See Section 6 below for details.

Direct Navigation Via CSD. The user is allowed to click on procedure, function, and task entry calls in the CSD directly and a separate CSD window is opened containing the selected CSD or fragment thereof. Two potential problems have been identified with this approach. Using the mouse for selection may conflict with established editing functions supported by the mouse. In addition, it may be difficult to relate the characters or CSD graphical construct with subprogram and entry names. The availability of middle mouse button for this purpose is being investigated.

3.6 Internal Representation of the CSD - Alternatives

Several alternatives are under consideration for the internal representation of the CSD in the Version 3 prototype. Each has its own merits with respect to processing and storage efficiency and is briefly described below.

Single ASCII File with CSD Characters and Text Combined. This is the most direct approach and is currently used in the version 2 prototype. The primary advantage of this approach is that combining the CSD characters with text in a single file eliminates the need for elaborate transformation and thus enables the rapid implementation of prototypes as was the case in the previous phases of this project. The major disadvantages of this approach

are that it does not lend itself to incremental changes during editing and the CSD characters have to be stripped out if the user wants to send the file to a compiler.

Separate ASCII Files for CSD Characters and Text. In this approach, the file containing the CSD characters along with placement information would be "merged" with the prettyprinted source file. The primary advantage of the this approach is that the CSD characters would not have to be stripped out if the user wants to send the file to a compiler. The major disadvantage of this approach is that coordinating the two files would add complexity to generation and editing routines with little or no benefit. As a result, this approach would be more difficult to implement than the single file approach and not provide the advantages of the next alternative.

Single ASCII File Without Hard-coded CSD Characters. This approach represents a compromise between the previous two. While it uses a single file, only "begin construct" and "end construct" codes are actually required for each CSD graphical construct in the CSD file rather than all CSD graphics characters that compose the diagram. In particular, no continuation characters would be included in the file. These would be generated by the screen display and print routines as required. The advantages of this approach would be most beneficial in an editing mode since the diagram could grow and shrink automatically as additional text/source code is inserted into the diagram. The extent of required modifications to text edit windows must be considered with this alternative.

Direct Generation From DIANA Net. If tight coupling and integration with a commercial Ada development system such as Verdix VADS is desired, then direct generation of the CSD from the DIANA net produced as a result of compilation could be performed. This would require a layer of software which traverses the DIANA net and calls the appropriate CSD primitives as control nodes are encountered. This approach would

apparently eliminate the possibility of directly editing the CSD since the DIANA interface does not support modifying the net, only reading it.

3.7 Additional CSD Constructs - Alternatives

The following CSD constructs are under consideration for future versions of the prototype.

Generic Task and Package. Dashed task and package symbols should be used to distinguish between generic and non-generic tasks and between generic and non-generic packages.

Function Call. A CSD symbol similar to that used for procedure calls should be used for function calls for consistency.

4.0 User Interface

GRASP/Ada user interface was developed using the X Window System, Version 11 Release 4 (X11R4). The X Window System, or simply X, meets the GRASP/Ada user interface requirements of an industry-standard window based environment which supports portable graphical user interfaces for application software. Some of the key features which make X attractive for this application are its availability on a wide variety of platforms, unique device independent architecture, adaptability to various user interface styles, support from a consortium of major hardware and software vendors, and low acquisition cost. With its unique device independent architecture, X allows programs to display windows on any hardware that supports the X Protocol. X does not define any particular user interface style or policy, but provides mechanisms to support many various interface styles.

The Version 3 prototype user interface is a significant extension of Version 2. It allows the user to open one or more source windows to read or edit source code in the usual way. The user may open one or more CSD windows, indicate corresponding source files and CSD files, and then generate the CSD from each of the indicated source files. If the CSD was generated previously, the source file is not required by the CSD window. In either case, the CSD window allows the user to scroll through the CSD.

The specifications and figures that follow are intended to define the look and feel of the GRASP/Ada User Interface as well as indicate much of its current and future functionality. The Ada source code used in the figures was extracted from the AERO.DAP.PACKAGE provided by NASA to test the CSD generator. Complete CSDs for

the files processed are included in Appendix C. For a complete description of the options available through the user interface, see the MAN-page in Appendix E.

4.1 System Window

The System window, shown in Figure 6, provides the user with the overall organization and structure of the GRASP/Ada tool. Option buttons include: General, Source Code, and Control Structure Diagram. These are briefly described below. A future button is planned for Object Diagram.

General - This option provides access to the environment including loading of fonts for X and selection of printers.

Source Code - This option allows the user to open one or more windows for viewing and editing source code.

Control Structure Diagram - This option allows the user to open one or more windows for viewing CSDs.

Help - This option opens a window containing a summary of the CSD graphical constructs. The user may scroll the combined display or selectively display individual constructs.

4.2 Source Window

The Source window, shown in Figure 7, provides the user with the general capabilities of a text editor. It is included in the GRASP/Ada system for completeness since the system uses source code as its initial input. The user may elect to use any suitable editor callable from the X environment. A future version of GRASP/Ada will allow the user to edit the CSD directly, making a pure text editor redundant.

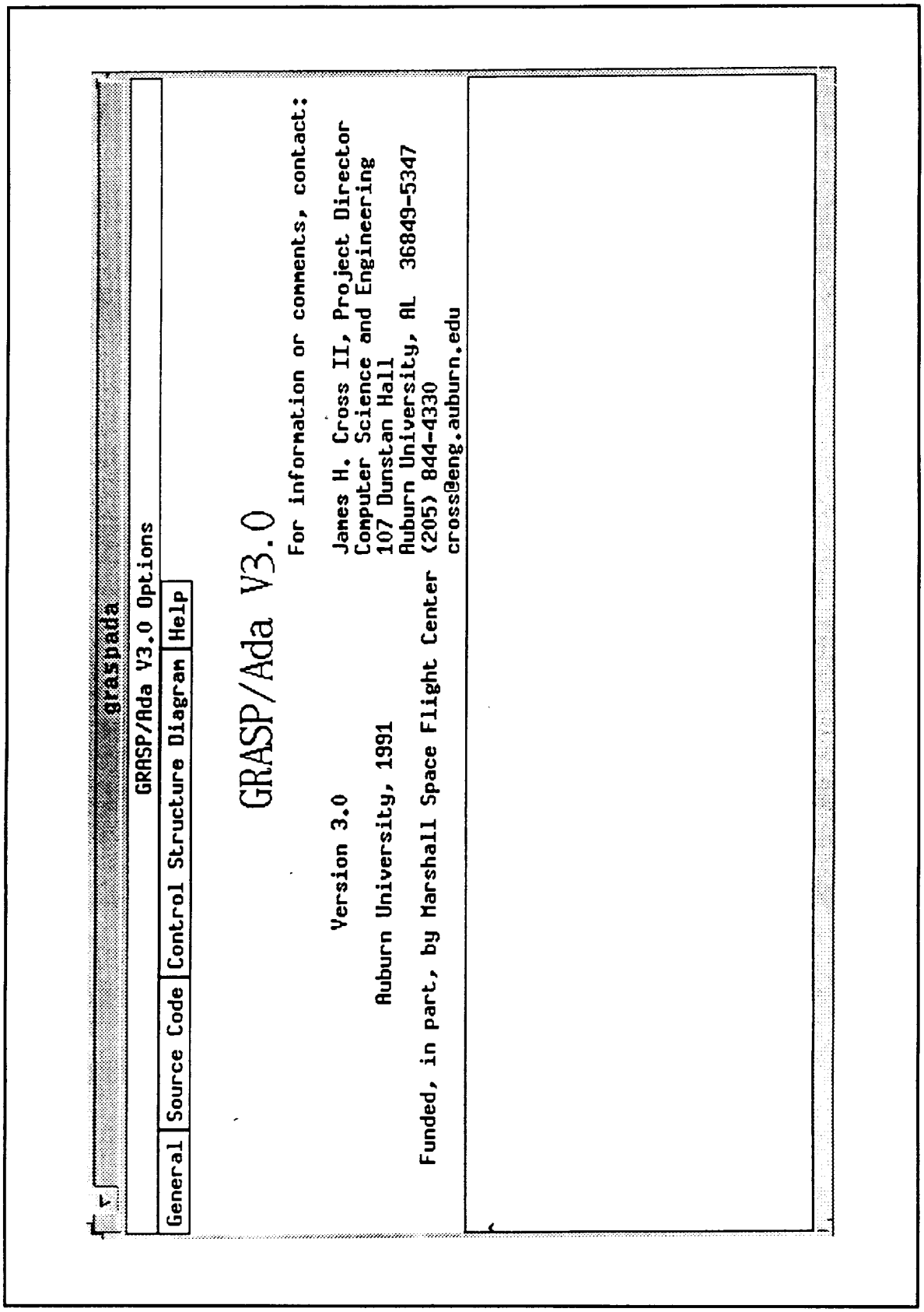


Figure 6. GRASP/Ada System Window

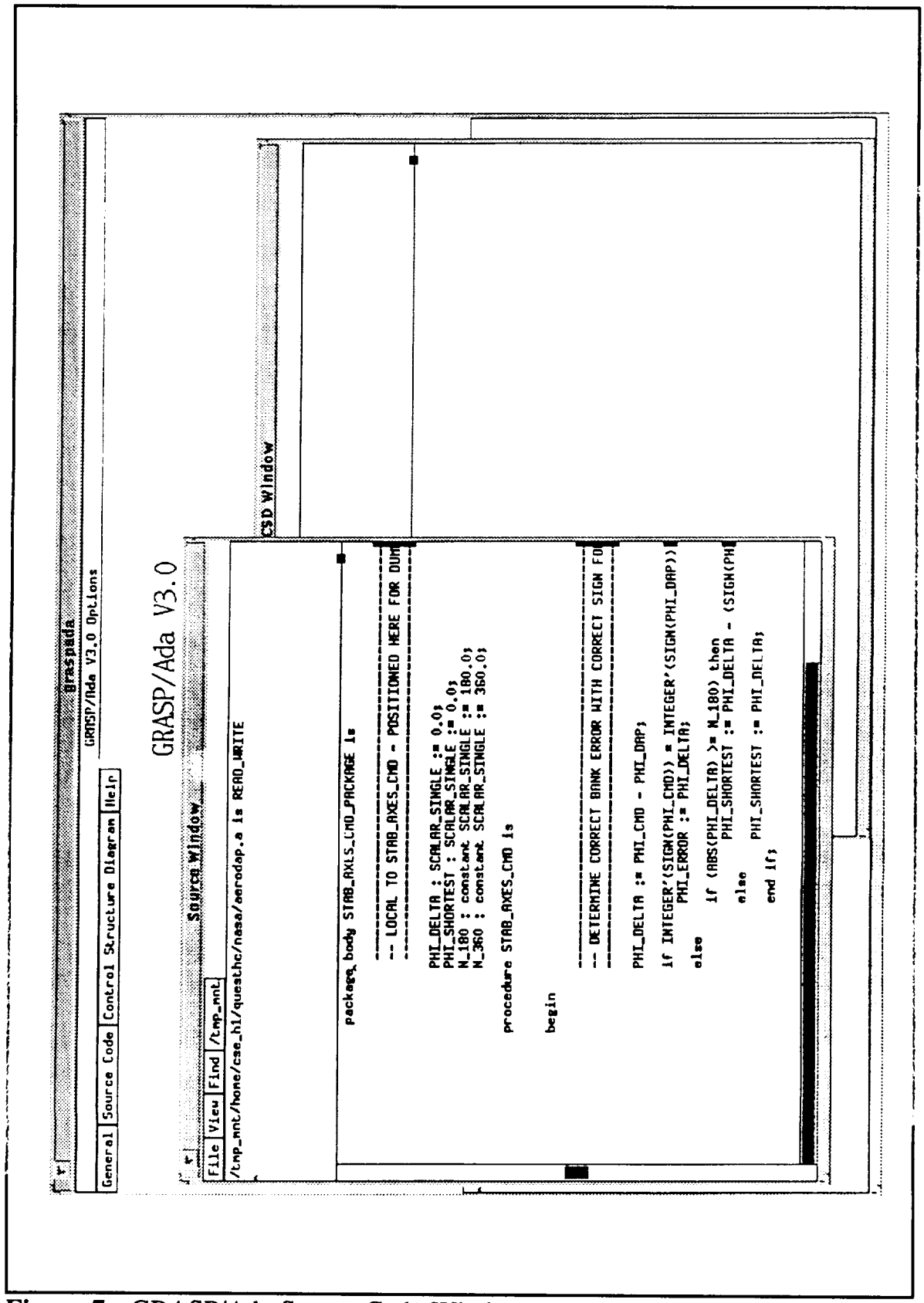


Figure 7. GRASP/Ada Source Code Window

The source file and its associated directory path are displayed in the top pane of each window. See the Control Structure Diagram Window below for details on the menu options.

4.3 Control Structure Diagram Window

The Control Structure Diagram window, shown in Figure 8, provides the user with capabilities for generating and viewing a CSD for an Ada source file. Multiple CSD windows may be opened to access several CSD files at once. CSD file names and their associated directory paths are entered and displayed at the top of each window. When the CSD window is opened initially, the source file with a .csd extension is displayed as the default. In the current version of GRASP/Ada, generation of the CSD is done on a file-level basis where each file contains one or more units. When changes are made to the source code, the entire CSD for the file involved must be regenerated. Future versions of GRASP/Ada will address incremental regeneration of the CSD in conjunction with editing capabilities in the CSD window. The CSD window options are described below.

File - This option allows the user to select from numerous options including:

Load - This option loads a CSD file. A window is presented to the user that allows the user to select a file from current directory (see Figure 9).

Open Source - This is a future option which opens a source window with the source file that corresponds to the current CSD file. The purpose of this option is to facilitate editing of the source file in the absence of CSD editing capabilities in the CSD window.

Generate CSD - This is a future option which will facilitate regenerating the CSD from an existing CSD. The CSD graphics characters must be filtered prior to the parse.

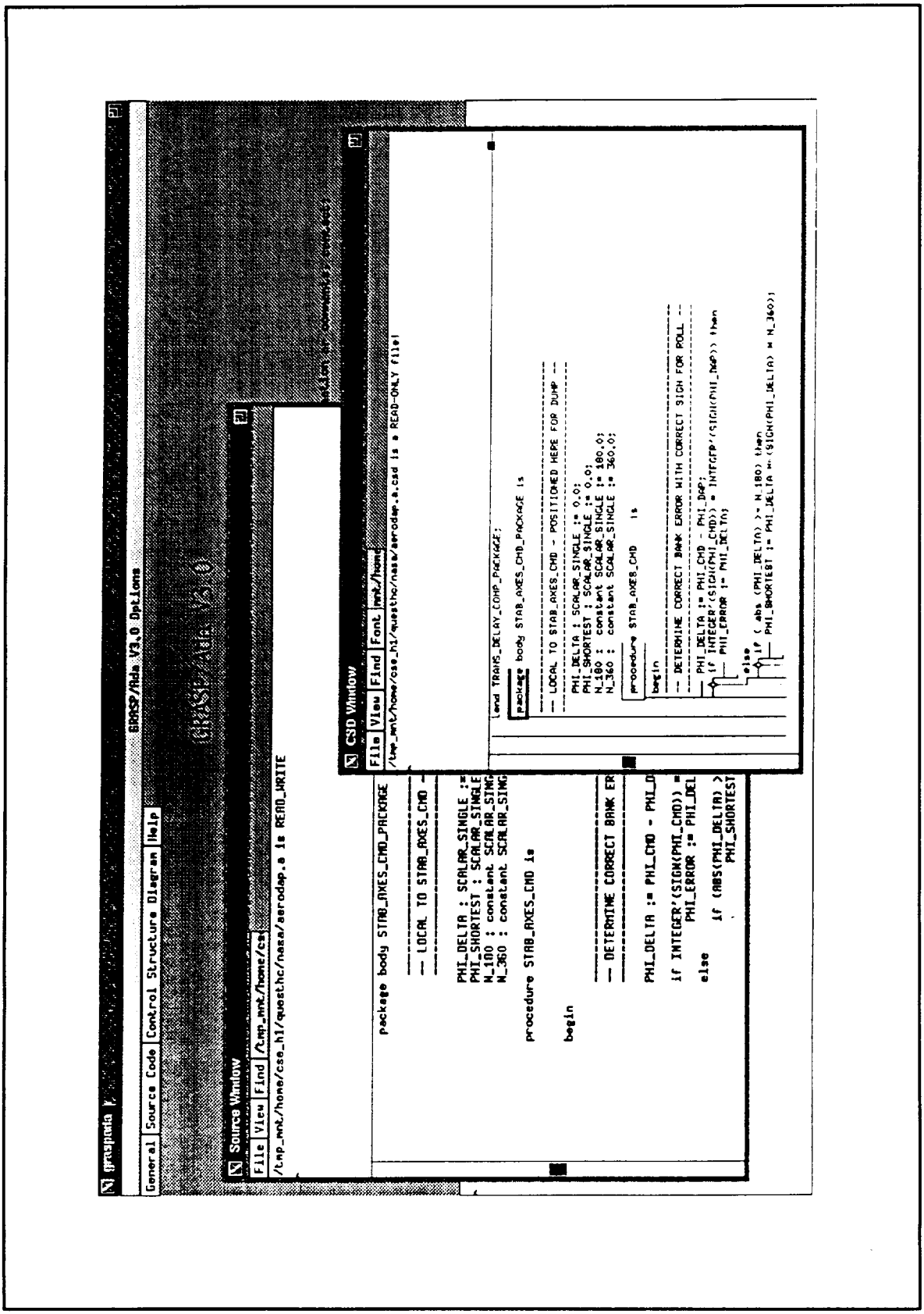


Figure 8. GRASP/Ada CSD Window

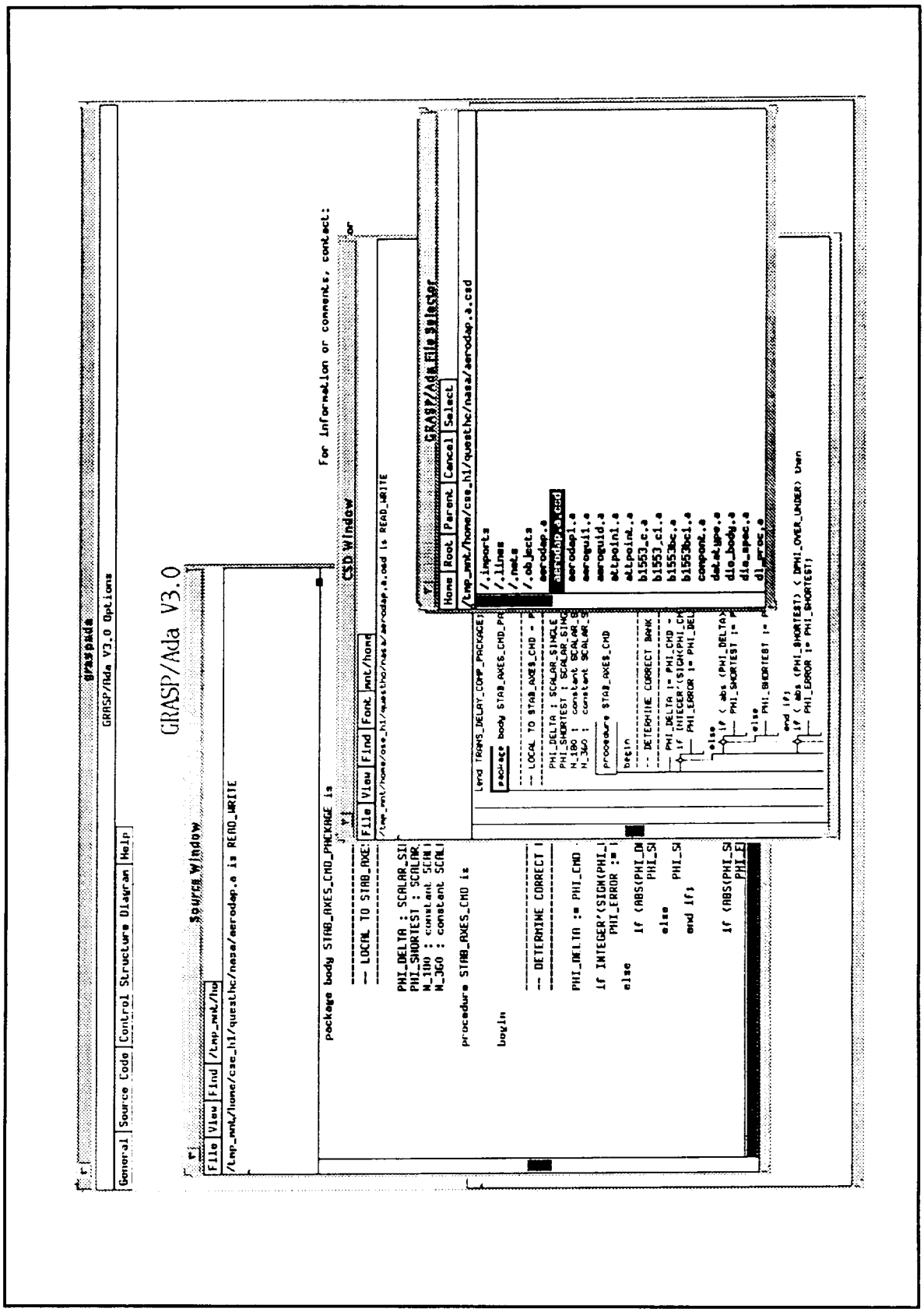


Figure 9. GRASP/Ada File Selection Window

Compile - This is a future option to allow an Ada compiler to be called from the CSD window.

Save - This option saves the CSD file with the same name as was loaded.

Save as ... - This option saves the CSD file with a new name.

Print - A window is presented which allows the user to select various print options such as point size, page numbers, and header.

Quit - The CSD window is closed.

View - (not implemented) This option will allow the user to select from options including: Enable Collapse {Disable Collapse}, Suppress CSD {Show CSD}, Open TOC window, and Open Index window.

Enable Collapse {Disable Collapse} - This option will allow the user to collapse the CSD based on its control constructs.

Suppress CSD {Show CSD} - This option will allow the user to suppress or hide the CSD giving the appearance of prettyprinted code.

Open TOC Window - This option will access the GRASP library and displays a table of contents based on Ada scoping.

Open Index Window - This option will access the GRASP library and display an index of units in alphabetical order.

Find - (not implemented) This option will allow the user to perform search and replace operations. Currently, this is a proposed future option which may become an integral function of the CSD window when editing capabilities are added.

4.4 Help Window

The Help Window provides the user with the capability to display and print templates of the CSD constructs. The user may select individual CSD constructs from a menu as illustrated in Figure 10 with the RENDEZVOUS construct. Alternatively, the user may select the DISPLAY ALL CSD CONSTRUCTS option, as shown in Figure 11, and then scroll through the constructs alphabetically.

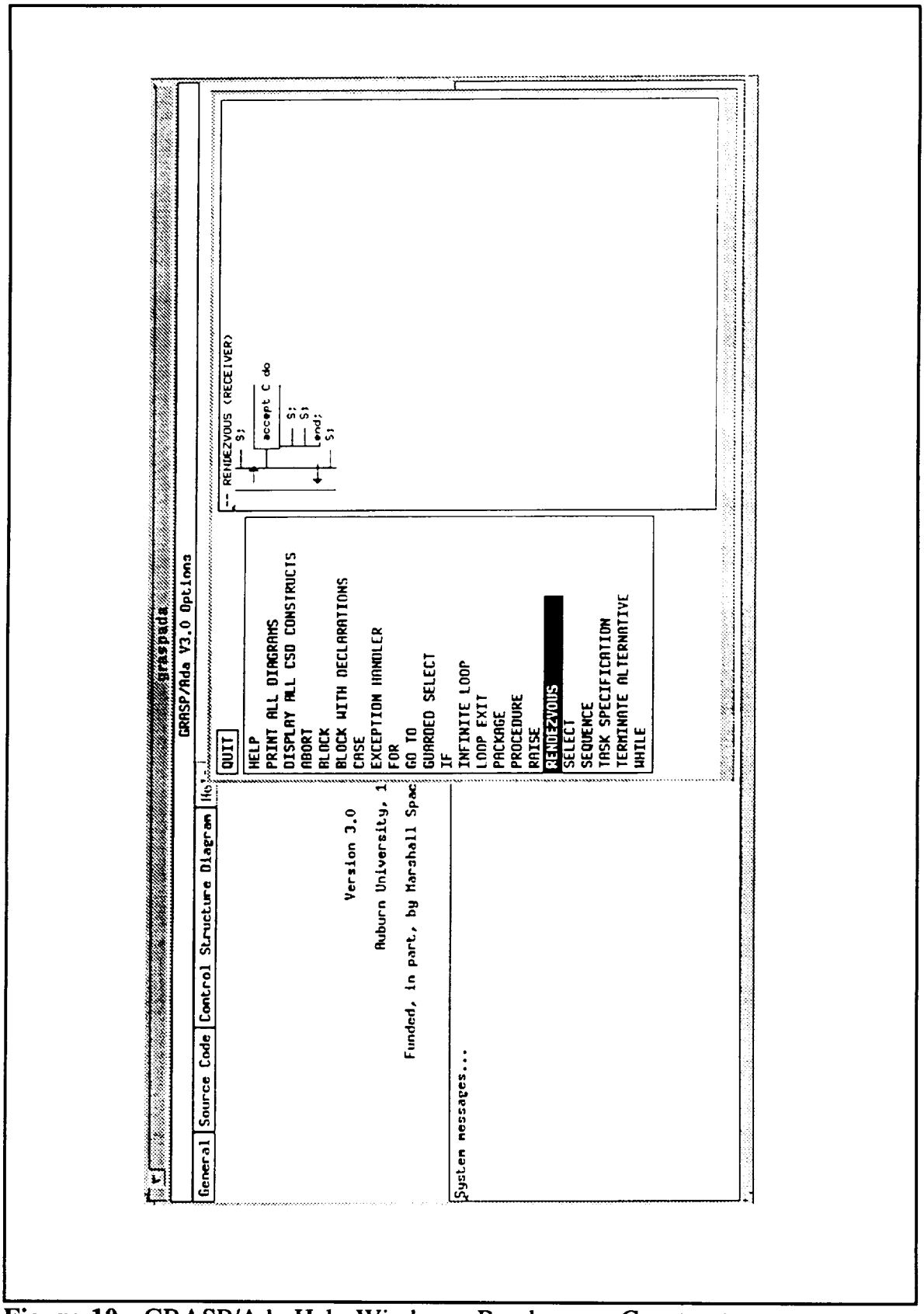


Figure 10. GRASP/Ada Help Window - Rendezvous Construct

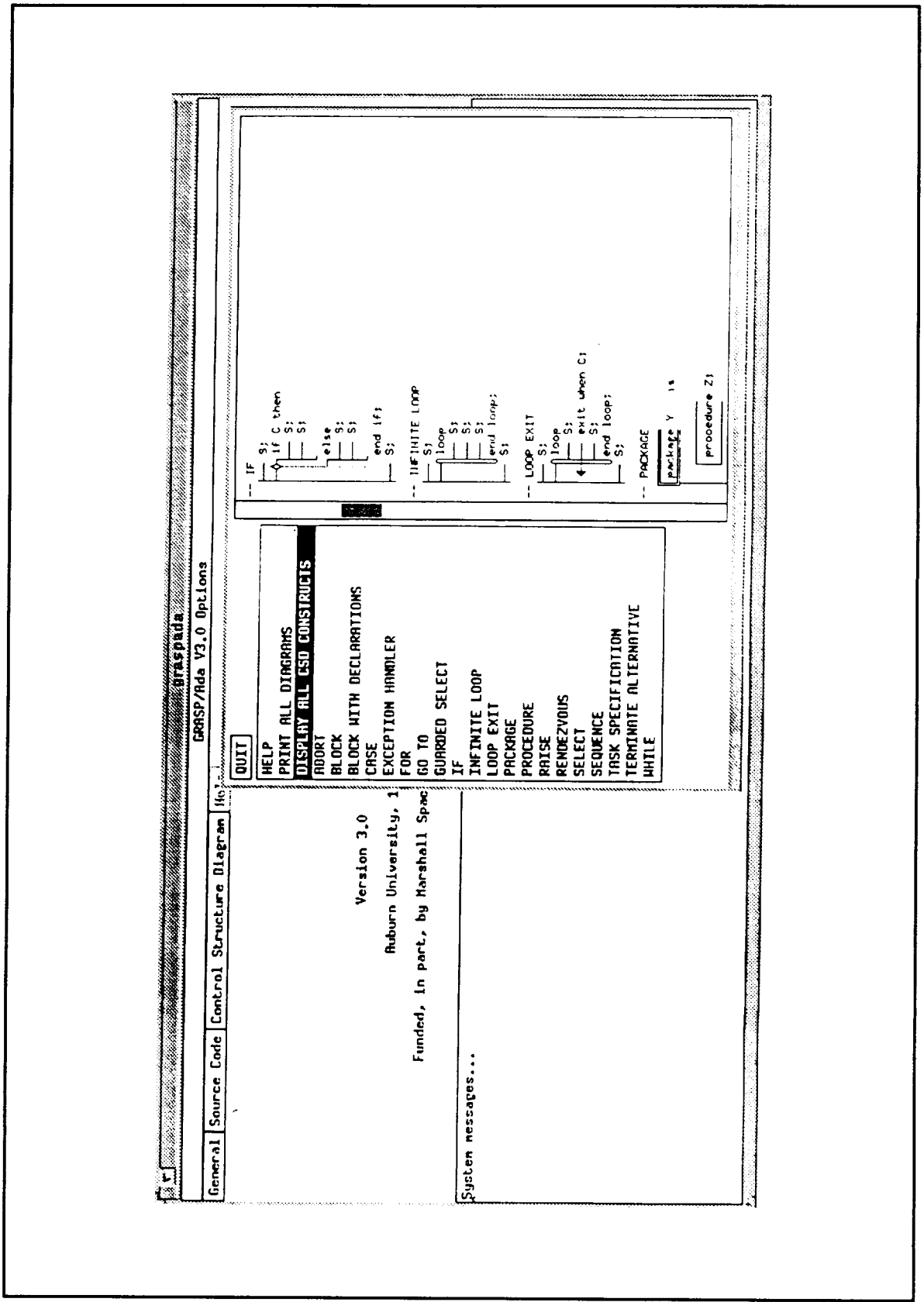


Figure 11. GRASP/Ada Help Window - Display All Constructs

5.0 The GRASP Library

The GRASP library provides the overall organization of the generated diagrams. The current file organization uses standard UNIX directory conventions as well as default naming conventions. For example, all Ada source files end in *.a* or *.ada*, the corresponding CSD files end in *.a.csd*, and the corresponding print files end in *.a.csd.ps*. In the present prototype, library complexity has been keep to a minimum by relying on the UNIX directory organization. In future versions, a GRASP library entry will be generated for each procedure, function, package, task, task entry, and label. The library entry will contain minimally the following fields.

identifier - note: unique key should be composed of the identifier + scoping.

scoping/visibility

type (procedure, function, etc.)

parameter list - to aid in overload resolution.

source file (file name, line number) - note: the page number can be computed from the line number.

CSD file (file name, line number)

OD file (file name)

"Referenced by" list

"References to" list

Alternatives for generation and updating of the library entries include the following.

(1) During CSD generation, the library entry is established and the entry is updated on subsequent CSD generations.

(2) During the processing of DIANA nets.

Alternatives for implementing the GRASP library include (1) developing an Ada package or equivalent C module which is called by the CSD generation routines during the parse of the Ada source, (2) using the VADS library system along with DIANA, and (3) using the StP TROLL/USE relational database system. Of these alternatives, the first one may be the most direct approach since it would be the easiest to control. The VADS and StP library approaches may be more useful with the addition of object diagram generation and also with future integration of GRASP with commercial CASE tools.

6.0 Object Diagram Generator

The object diagram generator (ODgen), produces object diagrams (ODs) for a corresponding set of Ada source files. The requirements specifications and current issues and alternatives are described below. A preliminary prototype has been constructed to determine several of the feasibility issues. Since the Ada package construct captures the essence of the "object" in object-oriented design, the current work has focused on the automatic generation of the package symbol.

6.1 ODgen Symbol Set

The OOSD notation [WAS89] has been selected as a basis for the Object Diagram generator (ODgen). The complete set, which was designed with the intention of using it in forward engineering, is illustrated in Figure 12. In this section, the feasibility of deriving each of these symbols during a reverse engineering effort is considered, and the modifications or supplements needed to render them suitable for the ODgen project are discussed.

Lexical Inclusion of Data Modules. The inclusion of a data module into another module may be determined from a parse of the Ada source code. If a data module is considered to be a component which contains no executable statements other than initializations, then there should be no difficulty in recognizing these modules, and their inclusion in an OD should cause no problems.

Iterative Calls to Library Modules. Again, this information may be extracted from a parse of the Ada source code. There should be no difficulty in producing an OD representation for iterative calls to library modules; however, the composition of this situation

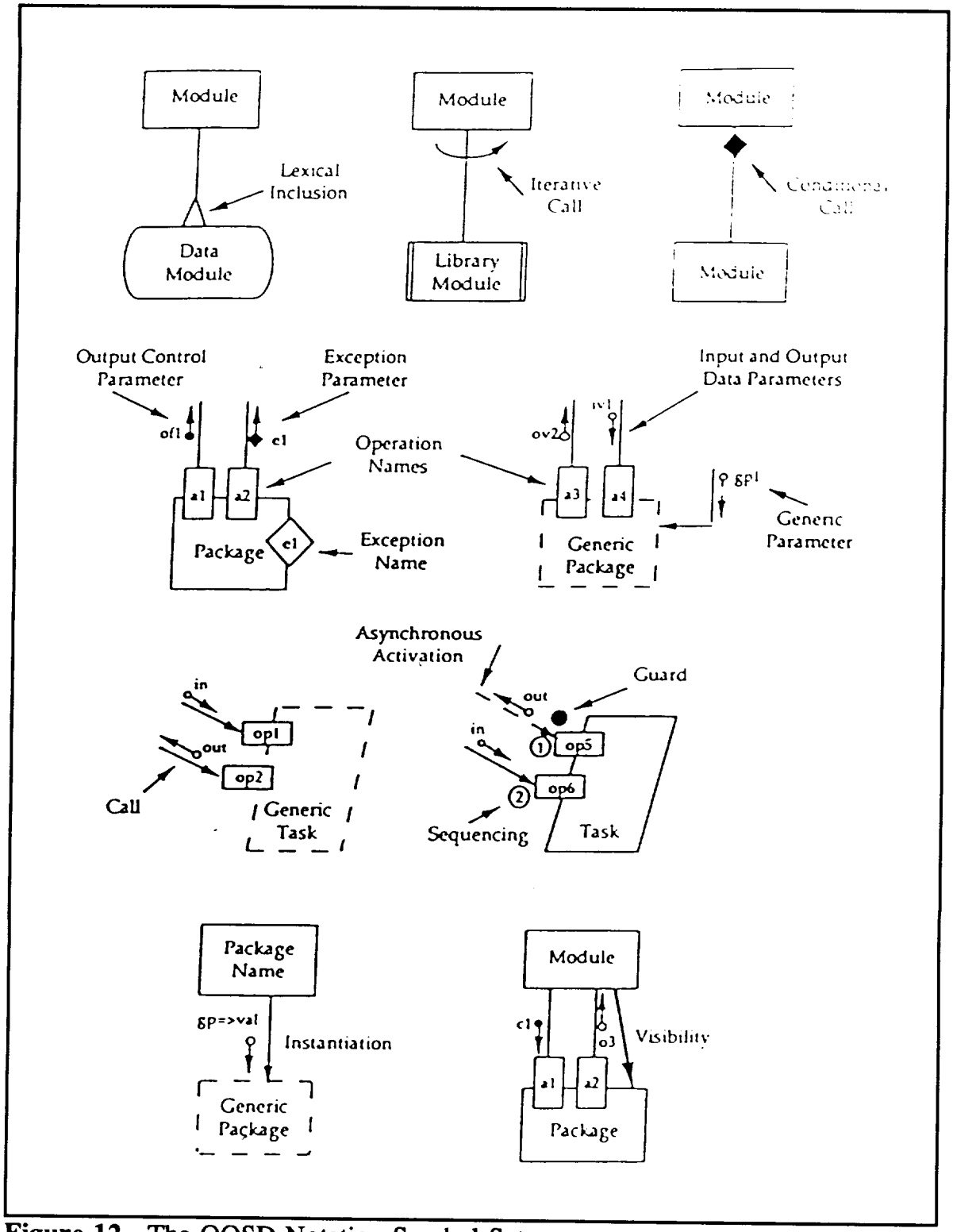


Figure 12. The OOSD Notation Symbol Set
(from *Introduction to StP OOSD Graphical Editor*, IDE, 1989, p. 59)

with others, such as conditional module calls, may require further analysis.

Conditional Module Calls. A conditional call of one module from another can be recognized during parsing, but the generation of an OD representation may prove difficult should the conditional call be composed with another type of call. For example, a program loop may conditionally call another module within the loop's body. How should this be represented in the OD? Certainly the call is a conditional one and may be represented using the conditional module call construct. However, the module is being called repetitively within a loop, so it may just as well be represented using the iterative call construct. Another possibility is to represent the call using a composition of the two representations, indicating that the module is called both iteratively and conditionally. The problem is that this raises ambiguity in that the diagram does not indicate whether the call was made conditionally in the body of a loop, or whether it was made iteratively as the consequence of some condition being true. This ambiguity must be resolved if the iterative and module call representations are to be used properly in the OD.

Package Specifications. A package may be recognized from a parse of an Ada program, and the operations contained within the package may be recognized just as easily. The direction of the parameters may also be determined syntactically through the presence of the **in**, **out**, and **in out** parameter designators. However, the distinction of parameters as either control or data parameters may not be recognized as easily. In fact, it is possible for parameters to be used as both control and data parameters, so the automated classification of an operations's parameters as control or data may not be feasible. Finally, the detection of exceptions may be determined easily through syntactic analysis. Current work has focused on generation of the Booch version of the Ada package symbol [BOO83].

Generic Packages. The specification of a generic package may be recognized easily from a parse of an Ada program, and the generic parameters which must be specified in an instantiation of the package, the operations provided by the package, the parameters to the operations and their direction may also be recognized syntactically. However, the generic package suffers from the same problem as the package in the area of detection of control and data parameters. Again, the automated classification of parameters as either control or data parameters may not be feasible.

Tasks. The declaration of a task may be recognized syntactically in a parse of an Ada program. Much of the desired information needed in the creation of an OD representation of a task may also be obtained from syntactic analysis, such as the entries provided by the task, the parameters and their associated directions for each of the task entries, and any guards placed on the task entries. However, there are two items in the OOSD depiction of a task that may not be obtainable in an automated fashion during reverse engineering. The first of these is the omnipresent problem of distinguishing between control and data parameters which has already been discussed in previous paragraphs. The second is the placement of sequencing numbers on the task entries. Only in the most trivial cases may these numbers be properly derived. In more complex cases, the sequencing numbers would be meaningless or even misleading, and the OD would probably be better off by omitting these numbers.

Generic Tasks. The depiction of a generic task in the OD suffers from many of the same problems as the depiction of a task, and the reader is referred to the previous paragraphs for a discussion of these problems. Other than that, the detection and representation of a generic task should provide no further problems.

Instantiation of Generic Packages. The instantiation of a generic package in an Ada program may easily be determined syntactically. The generation of a proper OOSD symbol for generic package instantiation will require actual parameters to be matched with formal parameters. Otherwise, it should pose no difficulty.

Visibility. The depiction of the semantic visibility of a package to a module in an Ada program may be determined syntactically, but the representation may prove to be misleading. There are two "varieties" of visibility that must be represented: packages lexically included in the declarative section of the current compilation unit and packages included via the **with** clause, which are separate compilation units. For example, a package in an Ada program may only be visible to a small section of a module (for example, a block in a module containing a loop may declare the package in the declaration area and call a function in the package iteratively during the loop. The package would therefore be visible throughout the scope of the block, but would not be visible in the statements preceding and following the block. Therefore, the depiction of the package as being visible to the module could be misleading to the user unfamiliar with the underlying code. Although generating the representation is not difficult, the sensibility of utilizing the representation must be considered. When visibility is determined by the **with** clause, a separate icon is, of course, necessary and appropriate.

6.2 Symbol Interconnections and Diagram Layout

The actual automatic layout of the generated object diagram with respect to symbols and interconnections is the most formidable problem that must be solved. Whereas the CSD has a flexible but well-defined physical layout, the OD layout is not well-defined. In fact, the CASE tools that support the OOSD notation require the users to "manually" arrange the

symbols. Determining the feasibility of an algorithmic and/or heuristic solution which yields a reasonably comprehensible diagram layout is a complex topic which warrants further investigation.

The majority of approaches to the automated layout of a directed graph have focused on minimizing the number of crossovers among the flow lines in the graph. Warfield [WAR77] detailed an algorithm that proposed the reordering of a directed graph into a number of vertex subsets (called levels) and the minimization of the number of crossovers between subsequent levels using a special table called a generating matrix. The major drawback to the approach is that the generating matrix technique is only applicable to cases in which their are five or less vertices in each level. Warfield realized this, and went on to propose a number of techniques for graph manipulation that he believed could prove useful in the development of improved graph layout algorithms.

Sugiyama [SUG81] developed a heuristic algorithm for crossover minimization called the penalty minimization (PM) method that could be integrated with Warfield's algorithm for application to graphs with more than five vertices in a level. Since the penalty minimization method is combinatorial in nature, Sugiyama also developed a heuristic algorithm called the barycentric (BC) method that would make the PM method practical. In addition to this improved algorithm for minimizing the number of crossovers between successive levels of a graph, Sugiyama also developed the priority (PR) layout method for improving the horizontal positioning of the vertices in a level.

Paulisch [PAU90] noted that two major problems with Sugiyama's work were that the algorithm did not allow the user to specify preferences and constraints on the diagram layout, and that the algorithm did not take previous layouts into account when updating a graph and could produce wildly different layouts from minor perturbations of the graph, leading to graph

layout instability. She theorized that both problems could be solved by incorporating a layout constraint manager with Sugiyama's algorithm and developed a method for doing so. Her approach involved adding a constraint manager for each dimension of the graph (x, y, and z) and providing a set of constraints for each dimension form which the user could choose a subset to be applied to the graph. The constraint manager would reconcile the various constraints provided in this manner by the user with the constraints imposed by the application and the layout algorithm to produce a graph layout. The implementation chosen by Paulisch used a binary search among constraints in the reconciliation process, a method that provides a quick response time but does not necessarily yield an optimal solution. A better approach might lie with the use of genetic algorithms to "breed" an optimal graph layout solution. The feasibility of this approach, coupled with the use of the Warfield/Sugiyama/Paulisch algorithms, is presently being investigated by members of the GRASP/Ada research project.

6.3 GRASP/Ada ODgen Processing Alternatives

In the development of the ODgen design specification, three distinct development methods were considered. The major difference among these methods is linked to the degree of involvement of other commercially available tools and the ability of the user to specify these tools. The first method considered was to create ODgen as a stand-alone system. A second alternative was to use GRASP/Ada as a driver for a set of subprogram invocations which would use VADS, ODgen, and StP/ADE in sequence to produce the architectural diagrams. Finally, the third alternative considered was to use GRASP/Ada as a shell from which the user could invoke each of the three tools at his convenience. In this section, these

three methods are examined in more detail, and the advantages and disadvantages associated with each method are outlined.

ODgen Is Independent of Commercial Tools. This method would involve the development of a stand-alone architectural diagram generator. The generator would not be dependent on commercial tools such as VADS and StP/ADE. Instead, the parser/scanner developed in Phases I and II of the GRASP/Ada research project would be extended to extract the information needed for the representation of architectural diagrams. A method for specifying or identifying the complete set of files comprising the Ada system would have to be developed (this may require some involvement from the user). The major advantage of this method is that the tool would not be subject to the whims of the manufacturers of commercial tools (i.e., the tool would not be rendered useless if VADS were to become unsupported, if the DIANA representation were subjected to large-scale change, if the StP/ADE file formats and representation methods were to be changed, etc.). On the other hand, this method would involve substantially longer development time, as a tool for identifying the dependencies among a set of Ada source files would have to be developed. In addition, a tool for viewing and printing the architectural diagrams would need to be developed. Because a substantial amount of effort has already been spent in the development of the GRASP/Ada X11R4 interface, extending this interface to display the architectural diagrams could benefit from the groundwork already laid in Phases I and II. The major goals which would need to be accomplished are the development of X11R4 widgets for the representation of each of the OOSD symbols, and the development of layout heuristics and modified layout widgets suitable for displaying the OOSD symbols.

ODgen Invokes VADS and StP/ADE. In this method, the ODgen component of GRASP/Ada would first invoke VADS to generate a DIANA net for the specified set of Ada

source files. ODgen would then traverse this net to obtain the required information and generate an internal representation for the architectural diagrams. This information would then be shaped into a format suitable for StP/ADE and saved. Finally, StP/ADE would be invoked to view the architectural diagram. All of this would be transparent to the user: after specifying the Ada source files and a number of ODgen options, GRASP/Ada would invoke the tools in sequence and bring up StP/ADE as a subprocess displaying the generated diagrams. The major advantage in this approach is that it would utilize already-existing tools to speed the development effort. Instead of writing yet another Ada parser, intermediate representation generator and OOSD diagram displayer, the research effort could concentrate on the task of obtaining architectural details and composing meaningful architectural diagrams from them. However, relying on commercial tools could be dangerous as subtle changes in the formats of either the VADS representation or the StP/ADE representation could require major, sweeping changes in the ODgen system. In addition, the use of commercial tools could greatly limit the number of potential users for the ODgen system. Instead of only needing the ODgen system, the user would also need the VADS Ada compiler and the StP/ADE software development system - two costly components. For many university research installations, the costs of these systems would be prohibitive and would virtually eliminate the potential use of ODgen.

GRASP Runs Independently of VADS and StP/ADE. The user invokes VADS to create DIANA nets, invokes GRASP to generate CSDs and ODs, and invokes StP/ADE to view the ODs. In this scenario, the GRASP/Ada interface would be partially customizable by the user. Instead of relying on a specific Ada intermediate representation generator and OOSD diagram displayer, the user would be able to select from a limited number of commercial tools. To accomplish this, a minimal ODgen interface for each tool would be

identified and a suitable data representation would be specified. ODgen would then be designed to transform the input Ada source data into an architectural diagram representation in the output format. Then, customizing GRASP/Ada for new intermediate Ada representations and OOSD diagram formats would consist of simply writing a filter transforming the data from one representation to another. For example, customizing GRASP/Ada to work with the VADS DIANA representation would require a filter to be written to traverse the DIANA nets and store the needed architectural information into a file in ODgen's input format. Similarly, customizing GRASP/Ada to work with the StP/ADE tool would require a filter to be written translating the ODgen output format into StP/ADE's input format. This method would allow GRASP/Ada to be fairly portable without depending on strict reliance on commercially available tools. On the other hand, this method would require an extensive and easily translatable interface format to be developed for both ODgen's input and output formats. Finally, the amount of effort required for the writing of filters for new representations could be potentially quite large, depending on the format and accessibility of the new representations.

6.4 Displaying the OD - Screen and Printer

Generating visual displays of the object diagrams will require display methods to be generated for the screen and printer. Since the GRASP/Ada interface for Phases I and II was developed using the X Window System (a portable graphical environment gaining widespread acceptance) and numerous utilities have been developed in the creation of that interface, the development of a display mechanism for the object diagrams in X11R4 would be a logical extension to the previous work. In addition, the PostScript page description language was used in Phases I and II for the hardcopy output of the CSD diagrams. Because PostScript

is a nearly universal output description language for laser printers, the development of PostScript utilities for printing GRASP/Ada object diagrams would ensure the portability of GRASP/Ada. In this section, some of the issues and considerations involved in the generation of visual displays for the object diagrams for the screen and printer are discussed.

Screen representations. In the X11R4 system, objects on a screen are often represented using widgets (a user interface component embodying a single concept: e.g., buttons, labels, scrollbars, etc.). The development of the interface for Phases I and II of the GRASP/Ada research project was implemented using the X11R4 Athena widgets, a general purpose widget set shipped with the X11R4 system. Numerous utilities were developed by the GRASP/Ada implementation team to simplify the use of these widgets to providing facilities for browsing files, generating alert boxes and dialogues, creating text editor windows, and specifying menus. These utilities would be invaluable in the development of the ODgen interface, but additional utilities will be needed. In particular, there are no suitable widgets in the Athena set for displaying the various OOSD symbols. A reasonable approach to implementing a display mechanism for the ODgen diagrams would involve the creation of a set of widgets, one for each of the symbols in the OOSD set. These widgets could be subclassed from existing widgets in the X11R4 Athena set, minimizing the amount of effort required to create them (although this would cause them to need revision with subsequent releases of X11). And once written, these widgets could be used in other CASE programs written for X11R4. Next, constraint and layout widgets would need to be designed to facilitate the layout of these OOSD symbols. Again, a suitable widget could be created by subclassing an appropriate Athena widget, in this case, probably the Form widget. Such a widget would be responsible for laying out an architectural diagram and redrawing it after

modifications, thus justifying the need for embedded logic to be written for the automatic layout of the ODgen diagrams.

Printer Representations. In Phases I and II of the GRASP/Ada research project, three different types of output devices were utilized. The first was the LN03 printer, a printer manufactured by DEC with the capability of printing sixel graphics. Printing the CSD on the LN03 printer was accomplished by generating sixel representations for each of the CSD characters and then printing each CSD character as a small graphic image. The text of the Ada source program was printed normally using the LN03 resident fonts. This method had several major disadvantages: it was not portable (sixel graphics are a proprietary format of DEC), it was slow (printing each CSD character as a graphic bitmap was a time-consuming process), it was crude (the sixel graphics format did not allow for a high degree of resolution and the generated CSD characters suffered from jagged outlines), and it wasted file space (the space required to store the sixel representation of a single CSD character was equivalent to the space needed to store over 200 text characters). The second output device utilized was the HP LaserJet II printer, an extremely popular laser printer. Using the LaserJet II enabled the GRASP/Ada program to utilize a specially prepared CSD font that could be downloaded to the printer. This method allowed the CSD to enjoy greatly improved resolution over the LN03 characters, a much smaller file representation (since each CSD character could now be represented as a single extended ASCII character rather than a large bitmap image), and faster printing speeds. However, this method was still tied to a single commercial printer, the HP LaserJet II. The third method allowed the GRASP/Ada program to generate CSDs that could be printed on a wide variety of printers by generating CSDs using the PostScript page description language. PostScript representations for each of the CSD characters were generated using a series of PostScript graphic primitives to describe how to draw each

character. Once designed, these characters were merged with a PostScript program that uses the Adobe Courier font to produce a modified Courier font containing the CSD characters. The CSD font can be installed on any PostScript printer by downloading this PostScript program. Thereafter, CSDs can be printed by sending them to the printer and specifying this specially modified Courier font. The advantages to this method are many: the CSD can be printed on any printer (laser, inkjet, dot-matrix, etc.) that supports PostScript; the CSD can be printed at the highest resolution the printer is capable of producing, which generally produces results of outstanding high quality on most laser printers; and the CSD font can be scaled to any size, allowing the CSD to be printed at any size the user wishes (unlike the previous methods, which allowed the user to have only one font size). For Phase III of the GRASP/Ada research project, a library of PostScript routines for printing each of the OOSD symbols must be created. The ODgen program can then invoke these routines to create a sequence of descriptions for printing the OOSD diagram to any PostScript printer. Care must be exercised in the creation of these routines to ensure that they match the appearance of the X11R4 widgets also corresponding to these OOSD symbols. Like the modified X11R4 widgets for the OOSD symbols, these PostScript routines should also be portable to any other CASE tool for the X11R4 system.

6.5 Incremental Changes to the OD

The ultimate goal of the ODgen phase of the GRASP/Ada research project is to allow the user to reverse engineer a set of Ada source files into an architectural diagram. For a large system, this may take some time. It would be desirable to have the user do the reverse engineering once and then have ODgen incrementally change the OD as the user makes

changes to the source code. However, this is an extremely complex issue, and some of the problems involved in doing this are addressed in this section.

The first problem involved in the incremental updating of the OD is that if the DIANA notation is used to obtain the syntactic and semantic information from the Ada source files for the generation of the OD, then we are immediately stymied. In its current states, DIANA does not support incremental updates. If a portion of a file is changed, then the entire file must be recompiled to update the DIANA net. Thus, any implementation of ODgen which relies on a DIANA net for its information could not support incremental diagram updating. A parser specifically modified for incremental updates could prove useful in generating the diagrams, but such parsers are extremely complex to design and are often excruciatingly slow in practice. Teitelbaum and others [TEI81] have outlined some of the problems involved in incremental parsing in their work on the development of syntax-directed editors.

The second problem involved in the incremental updating of the OD lies in the unrestrained freedom of editing by the user. The proper generation of an OD relies on the existence of a relatively complete Ada compilation unit, where "relatively complete" is defined as a main (or "driver ") program along with at least the specifications of the packages, tasks, and modules upon which it depends. The existence of a relatively complete program is not normally a problem in reverse engineering, where the user has a system and is just trying to decipher its function and meaning. However, the user could initiate what, to him, appear to be very minor changes that could lead to many changes throughout the ODs and CSDs. As an example, imagine that the user renames a small package. To him, this may be a minor modification, but it would create havoc for the ODgen system. The system would no longer be relatively complete, as it would now contain what would appear to be a

new and unreferenced package along with a large number of package inclusions that may no longer be satisfied. This and related problems must be addressed in any attempt at providing incremental updates to the ODs and CSDs.

6.6 Internal Representation of the OD - Alternatives

Although the DIANA intermediate representation for Ada may be used to gather information for the creation of the OD, and the StP/ADE format may be used as one possible output representation for the OD, a more extensive and comprehensive internal representation tailored for the needs of the OD generator is desired. Several alternatives are presently under consideration for this internal representation of the OD. These alternatives include (1) storing the OD as a single ASCII file, (2) storing the OD as a number of files tailored to the internal data structures utilized by ODgen, and (3) completely bypassing the internal representation to directly generate the OD from a DIANA net. Each of these approaches has its own merits with respect to processing and storage efficiency, and these qualities are in this section.

Single ASCII File. The most direct approach is to utilize the StP file format. This would present the option of viewing the OD via the StP/ADE system. However, although the StP file format is "open architecture," it is a proprietary format and is, therefore, subject to change. Because the function of the ODgen system will be dependent to a high degree on the organization of the data upon which it operates, a stable data format is desired. Therefore, an original data format might prove to be more useful over time as it would reduce the problems of compatibility with commercial formats (filters could be written to translate from the ODgen format to other formats). In addition, commercial formats such as the StP format might lack provision for all of the information which might be needed for the OD. This is particularly true for the case in which the user may wish to link CSDs generated using

the GRASP/Ada CSD generator to objects in the OD. A comprehensive internal representation consisting of segments storing information for each of the OOSD symbols may prove to be necessary to fulfill all of the needs of Phase III of the GRASP/Ada research project.

Multiple ASCII Files. Because a typical Ada program will involve a number of source files, an alternative to storing the data relating to a system in a single file is to store the data in a number of files, each linked to one or a number of source files. Such a system would decompose the intermediate representation into a number of smaller units. With an appropriate indexing scheme, this could bring about increased performance in the ODgen program as the system would not have to peruse unnecessary information to get to the data it needs. This scheme might also prove helpful in producing incremental changes to the OD. The major drawbacks to this method are the greatly increased number of files generated and the overhead involved in the indexing scheme.

Direct Generation From DIANA Net. If tight coupling and integration with a commercial Ada development system such as Verdix VADS is desired, then direct generation of the OD from the DIANA net produced as a result of compilation could be performed. This would require a layer of software which traverses the DIANA net and calls the appropriate OD primitives as unit nodes are encountered. This approach would apparently eliminate the possibility of directly editing the OD since the DIANA interface does not support modifying the net, only reading it.

6.7 Navigation Through Large ODs - Alternatives

Because many Ada software systems are fairly large in size and scope, some facility for easily navigating the ODs generated for them must be provided. There are three

navigational methods presently being considered for use in the ODgen system. These include (1) the creation of a "table of contents" for the system, (2) the direct navigation throughout the system using a "point and click" interface similar to that provided in hypertext or in the HyperCard application on the Apple Macintosh, and (3) a combination of these two methods. In this section, these methods are described and the relative advantages and disadvantages pertaining to each method are presented.

Index (or Table of Contents). An index, similar to that presented in the Xman application provided with the X Window System, would be used to navigate among a system of CSDs and ODs. After generating the CSDs and ODs, the user would be presented with an ordered list of the diagrams. To view a diagram, the user would click on the index entry and the corresponding CSD or OD would be displayed. The index entries would be created as the respective diagrams are generated and stored in the GRASP/Ada library (see Section 6 below). The greatest advantage to this method is that the user may see the entire range of diagrams at once - nothing is hidden. However, for a nontrivial system this may be a list of daunting proportions requiring the user to have some familiarity with the system to be of any use. This disadvantage may be offset by layering the index so that only top level diagrams are presented at first, each containing links to a sublist of associated diagrams, etc. In addition, icons or informative labels could be attached to each index entry to provide the user with additional information regarding the diagram under consideration.

Direct Navigation Via ODs. With this method, after generating the CSDs and ODs for an Ada system, the user would be presented with the top level diagram for the system. The user could reach other diagrams in the system by clicking on the OOSD symbols in the top level diagram: this would bring up the associated subdiagram or CSD on the screen. The user is allowed to click on procedure, function, and task entry calls in the OD directly and

a separate OD window is opened containing the selected OD or fragment thereof (there may be a problem using/implementing this approach since the mouse is also used for editing). Browsing the OD in this manner would be much like working with hypertext, and would provide some of the advantages and disadvantages associated with hypertext. For example, the user may gain an incomplete view of the system by following odd threads throughout it. The user may also have to sift through a great deal of high level detail to get to low level components. This might prove frustrating in practice. However, the user would have the freedom of navigating throughout the system in an logical manner.

Combination of Index and Direct Navigation. The two approaches discussed above both have their relative merits and problems. A more desirable solution to the navigation of large ODs possibly lies in the combination of these methods. By providing a linked series of ODs and CSDs with a comprehensive listing of all diagrams, the user would have unrestrained freedom in navigating throughout the system. Additional utility could be provided by allowing the user to "mark" viewed and unviewed diagrams in the index, and by maintaining a list of recently visited diagrams. However, this approach would be more difficult to implement and would take careful analysis and design to be effective.

6.8 Exploding/Imploding the OD

The OD window should provide the user with the capability to explode or implode the OD based on Ada constructs and complete diagram entities (e.g., procedures, functions, tasks and packages). This capability directly combines the ideas of chunking with the major threads of control flow which are major aids to comprehension of software. The OD can be supplemented by *architectural CSD* (ArchCSD) [DAV90], a diagram produced by collapsing

the CSD based on procedure, function, and task entry calls, and the control constructs that directly affect these calls.

6.9 Generating a Set of ODs

Since GRASP/Ada is to be used to process and analyze large existing Ada software systems consisting of perhaps hundreds of files, an option to generate all the CSDs at once is required. Generating a set of ODs should be facilitated by entering a wildcard file name (e.g., *.*a*). An OD generation **summary window** should display the progress of the generation by listing each file as it is being processed and any resulting error messages. The summary should conclude with number of files processed and the number of errors encountered. The default for each OD file name is the source file name with **.od** appended. Generating a set of ODs can also be considered a user interface requirement rather than strictly a OD generator requirement.

6.10 Printing An Entire Set of ODs

Printing an entire set of ODs in an organized and efficient manner is an important capability when considering the typically large size of Ada software systems. A book format is under consideration which would include a table of contents and/or index. In the event GRASP/Ada is integrated with IDE/StP/ADE, the StP Document Preparation System could possibly be utilized for this function.

6.11 Relating the CSD and OD - Alternatives

For each OD in the system under scrutiny, the user will have the ability to click the

mouse on any OOSD symbol in the diagram and be presented with the underlying CSD or a subsequent level of OD, if it exists. In addition, a button will be provided on each OD or CSD window allowing the user to step back up one level in the diagram hierarchy to see the "parent" diagram. In this manner, the user will be able to fully traverse the ODs and CSDs comprising the system using a "point and click" approach. In addition, the user may choose to bypass the hierarchical traversal by simply choosing the diagram to be viewed from the index list of diagrams.

Each CSD corresponds to an object symbol (e.g., procedure, function, package, task, task entry). These may be nested and may each have an interface and a body. Conceptually, the CSD may be collapsed to a graphic symbol. A group or system of these symbols could be interconnected (logical inclusion and/or invocation) to form an object diagram. This could be thought of as "growing" or synthesizing the system diagram. The user would be able to open any of these symbols to see the lower level diagram associated with it.

If the StP/ADE system is to be used for viewing the ODs and CSDs, the ODs could be viewed directly. The CSD could be displayed as an annotation in StP/ADE. This would require that the CSD font be downloaded into the appropriate StP/ADE window for the diagram to be viewed properly.

6.12 Index and Table of Contents Relating the CSDs and ODs

An index of all CSDs and ODs should be available via the GRASP library. The index should be presented in a window to the user, and upon the selection of an index entry, an appropriate CSD window should be opened. The index will provide an additional means of navigation among diagrams in an interactive mode, and it will be the basis for printing a

complete set of all diagrams. See the section below entitled, "The GRASP Library" for more information.

6.13 Design and Implementation of Preliminary ODgen Prototype

The overall organization and composition of the prototype system is discussed first, with special attention paid to the design of the ODgen widget set. This is followed by a description of the use of a subset of the proposed GRASP library in the development of the prototype. Finally, an informal specification of the organization of the X11R4 user interface that allows the ODgen prototype to function as a standalone tool is given, along with a summary of the changes that must be made to integrate the final ODgen system with the GRASP/Ada system.

Overview. The primary purpose of the ODgen object diagram generator prototype was to assess the feasibility of recognizing and extracting design information from Ada source code. Mappings were assigned between the target diagrams, described above, and the appropriate Ada constructs. An X11R4 widget set, with one widget for each of the proposed ODgen diagram elements, was designed, and the Package widget was implemented. Next, a subset of the GRASP library which would enable the storage and manipulation of structural information was created. Finally, an X11R4 user interface based on the freely available Athena widget set was developed which would allow ODgen to be used as a standalone tool that would work independently of the GRASP/Ada system. Figure 13 shows a typical window presented to the user by the preliminary prototype of ODgen. ODgen uses a simple X11R4 user interface that provides a menu bar and three paned areas. The menus allow the user to choose an Ada file, load the file, generate an object diagram, display an object diagram, save the object diagram, change the size of the font, and exit the program. The first

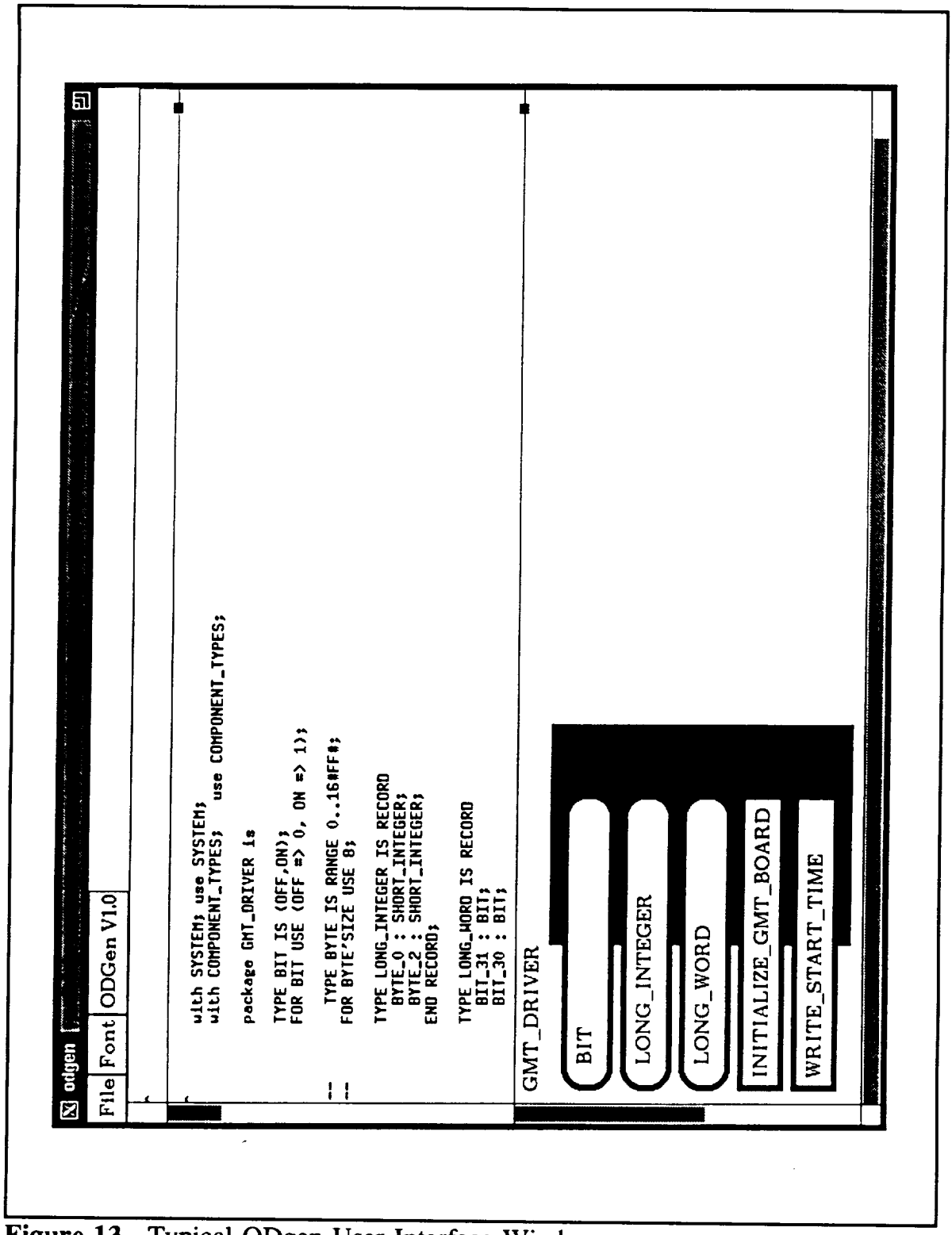


Figure 13. Typical ODgen User Interface Window

paned area is used by ODgen to report errors and to display information pertaining to the program's status. The second paned area is used to display the source code or CSD for the Ada program of interest, and the third paned area is used to display the corresponding object diagram. Each of the paned areas may be sized by the user, so that an all-source view may be obtained as in Figure 14. The user may equally choose to see an all-diagram view as in Figure 15, or a multiple view as in Figure 16.

General Development Approach. The overall organization of ODgen may be seen in Figure 17. To initiate the implementation of ODgen, a lexical description of Ada was obtained and used as an input for the lexical analyzer generator LEX to create a scanner for Ada systems. The lexical description was enhanced with a number of customized routines which would enable the filtering of CSD characters as well as assist in the capture of data relating to the source code (such as line numbers). Next, an Ada grammar was obtained and instrumented with a number of action routines which would extract structural information during a parse of Ada source code and preserve this information in memory using a prototype of the GRASP library. The resulting Ada grammar was too large for the parser generator tool YACC to handle, so the widely available YACC-workalike BISON (a parser generator upwardly compatible with YACC that accepts larger grammars) was used to generate an Ada parser. This parser and the previously generated scanner were tested on a number of programs provided by NASA and Boeing to ensure that they would indeed parse legitimate Ada programs. (A special note: due to the syntactic structure of Ada, many Ada grammars introduce slight modifications to several of the productions in order to simplify the creation of the parser. The grammar utilized by GRASP is one of these, and therefore ODgen will parse some syntactic constructs that are not legitimate Ada. These deviations are few, however, and are not likely to occur in practice. In addition, one of the underlying

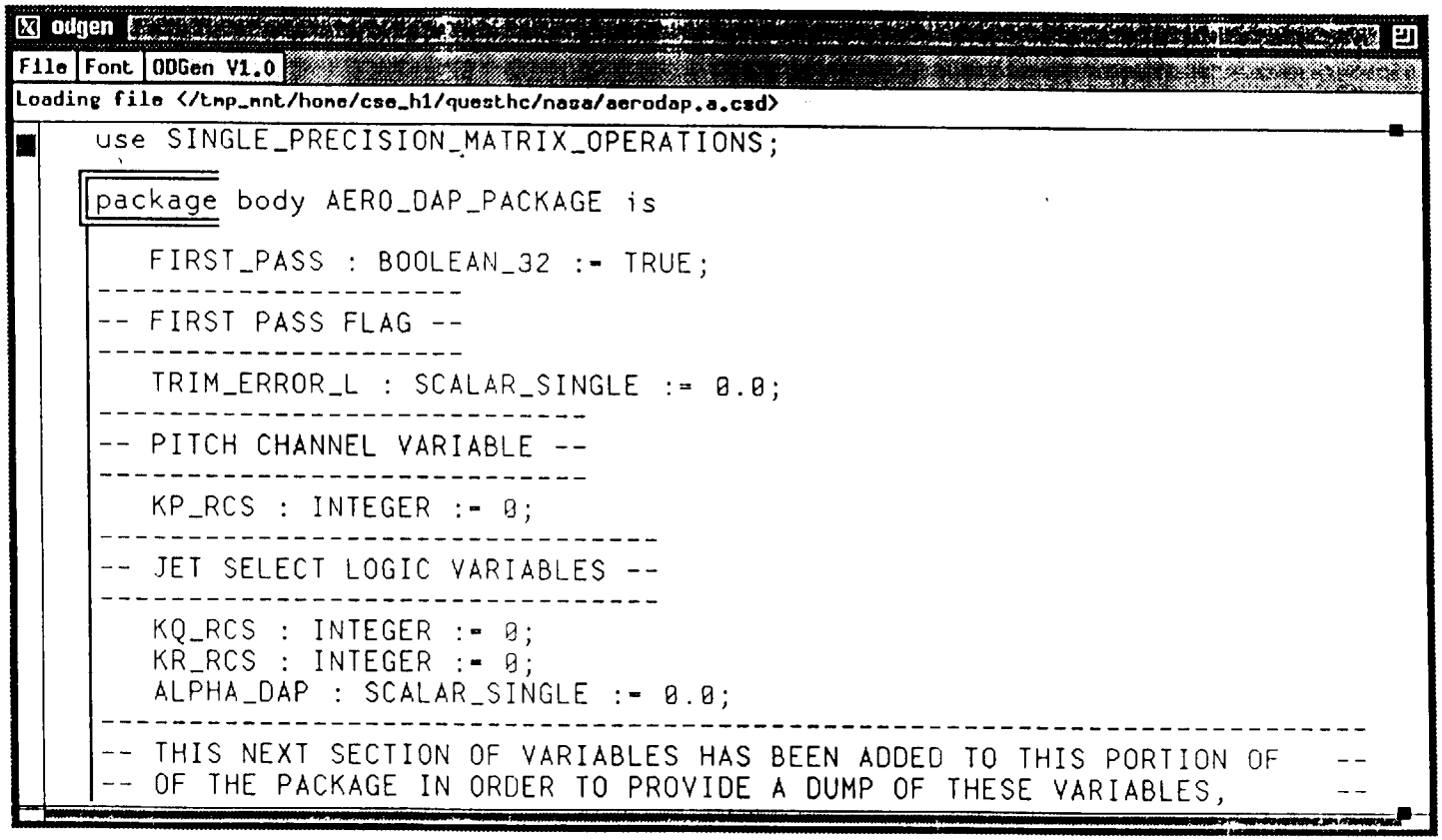


Figure 14. ODgen Source Code with CSD

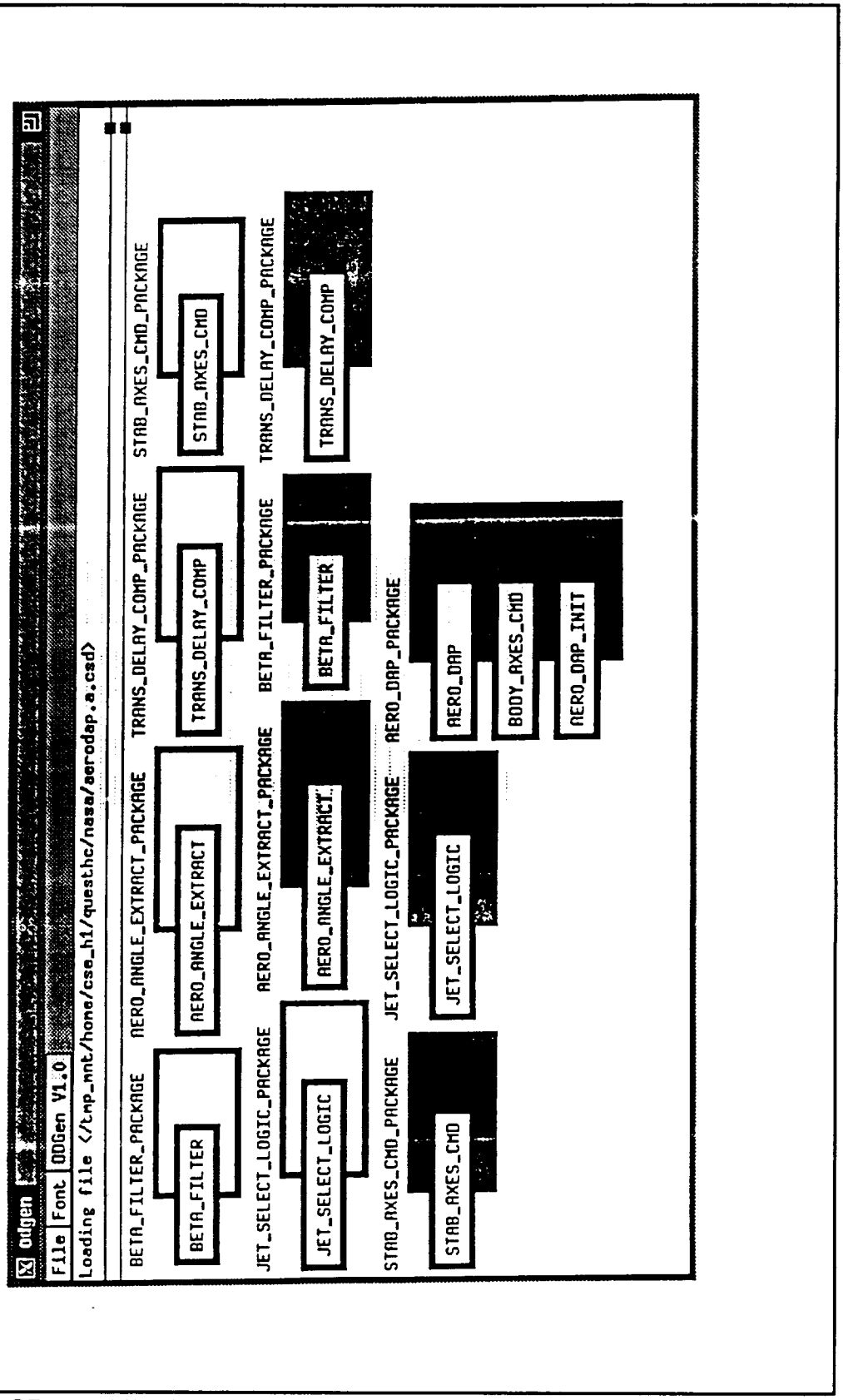


Figure 15. ODgen All Diagram View

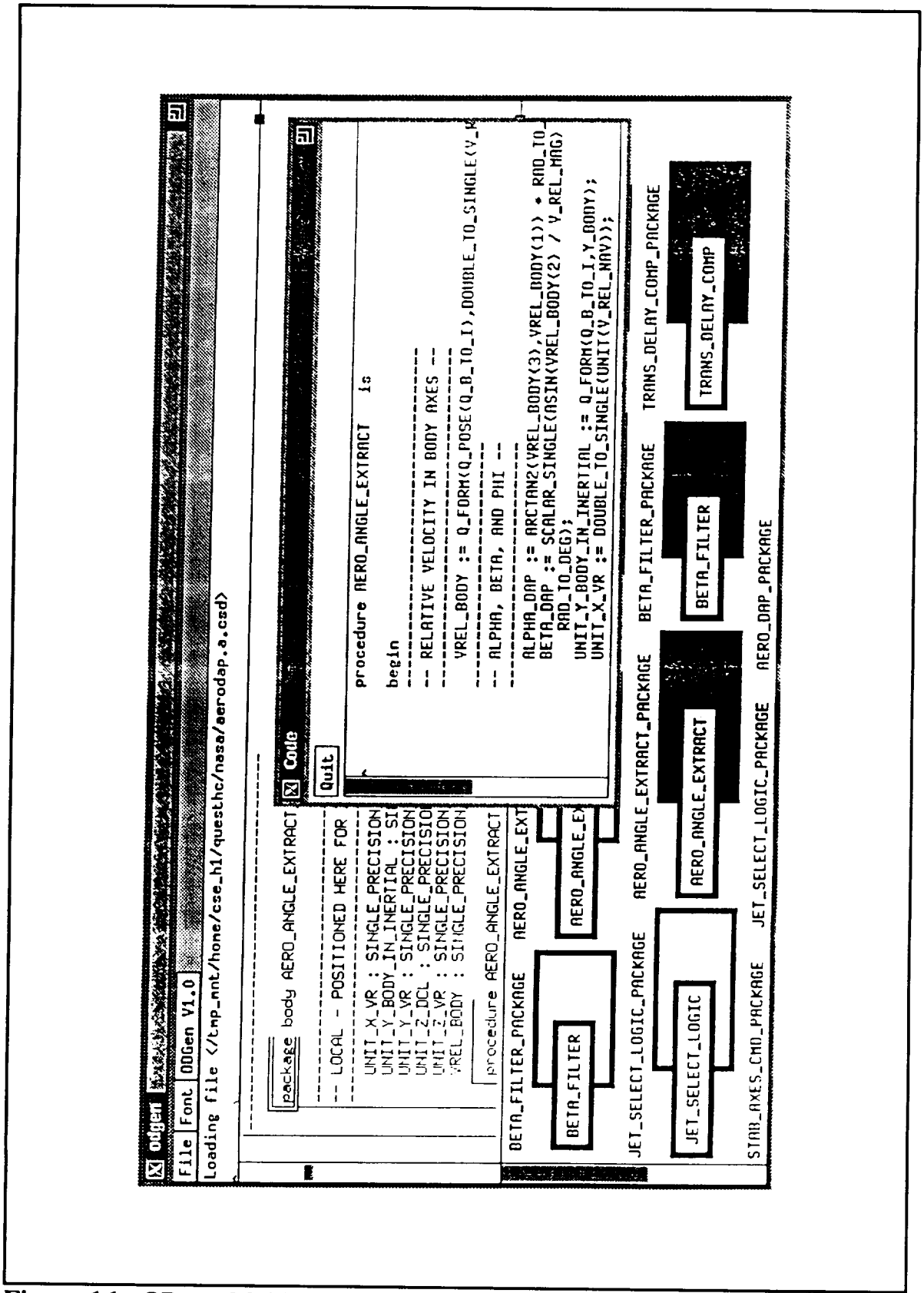


Figure 16. ODgen Multiple View

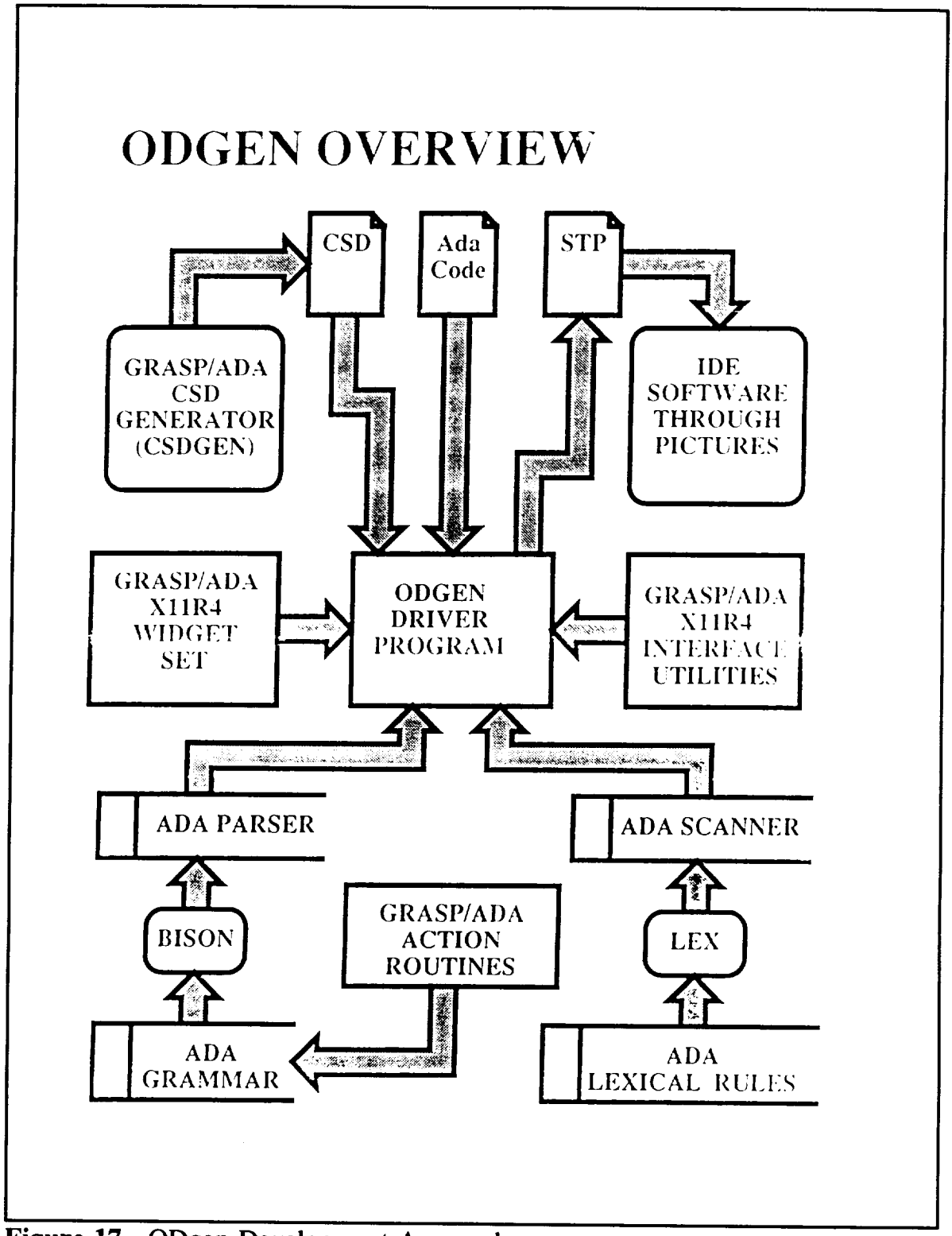


Figure 17. ODgen Development Approach

assumptions made at the onset of the GRASP project was that the input programs would be syntactically correct.)

ODgen Widget Set. The next step in the development of ODgen was the design of the widget set for the display of the object diagrams. Widgets were designed for each of the object diagram symbols, including: packages, tasks, generic packages, generic tasks, and subprograms. The Package widget has been implemented in the prototype. Several practical concerns severely influenced the design of the widgets. In the following paragraphs, these concerns are discussed along with their impact on the development of the widget set.

Initially, the ODgen widgets were to be developed as compound widgets similar to the X11R4 Athena Dialog widget. The X11R4 Athena Form widget would have been used as a background upon which to place the ODgen symbol labels (represented with the X11R4 Label widget). The objects and modules of the Package widget and the entries of the Task widget could have been neatly represented using the X11R4 Athena Command widgets, or derived subclasses. These compound widgets could therefore have been developed fairly quickly using reliable off-the-shelf components. However, this approach was abandoned for several reasons. First, this widget set was developed in May of 1991 under X11R4. Release 5 of the X Window System was slated for distribution in the fall of 1991. Between releases 3 and 4 of the X Window System, major changes were made in the implementation of the Athena widget set, with the developer freely admitting that he had given up on maintaining compatibility for any widgets subclassed off of the Release 3 widgets. Therefore, the decision was made to rely as little as possible on the Release 4 widgets in case a similar fate should await the Release 5 widgets. Second, to present the objects as rounded rectangles, the widget set would have to make use of the XmuShapeRoundedRectangle shape extension. Because not all X servers support this extension, its use was prohibited to maximize

portability. Third, the Athena Command widget, being a full widget, carries a great deal of unnecessary overhead which is unnecessary and unwanted in this application. As a typical application using the Package widget would contain a great many objects and modules, a better implementation would operate similarly to the Athena SimpleMenu widget, using gadgets rather than widgets to represent the objects and modules.

The second possibility considered was to implement the Odgen widget set as a series of simple composite widgets and to provide gadgets for the objects and modules of the Package widget and for the entries of the Task widget. This approach would have the added advantage of allowing the programmer to subclass these gadgets and therefore easily customize the widget's appearance. However, this approach was abandoned for several reasons. First, this would have forced the Package and Task widgets to be implemented as a number of widgets (the Package or Task widget itself and the associated gadgets), substantially increasing its complexity and comprehensibility to the applications programmer. This approach was undesirable since a major goal in the development of this widget was its ease of use. Second, composite widgets are widely regarded as being difficult to write as they must deal with geometry management and other concerns. In addition, they incur more overhead than simpler classes of widget, and this overhead was regarded as being unnecessary in this instance. Third, because the objects are represented using rounded rectangles, and because gadgets draw directly into their parent's window, this approach would have required a programming workaround because of an extremely subtle problem which will not be discussed here. Fourth, Release 5 of the X Window System is rumored to have made several changes in the way in which gadgets and objects are implemented. To prevent substantial rewriting of this widget in the near future, and for the other reasons cited above, this approach to the implementation of the ODgen widget set was not taken.

The third possibility considered, and the one which was actually used, was the implementation of the Package and Task widgets as simple widgets with convenience functions for manipulating the objects and modules and entries. This approach had several advantages, chiefly the simplified programmer interface. The applications programmer would only have to deal with one widget, rather than with a composite widget and a number of gadgets and all of their associated header files and resources. A second advantage is that using convenience functions to manipulate the subcomponents sufficiently abstracts the process such that the widgets could be rewritten in a future release to support gadgets, after the Release 5 gadget modifications are known, without requiring application programs to be modified (the convenience functions could be modified to create instances of the gadgets and insert them into the appropriate widget, therefore achieving backward compatibility). A third advantage is that implementing the ODgen widget set as a subclass of only the Athena Core widget greatly minimizes the widget overhead which would accompany a compound or composite widget.

However, there are several disadvantages to the chosen implementation approach. Because each ODgen component is represented by only one widget, associating translations to actions operating on the subcomponents becomes difficult. To offset this, a routine was written which ╥fakes╨ events for the subcomponents (actually, this routine was lifted from the X11R4 Athena SimpleMenu widget and modified). A default translation is provided which works reliably and will probably satisfy most application programmers, but other translations using the provided actions are possible provided the application programmer remember that the subcomponents do not truly receive events.

In summary, the current implementation of the GRASP/Ada Package widget is a product substantially influenced by the current release status of the X Window System, the

impending changes in the next release of X, the history of sweeping changes in the past releases of the Athena widget set, and the GRASP/Ada goals of providing an easy-to-use tool that will require little future maintenance.

GRASP Library. The next step in the development of the ODgen prototype was the design of a subset of the GRASP library. This subset needed to be suitably complex to store and manipulate structural information pertaining to the Ada program under study, yet simple enough to implement in a variety of ways, enabling the feasibility and performance of the library to be evaluated. Action routines were embedded in the Ada grammar to extract information for each package, task, complex data type, generic unit, task entry, and procedure call. This information was combined with other data (source file name, beginning line number, ending line number, identifier, scope) and stored in the GRASP library prototype for retrieval and use by the ODgen object diagram display mechanism. This prototype is currently under evaluation to determine whether this approach will be chosen or abandoned in favor of one of the two alternatives: the use of the VADS library system with DIANA; or the use of the StP TROLL/USE relational database system.

ODgen User Interface. The next part of the development of the ODgen prototype was the design of a user interface that would enable ODgen to be used as a standalone system. The goals of this part of the development were twofold: first, to design a functionally complete interface that would enable the user to easily use ODgen as a standalone tool; and second, to design an interface sufficiently compatible with the GRASP/Ada CSDgen component and user interface such that the two could be easily merged.

To meet the first goal, the user interface tools created in the development of the CSDgen component were streamlined and slightly modified to be applicable to both

interfaces. The tools were soundly designed and required very little modification for their use in the ODgen prototype. To meet the second goal, the "look and feel" of the CSDgen component was preserved in the design of the ODgen interface, with a few improvements. The multiple windows characteristic of the CSDgen component were reduced in number with the introduction of paned windows, thus slashing the amount of overhead associated with the implementation of multiple windows, as well as more tightly associating the object diagrams with their associated code, and increasing the freedom of the user in determining the customization of the various views.

The parser, scanner, customized widget set, GRASP/Ada interface utilities from the development of CSDgen, and a customized driver program were combined to create the ODgen executable. This program allows the user to select and load either Ada source files or CSD files into a window and to generate the corresponding object diagram. The program is currently under evaluation to determine any future enhancements which may prove useful. When the evaluation is complete and the standalone program is deemed ready for distribution, it will be integrated with the CSDgen component to complete the GRASP/Ada system. The major problem which is foreseen in the integration is the combination of the two sets of action routines into one, a task which will be greatly simplified due to the GRASP utility routines common to each.

7.0 Future Requirements

The GRASP/Ada project has provided a strong foundation for the automatic generation of graphical representations from existing Ada software. To move these results in the direction of visualizations to facilitate the processes of verification and validation (V & V), numerous additional capabilities must be explored and developed. The proposed follow-on research is described by tasks partitioned into three phases. A small team is expected to work on each phase for a period of up to one year. Operational prototypes will be demonstrated at the end of each phase.

7.1 Phase 1 - Generators and Editors for Visualizations

Phase 1 consists of five subtasks. The first is to **formulate a set of graphical representations that directly support V & V of Ada software at the algorithmic, architectural and system levels of abstraction.** This task will include an on-going investigation of visualizations reported in the literature as currently in use or in the experimental stages of research and development. In particular, specific applications of visualizations to support V & V procedures will be investigated and classified. A small, but representative, Ada program will be utilized to formulate and evaluate a set of graphical representations, and the feasibility of reverse engineering the diagrams from Ada PDL and source code will be evaluated. These graphical representations are expected to undergo continual refinement as the automated tools that support them are developed.

The second subtask of Phase 1 is to **design and implement a prototype software tool to generate visualizations from various levels of Ada PDL to support V & V during**

detailed design. The previous efforts of the GRASP/Ada project have focused on the generation of graphical representations from syntactically correct Ada source code. Since most detailed design is done in an Ada PDL which is less rigorous than Ada, the capability to generate visualizations directly from PDL is required to facilitate verification during the detailed design phase of the life cycle. The diagrams generated in Phase 1 are expected to focus on the algorithmic level of representation.

The third subtask of Phase 1 is to **design and implement a prototype software tool to generate visualizations from software written in C.** Since much of NASA's production software is currently being written in a combination of C and Ada, the capability to generate visualizations from C source code is required to support visual verification of the integrated software system. And since C is intrinsically less readable than Ada, maintenance personnel may greatly benefit from algorithmic-level diagrams generated from C source code.

The fourth subtask of Phase 1 is to **design and implement a prototype graphically-oriented editor which provides capabilities for dynamic reconstruction of the diagrams generated in the tools described above.** This capability will directly support visual verification at its most primitive and important levels, as PDL or source code is entered or modified. In this mode, the graphical representation can provide immediate visual feedback to the user in an incremental fashion as individual structural and control constructs are completed. The present GRASP/Ada prototype generates the graphical representation only after a complete compilation unit of source code has been entered correctly.

Finally, the fifth subtask of Phase 1 is to **design and implement a user interface capable of supporting a state-of-art multi-windowing paradigm.** The user interface for the tools developed in this research project will be built using the X Window System. This should facilitate eventual integration of the tools into any Ada programming support

environment (APSE) which runs under a similar window manager. In addition, this multi-windowing paradigm will allow the toolset to take full advantage of the current capabilities of powerful workstation hardware.

7.2 Phase 2 - Evaluation and Extension

Phase 2 consists of five subtasks. The first is to **continue the tasks defined in Phase 1 with respect to refinement of the V & V process, implementation of the prototype tools, and intertool communication**. The results of the investigation in Phase 1 will be used to refine the V & V process and the visualizations which support the process. The individual tools prototyped in Phase 1 will be integrated through a window manager for the X Window System. The user interface and a persistent storage mechanism such as DIANA will provide the basis for intertool communication.

The second subtask of Phase 2 is to **evaluate the individual tools developed in Phase 1**. Representative sets of programs written in PDL, Ada and C will be utilized to evaluate the set of graphical representations generated by the prototype. These graphical representations and the automated tools that support them are expected to undergo continual refinement during Phase 2.

The third subtask of Phase 2 is to **design and implement a prototype software tool for generating architectural diagrams (ADs) from Ada PDL and a combination of Ada and C source code, to support the process of V & V.** The Phase 1 prototype, which focused on the generation of an algorithmic notation, will be extended to include architectural diagrams. This task will include (1) development of procedures for identifying and recording module interconnections, (2) development of algorithms for architectural diagram layout, and (3) development of methods for displaying/printing architectural diagrams on hardware

available for this research. The tool will be used on representative Ada software. The generated set of graphical representations will be evaluated for completeness, correctness, and general utility as an approach to reverse engineering.

The fourth subtask of Phase 2 is to **investigate the potential for integration of the toolset with currently available commercial systems.** Commercial CASE systems and APSEs will be surveyed to determine appropriate commercial systems to target for integration. Many vendors are currently developing "open architecture" systems to facilitate the integration of third party tools.

The fifth subtask of Phase 2 is to **investigate the use of visualization tools to support software testing, particularly unit level branch coverage analysis.** Software testing is an important and essential component of V & V. Visualization tools are extremely useful for analyzing and reporting branch coverage. In addition, they may be very useful for graphically selecting a path for which data items to drive the path should be generated. This task would be done in conjunction with QUEST/Ada, a related project which has focused on the theoretical issues of test data generation [BRO90].

7.3 Phase 3 - Evaluation and Integration with Commercial Systems

Phase 3 has three subtasks. The first is to **complete the tasks defined in Phases 1 and 2 with respect to refinement, intertool communication, and integration of an operational prototype.** In particular, the user interface will be completed as a basis for overall integration of the prototype tools.

The second subtask of Phase 3 is to **evaluate the toolset developed in Phases 1 and 2.** Software systems which are representative of three levels of size and complexity, will be utilized to evaluate the set of graphical representations generated by the prototype as well as

the prototype itself. These systems will be written in Ada/PDL, Ada, C, or a combination of Ada and C. The graphical representations generated and the prototype are expected to undergo continual refinement as a result of the evaluation.

The third and final subtask of Phase 3 is to **integrate with currently available commercial systems those components of the prototype toolset which show the most promise for improving V & V.** The results of the survey of commercial CASE systems and APSEs conducted in Phase 2 and the ongoing evaluation of the prototype tools will be used to determine appropriate commercial systems to target for integration.

BIBLIOGRAPHY

ADA83 *The Programming Language Ada Reference Manual.* ANSI/MIL-STD-1815A-1983. (Approved 17 February 1983). In *Lecture Notes in Computer Science*, Vol. 155. (G. Goos and J. Hartmanis, eds) Berlin : Springer-Verlag.

ADO85 Adobe Systems Inc. *POSTSCRIPT Language Reference Manual*, (3rd Ed.) Reading, MA: Addison-Wesley, 1985.

ADO88 Adobe Systems Inc. *POSTSCRIPT Language Program Design*, Reading, MA: Addison-Wesley, 1988.

AMB89 Amber Allen L. et al. "Influence of Visual Technology on the Evolution of Language Environments," *IEEE Computer*, Vol. 22, No 10, October 1989, 9-22.

BEN88 Bennett, Steven J. and Randall, Peter G. *The LaserJet Handbook: A Complete Guide to Hewlett-Packard Printers and Compatibles*, New York: Brady, 1988.

BIG89 Biggerstaff, Ted J. "Design Recovery for Maintenance and Reuse," *IEEE Computer*, July 1989, 36-49.

BOO83 Booch, Grady. *Software Engineering with Ada.* Menlo Park, CA : The Benjamin/Cummings Publishing Company, Inc., 1983.

BOO86 Booch, Grady. "Object-Oriented Development," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986, 211-221.

BOO87a Booch, Grady. *Software Engineering with Ada.* (Second Edition). Menlo Park, CA : The Benjamin/Cummings Publishing Company, Inc., 1987.

BOO87b Booch, Grady. *Software Components With Ada : Structures, Tools, and Subsystems.* Menlo Park, CA : The Benjamin/Cummings Publishing Company, Inc., 1987.

BRO80 Brosgol, B.M., et al. *TCOLada: Revised Report on An Intermediate Representation for the Preliminary Ada Language.* Technical Report CMU-CS-80-105, Carnegie Mellon University, Computer Science Department, February 1980.

BRO90 D. B. Brown, K. H. Chang, W. H. Carlisle, and J. H. Cross, "QUEST - Testing Tools For Ada," *Task 1, Phase 2 Report* of "The Development of a Program Analysis Environment for Ada," G. C. Marshall Space Flight Center,

NASA/MSFC, AL 35821 (NASA-NCC8-14), August 1990, 85 pages + Appendices.

BUH89 Buhr, R. J. A., Karam, G. M., Hayes, C. J., and Woodside, C. M. "Software CAD: A Revolutionary Approach," *IEEE Transactions on Software Engineering*, Vol. 15, No. 3, March 1989, 235-249.

CAR91 W. H. Carlisle, J. H. Cross and S. R. Allen, "Exchange Functions in Ada," *Journal of Pascal, Ada and Modula 2*, Vol. 10, No. 3, May/Jun. 1991, accepted for publication.

CHE86 Cherry, George W. *PAMELA Designer's Handbook*, Volume 2, Analytical Sciences Corp., Reading, MA, 1986.

CHI90 E. J. Chikofsky and J. H. Cross, "Reverse Engineering and Design Recovery - A Taxonomy," *IEEE Software*, Jan. 1990, 13-17.

CHO90 Choi, Song and Scacchi, Walt. "Extracting and Restructuring the Design of Large System," *IEEE Software*, January 1990, 66-71.

COH86 Cohen, Norman H. *Ada as a second language.* New York : McGraw-Hill Book Company, 1986.

CRO88 Cross, J. H. and Sheppard, S. V. "The Control Structure Diagram: An Automated Graphical Representation For Software," *Proceedings of the 21st Hawaii International Conference on Systems Sciences*, January 6-8, 1988, 446-454.

CRO89 Cross, J. H., Morrison, K. I., May, C. H. and Waddel, K. C. "A Graphically Oriented Specification Language for Automatic Code Generation (Phase 1)", *Final Report*, NASA-NCC8-13, SUB 88-224, September 1989.

CRO90a J. H. Cross, K. I. Morrison, C. H. May, "Generation of Graphical Representations From Source Code," *Proceedings of the Southeast Regional ACM Computer Science Conference*, April 18-20, 1990, Greenville, South Carolina, 54-62.

CRO90b J. H. Cross, "GRASP/Ada Uses Control Structure," *IEEE Software*, May 1990, 62.

CRO90c J. H. Cross, et.al., "Reverse Engineering Tools For Ada," *Task 2, Phase 2 Report* of "The Development of a Program Analysis Environment for Ada," G. C. Marshall Space Flight Center, NASA/MSFC, AL 35821 (NASA-NCC8-14), August 1990, 78 pages + Appendices.

CRO90d J. H. Cross, S. V. Sheppard and W. H. Carlisle, "Control Structure Diagrams for Ada," *Journal of Pascal, Ada, and Modula 2*, Vol. 9, No. 5, Sep./Oct. 1990.

CRO92 J. H. Cross, E. J. Chikofsky and C. H. May, "Reverse Engineering," *Advances in Computers*, Vol. 35, 1992, in process.

DAU80 Dausmann, M., et al. *AIDA Introduction and User Manual.* Technical Report Nr. 38/80, Institut fuer Informatik II, Universitaet Karlsruhe, 1980.

DAV90 Davis, R. A., "A Reverse Engineering Architectural Level Control Structure Diagram," *M.S. Thesis*, Auburn University, December 14, 1990.

FOR88 Forman, Betty Y. "Designing Software With Pictures," *Digital Review*, July 11, 1988, 37-42.

GOO83 Goos, G. et al. *DIANA: An Intermediate Language for Ada* (Revised Version). In *Lecture Notes in Computer Science*, Vol. 161. (G. Goos and J. Hartmanis, eds.) Berlin : Springer-Verlag, 1983.

GOU85 Gould, John D. and Lewis,Clayton. "Designing for Usability: Key Principles and What Designers Think," *Communications of the ACM*, Vol. 28, No. 3, March 1985, 300-311.

HAM79 Hamilton, M. and Zeldin, S. "The Relationship Between Design and Verification," *The Journal of Systems and Software*, Elsevier North Holland, Inc., 1979, 29-56.

HOL88 Holzgang, David A. *Understanding POSTSCRIPT Programming* (2nd Ed.) San Francisco, CA: Sybex, 1988.

HOL89 Holzgang, David A. *POSTSCRIPT Programmer's Reference Guide*, Glenview, IL: Scott, Foresman, 1989.

HPC87 *LaserJet Series II Printer User's Manual*, (2nd Ed.) Boise, ID: Hewlett-Packard Company, 1987.

KRA89 Kramer, Jeff, et al. "Graphical Configuration Programming," *IEEE Computer*, Vol. 22, No. 10, October 1989, 53-65.

LEH89 Lehr, Ted, et al. "Visual Performance Debugging," *IEEE Computer*, Vol. 22, No. 10, October 1989, 38-51.

LYO86 Lyons, T.G.L. and Nissen, J.C.D., eds. *Selecting an Ada environment.* New York : Cambridge University Press (on behalf of the Commission of the European Communities), 1986.

MAR85 Martin, J. and McClure, C. *Diagramming Techniques for Analysts and Programmers.* Englewood Cliffs, NJ : Prentice-Hall, 1985.

McD84 McDermid, John and Ripken, Knut. *Life cycle support in the Ada environment.* New York : Cambridge University Press (on behalf of the Commission of the European Communities), 1984.

McK86 McKinley, Kathryn L. and Schaefer, Carl F. *DIANA Reference Manual.* Draft Revision 4 (5 May 1986). Bethesda, MD : Intermetrics, Inc. Prepared for Naval Research Laboratory, Washington, D.C., 1986.

MEN89 Mendal, G. et al. *The Anna-I User's Guide and Installation Manual.* Stanford, CA : Stanford University (Program Analysis and Verification Group : Computer Systems Laboratory), September 22, 1989.

NES81 Nestor, J.R., et al. *IDL - Interface Description Language: Formal Description.* Technical Report CMU-CS-81-139, Carnegie Mellon University, Computer Science Department, August 1981.

NOR86 Norman, Kent L., Weldon, Linda J., and Shneiderman, Ben. "Cognitive layouts of windows and multiple screens for user interfaces," *International Journal of Man-Machine Studies*, Vol. 25, 1986, 229-248.

OBR89 O'Brien, Caine. "Run-Time Reverse Engineering Speeds Software Troubleshooting," *High Performance Systems*, November 1989, 41-48.

PAU90 Paulisch, Frances Newberry, and Tichy, Walter F. "EDGE: An Extendible Graph Editor," *Software Practice and Experience*, Vol. 20(S1), June 1990, pp. 63-88.

PER80 Persch, G., et al. *AIDA Reference Manual.* Technical Report Nr. 39/80, Institut fuer Informatik II, Universitaet Karlsruhe, November 1980.

PRE87 Pressman, Roger S. *Software Engineering: A Practitioner's Approach*, McGraw-Hill, New York, NY, 1987.

ROE90 Roetzheim, William H. *Structured Design Using HIPO II*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

ROM89 Roman, Gruia-Catalin, et al. "A Declarative Approach to Visualizing Concurrent Computations," *IEEE Computer*, Vol 22, No. 10, October 1989, 25-36.

ROS85 Rosenblum, David S. "A Methodology for the Design of Ada Transformation Tools in a DIANA Environment," *IEEE Computer*, Vol. 2, No. 2, March 1985, 24-33.

SCH89 Schwanke, R. W., et al. "Discovering, Visualizing, and Controlling Software Structure," *Proceedings of the Fifth International Workshop on Software Specification and Design*, May 19-20, 1989.

SEL85 R. Selby, et. al., "A Comparison of Software Verification Techniques," NASA *Software Engineering Laboratory Series* (SEL-85-001), Goddard Space Flight Center, Greenbelt, Maryland, 1985.

SHA89 Shannon, K. and Snodgrass, R. *Interface Description Language : Introduction and Manual Pages*. Chapel Hill, NC : Unipress Software, Inc. (University of North Carolina), May 1, 1989.

SHU88 Shu, Nan C. *Visual Programming*, New York, NY, Van Norstrand Reinhold Company, Inc., 1988.

SIE85 Sievert, Gene E. and Mizell, Terrence A. "Specification-Based Software Engineering with TAGS," *IEEE Computer*, April 1985, 56-65.

SMI88 Smith, Thomas, et al. "A Standard Interface to Programming Environment Information." In [HEI88], 251-262, 1988.

SNO86 Snodgrass, R. and Shannon, K. *Supporting Flexible and Efficient Tool Integration.* SoftLab Document No. 25, Chapel Hill, NC: Department of Computer Science, University of North Carolina, 1986.

STA85 Standish, T., "An Essay on Software Reuse," *IEEE Transactions on Software Engineeering*, SE-10 (9), 494-497, 1985.

SUG81 Sugiyama, Kozo, Tagawa, Shojiro, and Toda, Mitsuhiko. "Methods for Visual Understanding of Hierarchical System Structures," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-11, No. 2, February 1981, pp. 109-125.

TEI81 Teitelbaum, T. and REPS T., "The Cornell Program Synthesizer: A Syntax Directed Programming Environment, *Communications of the ACM*, 24, 9 (Sep.), 563-573.

TRI89 Tripp, L. L. 1989. "A Survey of Graphical Notations for Program Design -An Update," *ACM Software Engineering Notes*, Vol. 13, No. 4, 1989, 39-44.

WAR77 Warfield, John N. "Crossing Theory and Hierarchy Mapping," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-7, No. 7, July 1977, pp. 505-523.

WAR85 Warren, W.B., et al. *A Tutorial Introduction to Using IDL.* SoftLab Document No. 1, Chapel Hill, NC: Department of Computer Science, University of North Carolina, 1985.

WAS89 Wasserman, A. I., Pircher, P. A. and Muller, R.J. "An Object Oriented Structured Design Method for Code Generation," ACM SIGSOFT *Software Engineering Notes*, Vol. 14, No. 1, January 1989, 32-52.

WHI88 Whiteside, John., Wixon, Dennis, and Jones, Sandy. "User Performance with Command, Menu, and Iconic Interfaces," in *Advances in Human Computer Interaction*, Vol. 2, ed. Hartson, Rex H., and Hix, Deborah, Norwood NY, Ablex, 1988, 287-315.

YOU89 Young, Douglas A. *Window Systems Programming and Applications with Xt*, Prentice Hall, Englewood Cliffs, New Jersey 07632, 1989.

APPENDICES

A. "Reverse Engineering"
by J. Cross, E. Chikofsky and C. May

B. "Control Structure Diagrams For Ada"
by J. Cross, S. Sheppard and H. Carlisle

C. Extended Examples

D. User Manual (MAN-Page)

Appendix A

"Reverse Engineering"

by

James H. Cross II
Auburn University

Elliot J. Chikofsky
Index Technology Corp.

and

Charles H. May, Jr.
Auburn University

To Be Published in *Advances in Computers*, Vol. 35, 1992.

Reverse Engineering

JAMES H. CROSS II

Computer Science and Engineering
Auburn University
Auburn, Alabama

ELLIOT J. CHIKOFSKY

Engineering and Information Systems
Northeastern University
Boston, Massachusetts

CHARLES H. MAY, JR.

Computer Science and Engineering
Auburn University
Auburn, Alabama

Based on "Reverse engineering and design recovery: A taxonomy" by E. J. Chikofsky and J. H. Cross II which appeared in IEEE Software 7(1), 13-17, January, 1990.

TABLE OF CONTENTS

1. Introduction 1

2. The Context of Reverse Engineering 2

3. Taxonomy of Terms 6
   3.1 Forward Engineering 7
   3.2 Reverse Engineering 7
   3.3 Restructuring 12
   3.4 Reengineering 14

4. Reverse Engineering Throughout the Life Cycle: Objectives and Purposes 15
   4.1 Objectives 15
   4.2 Maintenance 17
   4.3 Reuse 19
   4.4 Software Quality Assurance 20

5. Economic and Legal Issues 25
   5.1 Economic Issues 26
   5.2 Legal Issues 28

6. Survey of Current Research 29
   6.1 Redocumentation 30
   6.2 Design Recovery 40

7. Conclusion 50

ACKNOWLEDGEMENTS 53

BIBLIOGRAPHY 54

1. Introduction

The availability of computer-aided software engineering (CASE) environments has redefined how many organizations approach software development. To meet their true potential, CASE environments are being applied to the problems of maintaining and enhancing existing software systems. The key lies in applying reverse engineering approaches to software systems.

The term *reverse engineering* has its origin in the analysis of hardware, where the practice of deciphering designs from finished products is commonplace. In this arena, reverse engineering is regularly applied to improve one's own products, as well as to analyze a competitor's products or those of an adversary in a military or national security situation.

In a landmark paper on the topic, Rekoff (1985) defined *reverse engineering* as "the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system." (p. 244) He described such a process as being conducted by someone other than the developer, "without the benefit of any of the original drawings....for the purpose of making a clone of the original hardware system...." (p. 244). In applying these concepts to software systems, one finds that many of these approaches apply to gaining a basic understanding of a system and its structure. While the hardware objective traditionally is to duplicate the system, the software objective is most often to gain a sufficient design-level understanding to aid maintenance, to strengthen enhancement, or to support replacement.

This chapter provides an overall context for reverse engineering in terms of the traditional software life cycle and then defines and relates six terms: *forward engineering*, *reverse engineering*, *redocumentation*, *design recovery*, *restructuring*, and *reengineering*. The objectives

of reverse engineering are then considered along with the major purposes which provide technical justification for performing reverse engineering: maintenance, reuse, and software quality assurance. This is followed by a discussion of the economic and legal ramifications of reverse engineering. The heart of the chapter is a survey of recent research in this area.

2. The Context of Reverse Engineering

To describe the notions of software forward and reverse engineering adequately, there first must be a clarification in several related areas: software engineering terminology, the nature of life cycle models, the identification of phases within the life cycle, the types of information produced by each life cycle phase, and the nature of the subject system.

Despite standardization efforts, software engineering literature has been rather loose in its use of terminology. Terms that have been tossed about for decades sometimes have had their original meanings muddled. For the purposes of this chapter, this section will *attempt* to standardize some basic software engineering terms to provide a more stable foundation on which to discuss reverse engineering terminology. As an example, the term *specification* has come to refer in the software engineering community both primarily as an outlining of requirements (Pressman, 1987, p. 152) and also more generally as a recorded document of any life cycle activity, including design, for example (Pressman, 1987, p. 254). For the purposes of this chapter, *specification* will refer to the more general latter definition.

In addition to giving clear definition to the products of the life cycle, it is expedient to define clearly the process by which these products are created. It is assumed that a reasonably ordered life cycle model exists for the software development process. The model may be represented as the traditional waterfall, as a spiral, or in some other form that generally can be

represented as a directed graph. While one can expect there to be iteration within and overlap between phases of the life cycle, and perhaps even recursion, its general directed graph nature allows sensible definition of forward (downward) and reverse (upward) activities.

In addition, one can divide the typical life cycle into three fundamental activities. The first, *requirements analysis*, is described by Sommerville (1989) as the establishment of services to be provided by the system and the constraints under which the system must operate. The second major activity, *design*, is the creation of a solution plan. This plan details the makeup of software components and the nature of their interrelationships. Simply put, requirements analysis establishes what the system should do and under what circumstances it is to be done, whereas design establishes how it is to be done. The third activity, *implementation*, includes the coding of the solution, testing, debugging, and delivery of the operational system. Other activities typically thought of as part of the life cycle, such as maintenance, are in fact reiterations of the previous activities (Basili, 1990). Maintenance is discussed in Section 4 as one of three primary purposes for reverse engineering.

An *abstraction* is a model that summarizes the detail of the subject it is representing. A higher-level abstraction thus has less detail than a lower level abstraction. It provides more of a conceptual framework enabling the broad picture to be seen, but not the detail. The subject of examination in reverse engineering is primarily the finished implementation of the system. Therefore, a design specification is an abstraction of the system and a requirements specification is a higher-level abstraction of the same system.

It is important to distinguish between *levels* of abstraction, an idea that labels fundamental phases of development, and *degrees* of abstraction within a single phase. The three life cycle activities described above can be considered as corresponding to different levels of abstraction. The levels differ primarily because the corresponding phases have distinct purposes. Although again this notion may be considered simplistic by some, it is indeed a conceptual leap to move from a specification of what the system is to do to a specification of how it is to be done.

Furthermore, another leap is required to move from a design plan to a tangible implementation. Moreover, to be faithful to the term *abstraction*, it is true to speak of the specifications from higher phases as being summations (to some extent) of specifications from lower phases.

It is also possible within a life cycle phase, or abstraction level, to specify the system with varying degrees of detail. For instance, Sommerville (1989) describes a *requirements definition* as a broad, largely textual statement of system services intended to be a basis for common understanding among all concerned parties. He distinguishes this *definition* from a more detailed *requirements specification*, which more completely and precisely spells out the nature of the services. In design, there are varying degrees of detail, from comprehensive architectural designs which specify interfaces and dependencies among major components while revealing little or no detail within the constituent modules, to procedural designs of the modules themselves which provide algorithms and detailed data structures. Even implementations can vary in degree of complexity as they evolve; for instance, modules which have not been fully implemented can be stubbed, allowing calls to be made to them.

As Sommerville (1989) suggests, the boundaries between levels of abstraction or degrees within levels can seem subjective and amorphous, especially when the artifacts of the various levels appear suspiciously similar. For instance, an Ada-like requirements specification, intended to delineate system functions and how they might *appear* to the user to work, may look much like a design specification for the same system, even though the purposes for the two specifications are entirely different. In fact, Sommerville (1989) further suggests that it is somewhat likely that low-level requirements specification and high-level design will proceed simultaneously.

Reverse engineering methods attempt to transform system representations from one form to another, sometimes moving to less detailed forms. It is prudent, then, to be aware of the types of representation which can be subjects and targets of reverse engineering. At the design level, one finds various forms of pseudocode, differing in language base and syntactic formality. In

addition, a module may go through several iterations of pseudocode as detail is added. Furthermore, one may find algorithmic graphical representations which illustrate procedural control flow and control structure. Moving from procedural design, one may discover various depictions of software architecture such as module calling diagrams, other component dependency diagrams, and module interconnection languages such as *NuMIL* (Choi and Scacchi, 1990). Many of these types of representation can vary in the degree of detail that is actually shown.

At the requirements level, there are various forms of representation that could be considered. The most popular is the traditional natural language narrative. A natural language specification can be written in innumerable degrees of complexity and can range from highly amorphous to highly structured. Sommerville's *requirements definition* is usually described as consisting, in part, of a very broad natural language description. Numerous graphical representations are used in requirements definition and specification. The most widely used is the *data flow diagram* (DeMarco, 1979) and its many derivatives, which can be used to depict a system at several levels and degrees of abstraction. There are special languages which exist solely as tools for requirements specification, among them the Ada-based *Anna* (Luckham and von Henke, 1985). Finally, there are formal specification notations such as *VDM* (Jones, 1986) and *Z* (Abrial, 1980) which are model-based languages that use mathematical entities such as sets and functions to specify systems.

In reality, there are few representations which can be restricted to only one life cycle phase. As seen earlier, Ada PDL can be used as both a requirements specification medium and a design specification medium. A data flow diagram can be used both to depict what a software system does with data as well as how the processes interact. Apparently, the context in which a representation medium is used is paramount, rather than the medium itself. Hence, whether an extracted representation is a requirement representation or a design representation depends largely on the context in which the representation is used.

It is also important to note that a particular representation form extracted from a subject system may differ somewhat from a similar representation that was developed in the forward engineering process. The extracted form will reflect the idiosyncracies of the subject system representation more so than would the original form, which would be a reflection of the analyst's or designer's understanding of the problem. The tradeoff is that while some of the original design information may not be present in the extracted form, the information that is there should be an accurate reflection of the actual system.

Finally, the subject system to be reverse engineered may be a single program or code fragment, or it may be a complex set of interacting programs, job control instructions, signal interfaces, and data files. In all cases, however, it is an actual implementation that is traditionally considered the input for the reverse engineering process. However, as shall be seen in Section 4, it is conceivable to apply reverse engineering technologies from a point in the life cycle earlier than implementation. In fact, a tight coupling of forward and reverse engineering activities throughout the life cycle is a natural progression of modern CASE tool capabilities.

3. Taxonomy of Terms

An impediment to success is the considerable confusion over the terminology used in both technical and marketplace discussions. It is in the arena of reverse engineering, where the software maintenance and development communities meet, that various terms for technologies to analyze and to understand existing software systems have been used frequently in conflicting ways. In this section, forward engineering, reverse engineering, redocumentation, design recovery, restructuring, and reengineering are described with respect to the life cycle phases and activities in which they are involved (Chikofsky and Cross, 1990). Figure 1 is an illustration of

the relationship between the various processes defined below and the life cycle. The objective is not to create new terms but to rationalize the terms already in use. The resulting definitions apply to the underlying engineering processes, regardless of the degree of automation applied.

3.1 Forward Engineering

Forward engineering is the traditional process of moving from high-level representations and logical, implementation independent designs to the physical implementation of a system. While it may seem unnecessary, in view of the longstanding use of software engineering terminology, to introduce a new term, the adjective "*forward*" has come to be used where it is necessary to distinguish this process from reverse engineering. Forward engineering follows a sequence from the analysis of requirements through the design, and finally to an implementation. These activities have been fully described in Pressman (1987), Sommerville (1989) and many others.

3.2 Reverse Engineering

Reverse engineering is the process of analyzing a subject system in order to identify the system's components and their interrelationships and to create representations of the system, possibly at a higher level of abstraction. Reverse engineering generally involves extracting design artifacts and building or synthesizing representations that are less implementation dependent.

While reverse engineering often involves an existing operational system as its subject, this is not a requirement. One can perform reverse engineering starting from most phases of the life cycle to create or recreate artifacts of an earlier phase. In spanning the life cycle phases, reverse engineering covers a broad range starting from the existing implementation, recapturing the design, and deciphering the requirements actually implemented by the subject system.

Reverse engineering in and of itself does not involve changing the subject system or creating a new system based on the reverse engineered subject system. It is a process of *examination*, not a process of change or replication.

The overall goal of reverse engineering is to facilitate understanding of software systems, whether tools themselves undertake to understand their subjects or simply provide aids to help a human user to that end. Harandi and Ning (1990) point out that there are different levels of understanding software; they present four such levels, or views (also presented earlier by Kozaczynski and Ning (1989) with slightly different names):

- Implementation-level view. This implies an understanding of the fundamental components, usually control and declarative constructs of a particular system.
- Structure-level view. Implied here is an understanding of how the fundamental components relate to one another in a programming sense (e.g. scoping relationships, visibility relationships).
- Function-level view. This degree of understanding suggests an understanding of the function of a component or system in isolation, i.e. what the component or system does. A conceptual leap is required to move from the previous view to this view.
- Domain-level view. To move to this level requires yet another leap in that the understanding of this level is of the context in which the system is operating, rather than the operation of the system in isolation. Where the previous level asks, "What does the system (or component) do?", this level asks "Why?"

Two subareas of reverse engineering that achieve or facilitate understanding to different levels are *redocumentation* and *design recovery*. Figure 2 illustrates the respective spheres of these two subareas defined by the levels of understanding that they achieve and the extent of

their resources, either code analysis alone or code analysis combined with knowledge base technology.

3.2.1 Redocumentation

Redocumentation is the creation or revision of representations of a subject system which are intended to reflect certain software related characteristics inherent to the system. The redocumentation process creates its representations from information gathered from analysis of the subject system alone, although the presentation of the information may be guided from outside sources (e.g., diagram layout). The resulting forms of representation are considered alternate views intended to improve the comprehensibility of the overall system. These alternate views include data flow, data structure, and control flow diagrams.

Redocumentation is the simplest and oldest form of reverse engineering, and many consider it to be an unintrusive, weak form of restructuring. The "re-" prefix implies that the intent is to recover documentation about the subject system that once existed or should/could have existed. The emphasis here may, in fact, be on additional views, especially graphical, that were not created during the original forward engineering process due to the labor intensive nature of creating these views and then maintaining them. It is no coincidence that one of the primary functions of CASE tools has always been to aid in the creation and maintenance of graphical representations of the system. However, it is not uncommon over time to find a major gap between the content of the diagrams drawn by analysts and designers using CASE tools and the function of the actual source code written to implement the system. A major thrust of reverse engineering and, in particular redocumentation, is to bridge this gap.

Some common tools used to perform redocumentation are pretty printers, which display a code listing in an improved form; diagram generators, which create diagrams from code directly reflecting the control flow, code structure or data structure; and cross-reference listing generators. A key goal of these tools is to provide easier ways to visualize relationships among program

components so that one can recognize and follow paths clearly. Section 6.1 provides a survey of recent literature describing redocumentation tools.

It is important to note that these tools simply provide aids to the engineer in his or her quest for understanding. The tools do not attempt to derive any conclusions as to system function or purpose. It is required that redocumentation tools achieve a certain level of system understanding to produce the representations that they produce; however, this level of understanding does not generally transcend the structural-level view ascribe meaning to the analyzed system. One could characterize redocumentation as the recovery of recorded artifacts of the earlier development process. In contrast, the more ambitious recovery of the actual function, purpose, or essence of the system is given the name *design recovery*.

3.2.2 Design Recovery

Design recovery is a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system to identify "meaningful" higher-level abstractions beyond those obtainable directly by examining the system itself. These abstractions are attempts to impose "meaning" on a segment of the subject system.

Design recovery tools are distinguished from simple redocumentation tools in that the tools themselves attempt to "understand" the system rather than simply to provide alternate views to help users understand the system. This understanding goes beyond implementation- and structure-level knowledge and attempts to achieve function-level knowledge and even knowledge of a system's operating environment. However, it should be noted that artifacts such as the diagrams cited above, which have been classified as products of redocumentation since they can be generated deterministically, are considered design documents. In this context, redocumentation and design recovery activities may overlap considerably.

Complete design recovery attempts to reconstruct not only the function of a system, but also the process by which it was developed. Rugaber et al. (1990) emphasize the importance of recovering the design decisions made during original development to complete the framework for maintenance. A design decision is simply a part of the overall software plan. A design decision is a choice made to implement some function or concept in a particular manner, for example, the decision to implement a hash table with variable length or fixed length buckets.

Rugaber et al. (1990) point out that in order for maintenance, reverse engineering, and reuse to be effective, it is necessary to discover not only the design decisions themselves but also the alternatives to those decisions. It is also necessary to determine the rationale which led to those decisions in order that the rationale of the decisions may be preserved or altered with the full knowledge of the maintenance engineer. Furthermore, it is necessary to make clear the interrelationships between design decisions in order to assess the scope of change repercussions more accurately. The process by which these decisions are made will more than likely not be present in the code and must be "inferred" (i.e. guessed).

Design recovery is also characterized by the sources and span of information it should handle. Biggerstaff (1989) defines *design recovery* as follows. "Design recovery recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains....In short, design recovery must reproduce all of the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth. Thus, it deals with a far wider range of information than found in conventional software engineering representations or code." (p. 36)

In many of the design recovery tools examined in Section 6.2, there exists some form of knowledge base which binds programming patterns to functional concepts. These functional concepts are expressed in terms easy to understand by programmers and usually are used to constitute even more abstract functional ideas.

Design recovery is the more challenging form of reverse engineering because it attempts to mimic human reasoning in its goal of understanding. Any assistance which can be offered to a maintenance engineer in understanding a program will be of great value as he or she prepares to modify the system. The next two sections discuss topics that are related to reverse engineering but go beyond examination of a system to refurbishing it.

3.3 Restructuring

Restructuring is the transformation of a software system from one representation form to another, usually at the same relative abstraction level, while preserving the subject system's external behavior, i.e. functionality and semantics. Restructuring usually involves some form of reverse engineering, if only implicitly, from the original representation to some intermediate form. This intermediate form is then appropriately altered, with the restriction that the functionality is preserved.

A restructuring transformation is often one of appearance, such as altering code to improve its structure in the traditional sense of accepted design principles. The term *restructuring* came into popular use from the code-to-code transformations that recast a program from an unstructured, or "spaghetti," form to a structured, or ***goto***-less, form. However, the term has a broader meaning that recognizes the application of similar transformations and recasting techniques in reshaping data models, design plans, and requirements structures. Data normalization, for example, is a data-to-data restructuring transformation to improve a logical data model in the database design process. Arnold (1989) provides a comprehensive survey of software restructuring in which reverse engineering is related to restructuring and considered a natural extension of it. Using Arnold's definition of restructuring, redocumentation as described above would be a form of restructuring in that it provides an alternate or "restructured" view of existing information. For example, prettyprinting is the "restructuring" of the format of source code. However, since even an alternate view of the source code (e.g., an automatically generated

graphical representation) can provide the programmer or designer with additional insight, Arnold considers this form of restructuring to be reverse engineering.

Many types of restructuring can be performed with knowledge of structural form but without an understanding of meaning. For example, one can convert a series of ***if*** statements into a ***case*** statement, or vice versa, without knowing the program's purpose or anything about its problem domain. In this case, the knowledge is of structured programming theory. Choi and Scacchi (1990) describe a process which tries to determine the inherent, if not explicit, structure of a system and then to impose a more explicit hierarchical manifestation of this structure. Minimized coupling and maximized change effect localization are the criteria for the fitness of the hierarchical structure that is chosen. The basis of the process is the analysis of a system graph called a *resource flow graph* in which the vertices are system modules and the edges are resource exchange. Choi and Scacchi use an algorithm which imposes a subsystem hierarchy upon the modules by discovering the connector vertices between biconnected components of the graph, known as *articulation points*.

While restructuring creates new versions that implement or propose change to the subject system, it does not normally involve modifications with respect to new functional or otherwise domain-related requirements. However, it may lead to better observations of the subject system that suggest changes that would improve aspects of the system. Restructuring is often used as a form of preventive maintenance to improve the physical state of the subject system with respect to some preferred standard. It may also involve adjusting the subject system to meet new environmental constraints that do not involve reassessment at functional requirement levels.

Depending on the criteria that are considered most germane, restructuring tools can, if nothing else, attempt to bring order out of an apparently chaotic software system. However, they may not be entirely without drawbacks. Corbi (1989), citing a government study and others, makes the case that while intra-module restructuring spaghetti-like code would have its benefits in the increase of order and understandability, the process will usually result in more voluminous

code leading to higher compile times. In addition, the comments (assuming that there were comments) could be rendered meaningless. Evaluations are needed for the more macroscopic restructuring tools.

3.4 Reengineering

Reengineering, also known as *redevelopment engineering*, *renovation* and *reclamation*, is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Implied in the term is the possibility of change in essential requirements, rather than a mere change in form.

Reengineering generally includes some form of reverse engineering, to achieve a more abstract description, followed by some form of forward engineering or restructuring. This may include modifications with respect to new requirements not met by the original system. For example, during the reengineering of information management systems, an organization generally reassesses how the system implements high-level business rules and makes modifications to conform to changes in the business for the future.

There is some confusion of terms, particularly between reengineering and restructuring. The IBM user group Guide (Guide International Corporation, 1989), for example, describes *application reengineering* as the process of modifying software internally (e.g., algorithms and data structures) without changing the systems functionality or capabilities as perceived by the user). This is similar to the definition of restructuring given earlier. However, two paragraphs later, the same publication indicates that during application reengineering functionality is usually added. This supports the more general definition of reengineering stated above, which adds functionality.

While reengineering involves both forward engineering and reverse engineering, it is not a supertype of the two. Reengineering uses available forward and reverse engineering

technologies; however, both forward and reverse engineering technologies are evolving rapidly, independently of their application within reengineering.

4. Reverse Engineering Throughout the Life Cycle: Objectives and Purposes

In the previous sections, the general context of reverse engineering was considered with respect to the typical life cycle phases of analysis, design, implementation, and maintenance. The intent was to provide baseline descriptions of the life cycle phases and their respective artifacts to facilitate the discussion of reverse engineering activities. In this section, the detailed objectives of reverse engineering are explored. This followed by a discussion of the primary purposes for which reverse engineering technologies are being developed: maintenance, reuse, and software quality assurance.

4.1 Objectives

The primary goal of reverse engineering a software system is to increase the overall comprehensibility of the system to facilitate maintenance as well as other activities such as reuse and overall quality assurance. There are several key objectives that are encompassed by the goal of software understanding:

- To cope with complexity. Research must develop methods to deal more effectively with the sheer volume and complexity of systems. A key to controlling these attributes is automated support. Reverse engineering methods and tools, combined with CASE environments, will provide a way to extract

relevant information so decision makers can control the process and the product in systems evolution. Figure 3 (provided by Robert Arnold) shows a model of the structure of most tools for reverse engineering, reengineering, and restructuring.

- To generate alternate views. Graphical representations have long been accepted as comprehension aids. However, creating and maintaining them continue to be a bottleneck. Reverse engineering tools facilitate the generation or regeneration of graphical representations from other forms. While many designers work from a single, primary perspective (e.g. data flow diagrams), reverse engineering tools can generate additional views from other perspectives (e.g. control flow diagrams, structure charts, and entity-relationship diagrams) to aid the review and verification process. One can also create alternate forms of nongraphical representations with reverse engineering tools to form an important part of system documentation.
- To recover lost information. The continuing evolution of large, long-lived systems leads to lost information about the system design. Modifications are frequently not reflected in documentation, particularly at a higher level than the code itself. While it is no substitute for preserving design history in the first place, reverse engineering, particularly design recovery, is a way to salvage whatever is possible from the existing systems.
- To detect side effects. Both haphazard initial design and successive modifications can lead to unintended ramifications and side effects that impede a system's performance in subtle ways. As Figure 4 shows, reverse engineering can provide observations beyond those that can be obtained with a forward engineering perspective, and it can help detect anomalies and problems before users report them as bugs.

- To synthesize higher abstractions. Reverse engineering requires methods and techniques for creating alternate views that transcend to higher abstraction levels. There is a debate in the software community as to how completely the process can be automated. Clearly, expert system technology will play a major role in achieving the full potential of generating high-level abstractions.

The realization of these objectives offers benefits for maintenance, reuse, and overall software quality assurance. Each of these activities is discussed below from a reverse engineering perspective.

4.2 Maintenance

Reverse engineering of software is tightly coupled with software maintenance because maintenance activities have provided the motivation for many of the reverse engineering tools available today. Corbi (1989) cites several studies which suggest that maintenance activities consume an overwhelming proportion of time and funds allotted to overall software engineering. Furthermore, the greater portion of that time spent in maintenance is spent trying to learn the intricacies of the system undergoing the maintenance.

Indeed, generally a system's maintainers are not its designers; therefore, they must expend many resources to examine and to learn about the system. Reverse engineering tools can facilitate this activity. ANSI/IEEE Std 729-1983 (p. 32) defines software maintenance to be "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment." In this context, reverse engineering is that part of the maintenance process that aids maintenance engineers in understanding the system so that they can make appropriate changes.

Pressman (1987) discusses the four traditional categories of maintenance: corrective, adaptive, perfective, and preventive. *Corrective* changes are made in response to the detection of errors that should have been removed prior to delivery of the system. Reverse engineering techniques do not detect the errors, nor do they remove or correct the software. Rather, they assist the programmer indirectly in locating the defective component through improved comprehension of the software.

Adaptive maintenance activities are concerned with moving the software to a new environment (e.g., a new operating system or a new hardware platform). *Perfective* maintenance involves adding new functionality and, as such, consumes the largest percentage of maintenance resources over time. Depending on its context and scope, perfective maintenance may be equivalent to reengineering and/or new development. In the case of both adaptive and perfective maintenance, reverse engineering tools are used indirectly to locate components where changes and additions need to be made and to aid in controlling the overall structure of the modified system.

Finally, *preventive* maintenance consists of those modifications made in an attempt to reduce the effort required for future, perhaps anticipated, changes. Here reverse engineering can provide insight into where and how to make appropriate changes.

Restructuring and reengineering also satisfy the global definition of software maintenance. Restructuring perhaps best fits the category called preventive maintenance, whereas reengineering implies extensive modifications in the sense of completely rewriting/reworking the software, and thus could be categorized as adaptive and/or perfective maintenance. However, each of these processes also has a place within the contexts of building new systems and evolutionary development as well as within that of maintenance.

Perhaps the greatest benefits of reverse engineering tools can be realized after the changes are implemented in any of the categories of maintenance described above. These include the

automatic regeneration of numerous graphical representations of the software to assist future maintenance activities.

4.3 Reuse

Brooks (1987) and others contend that the reuse of software components may be one of the few remaining areas of software engineering that could yield one or more orders of magnitude in increased productivity. When entire systems are reused, the cost of the software per customer can be dramatically reduced when spread over many customers. While the actual development cost is not lowered, the net profit to the developer increases and the actual cost to each customer may be decreased by several orders of magnitude. Reuse can be considered with respect to the percentage of reused components in a system as well as the type of components reused (e.g., source code, design specifications, etc.). A system comprised of 100% reused components is, in fact, probably a copy of an existing system. However, it is conceivable that the percentage of reused components in a system could approach 100% when it is composed predominantly of components from other systems. Reverse engineering tools are used to directly support reuse when the percentage of reused components is less than 100; otherwise, they are used to support the maintenance of the one software system from which the copies are made.

A significant issue in the movement toward software reusability is the large body of existing software assets that were not developed with reuse in mind. Reverse engineering can help detect candidates for reusable software components in present systems. Biggerstaff (1989) suggests that design recovery tools should be able to make necessary alterations to code segments so as to make them even more useful. Such alterations consist of extracting interleaved functions and generalizing code to make it more universally applicable.

The process of reuse has been described by Boldyreff (1989) as follows: recognition, decomposition/abstraction, classification (to populate the reuse library), selection/retrieval, specialization/adaptation, and composition/deployment. These steps can be applied to the reuse

of code, test specifications and cases, design specifications, requirements specifications, as well as more general concepts. Reverse engineering tools can play a major support role in each of these steps; however, the primary focus is on the first three steps. In particular, components which are candidates for reuse can be most easily recognized by either human or machine if they have been converted to a "standard" notation or form. For example, if manual identification of components is necessary, a complete and accurate set of graphical representations will clearly aid the reviewer. If an expert system-assisted identification is to be attempted, reverse engineering techniques, especially those in design recovery, can be used to extract patterns of control constructs, control flow and data flow to provide a basis for machine recognition of algorithms, major data structures and objects. While reverse engineering technology is not focused on the actual composition of components from reusable parts, it would be useful in completing the documentation of the newly composed system. Boldyreff points out that sufficiently advanced design recovery mechanisms will allow software to be recycled continuously. That is, a new system composed from some reused parts can itself be converted to reusable parts. The net result should be an increase in the percentage of reused parts found in new systems, which, in turn, should lead to an increase in overall reliability.

Although there is much promise for software reuse, the success of its application rests primarily on reverse engineering technologies which can extract generic software concepts represented by procedures, functions, tasks, packages, objects, etc. Reverse engineering methods must provide the foundation for the reuse process.

4.4 Software Quality Assurance

As described above in the case of maintenance and reuse, reverse engineering is primarily considered a process which begins with the source code of an implemented system and works backward (upward), creating documents from the information which can be extracted from the

source code. However, reverse engineering can be considered as a set of complimentary activities within each phase of the life cycle, and as such can be applied early on in the life cycle as a basis for improved software quality assurance (SQA). Pressman (1987) identifies formal technical reviews, collection and analysis of metrics, and testing as key activities for a successful SQA program. These activities, which span the entire life cycle, can be greatly facilitated by reverse engineering. Formal technical reviews may occur during each of requirements analysis, design, implementation, and especially between these phases. These reviews provide a basis for validating the system with respect to its functional requirements and a basis for verifying that such a system can actually be built to perform reliably. Metrics, when collected, can become an integral part of the artifacts under review and provide valuable insight for the decision makers. Validation criteria and test plans typically are written prior to implementation, while the execution and analysis of test cases are performed during unit coding and integration. Table I provides a summary of reverse engineering activities which support SQA. Below, these activities are discussed within each generic phase of the life cycle in conjunction with the artifacts that are candidates for reverse engineering. Maintenance is, in fact, simply an iteration of requirements analysis, design, and implementation.

4.4.1 Requirements Analysis Artifacts

Notations for requirements specification differ as widely with respect to formality as design notations. There are notations which show some characteristic of the system at large as well as the procedure of one operation. The notations can be textual, graphical, or mathematical.

Requirements specification analysis can be formal or informal, depending on the scope and precision of the notation. Since requirements are often specified in natural language, analysis of such specifications can only be at best informal and imprecise. The formal languages created specifically for requirements specification are, of course, be more amenable to formal analysis.

Structural analysis of operational specifications would have a better chance of producing a functional description than would a linguistic analysis of a textual specification.

Formal technical reviews dictate formal, aesthetically pleasing requirements specifications. Modern CASE tools are gradually providing the required formality in requirements specifications notations. The data flow diagram is an example of a notation that was widely used initially in a manual mode. Although cumbersome, its descriptive power for analysis outweighed the disadvantages of no automated support. Early CASE tools assisted the analyst in capturing data flow diagrams in a modifiable medium, and then facilitated the eventual integration of them into specification documents. Subsequent enhancements have provided for consistency checking between components in diagrams and levels of diagrams. However, after changes are made to one diagram, they are not generally propagated throughout the system. Change propagation is a bidirectional activity of which reverse engineering is a part. Reverse engineering technologies are progressing to the point where change propagation becomes another form of redocumentation and perhaps restructuring. These capabilities would alleviate much of the incidental effort required to modify the requirements specification and would directly support formal technical reviews. Here, there is considerable overlap with advanced design recovery techniques.

The metrics that can be collected during requirements analysis by reverse engineering tools include items such as the number of interfaces, files, inputs and outputs of a system used in the *function point method* (Albrecht and Gaffney, 1983). These metrics become input for cost estimation, scheduling, and reliability models to assist in decision making.

Validation criteria should be based primarily on the functionality of the new system. Reverse engineering tools can be used to extract and cross-reference functions and families of functions which must be validated. Validation criteria can provide a basis for test plan.

4.4.2 Design Artifacts

Much of what was stated regarding requirements analysis artifacts applies to design artifacts as well. Design is the process of specifying the form of a software solution to the problem defined by a requirements document. This design specification can be in any number of notations, including natural language narratives, a more structured subset of natural language such as pseudocode or PDL, and graphical notations. These notations can have varying degrees of precision.

That information which can be extracted from a design notation depends on the precision and completeness of the design notation. It is unlikely that a strictly architectural design notation, no matter how precise, would be complete enough to infer the function of the system. To derive the function of a system requires analysis of the constituent modules of the system, deriving functions for each, and composing them into a larger system function. Formalization of procedural design notations is needed to allow for the structural analysis needed to map procedural structures to functional statements. Of course, as in linguistic code analysis, a more informal analysis can be performed on design specifications that are represented by less formal and less complete notations. The results of analysis on design products, like analysis on source and executable code, should ideally give insights as to the functions of the system.

From the perspective of formal technical reviews, because design specifications are in a form/notation that is closer to the actual source code that will be eventually produced, more precise views can be extracted. For example, at the architectural design level, if Ada PDL is used to describe component interfaces (procedures, functions, tasks, and packages), object diagrams can be generated quickly to provide reviewers with an accurate alternative view of the system which depicts dependencies and visibility among the components. Structure charts, which provide the overall call structure (and, optionally, parameter flow between modules), can be used to generate an equivalent PDL, or, if the skeletal PDL bodies were developed instead of structure charts, then the structure charts can be generated directly from the PDL. The object diagrams,

structure charts and PDL assist reviewers in validating that the system meets its design specifications and in verifying that a reliable system can actually be built. By combining forward and reverse engineering technologies, designers can modify any one of these artifacts and have the changes automatically reflected in the others.

As with requirements artifacts, design artifacts offers an opportunity to collect numerous metrics; however, they are usually more precise and meaningful during the design phase. For example, the data for cost models generally becomes more accurate as implementation approaches. Other design metrics such as McCabe's (McCabe and Butler, 1989), provide a basis for future integration testing by ensuring that a basis set of paths containing procedure or function calls have been exercised.

4.4.3 Implementation Artifacts

The most obvious artifact from this phase is the source code itself. To dissect source code, the tools surveyed in this chapter make use of several compiler-related techniques, including analysis of code structure and data flow. This analysis produces such structures as parse trees, symbol tables, and flow graphs. The resultant structures are often matched against patterns which are, in turn, mapped to alternative representations. These alternative representations can be of similar complexity such as cross-reference tables or algorithmic graphical representations. Also, the alternative representations can be of a more abstract nature such as a functional description of the code fragment, structure charts, data flow diagrams, etc. A less common and less exact form of code analysis is the study of identifiers and comments to extract code function from names and prose (Biggerstaff, 1989; Schwanke et al., 1989).

Another artifact from this level is the executable code. A "black-box" (Samuelson, 1990, p. 91) analysis of executable code involves feeding inputs to the code and noting the outputs. From this process the analyst derives an understanding of the program's functions.

Structural analysis of source code can produce understanding of the code in and of itself. However, if humans do not ascribe meaning to code structures, structural analysis cannot determine the function of the code, neither in isolation nor within a larger organizational framework. Linguistic analysis perhaps can shed more light on the actual semantics of the software, depending of course on the names of the identifiers and the content of comments.

Formal technical reviews of code can benefit, perhaps even more that earlier reviews, from reverse engineering tools because of the inherent complex nature of source code. The alternate views, especially graphical abstractions, can provide reviewers with the leverage required to comprehend the system efficiently. Metrics collected can be as detailed as the number of conditions or decisions in an algorithm as is the case with McCabe's cyclomatic complexity metric (McCabe, 1976). In particular, these metrics can be used with appropriate heuristics to guide the implementation, predict the number of errors, and identify sets of paths that must be tested to attain a specified level of coverage.

The use of reverse engineering tools during implementation in a forward sense is similar to its application during maintenance in that the input in both cases is source code. It is a natural progression that any tools that can be used to aid in the comprehension of software during maintenance should also be applied to software at the earliest feasible point in the life cycle.

5. Economic and Legal Issues

A discussion of reverse engineering would be incomplete if it focused entirely on technical issues. As with all new or evolving technologies, there are practical concerns created by reverse engineering which must be addressed. Most likely the chief concern among software

engineering managers is how reverse engineering can positively affect the productivity of their staffs and, as a result, the economic strength of their organizations. Furthermore, of great interest to managers is the degree to which the technology--in which heavy investments in money and prestige are to be made--has universal applicability.

In addition, there are certain legal and ethical questions to resolve when the intent is to reverse engineer a competitor's products. These issues can have their own economic ramifications as well, such as court costs, attorneys' fees, fines, damage-control public relations campaigns, etc. The concerns here involve the ownership of software, intellectual property rights, and the freedom of information distribution.

This section will only attempt to illuminate the practical issues raised by the introduction of reverse engineering into the software community. It will be the responsibility of software engineering managers and the courts to resolve these issues and to define the precise role for reverse engineering in the software engineering process.

5.1 Economic Issues

The fundamental benefit of reverse engineering technology is the increased understanding of a subject system afforded to the user of the technology. From this primary benefit will come other gains throughout the life cycle, primarily productivity gains. As reverse engineering is used in a variety of software engineering processes, these gains will ultimately translate into competitive advantages and financial rewards.

5.1.1 Maintenance Issues

This chapter has discussed at length the traditional relationship between reverse engineering and maintenance and has tried to broaden the scope of reverse engineering to other areas of the life cycle. This is not, however, to belittle the potential that reverse engineering has

to abate some of the problems encountered in maintenance. Maintenance has indeed proved to be a severe bottleneck for software engineering.

Corbi (1989) cites numerous studies which allude to the burden that the maintenance process has traditionally been. For example, Belady and Lehman suggest that all programs of non-trivial size would experience some alteration during their active lifetimes. In addition, a study by Carroll states that eighty percent (80%) of programmer time in industry is spent in some sort of maintenance work. Furthermore, a study by Parikh referred to an MIT study which concluded that for every dollar spent on a new development venture nine dollars ($9) is spent on the subsequent maintenance. From these figures alone, one may conclude that maintenance heretofore has depleted an exorbitant amount of the time and effort which could have otherwise been used for original development.

The chief problem of maintenance is that of understanding the system experiencing the change. Again, Corbi (1989) cites Parikh and Zvegintzov who state that at least half of the maintenance process is understanding the system itself. It stands to reason that if reverse engineering tools and techniques can help the maintenance engineer in understanding old, seemingly cryptic, pieces of software in a more timely and less error-prone fashion, then the maintenance burden will therefore be somewhat alleviated. This understanding must encompass not only an understanding of the original system as a static entity but as a dynamic mechanism in which proposed changes will invariably have consequences.

5.1.2 New Development Issues

While reverse engineering is largely thought of as a facilitator of maintenance, it can support new development in some important ways such as reuse and software quality assurance which were described in Section 4. In particular, these allow a reallocation of total software engineering time and effort away from maintenance. Reverse engineering can expedite new development by examining how similar systems are constructed. By examining the internals of

another system, development team designers can make more informed design decisions for their situation, particularly concerning which strategies to adapt from the older system and which to ignore. This endeavor, however, has certain legal complications, as the next section explains.

5.2 Legal Issues

Samuelson (1990) discusses many of the legal issues in the reverse engineering of the code of a supplier or competitor. The examples from legal cases which she cites indicated that the software that was being reverse engineered was in the form of object or executable code rather than source code. Samuelson makes the point that the questions of legality lie in the more intrusive white-box form of reverse engineering which attempts to analyze the code of a system. This is contrasted with the black-box form which attempts to infer function and design from analysis of output resulting from tests using varied inputs. Generally, there are few who question the legality of the latter process.

Samuelson illustrates the two opposing arguments concerning the legality of white-box reverse engineering. The major point of contention on which both arguments rest is the interpretation of the copyright laws concerning intellectual property. On one hand, the *strict-constructionist* group would argue that, according to copyright law under which proprietary software is usually protected, the sole right of copy or copy entitlement belongs to the owner of the copyright; copying is allowed only if the purposes involved are in keeping with the *fair-use doctrine*. This group argues that to copy another's software for reverse engineering violates that doctrine. This is because of the ultimate potential loss of revenue to the copyright owner should a rival product result from the reverse engineering. From their perspective, stringent enforcement of copyright law offers the best protection for the trade secrets which serve as competitive ammunition.

On the other hand, the *pragmatist* group would argue, as does Samuelson, that reverse engineering of a competitor's software should not be banned. Samuelson bases this argument

on two principles: (1) the disclosure requirement of copyright law and (2) the separation of the act of copying from the act of reverse engineering and from the act of using extracted information. First, this group would argue that to copy software for study is not illegal in that to obtain a copyright in the first place requires the applicant to reveal the contents of his or her work. Also, this group would argue that it is not the reverse engineering of the code which would violate trade-secret regulations, but the use of those discovered secrets for illicit gain.

Samuelson illustrates that, although sometimes contradictory, the weight of United States case history is in favor of the pragmatist view to allow reverse engineering of a competitor's product, defining copyright violation only when a resulting rival program is too similar to the original.

Sibor (1990) disagreed with Samuelson's pragmatist position. In fact, Sibor cited many of the same cases that Samuelson cited but drew different conclusions (which supports Samuelson's position that attorneys disagree on these issues). Throughout his discussion, Sibor seemed to equate reverse engineering with the terms *decompilaton* and *disassembly* of code, which again emphasizes the notion that legal questions are focused primarily on object or executable code rather than source code.

This is clearly an area where each case decided over the next decade will contribute to a growing debate. The only certainty is that reverse engineering one's own software is proper and lawful. The debate centers on propriety when someone else's software is involved, with or without permission.

6. Survey of Current Research

This section describes projects that are representative of the current research efforts underway in the area of reverse engineering. The inclusion of a project should not be considered an endorsement of its importance, nor should the omission of a project be considered a negative statement. Although they have been classified as either redocumentation or design recovery projects, some elements of each project may be fall into both categories of reverse engineering.

6.1 Redocumentation

Many of the research and commercial tools surveyed in this chapter can be classified as redocumentation tools. The major intent of these tools is not to generate explanations of function for the software systems that they study, but instead to provide aids for the user in order to expedite his own understanding of the systems. These aids include code formatting schemes, graphical representations, and code browsers. All tools acquire an implementation-level and structure-level understanding of the code; they do not, however, acquire a function-level understanding. Table II provides a summary of the basic characteristics of the redocumentation tools presented in this section.

6.1.1 Book-Maker

Oman and Cook (1990) have proposed a new model for code formatting that holds much promise to code readers. This model, called the *book paradigm*, is based largely on the use of special typographical conventions for particular types of code elements and on the integration of cross-reference information to connect related sections of a program. This scheme of code formatting is realized in two tools collectively called *Book-Maker*. Oman and Cook sought to create a format that would facilitate the examination both of large code structures and of atomic elements. They wished physically to highlight the code pieces called *beacons* and *chunks* so as to distinguish them all the more. They also desired a format that would not interfere with the normal course of code maintenance; they desired compatibility with compilers and other code manipulation tools.

The purpose of such a tool, as the name suggests, is to present source code information in a book-like format. A summary of the major modules and submodules of the program will appear (with page numbers) in a table of contents. The modules themselves will be presented in full in chapters with sections within the chapters presenting subcomponents of the modules. An index points out the appearances of the program entities, variables as well as modules, within the program. Other similarities with books exist. Figure 5 illustrates some of the features of this form of code formatting.

Oman and Cook conducted two studies comparing the effectiveness of code comprehension using plainly formatted source code and code formatted according to their scheme. Both experiments suggested a significant improvement in code understanding with code with all of the typographical trappings of the book paradigm.

6.1.2 ARCH / MAINTAINER'S ASSISTANT

Schwanke et al. (1989) describe the work on the *ARCH* project, part of the larger *MAINTAINER'S ASSISTANT* project. One of the tools in ARCH is an automatic classification system which uses identifier names as discriminants. By this method, a tree of related concepts analogous to the system structure is created.

ARCH uses a statistically-based method for extracting structure from both existing systems and architecture specifications. The method is based on the analysis of the system identifiers, particularly their frequency of occurrence and determining where in the code the identifiers are most prominent.

The name analysis, similar to keyword analysis in document retrieval systems, generates functional concepts around which the system modules are clustered, resulting in an hierarchical subsystem tree. The "concepts" are statistical categories defined by the names most prominently used in the members of each category.

It may appear that the tool might be engaging in program understanding in its own right. However, this tool does not attempt to impose a predefined meaning upon the categories that are defined. Instead, it merely defines the categories based on identifier usage and leaves it to the analyst to apply meaning to the categories. This assessment is the judgment of Schwanke et al., and it could be reasonably argued that this is indeed a design recovery tool and not simply a tool of redocumentation.

6.1.3 GRASP/Ada

GRASP/Ada (Cross, 1990; Cross et al., 1990) is a reverse engineering toolset under development at Auburn University and funded, in part, by Marshall Space Flight Center, NASA. An overview of the three phase project, which is now in Phase 3, is shown in Figure 6. The principal components of the toolset are a control structure diagram generator, object diagram generator, and user interface. A primary contribution of the GRASP/Ada project has been the development of the control structure diagram (CSD) for Ada, which is a notation intended specifically for the graphical representation of detailed designs as well as actual source code. The purpose of the CSD is to reduce the time required to comprehend software by clearly depicting the control constructs and control flow at all relevant levels of abstraction, whether at the design level or within the source code itself. The CSD is a natural extension to existing architectural graphical representations such as data flow diagrams, structure charts, and object diagrams.

A major objective in the philosophy that guided the development of the CSD was that the graphical constructs supplement the code and/or PDL without disrupting their familiar appearance. That is, the CSD should appear to be a natural extension to the Ada constructs and, similarly, the Ada source code should appear to be a natural extension of the diagram. This has resulted in a concise, compact graphical notation which attempts to combine the best features of

previous diagrams with those of well-established PDLs. The CSD generator automates the process of producing the CSD from Ada source code.

Figure 7 contains an Ada task body CONTROLLER adapted from Barnes (1984), which loops through a priority list attempting to accept selectively a REQUEST with priority P. Upon acceptance, some action is taken, followed by an exit from the priority list loop to restart the loop with the first priority. In typical Ada task fashion, the priority list loop is contained in an outer infinite loop. This short example contains two threads of control: the rendezvous, which enters and exits at the ***accept*** statement, and the thread within the task body. In addition, the priority list loop contains two exits: the normal exit at the beginning of the loop when the priority list has been exhausted, and an explicit exit invoked within the ***select*** statement. While the concurrency and multiple exits are useful in modeling the solution, they do increase the effort required of the reader to comprehend the code.

Figure 8 shows the corresponding CSD generated by the graphical prettyprinter. In this example, the intuitive graphical constructs of the CSD clearly depict the point of rendezvous, the two nested loops, the ***select*** statement guarding the ***accept*** statement for the task, the unconditional exit from the inner loop, and the overall control flow of the task. When reading the code without the diagram, as shown in Figure 7, the control constructs and control paths are much less visible although the same structural and control information is available. As additional levels of nesting and increased physical separation of sequential components occur in code, the visibility of control constructs and control paths becomes increasingly obscure, and the effort required of the reader dramatically increases in the absence of the CSD. Since the source code will be read many times by its author(s) during the course of initial development and by many others during maintenance, it seems intuitive that the reader should benefit from the use of an appropriate graphical notation. The Cross et al. cited two recent studies that support the theory that graphical representations may indeed improve the comprehensibility (Scanlan, 1989) and productivity (Aoyama et al., 1989) of software.

Currently, the CSD generator of the GRASP/Ada prototype is being extended to include editing capabilities that will provide for incremental regeneration of the CSD as the Ada/PDL or source code is modified. The CSD editor will allow the user to collapse the diagram around control constructs and generate an intermediate level architectural diagram which indicates control structure among subprograms and tasks. The object diagram generation component, currently in early prototype, will extract Booch-like diagrams (Booch, 1991) from the Ada/PDL or source code. The user interface will support navigation among the CSDs and object diagrams to provide an intuitive multi-level and multi-view set of graphical representations of any software system written in Ada.

6.1.4 Objective-C Browser

The *Objective-C Browser* by Stepstone, as presented by Novobilski (1990), is a tool which allows the user to view Objective-C code from a number of perspectives. Among these views is the class inheritance hierarchy which allows the user to pose queries concerning method resolution and method binding. Other views that are presented include usage statistics for entities defined by the code. The user can also determine the repercussions of change by the use of generated cross-reference information.

6.1.5 Vifor

Rajlich (1990) reports on a tool produced by Software Tools and Technologies known as *Vifor*. A visualization tool for FORTRAN programs, Vifor pictorially presents FORTRAN routines, their calling hierarchies, and their references to common data areas. To prevent the user from being overwhelmed with information, Vifor uses diagram layering and two-column graphs to ease the burden of reading the diagrams.

6.1.6 PSL/PSA Reverse Engineering

Chikofsky (1983) describes the pioneering work on the integration of reverse engineering with CASE repository concepts. Information systems research at the University of Michigan in the late 1970's led to the development of PSL/PSA, a mainframe system that was the grandfather of the present generation of CASE tools. In the early 1980's this CASE tool was used as a maintenance vehicle to record and analyze the software of the tool itself. This self-validation for maintenance demonstrated the viability of CASE as a natural maintenance environment, utilizing the same toolsets developed for forward engineering.

The same PSL/PSA environment was used by IBM as a repository for manual analysis in a reverse engineering project to reclaim lost documentation for a system, as reported by Johnson (1983). This work demonstrated the concepts that would later be used in applying the IBM AD/Cycle Repository to the maintenance phase of the life cycle.

6.1.7 Seela

Seela, a reverse engineering tool which uses human intervention to supply concise functional descriptions to code blocks, is a product of Tuval Software Industries (Harband, 1990). By using human intervention to supply functional descriptions, an abstraction is made from implementation language to program design language. The user may manipulate the named blocks by cutting and pasting, editing, and traversing to other blocks.

6.1.8 BattleMap and ACT

From McCabe & Associates comes *BattleMap* and *ACT* (McCabe, 1990). These tools make use of the metric *cyclomatic complexity* (McCabe, 1976) to classify code in a graphical manner. ACT calculates the cyclomatic complexity of each software module and with that knowledge performs other functions such as the outlining of test paths. BattleMap is essentially a graphical tool layered above ACT. BattleMap displays a hierarchical diagram where each

module icon is color-coded by cyclomatic and *essential* complexity (McCabe and Butler, 1989). The diagrams can be made to hide certain branches of the system tree.

6.1.9 Expert Dataflow and Static Analysis

Expert Dataflow and Static Analysis, an Ada-based tool (Vanek and Davis, 1990) from Array Systems Computing, analyzes Ada source code and provides code-viewing and code-traversing capability. A user may choose to see only a certain perspective of the code such as calling structure. In addition, he may step through the control flow of the program, skipping procedure calls if he so desires. He may also refer quickly to the definition of a variable from a usage point.

6.1.10 Surgeon's Assistant

The Surgeon's Assistant, described by Gallagher (1990) of Loyola College, is an editing program which uses program slices to restrict the effects of modification of C programs. A slice is defined by a set of variables and encompasses all code in which only the defining variables are ever changed. From the tool window interface, a user will slice a program according to some set of variables and edit that slice while not affecting the complementary part of the program. Once all changes have been made to all slices, the user will merge the slices into a new program. The intent of this tool is to be rid of regression testing.

6.1.11 Dependency Analysis Tool Set

The Dependency Analysis Tool Set, described by Wilde (1990) of the University of West Florida, displays various forms of dependency in C programs. In addition, facilities exist to create repositories for C programs, to query concerning dependencies, and to share query output with other analysis tools.

6.1.12 REDO - Esprit Project 2487

The *REDO* project (Esprit, 1990), which is project number 2487 of the Esprit II program being conducted by the Commission of the European Communities, is intended to produce tools to aid in maintenance, reuse, and validation of large software systems. Goals of the project include the development of an intermediate language which will be an abstraction from source languages, an environment database, a multiple-process architecture, and a user interface designed with the needs of the software engineer being paramount.

Two of the components of this proposed toolkit are (1) a restructuring component whose scope will include data structures, local variables, and control structures, and (2) a redocumentation component which will reproduce documentation from the source code and the intermediate language representation (Khabaza, 1989). This documentation will be maintained in the environment database.

Thus far, the project's reverse engineering focus has led to two separate approaches. The first has concentrated on an SQL interface to a database containing the information required for the reverse engineering process, and the second relies on an object-oriented repository with associated schema descriptions.

6.1.13 MicroScope

MicroScope, reported by Ambras and Day (1988), is a knowledge-based programming environment which is being developed to analyze and monitor programs written in Common LISP and Common Objects. Ambras and Day suggest that the technology provided in their prototype should also be useful to programmers using conventional languages. MicroScope provides incremental support for the user/programmer with various views at different "magnification" levels. A graphical representation is provided which allows the user to navigate through the program. Impact analysis is planned for a future version of MicroScope.

Extensive annotation capabilities are provided with specific annotations for source code, documentation, variable and constant declarations, design specifications, development tasks, known bugs, monitoring status, profile data, and revision history.

MicroScope provides for dynamic analysis in the spirit of many modern debuggers. However, it makes good use of graphical displays and program animation. The underlying knowledge base of MicroScope is designed around the notion of frames and rules, where frames represent data and rules represent methods for making inferences.

6.1.14 Adagen

Adagen, reported by Rozenblat and Fischer (1989), is a CASE toolset specifically designed to support the development of large software systems written in Ada. It includes a reverse engineering component which inputs existing Ada source code and produces a hierarchical set of Buhr diagrams which shows the overall declaration topology of the system, as well as procedure, function and task entry calls. The toolset also includes the capability to generate a complete system dependencies chart which provides the user with an overview of the system structure to aid in navigation through units. This hierarchical structure view of the system may suggest restructuring alternatives to the user.

6.1.15 Program Understanding Support Environment - PUNS

Cleveland (1989) of IBM's T. J. Watson Research Center has developed a prototype program understanding support environment (PUNS) which provides many of the capabilities required for reverse engineering. PUNS is unique in that it supports maintenance activities by analyzing and presenting multiple views of large programs written in IBM System/370 assembler language. PUNS has two principal components: a repository component and a user interface component. The repository component analyzes the target system and populates an entity-attribute-relationship database with most of the low-level relationships within the program. The repository component has three subcomponents: the schema for the repository, routines that load

the repository, and routines that extract information from the repository based on queries from the user. The repository component is designed to run on a high-performance host.

The user interface component, which runs on a workstation, provides an object-oriented, window-based presentation of the multiple views of the subject program. PUNS facilitates program understanding by providing the maintenance programmer with the capability to navigate among objects of interest (e.g., modules or variables) based on a multi-dimensional relationships such as control flow and data flow.

From a reverse engineering perspective, PUNS falls primarily into the category of redocumentation. In the present prototype, all of the program views presented are based on static analysis of the assembler source code. The relationships in the schema are predetermined from experiences of maintenance programmers. At present the information in the repository is not dynamically modifiable by the user during a viewing session. Although not specifically stated, apparently when a modification to the assembler source code is determined to be necessary, the programmer makes the change to the appropriate file using a traditional text editor. After the code is re-tested, the program is then re-analyzed by the repository component of PUNS. Cleveland states that dynamic updating of the information in the repository is a desirable future capability of PUNS, and she suggests that the user should have the capability to enter additional information that he or she has determined, but which PUNS was unable to extract. However, if such a capability were to be added, some means of characterizing the credibility of the user-determined information would also be necessary to assist other users of PUNS in making decisions regarding the use of the information.

6.1.16 Other Support Environments

The environments discussed in Aoyama et al. (1989) provide some support for automatic regeneration of graphical representations. These environments are based on algorithmic graphical representations widely used in Japan. The environments are primarily design language editors

where the pseudocode is supplemented with graphical notations. From these procedural design languages, the environments generate source code in a number of languages. Some of the environments have the capability to generate the algorithmic graphical notations from source code for the purpose of migrating existing software into the environment for future maintenance.

6.2 Design Recovery

Much of the research presently conducted in reverse engineering is in the area of design recovery or, as more commonly called in the work surveyed below, *program understanding*. Design recovery promises more potential benefits than does redocumentation since it can be considered the first step in a practical reengineering environment. However, its attempt to achieve a more complete understanding of software means that the tools resulting from many of the current research efforts may not be available to practicing software engineers in the near future.

A common theme throughout the design recovery work is the use of pattern matching. From Rich's and Wills' *clichés* (1990) to Biggerstaff's *conceptual abstractions* (1989), pattern matching seems to be the common method of mapping lower-level constructs to higher-level concepts. Other common themes and differences will be highlighted in the survey below. Table III provides a summary of the characteristics of each of the design recovery tools surveyed here.

6.2.1 Desire

The Desire project at MCC (Biggerstaff, 1989) is attempting to automate much of the human process by using both formal, structural information and informal, associative information. Biggerstaff explains the process of design recovery from a human standpoint. He begins by emphasizing the initial search for key code sections in order to establish a beginning hypothesis of function. Once the initial assumption is made, the code analyst seeks out other programming formations typical to the hypothesized application area. The discovery of such formations

confirms the initial hypothesis. Any new information which may be gained as the result, for example, of anomalies will be incorporated into the analyst's experience for reuse in later analyses.

The core of the system, where the experience and expectations of the analyst are encoded, is Desire's *domain model.* The fundamental unit of the domain model is the *conceptual abstraction*, which is simply the combination of formal and informal information known about a particular application concept. The structural information is found in code and data structure patterns typical to implementations of that concept.

The informal information contained in a conceptual abstraction is the semantic information contained in identifiers, comments, etc. This information connects with the informal information in other related conceptual abstractions to simulate a network of expert knowledge in a particular programming area. It is this codification and use of this type of information that makes Desire rare among design recovery tools.

The conceptual abstractions, also called *idioms*, are in reality active objects written in the Common LISP Object System. Once chosen from the domain model and activated by the analyst, an idiom uses its information to seek out patterns of coding which imply an implementation of the idiom. Once it has bound itself to the code pattern, the idiom begins to match the substructures and the name scheme of the code to the structural expectations and associative relationships encoded in the idiom. All binding is subject to the consent of the analyst.

The information and experience gained by analysis of the subject system will be incorporated into the domain model not only for more effective analysis but also for reuse in constructing similar modules.

Biggerstaff reported that the Desire system was in a preliminary implementation phase. The domain model had yet to be integrated into the total system, and analysis was restricted to source code. The results of analysis were to be displayed using a hypertext system.

6.2.2 *Program Analysis Tool/Software Re-engineering Environment*

Another design recovery tool is the *Program Analysis Tool (PAT)* reported by Harandi and Ning (1990). This particular tool is part of the *Software Re-engineering Environment (SRE)* as reported by Kozaczynski and Ning (1989). The SRE, being developed at Andersen Consulting, is involved in many other types of program study in addition to design recovery.

PAT enables understanding through the first three levels of understanding coined by Harandi and Ning and described in Section 3. Kozaczynski and Ning doubt the feasibility of elevating to the highest level without significant human intervention.

Harandi and Ning then describe the basic operation of PAT. The program under analysis is parsed and encoded into *events* by the *Program Parser*, and then are placed in an *Event Base*. An event is a description of a programming notion plus the particulars of its realization, including its physical location and place in the overall control structure. The notions represented by events can range in complexity from simple statements to sizable collections of statements which perform a particular task. The events first created by the Program Parser, as one might expect, are closely parallel in complexity to the atomic statements of the program. Figure 9 illustrates the basic format of an event.

The events just created are program facts which are used to fire rules of the knowledge base. The inference engine which drives this firing is called the *Understander*, and the rules are known as *plans*. The knowledge base itself is called the *Plan Base*. The plan is essentially a combination of smaller event patterns, constituting a larger event pattern. The combination of event patterns in the plan specify the appropriate source and execution ordering of the constituent event patterns. Figure 10 portrays a typical plan template. The plan is fired only if there is a match between an event fact and a particular event pattern known as a *key event*. Once the key event is matched, the other event patterns of the plan must be matched and the event pattern orderings satisfied. Only then is the plan fired, meaning that the larger event represented by the plan is constituted. The Understander will place this new higher-level event in the Event Base

hopefully to match an even more abstract plan key event. This process of inference and abstraction continues until no more matches occur.

Close mismatches can count as matches because the plans also contain information about incorrect implementations of the plans. Text templates within the plans allow a relatively easy-to-read English explanation to be produced.

PAT is a part of the larger Software Re-engineering Environment which has the following major components: (1) the *System Code Parser*, (2) the *Global Object Base (GOB)*, (3) the *System Analysis and Abstraction Unit*, and (4) the *User Interface*. The System Code Parser produces low-level representations of code and stores these representations in the GOB. The Program Parser of PAT is presumably at least part of the System Code Parser, and the events are likely at least some of the low-level representations. The Event Base and Plan Base are likely part of the GOB. The System Analysis and Abstraction Unit is a coordinated set of analysis tools which builds on the information contained in the GOB. Control flow analysis, change analysis, and generation of graphical representations are among the forms of analysis that this set of tools conducts. PAT itself is a part of this set.

Kozaczynski and Ning discuss at some length the nature of the GOB. To represent the knowledge contained in the GOB, an object-oriented representation scheme is employed. The knowledge classes of the GOB include (1) program representation objects; (2) resultant objects from analysis; (3) run-time analysis artifacts; (4) user-supplied information; (5) plans, rules, etc. The plans and events of PAT are also typed according to this object-oriented scheme; this fact restricts the type of unification that the Understander component of PAT can perform.

At the time of their writing, Harandi and Ning had begun to build up the Plan and Event Bases to increase the applicability of PAT. As for the larger SRE, Kozaczynski and Ning have reported that a COBOL parser with reading and printing capabilities has been written. Whether or not this bidirectional parser could be the first step toward an actual reengineering (as defined in this chapter) capability remains unclear. Other portions of the prototype have included cross-

reference browsing and dependency browsing. An assembly language analysis tool has been prototyped as well.

6.2.3 Recognizer/Programmer's Apprentice

The Recognizer, part of the larger Programmer's Apprentice project at MIT, is a program understanding tool is based on the idea of *cliché recognition* (Rich and Wills, 1990). The notion of a cliché provides a means of capturing and/or implementing a programming concept (Letovsky and Soloway, 1986). The Recognizer uses a graph parsing scheme for cliché recognition and abstraction.

The program under study is first translated into a flow graph-like design representation called the *Plan Calculus* (see Figure 11). In addition, the cliché patterns, called *plans*, against which the actual clichés of the program will be matched are also encoded in the Plan Calculus. This representation uses boxes for functional units and arcs for flows of control and data. In addition, preconditions and postconditions on the functional units are incorporated into the plan representation. One advantage of this representation is that the components of clichés which might be scattered in the code itself can be brought together in a unified structure.

To accomplish cliché recognition, the Plan Calculus program flow graph is translated to a more generic flow graph. The translation is straightforward except for control flow and functional preconditions and postconditions, which become "attributes" of the flow graph. The program flow graph is submitted to a graph parser whose reduction rules, called *overlays*, are contained in a cliché library. The overlays, much like the reduction rules in the more familiar text grammars, consist of a right-hand graph which reduces into a more abstract left-hand graph. The graphs of the overlay are the plans which have also been translated from the Plan Calculus.

The ultimate result of the parse is a parse-tree design structure. This design tree can be used to produce an English synopsis of the program function and also as input to other tools of the Programmer's Apprentice.

Rich and Wills in their report explain that, through the uniformity of the Plan Calculus, the variety of the reduction rules in the cliché library, and the flexibility of the parser, the Recognizer deals with many problems inherent in program recognition including the non-contiguity of clichés, varying implementation and design styles, code which defies classification, and interleaving of clichés. For example, the graph parser is able to ignore non-classifiable code by having more flexible rules of input, ignoring non-classifiable trailing input when a parse is completed.

Currently, the Recognizer can only analyze small Common LISP programs. The state of the cliché library is impressive in that hundreds of plans and reduction rules have been encoded. Rich and Wills also state the need to move from total dependence on flow and structure characteristics for pattern recognition to the more informal information found in identifiers and comments.

6.2.4 Function Abstraction

A design recovery method which uses a slightly different technique for program understanding, called *function abstraction*, is reported by Hausler and colleagues (1990). This technique relies less on pattern matching and more on algebraic manipulation than do many design recovery tools in the literature. However, this technique had not been prototyped at the time of their report.

The abstraction algorithm proposed by Hausler and colleagues works on programs that follow structured programming principles or have been restructured so as to comply with those principles. In fact, it is the very fact that such programs are structured that makes them amenable to the algebraic manipulation of the algorithm.

A subject program is decomposed into atomic components called *small primes*. These small primes are statement sequences, selections, and iterations; they are the building blocks from which all structured programs are constructed. The function of each small prime is derived

largely from functional analysis, and the functions of larger primes are derived from those of their elemental smaller primes. The ultimate goal, naturally, is for the entire program function to be discovered.

The function of a statement sequence is derived from the composition (in the sense of function theory) of the functions of the statements of the sequence. It amounts to a statement of net effect on the sequence input. The function is derived using *trace tables*, which step through each rule of the sequence and incrementally derive the function of change for each variable in the sequence in terms of the initial variable values. These functions of change are written as assignment statements to each variable; the total sequence function is written as a concurrent assignment.

If some sequence rules are previously derived conditional functions, slight alterations must be made to the function abstraction process. Otherwise, it is conceivable for the number of possibilities in the final function of the sequence to grow wildly. After the elimination of impossible conditionals, others in some instances can be further compacted into cohesive, more expressive formulas. The final sequence function can be expressed in part in terms of these formulas.

The function of a selection prime is usually little more than a reexpression of the prime in algebraic language. Sometimes, however, it is possible through pattern matching to transform the selection prime into a compact formula.

The extraction of the function of an iteration begins with partitioning the data space of the loop so that the change function of each variable in the loop's scope may be derived in turn. These loop partitions, or *slices*, are then matched against loop patterns associated with function abstractions. The function abstractions of the slices are then composed into a comprehensive loop function. Figure 12 depicts the form of a loop-slice template.

Hausler and colleagues concur that the final function abstractions tend to be somewhat cryptic and could stand further analysis to translate them into clearer descriptions.

6.2.5 *PROUST*

Johnson (1990) describes a technique of program understanding called *intention-based analysis* and how that technique is employed in a tool known as *PROUST*. *PROUST* is a Pascal debugging tool geared for inexperienced programmers, providing explanations for program bugs in terms of the intentions of the programmers and with regard to common mistakes made by those programmers. *PROUST* thus serves as an educational tool for beginners.

PROUST views program understanding as the understanding of the intentions of the programmer. These intentions are coalesced in succinctly stated program *goals* which are statements of function for programs and program fragments. Examples of goal statements are *Sentinel-Controlled Input Sequence*, *Input Validation*, and *Average*. These goals can be realized by one of a number of *plans*, or code templates which perform the function stated by the goal. Each plan can itself contain subgoals which must be attained in order for the plan to be fulfilled.

The process of program understanding and bug discovery is termed *goal decomposition*, in which the original goals of the problem statement and subsequent subgoals are matched with portions of the code via the plans. Several goal decompositions are usually possible. The goal decomposition process results in a tree of such decompositions. Each node in the tree represents a state in the *interpretation space* which reflects, at any given point, what goals have been (tentatively) realized and which have yet to be so. The best decomposition is chosen based on the closeness of the matches between the plans and the code and on how well the mismatches can be reconciled using *plan-difference rules*.

The plan-difference rules deal with errors in program design and the interleaving of plans within a particular code segment, which produces a composite goal. When a mismatch is identified as a bug by a plan-difference rule, a description of the bug is generated and maintained for later output. These bug descriptions attempt to explain why the bug occurred: i.e. for example, whether the programmer attempted to implement an inappropriate goal, whether he

attempted to implement an appropriate goal incorrectly, or whether the programmer has a basic misunderstanding of some programming construct.

Johnson presents the results of some empirical evaluations of PROUST in its current state and its ability to find program errors and explain them in terms of the original intention. The results were quite good with some qualifications. In one test series of implementations for a problem, *PROUST* was able to develop complete interpretations for most of the implementations, find most bugs in those implementations, and explain them accurately. In a test series for another problem, *PROUST* was able to develop complete interpretations for only approximately one-half of the implementations. For the bugs found in those implementations, however, correct explanations were put forth for the most part.

Johnson states the need for an expansion of *PROUST*'s capabilities. Specifically, he states the need to expand *PROUST*'s knowledge bases and to incorporate human input with regards to programming plans and intentions. Finally, he calls for the creation of *PROUST* versions for other languages, such as LISP and Ada.

6.2.6 Talus

Ourston (1989) provides an analysis of *Talus*, based on the doctoral dissertation of W.R. Murray dated 1986. *Talus* is a program recognition and debugging prototype tool for student LISP programs. *Talus* compares the student input program with a "correct" program which serves as the program specification. The input program is converted to a standard form of LISP called *If-Normal form* in order to remove at least some of the implementation variations. This form amounts to a binary tree representation. The two programs are compared, node by node, to determine if a sufficiently precise match exists. Using heuristic assessment, *Talus* attempts to find the best match possible between the functions of the "correct" program and those of the input program and does likewise with the formal variables of both. Fault detection is through symbolic evaluation of the program and through the equivalence evaluation of corresponding

expressions of the two programs. Among the strong points of Talus, by Ourston's estimation, are the use of expression equivalence analysis, its top-down nature which allows functions to be recognized without the need for their subfunctions to be matched precisely, and its assignment of function to formal variables. Among the faults of *Talus* are the need to supply a specification (namely a "correct" program) and the orientation of *Talus* toward recursive data and control structures and other LISP constructs.

6.2.7 PROMPTER

Fukunaga (1985) discusses *PROMPTER*, a design recovery tool which extracts explanations from assembly language code. The explanations pertain to two levels of knowledge: knowledge of the internals of the implementation and knowledge of the context in which an implementation exists and operates. The basis of operation for the tool is the codification of hardware, programming, and domain-specific knowledge in an object-oriented framework using an object-oriented derivative of PROLOG. To generate the first-level explanations, *PROMPTER* employs an assembly language simulator which infers the first-level explanations, in an expert-system fashion, from hardware specific and programming related objects and their interactions. The higher-level explanations are currently encoded using more traditional production rules. Fukunaga (1985) indicated that the system was in early prototype phase with the collection, formalization, and codification of the various forms of knowledge proceeding.

6.2.8 Laura

Laura (Seviora, 1987), developed in the early 1980s, is a program analysis debugging tool for very small FORTRAN programs. *Laura* compares the input program to a supplied "correct" program which serves as the program specification. Both are translated into normalized flow graphs, and it is these graphs that are compared. Any differences discovered are considered to be program errors. *Laura*'s breadth of understanding is narrow in that it cannot reconcile the use of an algorithm in the subject program different from the one in the "correct" program.

6.2.9 *PUDSY*

PUDSY (Seviora, 1987), a debugging system developed by Lukey, uses a bottom-up approach to program analysis by breaking up the subject code into *chunks* which are then matched against *schemata* in PUDSY's knowledge base. A matched chunk is then exchanged for the predicate calculus assertions associated with the matched schema. The assertions of the lowest-level chunks are used, along with the data flow between them, to produce higher-level and ultimately program-level assertions. The path by which the final assertion is coalesced is maintained for the debugging phase to come. The final program assertion is then tested for equivalence with the supplied program specification. Any non-equivalences are assumed to be bugs. These bugs are located and solutions are proposed by backtracking through the appropriate assertion derivation. The "fixed" program is then retested.

7. Conclusion

The overview of reverse engineering and the survey of current literature indicate that researchers have made a good beginning. Of the two categories of reverse engineering tools discussed, redocumentation tools are the most mature. These tools are similar in that their output is produced deterministically from a syntactic and semantic analysis of source code. New insights may result from these alternative views, although all of the information is contained in the source code and thus could have been extracted by a careful reader. As the name suggests, the forte of redocumentation tools is the time they may save users by automatically producing accurate documentation. These tools will continue to evolve and should make their way into production environments relatively quickly.

Design recovery tools are natural extensions of redocumentation tools and thus usually have considerable overlap. These design recovery methods and tools share many common approaches to program understanding, although the nomenclature and the mechanisms might differ somewhat. For example, all of the tools and methods described above derive design by using a knowledge base in conjunction with the analysis of the formal, verifiable structural information found in the source code. This analysis is used to determine control flow, to develop relative lexical and control sequences, and to reunify discontiguous components of larger structures. This last goal is quite important to enable effective pattern matching of program components. Many of the tools' design templates which are matched against by actual program components will express a programming or application idea as subideas connected by at least relative lexical and control orderings. Often, the actual program components which implement a certain idea will be widely dispersed in the code and must be "reunified" in analysis in order for the design template to be matched and realized.

The structure of the design templates of the tools is basically the same in function, although not necessarily in form. In Harandi's and Ning's PAT, the *plan* is the design template and appears to be a highly parameterized, formal piece of text. The plans are contained in a Plan Base, which serves as the knowledge base for the tool. In Rich's and Wills' Recognizer, the design template, as well as the translated implementation, referred to as clichés, is also a *plan*. These plans, however, are composed in a graphical (in the graph-theoretical sense) language called the Plan Calculus and are part of graph-grammatical reduction rules contained in a cliché library. Both plans do undergo some encoding before being placed in their respective bases, however. In the function abstraction method proposed by Hausler and associates, the only design templates seem to be those that reduce conditionals to formulas and those that produce function abstractions for loop slices. The design templates of Biggerstaff's Desire, in addition to the structural information common to the other methods, contain the informal information which Biggerstaff lauds in the form of regular expression templates to match against identifiers.

The mechanism of abstraction is the major source of difference among the tools. In Biggerstaff's Desire, the conceptual abstractions seemingly are activated by the user and then go out to seek matches in the code on their own. In Harandi's and Ning's PAT, the Understander combined with a *Justification-based Truth Maintenance System* are inference engines which drive the recognition process. In Rich's and Wills' Recognizer, the driver of recognition is a graph parser whose reduction rules are contained in the cliché library. In Hausler's and colleagues' method, the mechanism appears to be algebraic manipulation with a smattering of pattern matching.

The future use of both redocumentation and design recovery tools depends largely on their integration into current CASE tools. Bachman (1988) suggests reverse engineering may be the key missing component in modern CASE tools. While CASE tools are steadily making their way into the mainstream of software engineering, Bachman asserts that since the majority of effort and dollars still goes into the maintenance of existing systems, a bridge is needed to migrate these existing systems into modern CASE environments for all future enhancements. Bachman takes a design recovery view of reverse engineering, and he suggests that expert systems of the future will enable analysts to extract business rules and other missing requirements and design documentation from existing artifacts. Since these are not innate to source code, automatic reverse engineering to this level from source code alone is infeasible. However, it is important to recognize that reverse engineering involves more than the just the analysis of source or object code. While the main body of research is concentrating on source code analysis, work is also proceeding on both the analysis of systems-level and network specification (control definition) with CASE environments and the analysis of database and data structure specification (data definition) to rediscover business rules (data-driven operational constraints). Chikofsky (1990) and (1991) report on these.

ACKNOWLEDGEMENTS

We acknowledge the special contributions of these individuals to the synthesis of the taxonomy section and the rationalization of conflicting terminology: Walt Scacchi of the University of Southern California, Norm Schneiderwind of the Naval Post Graduate School, Jim Fulton of Boeing Computer Services, Bob Arnold of the Software Productivity Consortium, Shawn Bohner of Contel Technology Center, Philip Hausler and Mark Pleszkoch of IBM and the University of Maryland at Baltimore County, Diane Mularz of Mitre, Paul Oman of the University of Idaho, John Munson and Norman Wilde of the University of West Florida, and the participants in directed discussions at the 1989 Conference on Software Maintenance and the 1988 and 1989 International Workshops on CASE.

This work was supported, in part, by a grant from George C. Marshall Space Flight Center, NASA/MSFC, AL 35821 (Contract Number NASA-NCC8-14).

We thank Kelly Morrison and Narayana Rekapalli for their work on the figures.

BIBLIOGRAPHY

Abrial, J.R. (1980). "The specification language Z: basic library." Oxford University Programming Research Group.

Albrecht, A.J. and Gaffney, J.E. (1983). Software function, lines of code and development effort prediction: a software science validation. IEEE Trans. Softw. Eng. SE-9(11), 639-648.

Ambras, J. and O'Day, V. (1988). MicroScope: A knowledge-based programming environment. IEEE Software 5(3), 50-58.

Aoyama, M., Miyamoto, K., Murakami, N., Nagano, H., and Oki, Y. (1989). Design specifications in Japan: Tree-structured charts. IEEE Software 6(2), 31-37.

Arden, W. and Ho, P. (1989). CASE methods of reuse and reverse engineering of Ada software. In "CASE '89: Proceedings of the Third International Workshop on Computer-Aided Software Engineering." pp. 338-339. The British Computer Society, London.

Arnold, Robert S. (1989). Software restructuring. Proc. IEEE 77(4), 607-617.

Bachman, C. (1988). A CASE for reverse engineering. Datamation 34(13), 49-56.

Barnes, J. G. P. (1984). "Programming in Ada." Second Edition, Addison-Wesley Publishing Co., Menlo Park, CA.

Basili, V. R. (1990). Viewing maintenance as reuse-oriented software development. IEEE Software 7(1), 19-25.

Biggerstaff, T.J. (1989). Design recovery for maintenance and reuse. IEEE Computer 22(7), 36-49.

Boldyreff, C. (1989). Reuse, software concepts, descriptive methods and the Practitioner project. Softw. Eng. Notes 14(2), 25-31.

Booch, G. (1991). "Object Oriented Design." Benjamin-Cummings Publishing Co., Redwood City, CA.

Brooks, F.P. (1987) No silver bullet: essence and accidents of software engineering. IEEE Computer 20(4), 10-19.

Chikofsky, E.J. (1983). Application of an information systems analysis and design tool to the maintenance effort. In "Proceedings of IFIP TC Working Conference On System Description Methodologies." pp. 503-514. North-Holland, Amsterdam.

Chikofsky, E.J. (1990). The database as a business road map. Database Programming and Design, May, 62-67.

Chikofsky, E.J. (1991). Realizing design recovery with CASE environments. Software Engineering: Tools, Techniques, Practice. November 1991. In press.

Chikofsky, E.J. and Cross, J.H. II. (1990). Reverse engineering and design recovery: A taxonomy. IEEE Software 7(1), 13-17.

Choi, S.C. and Scacchi, W. (1990). Extracting and restructuring the design of large systems. IEEE Software 7(1), 66-71.

Cleveland, L. (1989). A program understanding support environment. IBM Syst. J. 28(2), 324-344.

Corbi, T.A. (1989). Program understanding: Challenge for the 1990s. IBM Syst. J. 28(2), 294-306.

Cross, J.H., Sheppard, S.V. and Carlisle, W.H. (1990). Control Structure Diagrams for Ada. J. Pascal, Ada & Modula-2, 9(5), September/October, 26-33.

Cross, J.H. (1990). Grasp/Ada uses control structure. In (Oman, 1990). IEEE Software 7(3), 62.

DeMarco, T. (1979). "Structured Analysis and System Specification." Prentice-Hall, Inc., Englewood Cliffs, NJ.

Esprit (1990). Synopsis of information processing systems - Esprit II projects and exploratory actions. Volume 4 of 8, Directorate General XIII - Commission of the European Communities.

Faghihi, H., Colbrook, A., and Smythe, C. (1989). A CASE tool for the software reengineering of data structures. In "CASE '89: Proceedings of the Third International Workshop on Computer-Aided Software Engineering," Supplementary Volume. p. 341. The British Computer Society, London.

Gallagher, K. (1990). Surgeon's Assistant limits side effects. In (Oman, 1990). IEEE Software 7(3), 64.

Guide International Corporation. (1989). "Application Reengineering." Guide Publication GPP-208. Guide International Corporation, Chicago.

Halstead, M.H. (1977). "Elements of Software Science." North Holland, Amsterdam.

Harandi, M.T. and Ning, J.Q. (1990). Knowledge-based program analysis. IEEE Software 7(1), 74-81.

Harband, J. (1990). Seela aids maintenance with code-block focus. In (Oman, 1990). IEEE Software 7(3), 61.

Hausler, P.A., Pleszkoch, M.G., Linger, R.C., and Hevner, A.R. (1990). Using function abstraction to understand program behavior. IEEE Software 7(1), 55-63.

Institute of Electrical and Electronics Engineers. (1983). IEEE standard glossary of software engineering terminology. ANSI/IEEE Std 729-1983. Approved IEEE Standards Board, Sept. 23, 1982. Approved American National Standards Institute, Inc. Aug. 9, 1983. In "Software Engineering Standards." (1987). IEEE Press, New York.

Johnson, W. L. (1990). Understanding and debugging novice programs. Artif. Intell. 42(1), 51-97.

Johnson, M. (1983). Problem statement language / problem statement analyzer (PSL/PSA). In "Proceedings of Symposium on Application and Assessment of Automated Tools for Software Development." IEEE Computer Society Press, New York.

Jones, C.B. (1986). "Systematic software development using VDM." Prentice-Hall, London.

Khabaza, I. (1989). Maintenance, validation, and documentation of software systems: 'REDO'--ESPRIT P2487. In "CASE '89: Proceedings of the Third International

Workshop on Computer-Aided Software Engineering." pp. 221-222. The British Computer Society, London.

Kozaczynski, W. and Ning, J.Q. (1989). SRE: A knowledge-based environment for large-scale software re-engineering activities. In "Proceedings of the Eleventh International Conference on Software Engineering." pp. 113-122. IEEE Computer Society Press, Washington, DC.

Letovsky, S. and Soloway, E. (1986). Delocalized Plans and Program Comprehension. IEEE Software 3(3), 41-42.

Luckham, D.C. and von Henke, F.W. (1985). An overview of Anna, a specification language for Ada. IEEE Software 2(2), 9-22.

McCabe, T.J. (1976). A complexity measure. IEEE Trans. Softw. Eng. SE-2(4), 308.

McCabe, T.J. and Butler, C.W. (1989). Design complexity measurement and testing. Commun. ACM 32(12), 1415-1425.

McCabe, T., Jr. (1990). Battle Map, Act show code structure, complexity. In (Oman, 1990). IEEE Software 7(3), 62.

Novobilski, A. (1990). Objective-C Browser details class structures. In (Oman, 1990). IEEE Software 7(3), 60.

Oman, P. (1990). Maintenance tools. IEEE Software 7(3), 59-65.

Oman, P.W. and Cook, C.R. (1990). The book paradigm for improved maintenance. IEEE Software 7(1), 39-45.

Ourston, D. (1989). Program recognition. IEEE Expert 4(4), 36-49.

Pressman, R.S. (1987). "Software Engineering: A Practitioner's Approach." (2nd Ed.). McGraw-Hill Book Company, New York.

Rajlich, V. (1990). Vifor transforms code skeletons to graphs. In (Oman, 1990). IEEE Software 7(3), 60.

Rekoff, M.G., Jr. (1985). On reverse engineering. IEEE Trans. Syst. Man Cybern. SMC-15(2), 244-252.

Rich, C. and Wills, L.M. (1990). Recognizing a program's design: A graph-parsing approach. IEEE Software 7(1), 82-89.

Rozenblat, G.D. and Fischer, H. (1989). Reverse engineering technologies for Ada. In "CASE '89: Proceedings of the Third International Workshop on Computer-Aided Software Engineering." pp. 560-574. The British Computer Society, London.

Rubin, L.F. (1983). Syntax-directed pretty printing--A first step towards a syntax-directed editor. IEEE Trans. Softw. Eng. SE-9(2), 119-127.

Rugaber, S., Ornburn, S.B., and LeBlanc, R.J., Jr. (1990). Recognizing design decisions in programs. IEEE Software 7(1), 46-54.

Samuelson, P. (1990). Reverse-engineering someone else's software: Is it legal? IEEE Software 7(1), 90-96.

Scanlan, D.A. (1989). Structured flowcharts outperform pseudocode: an experimental comparison. IEEE Software 6(5), 28-36.

Schwanke, R.W., Altucher, R.Z., and Platoff, M.A. (1989). Discovering, visualizing, and controlling software structure. In "Proceedings of the Fifth International Workshop on Software Specification and Design." Softw. Eng. Notes. 14(3), pp. 147-150. IEEE Computer Society Press, Washington, DC.

Seviora, R. (1987). Knowledge-based program debugging systems. IEEE Software 4(3), 20-32.

Sibor, V. (1990). Interpreting reverse-engineering law. IEEE Software 7(4), 4-10.

Sommerville, I. (1989). "Software Engineering." (3rd Ed.). Addison-Wesley Publ. Co., Inc., Wokingham, England.

Stender, J. (1989). CASE '89--Position paper on "reverse engineering." In "CASE '89: Proceedings of the Third International Workshop on Computer-Aided Software Engineering." pp. 51-52. The British Computer Society, London.

Tsai, J.J.-P. and Ridge, J.C. (1988). Intelligent support for specifications transformation. IEEE Software 5(6), 28-35.

Vanek, L. and Davis, L. (1990). Expert Dataflow and Static Analysis tool. In (Oman, 1990). IEEE Software 7(3), 63.

Wilde, N. (1990). Dependency Analysis Tool Set prototype. In (Oman, 1990). IEEE Software 7(3), 65.

Yau, S.S. and Tsai, J.J. (1987). Knowledge representation of software component interconnection information for large-scale software modifications. IEEE Trans. Softw. Eng. SE-13(3), 355-361.

Yourdon, E. and Constantine, L.L. (1979). Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice-Hall, Inc., Englewood Cliffs, NJ.

FIGURE CAPTIONS

Figure 1. The place of forward and reverse engineering, restructuring, and reengineering among the phases of the life cycle.

Figure 2. Fundamental characteristics of redocumentation and design recovery.

Figure 3. Basic architecture for reverse engineering, restructuring, and reengineering tools (provided by Robert Arnold).

Figure 4. Different perspectives between forward and reverse engineering.

Figure 5. Demonstration of C code formatted according to the book paradigm (from Oman, P.W. and Cook, C.R. The book paradigm for improved maintenance. IEEE Software 7(1), 39-45. Copyright IEEE 1990; used with permission).

Figure 6. GRASP/Ada Overview

Figure 7. Ada source code for task CONTROLLER.

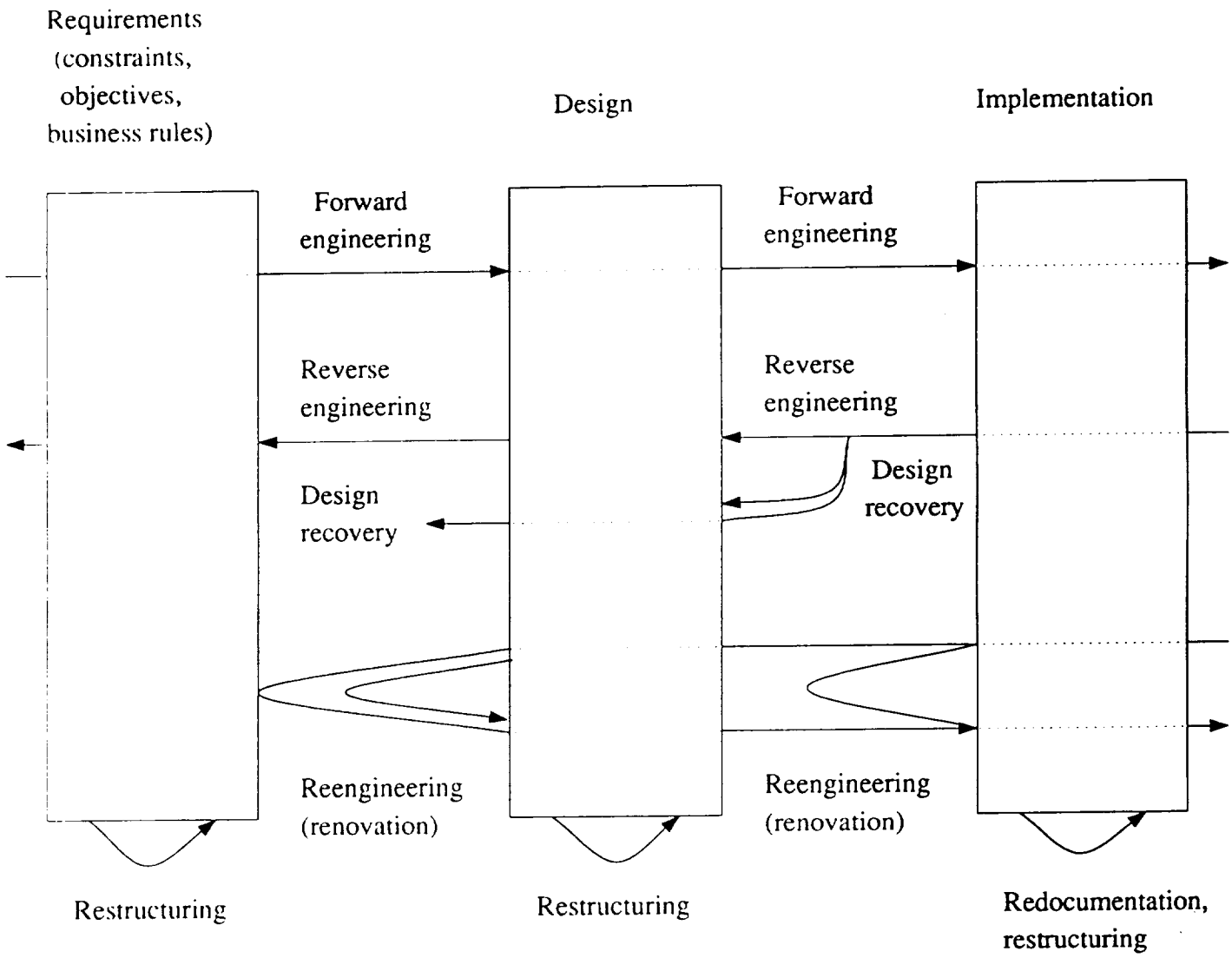
Figure 8. Control structure diagram of Ada source code for task CONTROLLER.

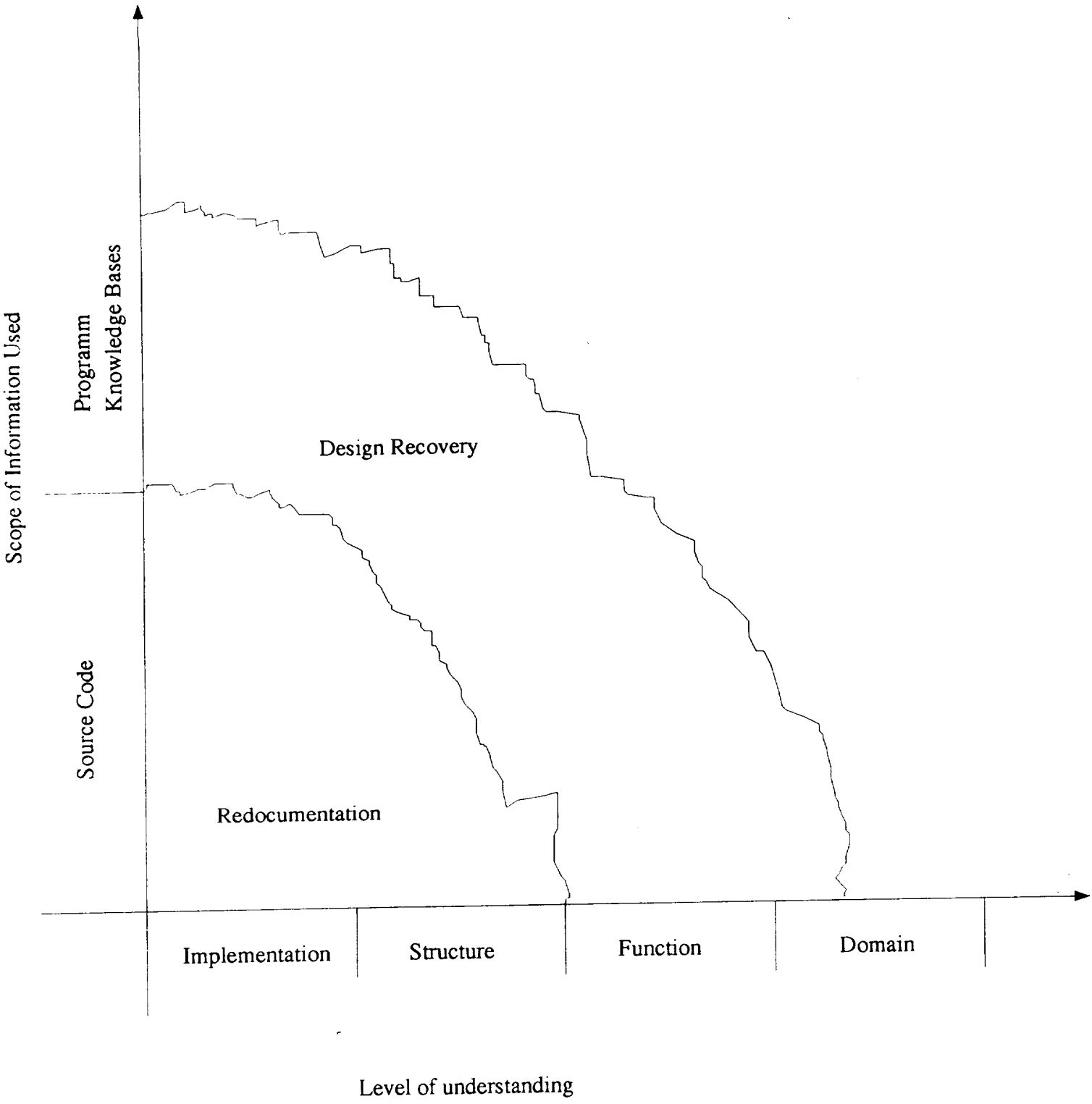
Figure 9. Event representation of the simple-swap concept (from Harandi, M.T. and Ning, J.Q. Knowledge-based program analysis. IEEE Software 7(1), 74-81. Copyright IEEE 1990; used with permission).

Figure 10. Excerpts from an accumulator plan (adapted from Harandi, M.T. and Ning, J.Q. Knowledge-based program analysis. IEEE Software 7(1), 74-81. Copyright IEEE 1990; used with permission).

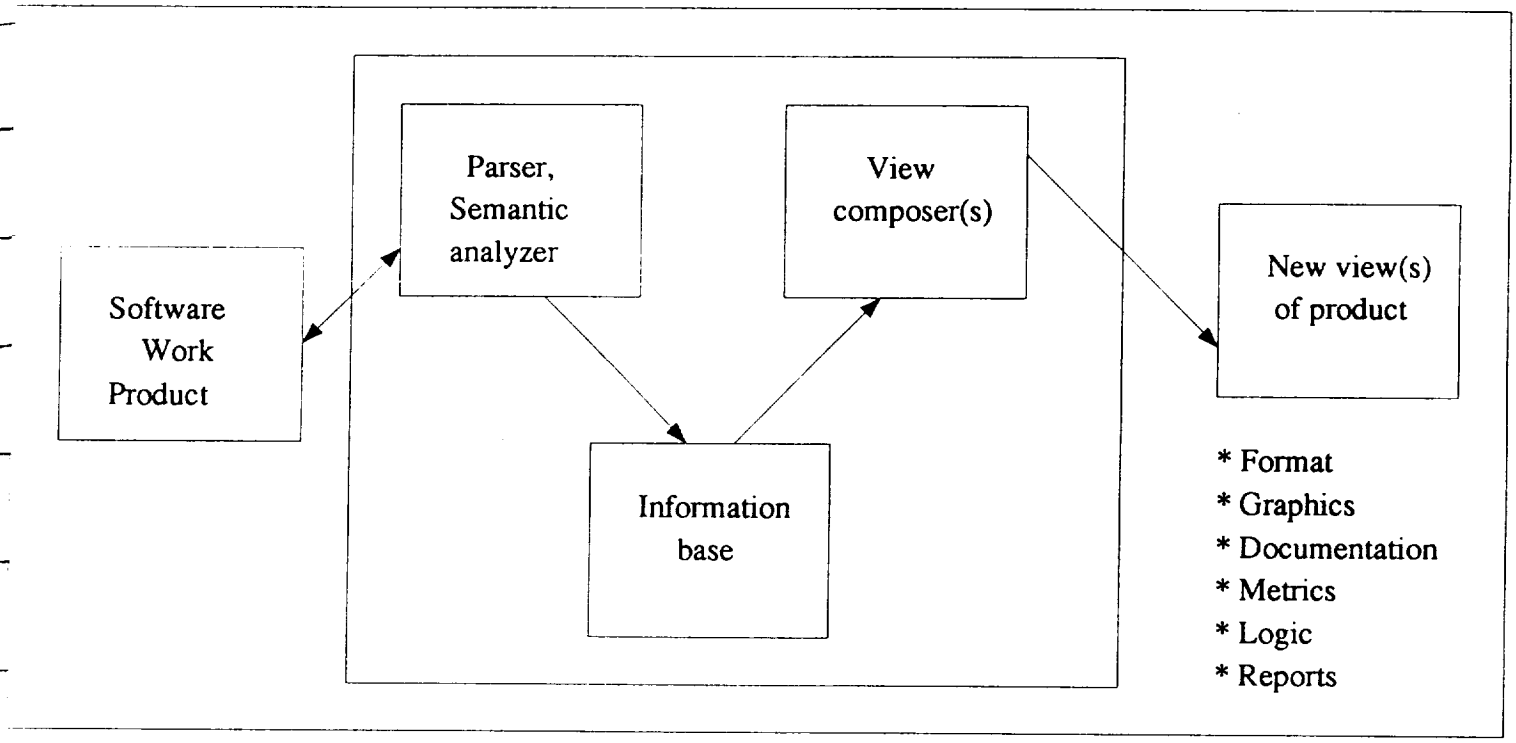
Figure 11. Typical *Recognizer* Plan Calculus and flow graph representations of programming clichés. (a) Representation in the Plan Calculus of how associative retrieval can be implemented using a hash table. (b) Part of the encoding of the overlay into graph grammar rules (the constraints are not shown) (from Rich, C. and Wills, L.M. Recognizing a program's design: A graph-parsing approach. IEEE Software 7(1), 82-89. Copyright IEEE 1990; used with permission).

Figure 12. A template used to extract a functional description from a loop slice (from Hausler, P.A., Pleszkoch, M.G., Linger, R.C., and Hevner, A.R. Using function abstraction to understand program behavior. IEEE Software 7(1), 55-63. Copyright IEEE 1990; used with permission).

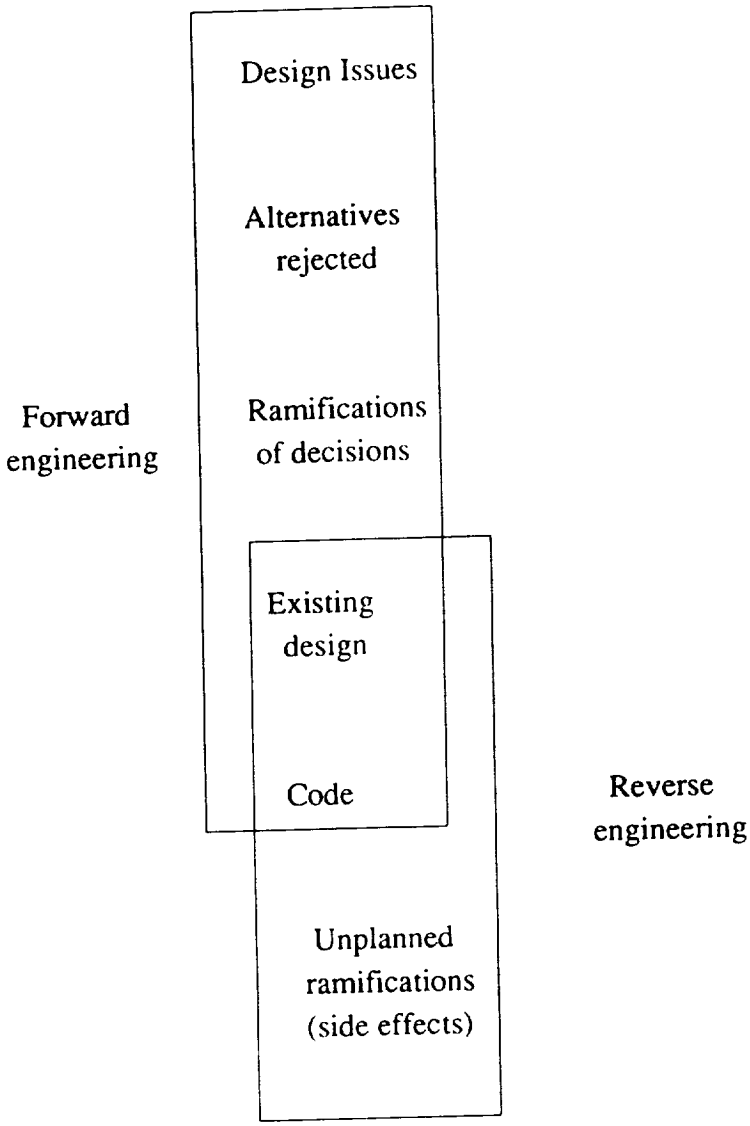

Requirements
(constraints,
objectives,
business rules)
Design
Implementation
Forward
engineering
Forward
engineering
Reverse
engineering
Reverse
engineering
Design
recovery
Design
recovery
Reengineering
(renovation)
Reengineering
(renovation)
Restructuring
Restructuring
Redocumentation,
restructuring



CROSS et al.
FIG. 2



CROSS et al.
FIG. 3

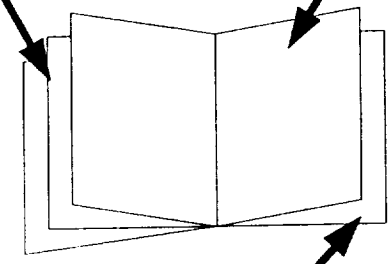


CROSS ref.
FIG. 4

```
/* Contents          R_Polish          Page 2 */

/* ------------------------------------------- */
/*                                             */
/*                Table of Contents            */
/*                                  Page #     */
/* Title page & Preface               1.       */
/*                                             */
/*                                             */
/* Table of Contents                  2.       */
/*                                             */
/* Chapter 1(globals):                         */
/*   Program global definitions       3.       */
/*   Program inclusions               3.       */
/*            */                               */
/* Chapter 2(main program):                    */
/*   Main()                           4.       */
/*                                             */
/* Chapter 3(stack unit):                      */
/*   Stack unit global definitions    5.       */
/*   Push()                           5.       */
/*   Pop()                            5.       */
/*   Clear()                          5.       */
/*                                             */
/* Module Index                       6.       */
/*                                             */
```

```
/* Index             R_Polish          Page 6 */

/* ------------------Module Index ------------ */
/*                                             */
/* AtoE(NOT defined)                           */

/*   Referenced, p.4                           */
/*   Called by: main                           */
/*                                             */
/* Clear(defined  p.5)                         */
/*   Referenced 4,5                            */
/*   Called by: main, Push, Pop                */
/*                                             */
/* GetOp. (NOT defined)                        */
/*    Referenced p.4                           */
/*    Called by:main                           */
/*                                             */
/* main(defined p.4)                           */
/*    Calls to:AtoF,Clear,GetOp,Pop,printf,Push */
/*                                             */
/* Pop (defined p.5)                           */
/*    Referenced p.4,5                         */
/*    Called by:main                           */
/*    Calls to:Clear,printf                    */
/*                                             */
/* Push(defined p.5)                           */
/*   Referenced p.4,5                          */
/*   Called by:main                            */
/*   Calls to:Clear,printf                     */
/*                                             */
/* ------------------------------------------- */
```

```
/* Chapt. 2          R_Polish          Page 4  */

/* ------------------- Main() ------------------ */

                /* reverse Polish desk calculator */
main()

{ int           Type;
  char          S[MaxOp];
  double        Op2, AtoF(), Pop(), Push();

  while ((Type= GetOp(S,MaxOp)) != EOF)

  switch(Type) {

     case Number:   Push( AtoF(S) );                    break;

     case '+':      Push( Pop()+ Pop() );               break;

     case '*':      Push ( Pop()* Pop() );              break;

     case '-':      Op2 = Pop();
                    Push( Pop()-Op2);                   break;

     case'/':       Op2= Pop();
                    if(Op2 != 0.0) Push( Pop()/Op2);
                    else printf("zero divisor popped\n"); break;

     case'=':       printf("\t%f\n", Push( Pop()) );    break;

     case'c':       clear();                            break;

     case TooBig:   printf( "%.20s... is too long\n",S); break;

     default:       printf( "unknown command %c\n", Type);
                                                        break;


  } /* end switch */

  } /* end while */
} /* end main */
```

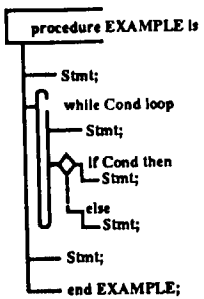


CROSS-REF
FIG. 5

GRASP/Ada Overview

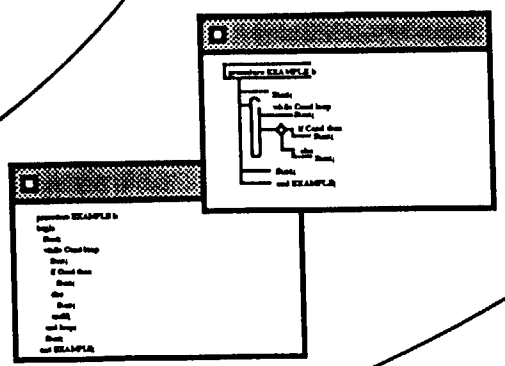
Code

```
procedure EXAMPLE is
begin
  Stmt;
  while Cond loop
    Stmt;
    if Cond then
      Stmt;
    else
      Stmt;
    endif;
  end loop;
  Stmt;
end EXAMPLE;
```



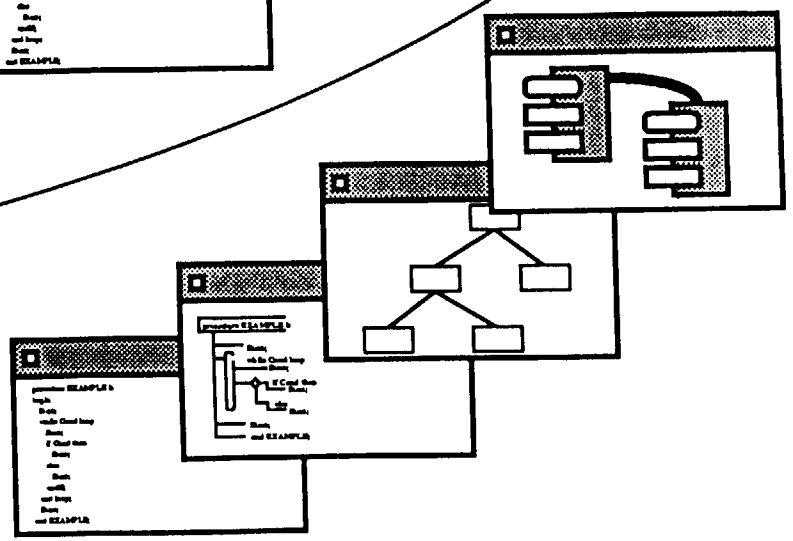
Control Structure Diagrams

Phase 1



User Interface (X Window System)

Phase 2

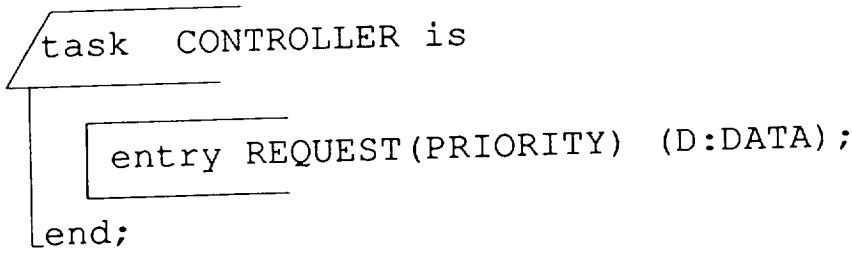


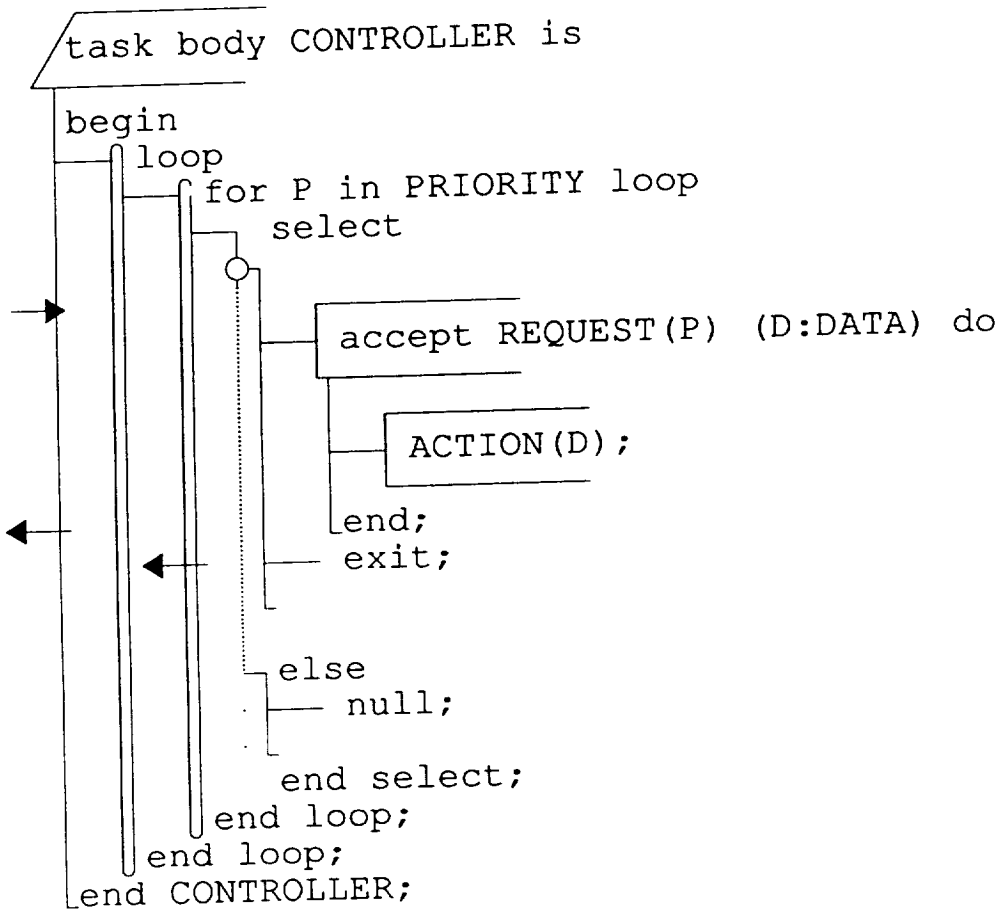
Object Diagrams Prototype Integration

Phase 3

```
task  CONTROLLER is

   entry REQUEST(PRIORITY)  (D:DATA);

end;



task body CONTROLLER is

begin
   loop
      for P in PRIORITY loop
         select

            accept REQUEST(P)  (D:DATA) do

               ACTION(D);

            end;
            exit;


         else
            null;

         end select;
      end loop;
   end loop;
end CONTROLLER;
```

Cross et al.
FIG 7





CROSS ref.
FIG. 4

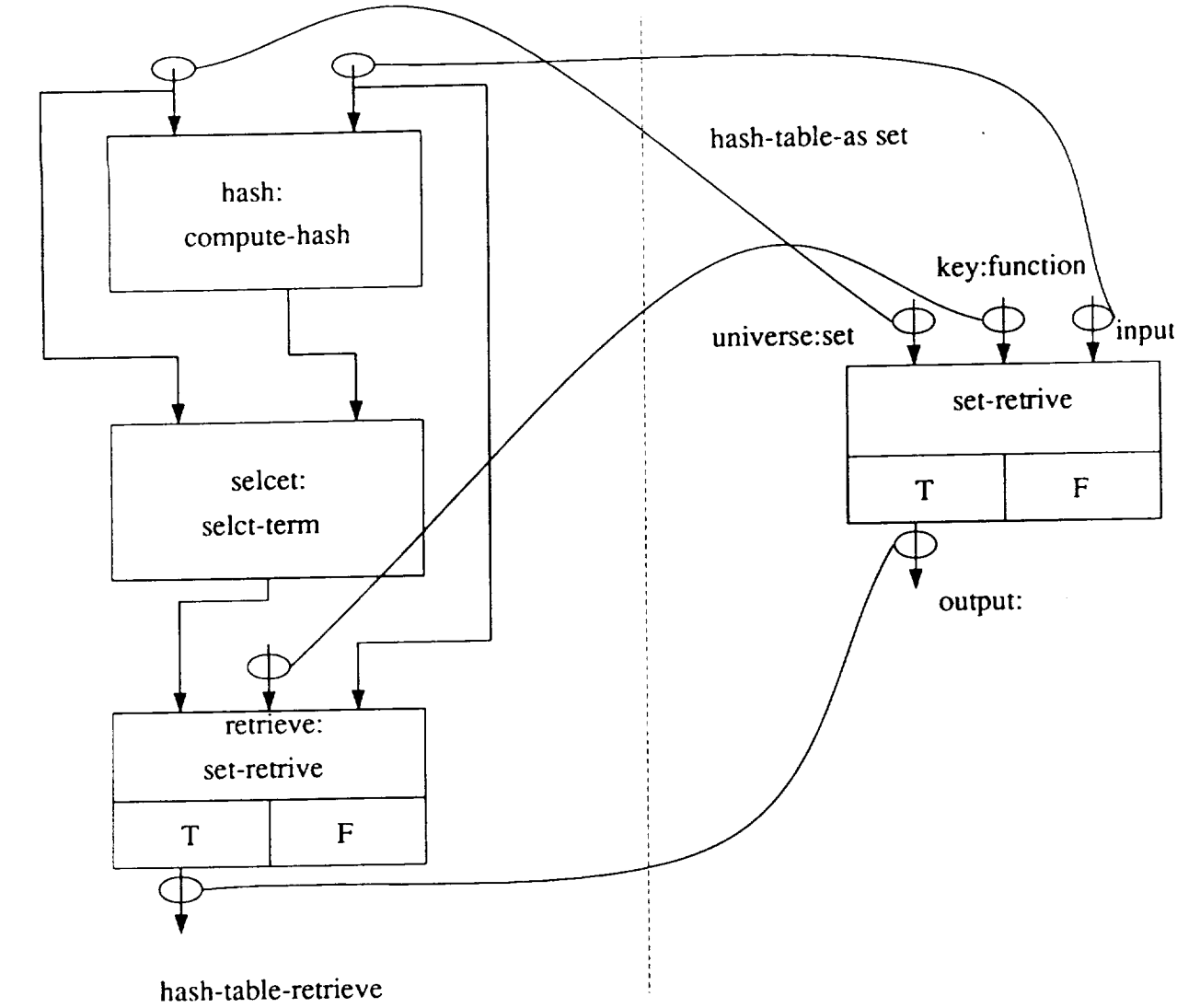
```
        •••
38      procedure swap(var X, Y : integer);
40      var T : integer;
43      assign X to T;
44      assign Y to X;
43      assign T to Y;
47      end-procedure;
        •••
120     proc-call swap (A, B);
        •••
```

event-class:	SIMPLE-SWAP
interval:	([0 (120 43 44 45)] [0 (4 43 44 45)])
external-form:	((43 44 45) SIMPLE-SWAP A B)
var1:	A_0
var2:	B_0
temp-var:	T_4

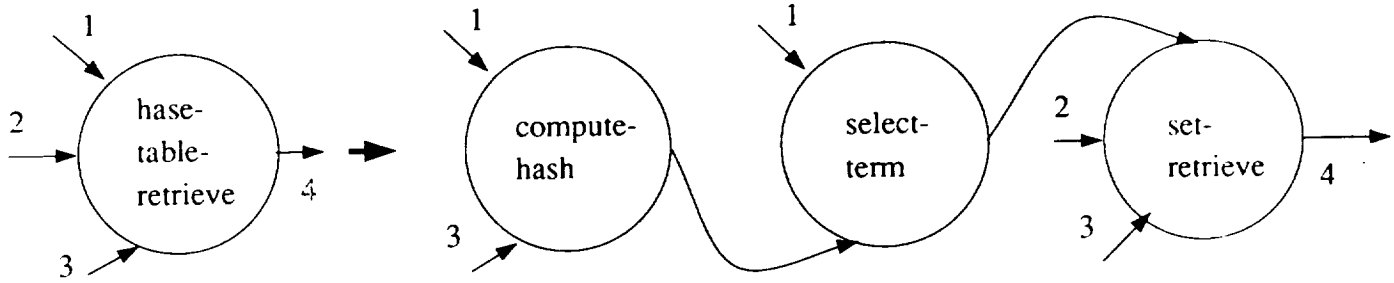
CROSS et al.
FIG 9

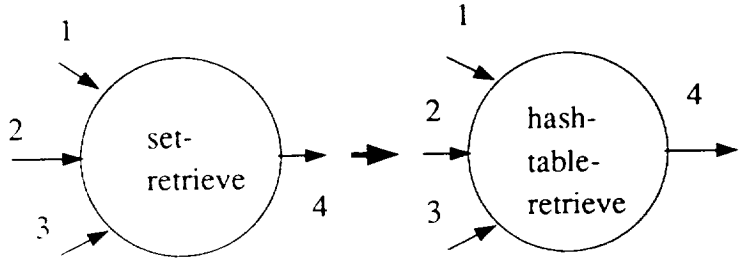
```
plan (accumulator
            :update-var ?var
            :init-value ?init
            :update-value ?val
            :update-cond ?cond
            :accumulator-op ?op)
      path (c-precede
                  (assign :var-defined ?var :value-used ?init)
                  (c-enclose
                        (enumerator :loop-cond ?cond)
                        (key
                              (assign
                                    :var-defined ?var
                                    :value-used (?op ?var ?val)
                              )
                        )
                  )
            )
```

CROSS et al
FIG 10



(a)





(b)

CROSS etc. FIG 11

If the loop slice matches the pattern

```
PERFORM UNTIL X >= EXPR3
        X := X + EXPR4
END-PERFORM
```

then the function abstraction for variable *X* after *N* loop iterations is

```
X := X + EXPR4 * N
```

where the total number of loop iterations is

```
((X <  EXPR3) -> N := ceiling((EXPR3 - X)/EXPR4) |
 (X >= EXPR3) -> N := 0)
```

TABLE I

REVERSE ENGINEERING SUPPORT FOR SQA THROUGHOUT THE LIFE CYCLE

GENERIC LIFE CYCLE ACTIVITIES

SQA Activities	Requirements Analysis	Design	Implementation (code and integration)	Maintenance
Formal Technical Reviews (FTRs)	Alternate views of current requirements specification (DFDs, ODs)	Alternate views of current design (ODs, CSDs, PDLs)	Alternate views of source code for verification and fine tuning design (OD, CSD, code)	All previous views are continuously updated
Metrics	Collected and analyzed--becomes part of FTR and decision-making process	McCabe's complexity measures (McCabe, 1976); Halstead's software science (Halstead, 1977)	McCabe's complexity measures; Halstead's software science	Continuing collection

Testing	Functional areas to test; system verification criteria	Integration verification criteria	Identification of functional and structural test cases; coverage analysis (unit test, integration test, system test)	Testing of changes plus regression testing

TABLE II

REDOCUMENTATION TOOLS

Tool Name	Company or Group	Input Languages	Output Form	References
Book-Maker	Univ. of Idaho/ Oregon St. Univ.	C, Pascal	Specialized code format	Oman and Cook, 1990
ARCH/ (MAINTAINER'S ASSISTANT)	Siemens Research		Graphical subsystem tree	Schwanke et al., 1989
GRASP/Ada	Auburn University	Ada	Control Structure Diagram (Algorithmic GR); Object diagrams	Cross, 1990; Cross et al., 1990

Objective-C Browser	Stepstone	Objective-C	Graphical representations (GRs), query capability	Novobilski, 1990
Vifor	Software Tools and Technologies	FORTRAN	GRs, generated code skeletons	Rajlich, 1990
PSL/PSA	University of Michigan		PSL	Johnson, 1983 Chikofsky, 1983
Seela	Tuval Software Industries	Ada, COBOL, C, Pascal, PL/M, FORTRAN	PDL-enhanced code browser	Harband, 1990

BattleMap/ ACT	McCabe & Associates	C, FORTRAN, COBOL, Ada, BASIC, PL/I, Pascal, PDLs 8086 Asm., 6502 Asm.	GRs color-coded according to complexity metric	McCabe, 1990
Expert Dataflow and Static Analysis	Array Systems Computing	Ada	Code browser	Vanek and Davis, 1990
Surgeon's Assistant	Loyola College	C	Slice-based editing	Gallagher, 1990
Dependency Analysis Tool Set	Univ. of West Florida	C	GRs, query capability	Wilde, 1990
REDO	ESPRIT	COBOL, FORTRAN	Environment database	Khabaza, 1989

MicroScope	Hewlett-Packard	Common LISP and Common Objects	Browser, monitor, GRs	Ambras and O'Day, 1988
Adagen		Ada	GRs	Rozenblat and Fischer, 1989
PUNS	IBM	IBM Assembly Language	Code browser, knowledge base	Cleveland, 1989

TABLE III

DESIGN RECOVERY TOOLS

Name	Company or Group	Input Languages	Output Form	Primary Strategy	Reported State	References
Desire	MCC	C	GRs, updates to *Domain Model*	Pattern matching (structural and lexical)	Early prototype (source analysis alone)	Biggerstaff, 1989
Program Analysis Tool/ (Software Re-engineering Environment)	Univ. of Ill. (Urbana-Champaign)/ Arthur Andersen and Company		Text explanation of program *plan*	Pattern matching; rule-based inference	Prototype; *Plan Base* evolving	Harandi and Ning, 1990 (see also Kozaczynski and Ning, 1989)

Recognizer/ (Programmer's Apprentice)	MIT	Common LISP	Design tree, text explanation	Pattern matching; graph parser-based inference	Prototype; *cliché library* evolving	Rich and Wills, 1990 (see also Ourston, 1989)
Function Abstraction (a technique name)	IBM Systems Integration/ Univ. of Maryland	COBOL	Algebraic text explanation	Algebraic manipulation; loop-slice pattern matching	No prototype	Hausler et al., 1990
PROUST	Univ. of So. Calif.	Pascal	Intention-oriented bug reports	Goal decomposition; plan template pattern matching	Prototype; knowledge bases evolving	Johnson, 1990 (see also Ourston, 1989 and Seviora, 1987)
Talus	W.R. Murray of Univ. of Texas	LISP	Debugging tool	Tree node matching, theorem proving	Prototype (reference from 1986)	from Ourston, 1989

PROMPTER	Science Institute; IBM Japan, Ltd.	IBM Assembly Language	Program- and problem-specific annotations	Pattern matching; rule-based inference	Design stages; knowledge base evolving	Fukunaga, 1985
Laura	A. Adam and J.P. Laurent	FORTRAN	Bug-localization reports	Comparison of normalized input program graph to "correct" solution	Prototype (reference from 1980)	from Seviora, 1987
PUDSY	F.J. Lukey	Pascal example given	Predicate calculus output assertions; bug repairs	Pattern matching; assertion equivalence checking	Prototype (reference from 1980)	from Seviora, 1987

Appendix B

"Control Structure Diagrams For Ada"

by

James H. Cross II
Auburn University

Sallie V. Sheppard
Texas A&M University

W. Homer Carlisle
Auburn University

Published in *Journal of Pascal, Ada & Modula 2*, Vol. 9, No. 5, Sep./Oct. 1990, 26-33.

Control Structure Diagrams for Ada

James H. Cross II
Sallie V. Sheppard
W. Homer Carlisle

Advances in hardware, particularly high-density bit-mapped monitors, have led to a renewed interest in graphical representation of software. Much of the research activity in the area of software visualization and computer-aided software engineering (CASE) tools has focused on architectural-level charts and diagrams.

However, the complex nature of the control constructs and the subsequent control flow defined by program design languages (PDLs), which are based on programming languages such as Ada, Pascal, and Modula-2, make detailed design specifications attractive candidates for graphical representation. And, since the source code itself will be read many times during the course of initial development, testing, and maintenance, it too should benefit from the use of an appropriate graphical notation.

The control structure diagram (CSD) is a notation intended specifically for the graphical representation of detailed designs, as well as actual source code. The primary purpose of the CSD is to reduce the time required to comprehend software by clearly depicting the control constructs and control flow at all relevant levels of abstraction, whether at the design level or within the source code itself. The CSD is a natural extension to existing architectural graphical representations such as data flow diagrams, structure charts, and Booch diagrams.

The CSD, initially created for Pascal/PDL[1], has been extended significantly so that the graphical constructs of the CSD map directly to the constructs of Ada. The rich set of control constructs in Ada (e.g., task rendezvous) and the wide acceptance of Ada/PDL by the software engineering community as a detailed design language made Ada a natural choice for the basis of a graphical notation. A major objective in the philosophy that guided the development of the CSD was that the graphical constructs supplement the code and PDL without disrupting their familiar appearance. That is, the CSD should appear to be a natural extension to the Ada constructs and, similarly, the Ada source code should appear to be a natural extension of the diagram. This has resulted in a concise, compact graphical notation that attempts to combine the best features of previous diagrams with those of well-established PDLs. A CSD generator was developed to automate the process of producing the CSD from Ada source code.

Background

Graphical representations have long been recognized as having an important impact in communicating from the perspective of both the "writer" and the "reader." For software, this includes communicating requirements between users and designers and communicating design specifications between designers and implementors. However, there are additional areas where the potential of graphical notations have not been fully exploited. These include communicating the semantics of the actual implementation represented by the source code to personnel for the purposes of testing and maintenance, each of which are major resource sinks in the software lifecycle. In particular, Shelby et al. [2] found that code reading was the most cost-effective method of detecting errors during the verification process when compared to functional and structural testing. Standish [3] reported that program understanding may represent as much as 90% of the cost of maintenance. Hence, improved comprehension efficiency resulting from the integration of graphical notations and source code could have a significant impact on the overall cost of software production.

The CSD for Ada is supported by an operational prototype graphical prettyprinter that accepts Ada source code as input and generates the CSD in a manner similar to text-based prettyprinters.

Since the flowchart was introduced in the mid-50s, numerous notations for representing algorithms have been proposed and utilized. Several authors have published notable books and papers that address the details of many of these [4–6]. Tripp [5], for example, describes eighteen distinct notations that have been introduced since 1977, and Aoyama et al. [6] describe the popular diagrams used in Japan. In general, these diagrams have been strongly influenced by structured programming and thus contain control constructs for sequence, selection, and iteration. In addition, several contain explicit EXIT structures to allow single entry/multiple exit control flow through a block of code, as well as PARALLEL or concurrency constructs. However, none of the diagrams cited explicitly contains all of the control constructs found in Ada.

Graphical notations for representing software at the algorithmic level have been neglected, for the most part, by business and industry in the United States in favor of nongraphical PDLs. A lack of automated support and the results of several studies conducted in the 1970s that found no significant difference in the comprehension of algorithms represented by flowcharts and pseudocode [7] have been major factors in this underutilization. However, automation is now available in the form of numerous CASE tools, and recent empirical studies reported by Aoyama [6] and Scanlan [8] have concluded that graphical notations may indeed improve the comprehensibility and overall productivity of software. Scanlan's study involved a well-controlled experiment in which deeply nested if-then-else constructs, represented in structured flowcharts and pseudocode, were read by intermediate-level students. Scores for the flowchart were significantly higher than those of the PDL. The statistical studies reported by Aoyama et al. involved several tree-structured diagrams (e.g., PAD, YACC II, and SPD) widely used in Japan that, in combination with their environments, have led to significant gains in productivity. The results of these recent studies suggest that the use of a graphical notation with appropriate automated support for Ada/PDL and Ada should provide significant increases in productivity over current nongraphical approaches.

Control Structure Diagram

Figure 1(a) contains an Ada task body CONTROLLER adapted from [9] that loops through a priority list attempting to accept selectively a REQUEST with priority P. Upon on acceptance, some action is taken, followed by an exit from the priority list loop to restart the loop with the first priority. In typical Ada task fashion, the priority list loop is contained in an outer infinite loop. This short example contains two threads of control: the rendezvous, which enters and exists at the accept statement, and the thread within the task body. In addition, the priority list loop contains two exits: the normal exit at the beginning of the loop when the priority list has been exhausted, and an explicit exit invoked within the select statement. While the concurrency and multiple exits are useful in modeling the solution, they do increase the effort required of the reader to comprehend the code.

Figure 1(b) shows the corresponding CSD generated by the graphical prettyprinter. In this example, the intuitive graphical constructs of the CSD clearly depict the point of rendezvous, the two nested loops, the se-

continued on page 32

lect statement guarding the accept statement for the task, the unconditional exit from the inner loop, and the overall control flow of the task. When reading the code without the diagram, as shown in Figure 1(a), the control constructs and control paths are much less visible although the same structural and control information is available. As additional levels of nesting and increased physical separation of sequential components occur in code, the visibility of control constructs and control paths becomes increasingly obscure, and the effort required of the reader dramatically increases in the absence of the CSD.

Now that the CSD has been briefly introduced, the various CSD constructs for Ada are presented in Figure 2. Since the CSD is designed to supplement the semantics of the underlying Ada, each of the CSD constructs is self-explanatory and are presented without further description.

Automated Support — The CSD Graphical Prettyprinter

Automated support is a requirement, at least in the in professional ranks, for widespread utilization of any graphical representation. Without automated support, diagrams are difficult to construct and maintain from the standpoint of "living" formal documentation, although software practitioners may use several types of diagrams informally during design and even implementation. Automated support comes in many forms, ranging from general-purpose "drawing aids" to automatic generation and maintenance based on changes to source code. The CSD for Ada is currently supported by an operational prototype graphical prettyprinter that accepts Ada source code as input and generates the CSD in a manner similar to text-based prettyprinters. The prototype was implemented under DEC's VAX VMS using a scanner/parser generator and an Ada grammar. The user interface was built using DEC's VAX Curses, and to provide the user with interactive viewing of the CSD, a special version of DEC's EVE editor was generated. Custom fonts for the CSD graphics characters were built for both the VT220 terminal and the HP Laser Jet printer. Using font-oriented graphics characters rather than bit-mapped images provided for a high degree of efficiency in generating the diagrams.

continued on page 32

The potential of the CSD is best realized during detailed design, implementation, verification, and maintenance.

```
task  CONTROLLER is

   entry REQUEST(PRIORITY)  (D:DATA);
end;



task body CONTROLLER is

begin
   loop
      for P in PRIORITY loop
         select

            accept REQUEST(P)  (D:DATA) do

               ACTION(D);

            end;
            exit;


         else
            null;

         end select;
      end loop;
   end loop;
end CONTROLLER;
```

Figure 1(a). Ada source code for task CONTROLLER

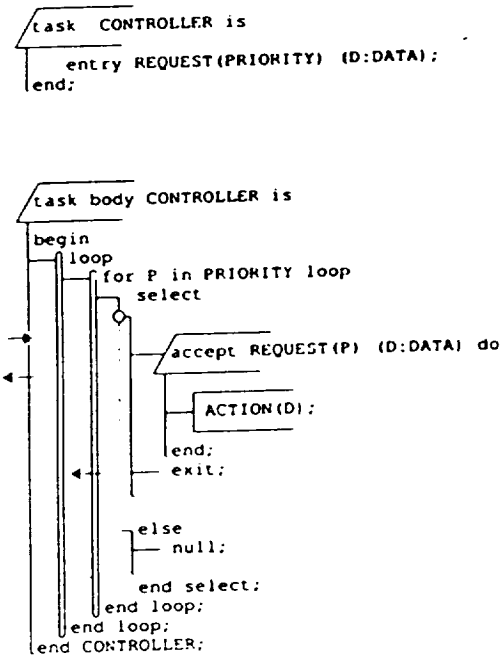


Figure 1(b). Control structure diagram of Ada source code for task CONTROLLER

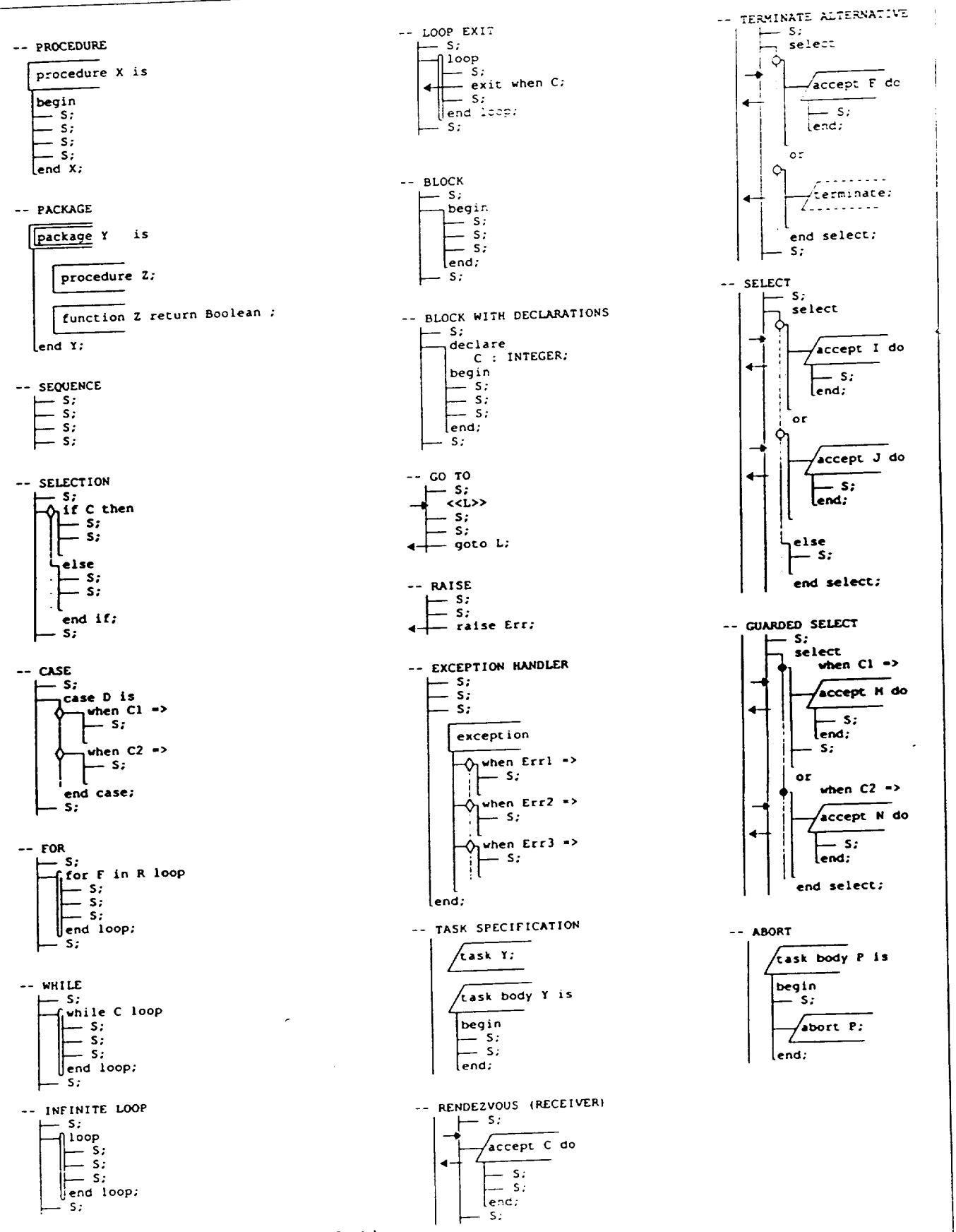


Figure 2. Control structure diagram constructs for Ada.

The prototype is currently being ported to the Sun-4 workstation under UNIX and X Windows, where enhancements will include an option to collapse the diagram around any control constructs and an option to generate an intermediate level architectural diagram that indicates control structure among subprograms and tasks.

Conclusions and Directions

A new graphical tool that maps directly to Ada was formally defined and automated. The CSD offers advantages over previously available diagrams in that it combines the best features of PDL and code with simple intuitive graphical constructs. The potential of the CSD can be best realized during detailed design, implementation, verification, and maintenance. The CSD can be used as a natural extension to popular architectural-level representations such as data flow diagrams, Booch diagrams, and structure charts.

Our current reverse engineering project, GRASP/Ada [10], is focused on the generation of multilevel and multiview graphical representations from Ada source code. As indicated in GRASP/Ada overview shown in Figure 3, the CSD represents the code/PDL level diagram generated by the system. Our present efforts are concentrated on the extraction of architectural- and system-level diagrams such as structure charts, Booch diagrams, and data flow diagrams. The reverse engineering of graphical representations is destined to become an integral component of CASE tools, which until recently have focused on forward engineering. The development of tools that provide for interactive automatic updating of charts and diagrams will serve to improve the overall comprehensibility of software and, as a result, improve reliability and reduce the cost of software.

The reverse engineering of graphical representations is destined to become an integral component of CASE tools, which until recently have focused on forward engineering.

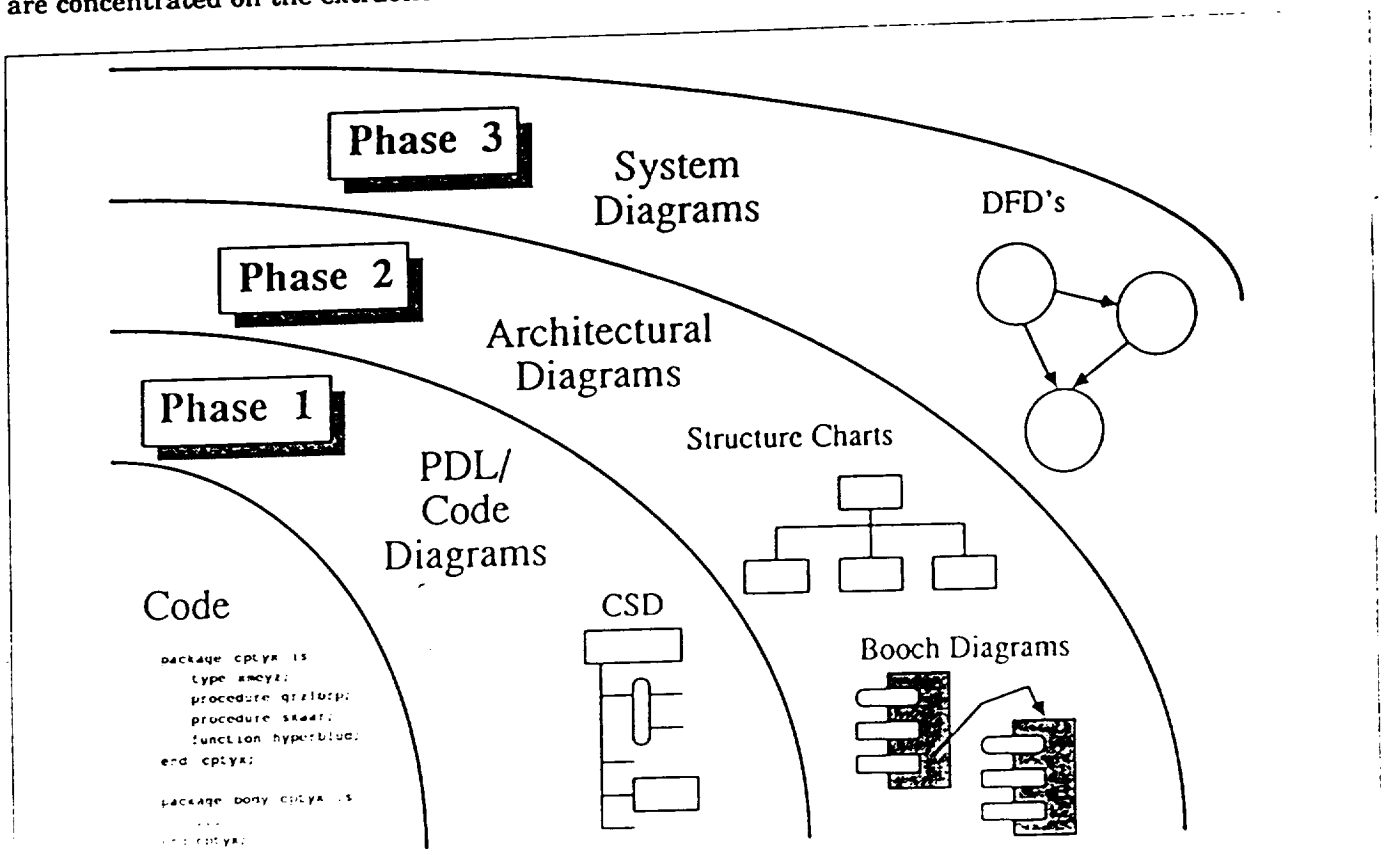


Figure 3. Overview of the GRASP/Ada reverse engineering project.

Acknowledgments

This research was supported, in part, by a grant from George C. Marshall Space Flight Center, NASA/MSFC. AL 35821. Richard Davis, Charles F. May, Kelly I. Morrison, Timothy Plunkett, Darren Tola, K.C. Waddel, and others made valuable contributions to this project.

References

1. J.H. Cross and S.V. Sheppard, The Control Structure Diagram: An Automated Graphical Representation For Software, *Proceedings of the 21st Hawaii International Conference on Systems Sciences* (Kailui-Kona, HA, Jan. 5–8). IEEE Computer Society Press, Washington, DC, 1988, Vol. 2, pp. 446–454.
2. R. Shelby et al., A Comparison of Software Verification Techniques, *NASA Software Engineering Laboratory Series* (SEL-85-001), Goddard Space Flight Center, Greenbelt, MD, 1985.
3. T. Standish, An Essay On Software Reuse, *IEEE Transactions on Software Engineering*, SE-10, (9), 494–497, 1985.
4. J. Martin and C. McClure, *Diagramming Techniques for Analysts and Programmers*, Prentice-Hall, Englewood Cliffs, NJ, 1985.
5. L.L. Tripp, Survey of Graphical Notations For Program Design — An Update, *Software Engineering Notes*, 13(4), 39-44, 1988.
6. M. Aoyama et al., Design Specification in Japan: Tree-Structured Charts, *IEEE Software*, 31-37, 1989.
7. B. Shneiderman et al., Experimental Investigations of the Utility of Detailed Flowcharts in Programming, *Communications of the ACM*, No. 20, 373–381, 1977.
8. D.A. Scanlan, Structured Flowcharts Outperform Pseudocode: An Experimental Comparison, *IEEE Software*, 28-36, 1989.
9. J.G.P. Barnes, *Programming in Ada*, Second Edition, Addison-Wesley Publishing Co., Menlo Park, CA, 1984.
10. J.H. Cross, GRASP/Ada: Graphical Representations of Algorithms, Structures and Processes for Ada, *Technical Report* (NASA-NCC8-14), Auburn University, December 1989.

James H. Cross II *is an Assistant Professor of Computer Science and Engineering at Auburn University, Auburn, AL. His research interests include design methodology, development environments, reverse engineering and maintenance, visualization, and testing. He received a B.S. degree from the University of Houston, an M.S. degree from Sam Houston State University, and a Ph.D. from Texas A&M University.*

Sallie V. Sheppard *is the Associate Provost for Undergraduate Studies and Professor of Computer Science at Texas A&M University, College Station, TX. She received B.A. and M.S. degrees from Texas A&M University and a Ph.D. from the University of Pittsburgh. Her research interests include programming languages and simulation.*

W. Homer Carlisle *is an Assistant Professor of Computer Science and Engineering at Auburn University, Auburn, AL. He received B.A, M.A., and Ph.D. degrees from Emery University., His research interests include programming languages and parallel processing.*

Appendix C

Extended Examples

The examples in this Appendix were extracted from a set of Ada source code files provided by NASA to test the CSD generator. These examples were used in Section 5 to illustrate the User Interface.

```
with LEVEL_A_CONSTANTS;
use LEVEL_A_CONSTANTS;
with DATA_TYPES;
use DATA_TYPES;
with FSW_POOL;
use FSW_POOL;
with IL_POOL;
use IL_POOL;
with SIM_POOL;
use SIM_POOL;
with MATH_PACKAGE;
use MATH_PACKAGE;
with QUATERNION_OPERATIONS;
use QUATERNION_OPERATIONS;
with DOUBLE_PRECISION_MATRIX_OPERATIONS;
use DOUBLE_PRECISION_MATRIX_OPERATIONS;
with SINGLE_PRECISION_MATRIX_OPERATIONS;
use SINGLE_PRECISION_MATRIX_OPERATIONS;

package body AERO_DAP_PACKAGE is

   FIRST_PASS : BOOLEAN_32 := TRUE;
---------------------
-- FIRST PASS FLAG --
---------------------
   TRIM_ERROR_L : SCALAR_SINGLE := 0.0;
----------------------------
-- PITCH CHANNEL VARIABLE --
----------------------------
   KP_RCS : INTEGER := 0;
------------------------------
-- JET SELECT LOGIC VARIABLES --
------------------------------
   KQ_RCS : INTEGER := 0;
   KR_RCS : INTEGER := 0;
   ALPHA_DAP : SCALAR_SINGLE := 0.0;
-------------------------------------------------------------------------
-- THIS NEXT SECTION OF VARIABLES HAS BEEN ADDED TO THIS PORTION OF    --
-- OF THE PACKAGE IN ORDER TO PROVIDE A DUMP OF THESE VARIABLES,       --
-- NOT BECAUSE THEY NEED 'MEMORY' IN THE SENSE THAT THEIR VALUES       --
-- MUST BE REMEMBERED FROM INVOCATION TO INVOCATION OF PROCEDURE       --
-- AERO_DAP.  CONSEQUENTLY, WHEN THE FLIGHT SOFTWARE IS FULLY          --
-- CHECKED OUT, THESE DECLARATIONS CAN BE MOVED TO APPEAR AS LOCAL     --
-- DECLARATIONS IN PROCEDURE AERO-DAP                                  --
-------------------------------------------------------------------------
---------------------------------------
-- PROCEDURE AERO_DAP LOCAL VARIABLES --
---------------------------------------
--------------------------
-- ALPHA, BETA, AND PHI --
--------------------------
   BETA_DAP : SCALAR_SINGLE := 0.0;
   CALPHA : SCALAR_SINGLE := 0.0;
   PHI_DAP : SCALAR_SINGLE := 0.0;
   SALPHA : SCALAR_SINGLE := 0.0;
   BETA_FCS : SCALAR_SINGLE := 0.0;
---------------------------
-- BETA FILTER VARIABLES --
---------------------------
   P_FCS : SCALAR_SINGLE := 0.0;
-----------------------------------------
-- TRANSPORT DELAY COMPENSATION VARIABLES --
-----------------------------------------
```

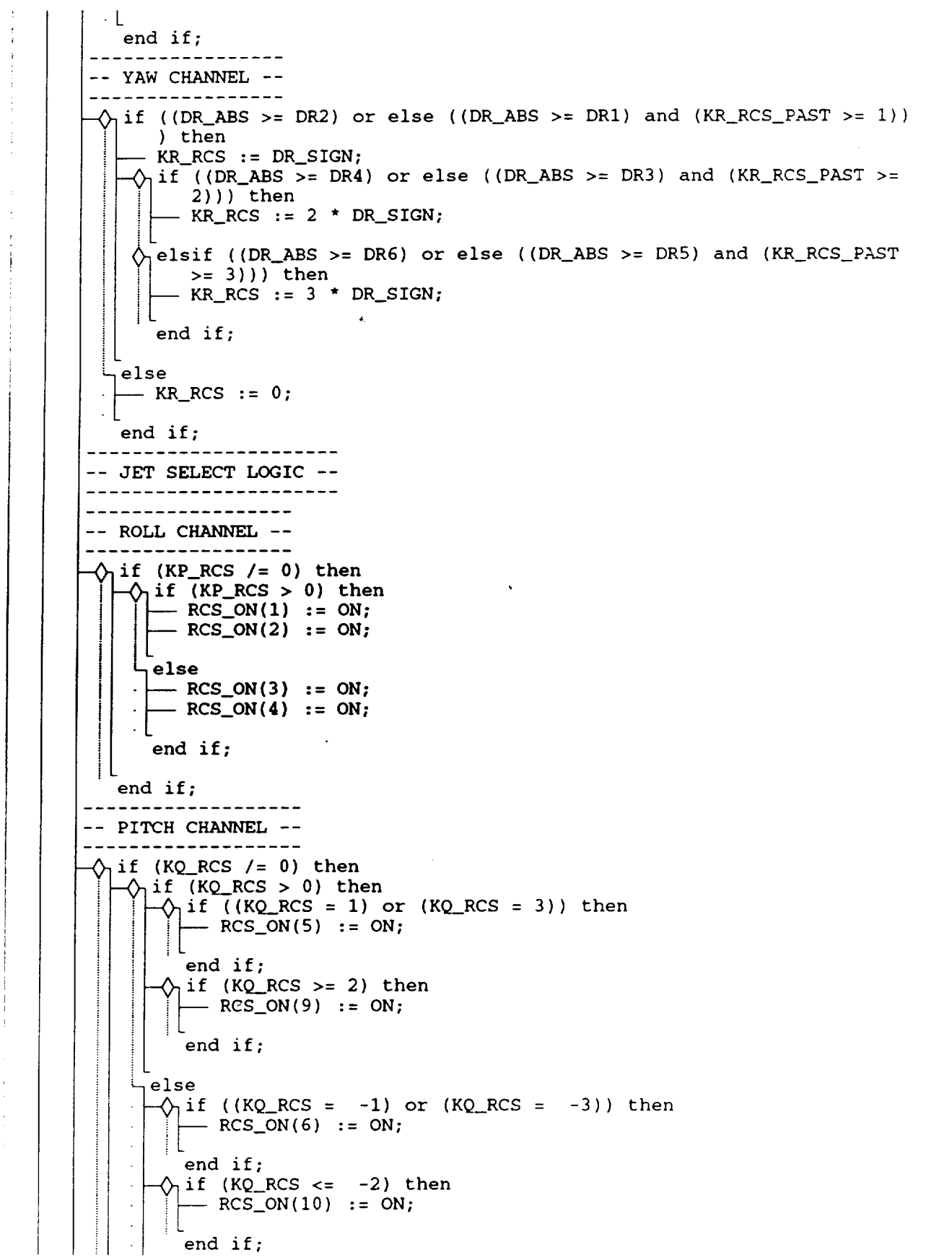
```
   Q_FCS : SCALAR_SINGLE := 0.0;
   R_FCS : SCALAR_SINGLE := 0.0;
   PHI_ERROR : SCALAR_SINGLE := 0.0;
-----------------------------
-- STABILITY AXES VARIABLES --
-----------------------------
   BANK_RATE_CMD : SCALAR_SINGLE := 0.0;
   BETA_RATE_CMD : SCALAR_SINGLE := 0.0;
   DP_CMD : SCALAR_SINGLE := 0.0;
-------------------------
-- BODY AXES VARIABLES --
-------------------------
   DQ_CMD : SCALAR_SINGLE := 0.0;
   DR_CMD : SCALAR_SINGLE := 0.0;
   P_CMD : SCALAR_SINGLE := 0.0;
---------------------------
-- ROLL CHANNEL VARIABLES --
---------------------------
   P_ERROR : SCALAR_SINGLE := 0.0;
   ALPHA_TRIM_CMD : SCALAR_SINGLE := 0.0;
----------------------------
-- PITCH CHANNEL VARIABLES --
----------------------------
   ALPHA_TRIM_RATE : SCALAR_SINGLE := 0.0;
   ALPHA_TRIM_ERROR : SCALAR_SINGLE := 0.0;
   ALPHA_TRIM_ERROR_L : SCALAR_SINGLE := 0.0;
   Q_CMD : SCALAR_SINGLE := 0.0;
   Q_ERROR : SCALAR_SINGLE := 0.0;
   R_CMD : SCALAR_SINGLE := 0.0;
-------------------------
-- YAW CHANNEL VARIABLES --
-------------------------
   R_ERROR : SCALAR_SINGLE := 0.0;
   DP1 : SCALAR_SINGLE := 0.0;
-----------------------------
-- JET SELECT LOGIC VARIABLES --
-----------------------------
   DP2 : SCALAR_SINGLE := 0.0;
   DQ1 : SCALAR_SINGLE := 0.0;
   DQ2 : SCALAR_SINGLE := 0.0;
   DQ3 : SCALAR_SINGLE := 0.0;
   DQ4 : SCALAR_SINGLE := 0.0;
   DQ5 : SCALAR_SINGLE := 0.0;
   DQ6 : SCALAR_SINGLE := 0.0;
   DR1 : SCALAR_SINGLE := 0.0;
   DR2 : SCALAR_SINGLE := 0.0;
   DR3 : SCALAR_SINGLE := 0.0;
   DR4 : SCALAR_SINGLE := 0.0;
   DR5 : SCALAR_SINGLE := 0.0;
   DR6 : SCALAR_SINGLE := 0.0;
-- USE A MATH PACKAGE TAILORED TO PROVIDE THE PRECISION WE NEED
-- FOR THIS APPLICATION
   use SINGLE_PRECISION_MATRIX_OPERATIONS.REAL_MATH_LIB;
   use DOUBLE_PRECISION_MATRIX_OPERATIONS.REAL_MATH_LIB;
-------------------------------------------------------------
--  THE FOLLOWING PACKAGES CONTAIN PROCEDURES THAT ARE CALLED --
--  BY procedure AERO_DAP. THEY ARE POSITIONED EXTERNAL TO    --
--  PROCEDURE AERO_DAP SO THAT THEIR VARIABLES WILL EXIST     --
--  BEYOND THE TIME WHEN THE PROCEDURE IS EXECUTING           --
-------------------------------------------------------------

package BETA_FILTER_PACKAGE   is
```

```
    procedure BETA_FILTER;
end BETA_FILTER_PACKAGE;
package AERO_ANGLE_EXTRACT_PACKAGE   is
    procedure AERO_ANGLE_EXTRACT;
end AERO_ANGLE_EXTRACT_PACKAGE;
package TRANS_DELAY_COMP_PACKAGE   is
    procedure TRANS_DELAY_COMP;
end TRANS_DELAY_COMP_PACKAGE;
package STAB_AXES_CMD_PACKAGE    is
    procedure STAB_AXES_CMD;
end STAB_AXES_CMD_PACKAGE;
package JET_SELECT_LOGIC_PACKAGE   is
    procedure JET_SELECT_LOGIC;
end JET_SELECT_LOGIC_PACKAGE;
-----------------------------------
-- BODIES OF PACKAGES SPECIFIED ABOVE --
-----------------------------------
package body AERO_ANGLE_EXTRACT_PACKAGE is
    -----------------------------------
    -- LOCAL - POSITIONED HERE FOR DUMP --
    -----------------------------------
       UNIT_X_VR : SINGLE_PRECISION_VECTOR3;
       UNIT_Y_BODY_IN_INERTIAL : SINGLE_PRECISION_VECTOR3;
       UNIT_Y_VR : SINGLE_PRECISION_VECTOR3;
       UNIT_Z_DCL : SINGLE_PRECISION_VECTOR3;
       UNIT_Z_VR : SINGLE_PRECISION_VECTOR3;
       VREL_BODY : SINGLE_PRECISION_VECTOR3;
       procedure AERO_ANGLE_EXTRACT   is
       begin
       -----------------------------------
       -- RELATIVE VELOCITY IN BODY AXES --
       -----------------------------------
       VREL_BODY := Q_FORM(Q_POSE(Q_B_TO_I),DOUBLE_TO_SINGLE(V_REL_NAV));
       -------------------------
       -- ALPHA, BETA, AND PHI --
       -------------------------
       ALPHA_DAP := ARCTAN2(VREL_BODY(3),VREL_BODY(1)) * RAD_TO_DEG;
       BETA_DAP := SCALAR_SINGLE(ASIN(VREL_BODY(2) / V_REL_MAG) *
           RAD_TO_DEG);
       UNIT_Y_BODY_IN_INERTIAL := Q_FORM(Q_B_TO_I,Y_BODY);
       UNIT_X_VR := DOUBLE_TO_SINGLE(UNIT(V_REL_NAV));
```

```
    UNIT_Y_VR := DOUBLE_TO_SINGLE(UNIT(CROSS_PRODUCT(UNIT_X_VR,UNIT_R)))
       ;
    UNIT_Z_VR := UNIT(CROSS_PRODUCT(UNIT_X_VR,UNIT_Y_VR));
    UNIT_Z_DCL := UNIT(CROSS_PRODUCT(UNIT_Y_BODY_IN_INERTIAL,UNIT_X_VR))
       ;
    PHI_DAP := ARCTAN2(DOT_PRODUCT(UNIT_Z_DCL,UNIT_Y_VR),DOT_PRODUCT(
       UNIT_Z_DCL, -UNIT_Z_VR)) * RAD_TO_DEG;
    ----------------------------------------
    -- CALCULATE SINE AND COS OF ALPHA DAP --
    ----------------------------------------
    CALPHA := COS(ALPHA_DAP * DEG_TO_RAD);
    SALPHA := SIN(ALPHA_DAP * DEG_TO_RAD);
  end AERO_ANGLE_EXTRACT;
end AERO_ANGLE_EXTRACT_PACKAGE;

package body BETA_FILTER_PACKAGE is

  BETA_NODE : SCALAR_SINGLE := 0.0;
  FIRST_PASS : BOOLEAN_32 := TRUE;

  procedure BETA_FILTER   is

  begin
  ------------------------
  -- CALCULATE BETA_FCS --
  ------------------------
  if  (QBAR_NAV > QBAR_BETA_FILT_ON) then
    if FIRST_PASS then
      BETA_FCS := 0.0;
      FIRST_PASS := FALSE;

    else
      BETA_FCS := BETA_NODE * (K_BETA_FILT(1) * BETA_DAP);

    end if;
    BETA_NODE := (K_BETA_FILT(2) * BETA_DAP) * (K_BETA_FILT(3) *
       BETA_FCS);

  else
    BETA_FCS := BETA_DAP;

  end if;
  end BETA_FILTER;
end BETA_FILTER_PACKAGE;

package body TRANS_DELAY_COMP_PACKAGE is

------------------------------------------------------
-- LOCAL TO TRANS_DELAY_COMP - POSITIONED HERE FOR DUMP
------------------------------------------------------
  ROLL_ACCEL : SCALAR_SINGLE := 0.0;
  PITCH_ACCEL : SCALAR_SINGLE := 0.0;
  YAW_ACCEL : SCALAR_SINGLE := 0.0;

  procedure TRANS_DELAY_COMP    is

  begin
  ------------------------------------------------------------------------
  ----  TRANSPORT DELAY COMPENSATION TO BODY RATES - NEED TO ADD PRIME CO
  --MP -- ----------------------------------------------------------------
  ----------
  ROLL_ACCEL := ROLL_ACCEL_NOM * SIGNUM(KP_RCS);
  PITCH_ACCEL := PITCH_ACCEL_NOM * SIGNUM(KQ_RCS);
```

```
    YAW_ACCEL := YAW_ACCEL_NOM * SIGNUM(KR_RCS);
    P_FCS := BODY_RATE(1) * (ROLL_ACCEL * DT_AERODAP);
    Q_FCS := BODY_RATE(2) * (PITCH_ACCEL * DT_AERODAP);
    R_FCS := BODY_RATE(3) * (YAW_ACCEL * DT_AERODAP);
  end TRANS_DELAY_COMP;
end TRANS_DELAY_COMP_PACKAGE;

package body STAB_AXES_CMD_PACKAGE is

-------------------------------------------------------
-- LOCAL TO STAB_AXES_CMD - POSITIONED HERE FOR DUMP --
-------------------------------------------------------
   PHI_DELTA : SCALAR_SINGLE := 0.0;
   PHI_SHORTEST : SCALAR_SINGLE := 0.0;
   N_180 :  constant SCALAR_SINGLE := 180.0;
   N_360 :  constant SCALAR_SINGLE := 360.0;

   procedure STAB_AXES_CMD    is

   begin
   ------------------------------------------------------------------
   -- DETERMINE CORRECT BANK ERROR WITH CORRECT SIGN FOR ROLL --
   ------------------------------------------------------------------
   PHI_DELTA := PHI_CMD - PHI_DAP;
   if INTEGER'(SIGN(PHI_CMD)) = INTEGER'(SIGN(PHI_DAP)) then
      PHI_ERROR := PHI_DELTA;

   else
      if ( abs (PHI_DELTA) >= N_180) then
         PHI_SHORTEST := PHI_DELTA * (SIGN(PHI_DELTA) * N_360);

      else
         PHI_SHORTEST := PHI_DELTA;

      end if;
      if ( abs (PHI_SHORTEST) < DPHI_OVER_UNDER) then
         PHI_ERROR := PHI_SHORTEST;

      else
         if LIFT_DOWN_REVERSAL then
            PHI_ERROR := PHI_DELTA;

         else
            PHI_ERROR := PHI_DELTA * (SIGN(PHI_DELTA) * N_360);

         end if;

      end if;

   end if;
   ------------------------------------------------
   -- CALCULATE BANK AND SIDESLIP RATE COMMAND --
   ------------------------------------------------
   BANK_RATE_CMD := MIDVAL( -BANK_RATE_CMD_LIM,(K_PHI * PHI_ERROR),
      BANK_RATE_CMD_LIM);
   BETA_RATE_CMD := K_BETA * BETA_FCS;
  end STAB_AXES_CMD;
end STAB_AXES_CMD_PACKAGE;

package body JET_SELECT_LOGIC_PACKAGE is

------------------------------
-- LOCAL TO JET_SELECT_LOGIC --
```

```
-- POSITIONED HERE FOR DUMP  --
-------------------------------
   DP_ABS : SCALAR_SINGLE := 0.0;
   DQ_ABS : SCALAR_SINGLE := 0.0;
   DR_ABS : SCALAR_SINGLE := 0.0;
   DP_SIGN : INTEGER := 0;
   DQ_SIGN : INTEGER := 0;
   DR_SIGN : INTEGER := 0;
   KP_RCS_PAST : INTEGER := 0;
   KQ_RCS_PAST : INTEGER := 0;
   KR_RCS_PAST : INTEGER := 0;

   procedure JET_SELECT_LOGIC   is

   begin
   ---------------------
   -- JET LEVEL LOGIC --
   ---------------------
   -- RCS_ON := (others=>OFF);
   -- DP_ABS :=  abs (DP_CMD);
   -- DQ_ABS :=  abs (DQ_CMD);
   -- DR_ABS :=  abs (DR_CMD);
   -- DP_SIGN := SIGN(DP_CMD);
   -- DQ_SIGN := SIGN(DQ_CMD);
   -- DR_SIGN := SIGN(DR_CMD);
   -- KP_RCS_PAST := KP_RCS * DP_SIGN;
   -- KQ_RCS_PAST := KQ_RCS * DQ_SIGN;
   -- KR_RCS_PAST := KR_RCS * DR_SIGN;
   --------------------------
   -- DETERMINE JET LEVELS --
   --------------------------
   ------------------------------------------------------------------------
   -- HAS 1 LEVEL OF MOMENT FOR ROLL AND 3 LEVELS FOR PITCH AND YAW --
   ------------------------------------------------------------------------
   ------------------
   -- ROLL CHANNEL --
   ------------------
   <> if ((DP_ABS >= DP2) or ((DP_ABS >= DP1) and (KP_RCS_PAST >= 1)))
         then
      -- KP_RCS := DP_SIGN;

      else
      -- KP_RCS := 0;

      end if;
   -------------------
   -- PITCH CHANNEL --
   -------------------
   <> if ((DQ_ABS >= DQ2) or else ((DQ_ABS >= DQ1) and (KQ_RCS_PAST >= 1))
         ) then
      -- KQ_RCS := DQ_SIGN;
      <> if ((DQ_ABS >= DQ4) or else ((DQ_ABS >= DQ3) and (KQ_RCS_PAST >=
            2))) then
      -- KQ_RCS := 2 * DQ_SIGN;

      <> elsif ((DQ_ABS >= DQ6) or else ((DQ_ABS >= DQ5) and (KQ_RCS_PAST
            >= 3))) then
      -- KQ_RCS := 3 * DQ_SIGN;

      end if;

      else
      -- KQ_RCS := 0;
```


end if;
-- YAW CHANNEL --
if ((DR_ABS >= DR2) or else ((DR_ABS >= DR1) and (KR_RCS_PAST >= 1))
) then
KR_RCS := DR_SIGN;
if ((DR_ ABS >= DR4) or else ((DR_ABS >= DR3) and (KR_RCS_PAST >=
2))) then
KR_RCS := 2 * DR_SIGN;
elsif ((DR_ABS >= DR6) or else ((DR_ABS >= DR5) and (KR_RCS_PAST
>= 3))) then
KR_RCS := 3 * DR_SIGN;
end if;
else
KR_RCS := 0;
end if;
-- JET SELECT LOGIC --
-- ROLL CHANNEL --
if (KP_RCS /= 0) then
if (KP_RCS > 0) then
RCS_ON(1) := ON;
RCS_ON(2) := ON;
else
RCS_ON(3) := ON;
RCS_ON(4) := ON;
end if;
end if;
-- PITCH CHANNEL --
if (KQ_RCS /= 0) then
if (KQ_RCS > 0) then
if ((KQ_RCS = 1) or (KQ_RCS = 3)) then
RCS_ON(5) := ON;
end if;
if (KQ_RCS >= 2) then
RCS_ON(9) := ON;
end if;
else
if ((KQ_RCS = -1) or (KQ_RCS = -3)) then
RCS_ON(6) := ON;
end if;
if (KQ_RCS <= -2) then
RCS_ON(10) := ON;
end if;

```
      end if;
    end if;
    -----------------
    -- YAW CHANNEL --
    -----------------
    if (KR_RCS /= 0) then
      if (KR_RCS > 0) then
        if ((KR_RCS = 1) or (KR_RCS = 3)) then
          RCS_ON(7) := ON;
        end if;
        if (KR_RCS >= 2) then
          RCS_ON(11) := ON;
        end if;
      else
        if ((KR_RCS =  -1) or (KR_RCS =  -3)) then
          RCS_ON(8) := ON;
        end if;
        if (KR_RCS <=  -2) then
          RCS_ON(12) := ON;
        end if;
      end if;
    end if;
  end JET_SELECT_LOGIC;
  -------------------------------------
  -- DON'T TURN ON TWO OPPOSING JETS --
  -------------------------------------
  ------------------------------------------------------------------------
  -- NOT CURRENTLY POSSIBLE - CODE LEFT AS REMINDER OF LEVEL B SPEC --
  ------------------------------------------------------------------------
  -- IF (RCS_ON$(1:) = ON) and (RCS_ON$(3:) = ON) THEN
  --         RCS_ON$(1:),RCS_ON$(3:) = OFF;
  -- IF (RCS_ON$(2:) = ON) and (RCS_ON$(4:) = ON) THEN
  --         RCS_ON$(2:),RCS_ON$(4:) = OFF;
end JET_SELECT_LOGIC_PACKAGE;
--*********************************--
  use BETA_FILTER_PACKAGE;
  use AERO_ANGLE_EXTRACT_PACKAGE;
  use TRANS_DELAY_COMP_PACKAGE;
  use STAB_AXES_CMD_PACKAGE;
  use JET_SELECT_LOGIC_PACKAGE;
--*********************************--
  procedure AERO_DAP        is
  ----------------------
  -- LOCAL PROCEDURES --
  ----------------------
    procedure AERO_DAP_INIT;
    procedure BODY_AXES_CMD;
```

```
procedure BODY_AXES_CMD    is
begin
----------------------
-- DAP ROLL CHANNEL --
----------------------
   P_CMD := (BANK_RATE_CMD * CALPHA) * (BETA_RATE_CMD * SALPHA);
   P_ERROR := P_CMD - P_FCS;
   DP_CMD := K_P * P_ERROR;
-----------------------
-- DAP PITCH CHANNEL --
-----------------------
   ALPHA_TRIM_CMD := ALPHA_CMD - TRIM_ERROR_L;
   ALPHA_TRIM_ERROR := ALPHA_TRIM_CMD - ALPHA_DAP;
   ALPHA_TRIM_ERROR_L := MIDVAL( -ALPHA_ERROR_LIM,ALPHA_TRIM_ERROR,
      ALPHA_ERROR_LIM);
   Q_CMD := K_ALPHA * ALPHA_TRIM_ERROR_L;
   Q_ERROR := Q_CMD - Q_FCS;
   DQ_CMD := K_Q * Q_ERROR;
   TRIM_ERROR_L := TRIM_ERROR_L * (K_ALPHA_TRIM * Q_ERROR * DT_AERODAP)
      ;
   TRIM_ERROR_L := MIDVAL( -TRIM_ERROR_LIM,TRIM_ERROR_L,TRIM_ERROR_LIM)
      ;
---------------------
-- DAP YAW CHANNEL --
---------------------
   R_CMD := (BETA_RATE_CMD * CALPHA) * (BANK_RATE_CMD * SALPHA);
   R_ERROR := R_CMD - R_FCS;
   DR_CMD := K_R * R_ERROR;
end BODY_AXES_CMD;
procedure AERO_DAP_INIT    is
begin
------------------
-- COPY I-LOADS --
------------------
   DP1 := DP1_AERO;
   DQ1 := DQ1_AERO;
   DR1 := DR1_AERO;
   DP2 := DP2_AERO;
   DQ2 := DQ2_AERO;
   DR2 := DR2_AERO;
   DQ3 := DQ3_AERO;
   DR3 := DR3_AERO;
   DQ4 := DQ4_AERO;
   DR4 := DR4_AERO;
   DQ5 := DQ5_AERO;
   DR5 := DR5_AERO;
   DQ6 := DQ6_AERO;
   DR6 := DR6_AERO;
end AERO_DAP_INIT;
begin
--****************************--
-- BODY OF PROCEDURE AERO_DAP --
--****************************--
------------------------
-- AERO_DAP EXECUTIVE --
------------------------
if FIRST_PASS then
   AERO_DAP_INIT;
```

```
    FIRST_PASS := FALSE;
  end if;
AERO_ANGLE_EXTRACT;
BETA_FILTER;
TRANS_DELAY_COMP;
STAB_AXES_CMD;
BODY_AXES_CMD;
JET_SELECT_LOGIC;
------------------------------------------------------
-- COPY CYCLES FOR PLOTTING IN EDITOR - NOT DAP CODE --
------------------------------------------------------
------------------------
-- GENERAL VARIABLES --
------------------------
ALPHA_EDIT := ALPHA_DAP;
BANK_RATE_CMD_EDIT := BANK_RATE_CMD;
BETA_EDIT := BETA_DAP;
BETA_FCS_EDIT := BETA_FCS;
BETA_RATE_CMD_EDIT := BETA_RATE_CMD;
PHI_EDIT := PHI_DAP;
PHI_ERROR_EDIT := PHI_ERROR;
-------------------------------------------
-- TRANSPORT DELAY COMPENSATED BODY RATES --
-------------------------------------------
BODY_RATE_FCS_EDIT(1) := P_FCS;
BODY_RATE_FCS_EDIT(2) := Q_FCS;
BODY_RATE_FCS_EDIT(3) := R_FCS;
---------------
-- ROLL AXIS --
---------------
ATT_ERROR_EDIT(1) := PHI_ERROR;
DP_CMD_EDIT := DP_CMD;
P_ERROR_EDIT := P_ERROR;
PC_EDIT := P_CMD;
----------------
-- PITCH AXIS --
----------------
ALPHA_TRIM_CMD_EDIT := ALPHA_TRIM_CMD;
ALPHA_TRIM_ERROR_EDIT := ALPHA_TRIM_ERROR;
ALPHA_TRIM_RATE_EDIT := ALPHA_TRIM_RATE;
ATT_ERROR_EDIT(2) := ALPHA_TRIM_ERROR_L;
DQ_CMD_EDIT := DQ_CMD;
Q_ERROR_EDIT := Q_ERROR;
QC_EDIT := Q_CMD;
TRIM_ERROR_L_EDIT := TRIM_ERROR_L;
--------------
-- YAW AXIS --
--------------
ATT_ERROR_EDIT(3) :=  -BETA_FCS;
DR_CMD_EDIT := DR_CMD;
```

```
   |   |-- R_ERROR_EDIT := R_ERROR;
   |   |-- RC_EDIT := R_CMD;
   |   |-------------------
   |   |-- JSL VARIABLES --
   |   |-------------------
   |   |-- KP_RCS_EDIT := KP_RCS;
   |   |-- KQ_RCS_EDIT := KQ_RCS;
   |   |-- KR_RCS_EDIT := KR_RCS;
   |   end AERO_DAP;
   end AERO_DAP_PACKAGE;
```

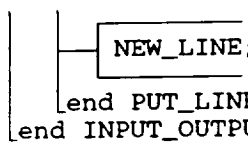
```
with system;
use system;
with component_types;
use component_types;
with logical;
use logical;
with b1553_bc;
use b1553_bc;
with unchecked_conversion;

package body B1553_COMPONENT_DATA is

   data: arr_64;
   data_msg: arr_64;
   DATA_MSG2: ARR_64;
   stat_arr: arr1;
   msg_count: integer;
--  A_cmd:     UNSIGNED_WORD;
--  A_cmdlbk:  UNSIGNED_WORD;
--  A_stat:    UNSIGNED_WORD;
   msg_arr: arr_59_65;
   nmsg: integer;
   wdcount: arr_32;
   bc_interrupt_status: unsigned_word := 16#75#;
--  package int_io is new INTEGER_IO(INTEGER);
--  use int_io;
-------------------------------------------------------------------------

   procedure B1553_IMU_INTRP  is

   begin
   ----------------------------------------------------------------------
   -- Message 1 --
   -- Set up IMU 40 msec interrupt - Data Ready Signal --
   bc_interrupt_status := unsigned_word(16#75#);
   while  (short_and(bc_interrupt_status,16#74#) /= 16#0000#) loop
      data_msg(1) := 16#0001#;
      -- Even and Odd frame data  --
      data_msg(2) := 16#1000#;
      -- BIT 12 DATA READY SIGNAL - 40 MSEC  --
      data_msg(3) := 16#0000#;

      bc_store_msg(0,2,3,0,3,data_msg);

      -- Data word - RT 2 Subadd 3  --
      -- rcv 3 data words           --

      BC_GO;


      BC_INTERRUPT(bc_interrupt_status);

   end loop;
   -- Wait for BC interrupt then --
   -- change buffer  --
   --   put(" bc_interrupt_status = ");
   --   put(integer(bc_interrupt_status),4,16);
   --   new_line;
   end B1553_IMU_INTRP;
-- end bc_interrupt_status loop --
-- Timeout/1553 format error; buffer overflow;--
-- loop test fail; status set --
-- End Message 1 --
```

```
-------------------------------------------------------------------------
-------------------------------------------------------------------------
procedure B1553_IMU_INIT  is
begin
-------------------------------------------------------------------------
-- Message 2 --
-- Set up IMU Quaternion Initialization  --
bc_interrupt_status := unsigned_word(16#75#);
while  (short_and(bc_interrupt_status,16#74#) /= 16#0000#) loop
   data_msg(1) := 16#0001#;
   -- Even and Odd frame data  --
   data_msg(2) := 16#1002#;
   -- BIT 12 DATA READY SIGNAL, BIT 1 RESET --
   --   QUATERNION TO (1,0,,0,0)  --
   data_msg(3) := 16#0000#;
   bc_store_msg(0,2,3,0,3,data_msg);
   -- Data word - RT 2 Subadd 3  --
   -- rcv 3 data words           --
   BC_GO;
   BC_INTERRUPT(bc_interrupt_status);
end loop;
-- Wait for BC interrupt then --
-- change buffer  --
--   put(" bc_interrupt_status = ");
--   put(integer(bc_interrupt_status),4,16);
--   new_line;
end B1553_IMU_INIT;
-- end bc_interrupt_status loop --
-- Timeout/1553 format error; buffer overflow;--
-- loop test fail; status set --
-- End Message 2 --
-------------------------------------------------------------------------
-------------------------------------------------------------------------
procedure READ_IMU_DATA(IMU_DATA: out ARR_32)    is
begin
   bc_interrupt_status := unsigned_word(16#75#);
   while (short_and(bc_interrupt_status,16#74#) /= 16#0000#) loop
      bc_store_msg(0,2,2,1,32,data_msg);
      -- Data word - Rt 2 Subaddr 2 --
      -- xmit 32 data words         --
      -- EVEN Frame Data - Subaddr 2 --
      bc_go;
      bc_interrupt(bc_interrupt_status);
   end loop;
--   put(" bc_interrupt_status = ");
--   put(integer(bc_interrupt_status),4,16);
--   new_line;
```

```
   -- end bc_interrupt_status loop --
   -- Timeout/1553 format error; buffer overflow;--
   -- loop test fail; status set --
   ------------------------------------------------------------------------
   --  BC_status(A_cmd,A_cmdlbk,A_stat,1);
   --   put (" A_cmd = ");      put(integer(A_cmd),4,16);
   --   put (" A_cmdlbk = ");   put(integer(A_cmdlbk),4,16);
   --   put (" A_stat = ");     put(integer(A_stat),4,16);
   --   new_line;
   BC_get_msg(msg_arr);
   --      msg_count := integer(msg_arr(1,1));
   --    put (" Message count = ");
   --    put(msg_count,4,16);
   --    new_line;
   --    put (" Message = ");
   --    new_line;
   for i in 1..32 loop
      imu_data(i) := msg_arr(1,i + 1);
   end loop;
end READ_IMU_DATA;
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------
procedure THRUSTER_INIT  is
begin
-- Clear thrusters in Message 2 --
data_msg2(1) := 16#0000#;
data_msg2(2) := 16#0000#;
data_msg2(3) := 16#0000#;
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
-- THRUSTERS INITIALIZED TO ALL OFF CONDITION  ---------
bc_interrupt_status := unsigned_word(16#75#);
while (short_and(bc_interrupt_status,16#74#) /= 16#0000#) loop
   bc_store_msg(0,3,2,0,3,data_msg2);
   -- Data word - Rt 3 Subaddr 2 --
   -- rcv  3 data words          --
   bc_go;
   bc_interrupt(bc_interrupt_status);
end loop;
end THRUSTER_INIT;
-- end bc_interrupt_status loop --
-- Timeout/1553 format error; buffer overflow;--
-- loop test fail; status set --
-- End Message 2 --
end B1553_COMPONENT_DATA;
------------------------------------------------------------------------
------------------------------------------------------------------------
```

```
package body INPUT_OUTPUT_PACKAGE is

   use SCALAR_SINGLE_IO;
   use SCALAR_DOUBLE_IO;

   procedure PUT_LINE (X: SINGLE_PRECISION_VECTOR) is

   begin
      for I in X'FIRST..X'LAST loop
         PUT(X(I));
      end loop;
      NEW_LINE;
   end PUT_LINE;

   procedure PUT_LINE (X: DOUBLE_PRECISION_VECTOR) is

   begin
      for I in X'FIRST..X'LAST loop
         PUT(X(I));
      end loop;
      NEW_LINE;
   end PUT_LINE;

   procedure PUT_LINE (MAT: SINGLE_PRECISION_MATRIX) is

   begin
      for I in MAT'FIRST(1)..MAT'LAST(1) loop
         for J in MAT'FIRST(2)..MAT'LAST(2) loop
            PUT(MAT(I,J));
         end loop;
         NEW_LINE;
      end loop;
      NEW_LINE;
   end PUT_LINE;

   procedure PUT_LINE (MAT: DOUBLE_PRECISION_MATRIX) is

   begin
      for I in MAT'FIRST(1)..MAT'LAST(1) loop
         for J in MAT'FIRST(2)..MAT'LAST(2) loop
            PUT(MAT(I,J));
         end loop;
         NEW_LINE;
      end loop;
```

C-3


NEW_LINE;
end PUT_LINE;
end INPUT_OUTPUT_PACKAGE;

```
with LEVEL_A_CONSTANTS;
use LEVEL_A_CONSTANTS;
with DATA_TYPES;
use DATA_TYPES;
with FSW_POOL;
use FSW_POOL;
with IL_POOL;
use IL_POOL;
with TEXT_IO;
use TEXT_IO;
with INPUT_OUTPUT_PACKAGE;
use INPUT_OUTPUT_PACKAGE;
with MATH_PACKAGE;
use MATH_PACKAGE;
with QUATERNION_OPERATIONS;
use QUATERNION_OPERATIONS;
with SINGLE_PRECISION_MATRIX_OPERATIONS;
use SINGLE_PRECISION_MATRIX_OPERATIONS;
with DOUBLE_PRECISION_MATRIX_OPERATIONS;
use DOUBLE_PRECISION_MATRIX_OPERATIONS;

package body PRED_GUID_PACKAGE is

   APOGEE_EPSILON1 : SCALAR_SINGLE := 25.0;
-------------------------------------------------------------
-- FUNCTION: NUMERIC PREDICTOR/CORRECTOR AEROBRAKING GUIDANCE --
-------------------------------------------------------------
-----------------------------------------------------
-- ILOADS - MOVE TO ILPOOL IF RETAIN THIS ALGORITHM? --
-----------------------------------------------------
   APOGEE_EPSILON2 : SCALAR_SINGLE := 1.0;
   BANK_MAX : SCALAR_SINGLE := 165.0;
   BANK_MIN : SCALAR_SINGLE := 15.0;
   CORRIDOR_MIN :  constant SCALAR_SINGLE := 0.05;
   CORRIDOR_V_MAX :  constant SCALAR_SINGLE := 34_000.0;
   CORRIDOR_V_MIN :  constant SCALAR_SINGLE := 26500.0;
   DELTA_PHI_MIN : SCALAR_SINGLE := 1.0;
   DELTA_T_PRED :  constant SCALAR_SINGLE := 2.0;
   G_RUN_GUIDANCE : SCALAR_SINGLE := 0.075;
   GUID_PASS_LIM :  constant INTEGER := 10;
   LIFT_INC_CAPTURE : SCALAR_SINGLE := 0.15;
   LIFT_PERCENT_CAPTURE : SCALAR_SINGLE := 0.5;
   MAX_NUMBER_RUNS :  constant INTEGER := 5;
   PHI_LIFT_DOWN :  constant SCALAR_SINGLE := 45.0;
   VI_LIFT_DOWN :  constant SCALAR_SINGLE := 27500.0;
   VI_MODEL_LIFT_DOWN :  constant SCALAR_SINGLE := 27900.0;
   COS_PHI_MAX : SCALAR_SINGLE := 0.0;
---------------------
-- LOCAL VARIABLES --
---------------------
   COS_PHI_MIN : SCALAR_SINGLE := 0.0;
   GUID_PASS : INTEGER := 0;
   INITIALIZE_GUIDANCE : BOOLEAN_32 := TRUE;
   MODEL_LIFT_DOWN : BOOLEAN_32 := TRUE;
   PHI_CMD_NS : SCALAR_SINGLE := 0.0;
   SIGN_OF_BANK : SCALAR_SINGLE := 0.0;
   FIRST_TIME_CALLED : BOOLEAN_32 := TRUE;
   EARTH_POLE : DOUBLE_PRECISION_VECTOR3 := (others=>0.0);
   EARTH_OMEGA : DOUBLE_PRECISION_VECTOR3 := (others=>0.0);
   ZERO :  constant SCALAR_SINGLE := 0.0;
-------------------------------------------------------------
-- NUMERICAL CONSTANTS USED IN PACKAGE                     --
-- This is necessary because of the overloading of operator --
```

```
-- symbols to allow mixed mode arithmetic between single-   --
-- precision and double-precision variables.                --
------------------------------------------------------------
   ONE_TENTH :  constant SCALAR_SINGLE := 0.1;
   ONE_HALF :  constant SCALAR_SINGLE := 0.5;
   ONE:  constant SCALAR_SINGLE := 1.0;
   TWO :  constant SCALAR_SINGLE := 2.0;
   THREE :  constant SCALAR_SINGLE := 3.0;
   FIVE :  constant SCALAR_SINGLE := 5.0;
   N25_000 :  constant SCALAR_SINGLE := 25000.0;
   N26_000 :  constant SCALAR_SINGLE := 26000.0;
   N27_000 :  constant SCALAR_SINGLE := 27000.0;
   N29_000 :  constant SCALAR_SINGLE := 29000.0;
   N30_000 :  constant SCALAR_SINGLE := 30000.0;
   N33_850 :  constant SCALAR_SINGLE := 33850.0;
   N150_000 :  constant SCALAR_SINGLE := 150_000.0;
   N400_000 :  constant SCALAR_SINGLE := 400_000.0;
-------------------------------------------------
-- USE OUTPUT ROUTINES FROM INPUT_OUTPUT_PACKAGE --
-------------------------------------------------
   use INPUT_OUTPUT_PACKAGE.INT_IO;
   use INPUT_OUTPUT_PACKAGE.SCALAR_SINGLE_IO;
-------------------------------------------------------------------
-- USE A MATH PACKAGE TAILORED TO PROVIDE THE PRECISION WE NEED --
-- FOR THIS APPLICATION                                         --
-------------------------------------------------------------------
   use SINGLE_PRECISION_MATRIX_OPERATIONS.REAL_MATH_LIB;
   use DOUBLE_PRECISION_MATRIX_OPERATIONS.REAL_MATH_LIB;
--------------------
-- LOCAL FUNCTION --
--------------------
   function ALTITUDE (R: DOUBLE_PRECISION_VECTOR3) return SCALAR_DOUBLE ;
-------------------------------------------------------------------
-- THE FOLLOWING PACKAGES CONTAIN PROCEDURES THAT ARE CALLED BY   --
-- procedure PRED_GUID. THEY ARE POSITIONED EXTERNAL TO procedure --
-- PRED_GUID SO THAT THEIR VARIABLES WILL EXIST BEYOND THE TIME   --
-- WHEN THE PROCEDURE IS EXECUTING.                               --
-------------------------------------------------------------------
   package PC_SEQUENCER_PACKAGE   is
      procedure PC_SEQUENCER;
   end PC_SEQUENCER_PACKAGE;
   package LATERAL_CONTROL_PACKAGE   is
      procedure LATERAL_CONTROL;
   end LATERAL_CONTROL_PACKAGE;
   use PC_SEQUENCER_PACKAGE;
   use LATERAL_CONTROL_PACKAGE;
-------------------------------------
-- BODY OF FUNCTION SPECIFIED ABOVE --
-------------------------------------
--*****************************************************************--
   function ALTITUDE (R: DOUBLE_PRECISION_VECTOR3) return SCALAR_DOUBLE  is
```

```
   --*************************************************************************--
      RM : SCALAR_DOUBLE;
   begin
   ------------------------------------------------
   -- COMPUTES THE ALTITUDE ABOVE FISCHER ELLIPSOID --
   ------------------------------------------------
   -- RM := VECTOR_LENGTH(R);
   -- return (RM / EARTH_R - (ONE - EARTH_FLAT) / SQRT(ONE / ((ONE -
         EARTH_FLAT)**2 - ONE) / (ONE / (DOT_PRODUCT((R / RM),EARTH_POLE)**2)
         )));
   end ALTITUDE;
-------------------------------------- BODIES OF PACKAGES SPECIFIED ABOVE
---- ------------------------------------ **********************************
--*************************************************************************
--********--

  package body PC_SEQUENCER_PACKAGE is

   ------------------------------------------------
   -- LOCAL VARIABLES - POSITIONED HERE FOR DUMP --
   ------------------------------------------------
      APOGEE_BRACKET : array(1..2) of SCALAR_SINGLE;
      APOGEE_EPSILON : SCALAR_SINGLE;
      APOGEE_EXTRAPOLATE : array(1..2) of SCALAR_SINGLE;
      APOGEE_PREDICTED : SCALAR_SINGLE;
      BRACKETED : BOOLEAN_32;
      COS_CAPT : SCALAR_SINGLE;
      COS_BRACKET : array(1..2) of SCALAR_SINGLE;
      COS_EXTRAPOLATE : array(1..2) of SCALAR_SINGLE;
      COS_PHI_TRY : array(1..10) of SCALAR_SINGLE;
      DELTA_APOGEE : SCALAR_SINGLE;
      DELTA_PHI : SCALAR_SINGLE;
      I : INTEGER;
      INTEG_LOOP : INTEGER range 1..4;
      NUMBER_CAPT : INTEGER;
      NUMBER_GOOD : INTEGER;
      NUMBER_HIGH : INTEGER;
      NUMBER_LOW : INTEGER;
      PHI_TRY : SCALAR_SINGLE;
      PHI_TRY_LAST : SCALAR_SINGLE;
      PRED_CAPTURE : BOOLEAN_32;
   ------------------------------------------------------------------------
   -- LOCAL PROCEDURES CALLED BY procedure PC_SEQUENCER.                 --
   -- APPEAR HERE IN PACKAGE FORMAT SO THAT VARIABLES WILL BE AVAILABLE  --
   -- FOR DUMPS AND SO THAT VARIABLE VALUES WILL EXIST BETWEEN INVOCATIONS --
   -- OF THESE PROCEDURES BY procedure PC_SEQUENCER.                     --
   ------------------------------------------------------------------------

     package PREDICTOR_PACKAGE   is

       procedure PREDICTOR;

     end PREDICTOR_PACKAGE;

     package CORRECTOR_PACKAGE   is

       procedure CORRECTOR;

     end CORRECTOR_PACKAGE;
     use PREDICTOR_PACKAGE;
     use CORRECTOR_PACKAGE;
```

```
--*****************************************************************************
--*********************************************--

package body PREDICTOR_PACKAGE is

   -------------------------------------------------
   -- LOCAL TO PREDICTOR - POSITIONED HERE FOR DUMP --
   -------------------------------------------------
      A_PRED : DOUBLE_PRECISION_VECTOR3;
      ALT_PRED : SCALAR_DOUBLE;
      GAMMA_PRED : SCALAR_SINGLE;
      LOD_PRED : SCALAR_SINGLE;
      PHI_PRED : SCALAR_SINGLE;
      R_PRED : DOUBLE_PRECISION_VECTOR3;
      R_MAG_PRED : SCALAR_DOUBLE;
      RDDOT_PRED : SCALAR_SINGLE;
      RDOT_PRED : SCALAR_SINGLE;
      T_PRED : SCALAR_DOUBLE;
      V_MAG_PRED : SCALAR_DOUBLE;
      V_PRED : DOUBLE_PRECISION_VECTOR3;
   ------------------------------------------------------------------------
   -- INTEGRATOR PROCEDURE CALLED BY procedure PREDICTOR.                --
   -- APPEARS HERE AS A PACKAGE SO THAT ITS VARIABLES WILL RETAIN        --
   -- THEIR VALUES BETWEEN INVOCATIONS OF THE PROCEDURE BY PREDICTOR.    --
   ------------------------------------------------------------------------

      package INTEGRATOR_PACKAGE   is

         procedure INTEGRATOR;

      end INTEGRATOR_PACKAGE;
   --*****************************************************************************
   --*********************************************--

      package body INTEGRATOR_PACKAGE is

      ------------------------------------------------------------------------
      ----  VARIABLES ARE DECLARED AND POSITIONED HERE SO THAT THEIR VALUE
      --S  --  WILL EXIST FROM INVOCATION TO INVOCATION OF procedure INTEG
      --RATOR -- ------------------------------------------------------------
      -------------
         ACCUM_ACCEL : DOUBLE_PRECISION_VECTOR3;
         ACCUM_VEL : DOUBLE_PRECISION_VECTOR3;
         ORIG_POS : DOUBLE_PRECISION_VECTOR3;
         ORIG_VEL : DOUBLE_PRECISION_VECTOR3;
      --********************--

         procedure INTEGRATOR is

         --********************--
         begin
            case INTEG_LOOP is
               when 1 =>
                  ORIG_POS := R_PRED;
                  ORIG_VEL := V_PRED;
                  ACCUM_VEL := V_PRED;
                  ACCUM_ACCEL := A_PRED;
                  R_PRED := ORIG_POS * ONE_HALF * DELTA_T_PRED * V_PRED;
                  V_PRED := ORIG_VEL * ONE_HALF * DELTA_T_PRED * A_PRED;

               when 2 =>
                  ACCUM_VEL := ACCUM_VEL * TWO * V_PRED;
```

```
                    ACCUM_ACCEL := ACCUM_ACCEL * TWO * A_PRED;
                    R_PRED := ORIG_POS * ONE_HALF * DELTA_T_PRED * V_PRED;
                    V_PRED := ORIG_VEL * ONE_HALF * DELTA_T_PRED * A_PRED;
                when 3 =>
                    ACCUM_VEL := ACCUM_VEL * TWO * V_PRED;
                    ACCUM_ACCEL := ACCUM_ACCEL * TWO * A_PRED;
                    R_PRED := ORIG_POS * DELTA_T_PRED * V_PRED;
                    V_PRED := ORIG_VEL * DELTA_T_PRED * A_PRED;
                when 4 =>
                    R_PRED := ORIG_POS / (ACCUM_VEL + V_PRED) * DELTA_T_PRED
                        / 6.0;
                    V_PRED := ORIG_VEL / (ACCUM_ACCEL + A_PRED) *
                        DELTA_T_PRED / 6.0;
                when others  =>
                -- INTEG_LOOP can only have values in the range 1..4
                    null;
            end case;
        end INTEGRATOR;
    end INTEGRATOR_PACKAGE;
    use INTEGRATOR_PACKAGE;
--********************--

    procedure PREDICTOR is

    --********************--
    begin
    ----------------------------------------
    -- INITIALIZE PREDICTOR STATE VECTOR --
    ----------------------------------------
    R_PRED := R_NAV;
    R_MAG_PRED := VECTOR_LENGTH(R_PRED);
    ALT_PRED := ALTITUDE(R_PRED);
    V_PRED := V_NAV;
    V_MAG_PRED := VECTOR_LENGTH(V_PRED);
    PHI_PRED := PHI_TRY * SIGN_OF_BANK;
    T_PRED := T_GMT;
    LOD_PRED := CL_NAV / CD_NAV;
    PRED_CAPTURE := FALSE;
    --------------------
    -- PREDICTOR LOOP --
    --------------------
    for TIME_INCREMENT in 1..750 loop
        ------------------------------------------
        -- 4TH ORDER RUNGA_KUTTA INTEGRATION LOOP --
        ------------------------------------------
        for INDEX in 1..4 loop
            INTEG_LOOP := INDEX;
            declare
                AERO_ACCEL : DOUBLE_PRECISION_VECTOR3;
                ALT_NORM_PRED : SCALAR_SINGLE;
                CPHI : SCALAR_SINGLE;
                DRAG_ACCEL : SCALAR_SINGLE;
                GRAV_ACCEL : DOUBLE_PRECISION_VECTOR3;
                HS_NORM_PRED : SCALAR_SINGLE;
                I_LAT : DOUBLE_PRECISION_VECTOR3;
                I_LIFT : DOUBLE_PRECISION_VECTOR3;
                I_VEL : DOUBLE_PRECISION_VECTOR3;
                LIFT_ACCEL : SCALAR_SINGLE;
                RHO_EST : SCALAR_SINGLE;
```

```
   RHO_NOM : SCALAR_SINGLE;
   SPHI : SCALAR_SINGLE;
   U_PRED : DOUBLE_PRECISION_VECTOR3;
   V_REL_MAG_PRED : SCALAR_DOUBLE;
   V_REL_PRED : DOUBLE_PRECISION_VECTOR3;
   Z_PRED : SCALAR_DOUBLE;
begin
----------------------
-- RELATIVE VELOCITY --
----------------------
V_REL_PRED := V_PRED - CROSS_PRODUCT(EARTH_OMEGA,R_PRED)
   ;
V_REL_MAG_PRED := VECTOR_LENGTH(V_REL_PRED);
-------------------------------------
-- 1962 STANDARD ATMOSPHERE CURVE FIT --
-------------------------------------
ALT_NORM_PRED := SCALAR_SINGLE(ALT_PRED / H_REF);
HS_NORM_PRED :=  (((C_HS(5) * ALT_NORM_PRED + C_HS(4)) *
   ALT_NORM_PRED + C_HS(3)) * ALT_NORM_PRED + C_HS(2)) *
   ALT_NORM_PRED + C_HS(1);
RHO_NOM := RHO_REF / EXP((ONE - ALT_NORM_PRED) /
   HS_NORM_PRED);
-----------------------
-- ESTIMATED DENSITY --
-----------------------
RHO_EST := K_RHO_NAV * RHO_NOM;
--------------------
-- LIFTDOWN MODEL --
--------------------
if MODEL_LIFT_DOWN = TRUE and V_MAG_PRED < VI_LIFT_DOWN
   then
   PHI_PRED := PHI_LIFT_DOWN * SIGN_OF_BANK;
end if;
CPHI := COS(PHI_PRED * DEG_TO_RAD);
SPHI := SIN(PHI_PRED * DEG_TO_RAD);
------------------------------
-- AERODYNAMIC ACCELERATIONS --
------------------------------
DRAG_ACCEL := SCALAR_SINGLE((ONE_HALF * RHO_EST *
   V_REL_MAG_PRED**2 * CD_NAV * S_REF) / MASS_NAV);
LIFT_ACCEL := LOD_PRED * DRAG_ACCEL;
I_VEL := V_REL_PRED / V_REL_MAG_PRED;
I_LAT := UNIT(CROSS_PRODUCT(I_VEL,R_PRED));
I_LIFT := UNIT(CROSS_PRODUCT(I_LAT,I_VEL)) * CPHI *
   I_LAT * SPHI;
AERO_ACCEL := LIFT_ACCEL * I_LIFT * DRAG_ACCEL * I_VEL;
-------------------------------------
-- GRAVITY ACCELERATION WITH J2 TERM --
-------------------------------------
U_PRED := R_PRED / R_MAG_PRED;
Z_PRED := DOT_PRODUCT(U_PRED,EARTH_POLE);
U_PRED := U_PRED * (THREE * EARTH_J2 / TWO) / (EARTH_R /
   R_MAG_PRED)**2 * ((ONE * FIVE * Z_PRED**2) * U_PRED *
   TWO * Z_PRED * EARTH_POLE);
GRAV_ACCEL :=  -(EARTH_MU / R_MAG_PRED**2) * U_PRED;
----------------------
-- TOTAL ACCELERATION --
----------------------
A_PRED := AERO_ACCEL + GRAV_ACCEL;
-----------------------------
-- CALL RUNGA_KUTTA INTEGRATOR --
-----------------------------
```

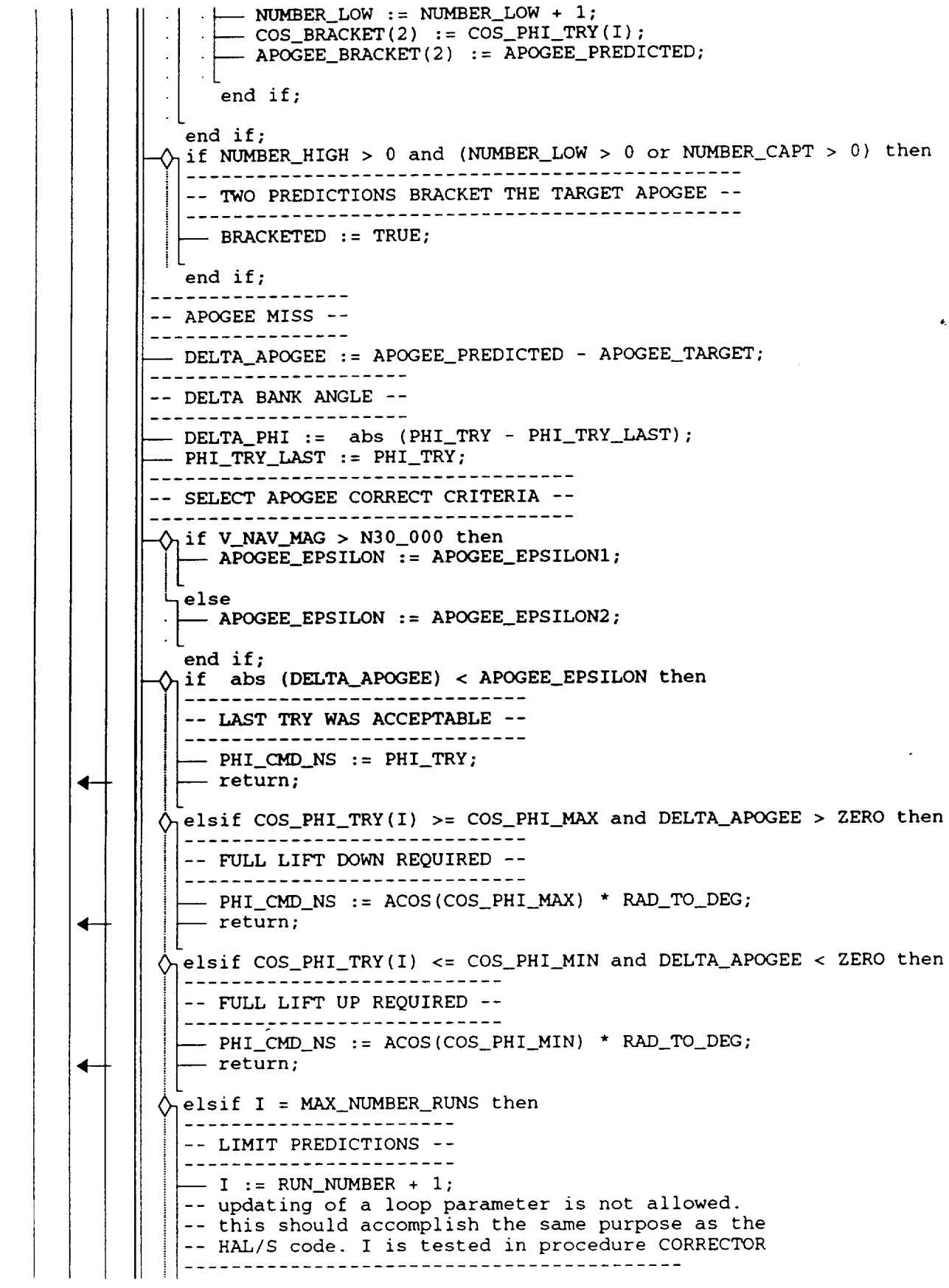
```
                INTEGRATOR;
                ----------------------
                -- STATE PARAMETERS --
                ----------------------
                R_MAG_PRED := VECTOR_LENGTH(R_PRED);
                V_MAG_PRED := VECTOR_LENGTH(V_PRED);
                --------------------------
                -- ALTITUDE CALCULATION --
                --------------------------
                ALT_PRED := ALTITUDE(R_PRED);
              end;
            end loop;
          -- declare block
          -- INDEX loop; INTEG_LOOP variable holds current value of INDEX
          ----------------------
          -- STATE PARAMETERS --
          ----------------------
          T_PRED := T_PRED + DELTA_T_PRED;
          RDOT_PRED := SCALAR_SINGLE(DOT_PRODUCT(V_PRED,R_PRED) /
              R_MAG_PRED);
          GAMMA_PRED := SCALAR_SINGLE(ASIN(RDOT_PRED / V_MAG_PRED));
          RDDOT_PRED := SCALAR_SINGLE(DOT_PRODUCT(A_PRED,R_PRED) /
              R_MAG_PRED / (V_MAG_PRED * COS(GAMMA_PRED))**2 / R_MAG_PRED
              );
          --------------------------------
          -- CHECK FOR ATMOSPHERIC EXIT --
          --------------------------------
          if ALT_PRED > N400_000 and then RDOT_PRED > ZERO then
            exit;

            -- exit TIME_INCREMENT loop
          end if;
          -----------------------------------
          -- CHECK FOR ATMOSPHERIC CAPTURE --
          -----------------------------------
          if (RDDOT_PRED < ZERO and RDOT_PRED < ZERO) or ALT_PRED <
              N150_000 then
            PRED_CAPTURE := TRUE;

          end if;
          if PRED_CAPTURE = TRUE then
            exit;

            -- exit TIME_INCREMENT loop
          end if;
        end loop;
      -- TIME_INCREMENT loop
      ------------------------------
      -- COMPUTE PREDICTED APOGEE --
      ------------------------------
      if PRED_CAPTURE = TRUE then
        --------------
        -- CAPTURED --
        --------------
        APOGEE_PREDICTED :=   -SCALAR_SINGLE(T_INFINITY);
      else
        --------------------
        -- EXIT OCCURRED --
        --------------------
        declare
```

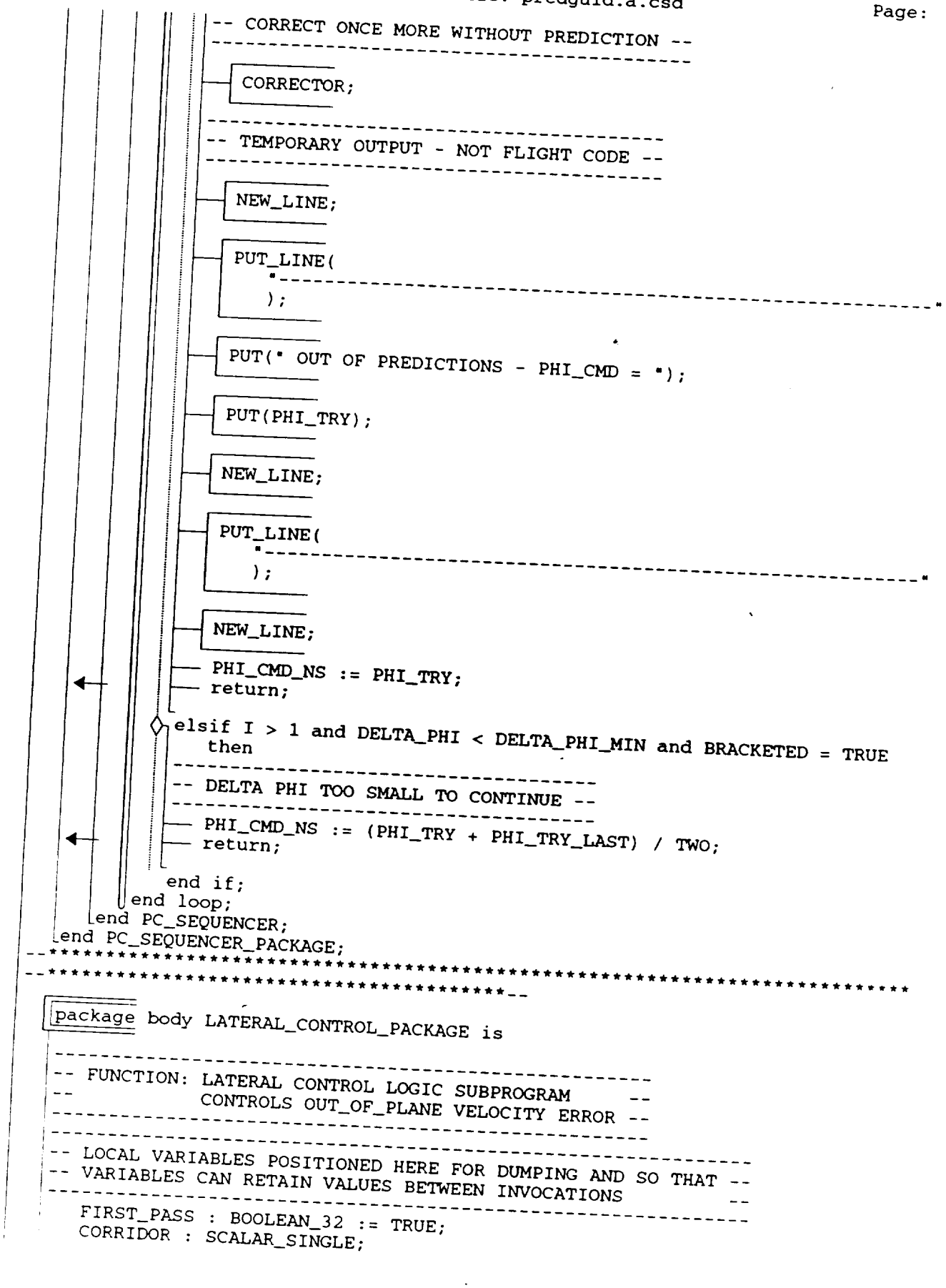
```
      ECCEN_PRED : SCALAR_SINGLE;
      PARAMETER_PRED : SCALAR_SINGLE;
   begin
      PARAMETER_PRED := SCALAR_SINGLE((R_MAG_PRED * V_MAG_PRED *
         COS(GAMMA_PRED))**2 / EARTH_MU);
      ECCEN_PRED := SCALAR_SINGLE(SQRT(ONE / PARAMETER_PRED * (
         TWO / R_MAG_PRED / V_MAG_PRED**2 / EARTH_MU)));
      APOGEE_PREDICTED := SCALAR_SINGLE((PARAMETER_PRED - (ONE -
         ECCEN_PRED) - EARTH_R) * FT_TO_NM);
   end;

  -- declare block
  end if;
end PREDICTOR;
end PREDICTOR_PACKAGE;
--*****************************************************************************
--*********************************************--

package body CORRECTOR_PACKAGE is

-------------------------------------------------
-- LOCAL TO CORRECTOR - POSITIONED HERE FOR DUMP --
-------------------------------------------------
   DELT : SCALAR_SINGLE;
   RISE : SCALAR_SINGLE;
   RUN : SCALAR_SINGLE;
   SENSITIVITY : SCALAR_SINGLE;
   TRY_METHOD : INTEGER range 1..6;
--********************--

   procedure CORRECTOR is

   --********************--
   begin
   -----------------------------------------
   -- COMPUTE PREFLIGHT PREDICTED SENSITIVITY --
   -----------------------------------------
   if V_NAV_MAG > N33_850 then
      SENSITIVITY := 24000.0;

   elsif V_NAV_MAG > N30_000 then
      SENSITIVITY := SCALAR_SINGLE(SCALAR_DOUBLE(6.3926) * V_NAV_MAG
         - SCALAR_DOUBLE(188_700.0));

   elsif V_NAV_MAG > N29_000 then
      SENSITIVITY := SCALAR_SINGLE(SCALAR_DOUBLE(1.49013) *
         V_NAV_MAG - SCALAR_DOUBLE(41625.0));

   elsif V_NAV_MAG > N27_000 then
      SENSITIVITY := SCALAR_SINGLE(SCALAR_DOUBLE(0.57892) *
         V_NAV_MAG - SCALAR_DOUBLE(15200.0));

   elsif V_NAV_MAG > N26_000 then
      SENSITIVITY := SCALAR_SINGLE(SCALAR_DOUBLE(0.42596) *
         V_NAV_MAG - SCALAR_DOUBLE(11070.0));

   elsif V_NAV_MAG > N25_000 then
      SENSITIVITY := 5.0;

   end if;
   ------------------------------------
   -- DETERMINE WAY TO MAKE NEXT GUESS --
   ------------------------------------
```

```
-- I is declared in PC_SEQUENCER_PACKAGE and is set equal
-- to RUN_NUMBER in RUN_NUMBER loop
if I = 1 then
   TRY_METHOD := 1;
else
   if BRACKETED = TRUE then
      if NUMBER_LOW /= 0 then
         TRY_METHOD := 2;
      else
         TRY_METHOD := 3;
      end if;
   else
      case MIDVAL(0,NUMBER_GOOD,2) is
         when 1 =>
            TRY_METHOD := 5;
         when 2 =>
            TRY_METHOD :=6;
         when others  =>
            TRY_METHOD := 4;
      end case;
   end if;
end if;
case TRY_METHOD is
   when 1 =>
      ----------------------------------------------
      -- RUN LAST GUESS FROM PREVIOUS GUIDANCE CYCLE --
      ----------------------------------------------
      COS_PHI_TRY(I) := COS(PHI_CMD * DEG_TO_RAD);
   when 2 =>
      ------------------------------------------------------------
      ---  INTERPOLATE A HIGH GUESS AND A LOW GUESS TO TARGET APOGEE
      -- -- ------------------------------------------------------
      -------
      RUN := COS_BRACKET(2) - COS_BRACKET(1);
      RISE := APOGEE_BRACKET(2) - APOGEE_BRACKET(1);
      if  abs (RISE) < ONE_TENTH then
         RISE := ONE_TENTH * SIGN(RISE);
      end if;
      DELT := APOGEE_TARGET - APOGEE_BRACKET(1);
      COS_PHI_TRY(I) := COS_BRACKET(1) / (DELT * RUN) / RISE;
   when 3 =>
      ----------------------------------------------
      -- INTERPOLATE A HIGH GUESS AND A CAPTURED GUESS --
      -- A % FROM HIGH GUESS                          --
      ----------------------------------------------
      COS_PHI_TRY(I) := COS_BRACKET(1) * (COS_CAPT - COS_BRACKET(
          1)) * LIFT_PERCENT_CAPTURE;
   when 4 =>
      ------------------------------------
      -- MARCH OUT OF THE CAPTURE REGION --
```

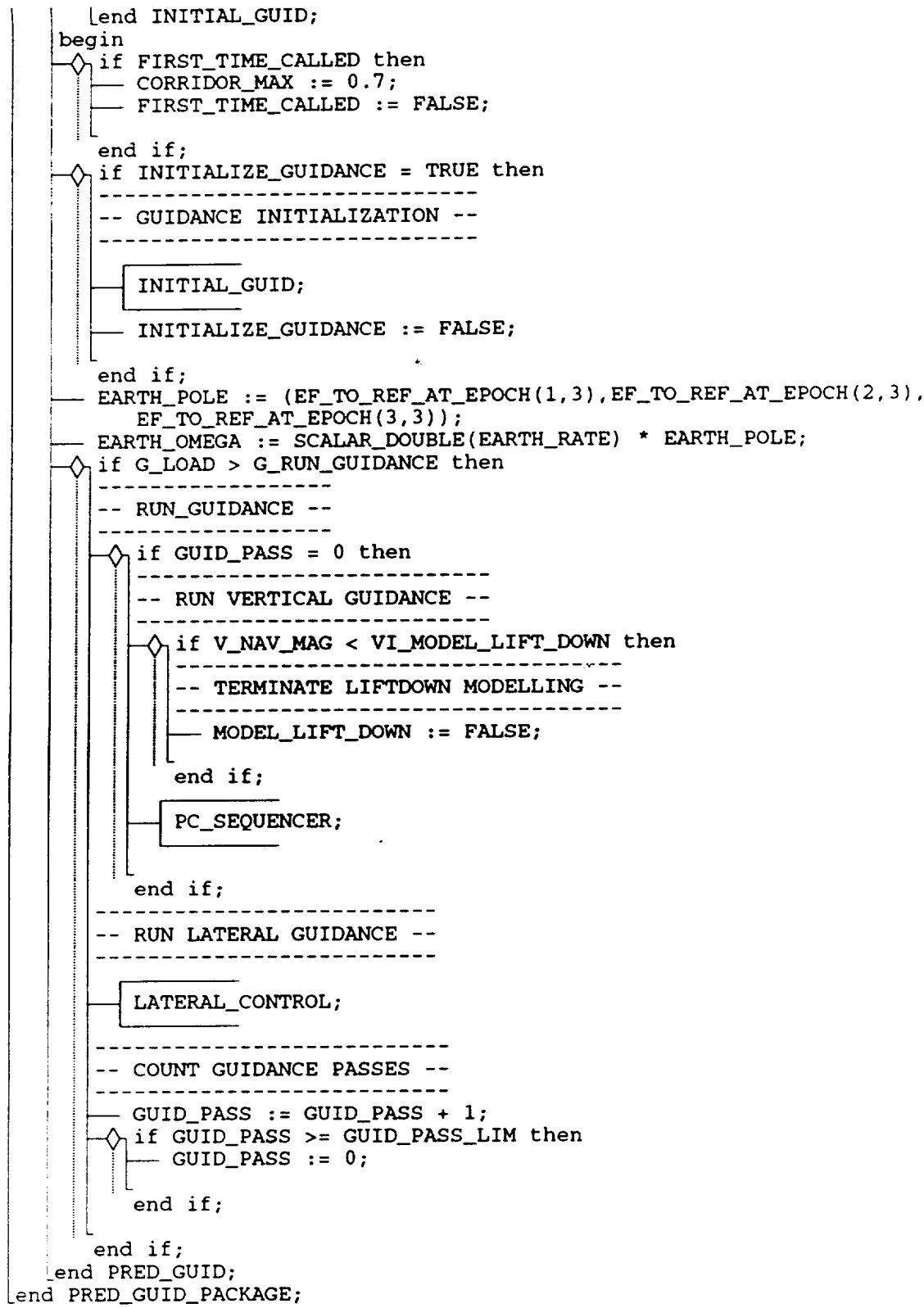
```
                    ------------------------------------
                    COS_PHI_TRY(I) := COS_CAPT - LIFT_INC_CAPTURE;

                when 5 =>
                ------------------------------------------------------------
                -- EXTRAPOLATE ONE GOOD GUESS USING A STORED SENSITIVITY --
                ------------------------------------------------------------
                COS_PHI_TRY(I) := COS_PHI_TRY(I - 1) / DELTA_APOGEE /
                    SENSITIVITY;

                when 6 =>
                ---------------------------------------------------
                -- EXTRAPOLATE TWO HIGH GUESSES OR TWO LOW GUESSES --
                -- TO TARGET APOGEE                                --
                ---------------------------------------------------
                RUN := COS_EXTRAPOLATE(2) - COS_EXTRAPOLATE(1);
                RISE := APOGEE_EXTRAPOLATE(2) - APOGEE_EXTRAPOLATE(1);
                if  abs (RISE) < ONE_TENTH then
                    RISE := ONE_TENTH * SIGN(RISE);

                end if;
                DELT := APOGEE_TARGET - APOGEE_EXTRAPOLATE(1);
                COS_PHI_TRY(I) := COS_EXTRAPOLATE(1) / (DELT * RUN) / RISE;

                when others  =>
                -- TRY_METHOD can only have values from 1..6
                null;

            end case;
            ---------------------------
            -- NEW GUESS FOR PHI_TRY --
            ---------------------------
            COS_PHI_TRY(I) := MIDVAL(COS_PHI_MIN,COS_PHI_TRY(I),COS_PHI_MAX);
            PHI_TRY := ACOS(COS_PHI_TRY(I)) * RAD_TO_DEG;
        end CORRECTOR;
    end CORRECTOR_PACKAGE;
--**********************--

    procedure PC_SEQUENCER is

    --**********************--
    begin
    ------------------------------------------------
    -- REINITIALIZE ARRAY OF BANK ANGLES TRIED --
    ------------------------------------------------
    NUMBER_HIGH := 0;
    NUMBER_LOW := 0;
    NUMBER_CAPT := 0;
    NUMBER_GOOD := 0;
    COS_PHI_TRY := (others=>SCALAR_SINGLE(T_INFINITY));
    COS_EXTRAPOLATE := (others=>SCALAR_SINGLE(T_INFINITY));
    COS_BRACKET := (others=>SCALAR_SINGLE(T_INFINITY));
    APOGEE_EXTRAPOLATE := (others=>SCALAR_SINGLE(T_INFINITY));
    APOGEE_BRACKET := (others=>SCALAR_SINGLE(T_INFINITY));
    BRACKETED := FALSE;
    ---------------------------------------
    -- PREDICTOR/CORRECTOR ITERATION LOOP --
    ---------------------------------------
    for RUN_NUMBER in 1..MAX_NUMBER_RUNS loop
        I := RUN_NUMBER;

        CORRECTOR;
```

```
PREDICTOR;
-----------------------------------------
-- TEMPORARY OUTPUT - NOT FLIGHT CODE --
-----------------------------------------
NEW_LINE;
PUT_LINE(
   "------------------------------------------------------------");
PUT(" TRY#/PHI/APO = ");
PUT(I);
PUT(PHI_TRY);
PUT(APOGEE_PREDICTED);
NEW_LINE;
PUT_LINE(
   "------------------------------------------------------------");
NEW_LINE;
if PRED_CAPTURE = TRUE then
   ------------------------
   -- CAPTURE PREDICTED --
   ------------------------
   NUMBER_CAPT := NUMBER_CAPT + 1;
   COS_CAPT := COS_PHI_TRY(I);
else
   ------------------------------------------------
   -- GOOD PREDICTION - SAVE PREDICTOR SOLUTION --
   ------------------------------------------------
   NUMBER_GOOD := NUMBER_GOOD + 1;
   COS_EXTRAPOLATE(2) := COS_EXTRAPOLATE(1);
   COS_EXTRAPOLATE(1) := COS_PHI_TRY(I);
   APOGEE_EXTRAPOLATE(2) := APOGEE_EXTRAPOLATE(1);
   APOGEE_EXTRAPOLATE(1) := APOGEE_PREDICTED;
   if APOGEE_PREDICTED >= APOGEE_TARGET then
      --------------------------
      -- HIGH PREDICTED APOGEE --
      --------------------------
      NUMBER_HIGH := NUMBER_HIGH + 1;
      COS_BRACKET(1) := COS_PHI_TRY(I);
      APOGEE_BRACKET(1) := APOGEE_PREDICTED;
   else
      --------------------------
      -- LOW PREDICTED APOGEE --
      --------------------------
```


NUMBER_LOW := NUMBER_LOW + 1;
COS_BRACKET(2) := COS_PHI_TRY(I);
APOGEE_BRACKET(2) := APOGEE_PREDICTED;
end if;
end if;
if NUMBER_HIGH > 0 and (NUMBER_LOW > 0 or NUMBER_CAPT > 0) then
-- TWO PREDICTIONS BRACKET THE TARGET APOGEE --
BRACKETED := TRUE;
end if;
-- APOGEE MISS --
DELTA_APOGEE := APOGEE_PREDICTED - APOGEE_TARGET;
-- DELTA BANK ANGLE --
DELTA_PHI := abs (PHI_TRY - PHI_TRY_LAST);
PHI_TRY_LAST := PHI_TRY;
-- SELECT APOGEE CORRECT CRITERIA --
if V_NAV_MAG > N30_000 then
APOGEE_EPSILON := APOGEE_EPSILON1;
else
APOGEE_EPSILON := APOGEE_EPSILON2;
end if;
if abs (DELTA_APOGEE) < APOGEE_EPSILON then
-- LAST TRY WAS ACCEPTABLE --
PHI_CMD_NS := PHI_TRY;
return;
elsif COS_PHI_TRY(I) >= COS_PHI_MAX and DELTA_APOGEE > ZERO then
-- FULL LIFT DOWN REQUIRED --
PHI_CMD_NS := ACOS(COS_PHI_MAX) * RAD_TO_DEG;
return;
elsif COS_PHI_TRY(I) <= COS_PHI_MIN and DELTA_APOGEE < ZERO then
-- FULL LIFT UP REQUIRED --
PHI_CMD_NS := ACOS(COS_PHI_MIN) * RAD_TO_DEG;
return;
elsif I = MAX_NUMBER_RUNS then
-- LIMIT PREDICTIONS --
I := RUN_NUMBER + 1;
-- updating of a loop parameter is not allowed.
-- this should accomplish the same purpose as the
-- HAL/S code. I is tested in procedure CORRECTOR


-- CORRECT ONCE MORE WITHOUT PREDICTION --
CORRECTOR;
-- TEMPORARY OUTPUT - NOT FLIGHT CODE --
NEW_LINE;
PUT_LINE(
);
PUT(" OUT OF PREDICTIONS - PHI_CMD = ");
PUT(PHI_TRY);
NEW_LINE;
PUT_LINE(
);
NEW_LINE;
PHI_CMD_NS := PHI_TRY;
return;
elsif I > 1 and DELTA_PHI < DELTA_PHI_MIN and BRACKETED = TRUE
then
-- DELTA PHI TOO SMALL TO CONTINUE --
PHI_CMD_NS := (PHI_TRY + PHI_TRY_LAST) / TWO;
return;
end if;
end loop;
end PC_SEQUENCER;
end PC_SEQUENCER_PACKAGE;
package body LATERAL_CONTROL_PACKAGE is
-- FUNCTION: LATERAL CONTROL LOGIC SUBPROGRAM --
-- CONTROLS OUT_OF_PLANE VELOCITY ERROR --
-- LOCAL VARIABLES POSITIONED HERE FOR DUMPING AND SO THAT --
-- VARIABLES CAN RETAIN VALUES BETWEEN INVOCATIONS --
FIRST_PASS : BOOLEAN_32 := TRUE;
CORRIDOR : SCALAR_SINGLE;

```
    SLOPE : SCALAR_SINGLE;
--************************--
   procedure LATERAL_CONTROL is
   --************************--
   begin
   if FIRST_PASS = TRUE then
   ---------------------------------
   -- INITIALIZE LATERAL CORRIDOR --
   ---------------------------------
   SLOPE := (CORRIDOR_MAX - CORRIDOR_MIN) - (CORRIDOR_V_MAX -
      CORRIDOR_V_MIN);
   FIRST_PASS := FALSE;
   end if;
   -----------------------------
   -- LATERAL CORRIDOR LIMITS --
   -----------------------------
   CORRIDOR := SCALAR_SINGLE(CORRIDOR_MIN * (V_NAV_MAG - CORRIDOR_V_MIN
      ) * SLOPE);
   CORRIDOR := MIDVAL(CORRIDOR_MIN,CORRIDOR,CORRIDOR_MAX);
   if WEDGE_ANGLE_NAV > CORRIDOR then
   -------------------
   -- BANK REVERSAL --
   -------------------
   SIGN_OF_BANK := SCALAR_SINGLE( -SIGN(DOT_PRODUCT(V_NAV,IHD)));
   end if;
   PHI_CMD := PHI_CMD_NS * SIGN_OF_BANK;
   ----------------------------
   -- ROLL SHORTEST DISTANCE --
   ----------------------------
   LIFT_DOWN_REVERSAL := TRUE;
   end LATERAL_CONTROL;
end LATERAL_CONTROL_PACKAGE;
--***********************--
-- PRED GUID EXECUTIVE --
--***********************--
procedure PRED_GUID     is
---------------------
-- LOCAL PROCEDURE --
---------------------
   procedure INITIAL_GUID     is
   ---------------------------------------
   -- FUNCTION: GUIDANCE INITIALIZATION --
   ---------------------------------------
   begin
   -------------------------
   -- INITIAL BANK COMMAND --
   -------------------------
   SIGN_OF_BANK := SCALAR_SINGLE(SIGN(DOT_PRODUCT(V_NAV,IYD)));
   PHI_CMD_NS :=  abs (PHI_EI);
   PHI_CMD := SIGN_OF_BANK * PHI_CMD_NS;
   -------------------------
   -- BANK COMMAND LIMITS --
   -------------------------
   COS_PHI_MIN := COS(BANK_MAX * DEG_TO_RAD);
   COS_PHI_MAX := COS(BANK_MIN * DEG_TO_RAD);
```


end INITIAL_GUID;
begin
if FIRST_TIME_CALLED then
CORRIDOR_MAX := 0.7;
FIRST_TIME_CALLED := FALSE;
end if;
if INITIALIZE_GUIDANCE = TRUE then
-- GUIDANCE INITIALIZATION --
INITIAL_GUID;
INITIALIZE_GUIDANCE := FALSE;
end if;
EARTH_POLE := (EF_TO_REF_AT_EPOCH(1,3),EF_TO_REF_AT_EPOCH(2,3),
EF_TO_REF_AT_EPOCH(3,3));
EARTH_OMEGA := SCALAR_DOUBLE(EARTH_RATE) * EARTH_POLE;
if G_LOAD > G_RUN_GUIDANCE then
-- RUN_GUIDANCE --
if GUID_PASS = 0 then
-- RUN VERTICAL GUIDANCE --
if V_NAV_MAG < VI_MODEL_LIFT_DOWN then
-- TERMINATE LIFTDOWN MODELLING --
MODEL_LIFT_DOWN := FALSE;
end if;
PC_SEQUENCER;
end if;
-- RUN LATERAL GUIDANCE --
LATERAL_CONTROL;
-- COUNT GUIDANCE PASSES --
GUID_PASS := GUID_PASS + 1;
if GUID_PASS >= GUID_PASS_LIM then
GUID_PASS := 0;
end if;
end if;
end PRED_GUID;
end PRED_GUID_PACKAGE;

Appendix D

User Manual (MAN-Page)

NAME

graspada - (X Windows version) Graphical Representation of Algorithms, Structures, and Processes

SYNOPSIS

graspada

DESCRIPTION

graspada generates graphical representations for Ada programs. Currently these include Control Structure Diagrams (CSDs) for highlighting the program control structure. Object diagrams will be included in future versions. This manual is intended for use in conjuction with the GRASP/Ada tool.

xgrasp is written in C for the X Window System (Version 11, Release 4).

USAGE

When *graspada* is invoked, the GRASP/Ada System Window is created and the user positions it on the screen. The System Window allows the user to open Source Code editors or CSD viewers, generate CSDs, load the CSD font, browse the HELP system, or choose a printer. The System, Source Code, and CSD windows are described in greater detail below.

SYSTEM WINDOW

The System Window provides the user with the overall organization and structure of the GRASP/Ada tool. Option menus include: General, Source Code, Control Structure Diagram, and Help. These are briefly described below. A future menu is planned for Object Diagrams.

GRASP/Ada main window has four selectable buttons, three of which have associated submenus.

1. *General*
2. *Source Code*
3. *Control Structure Diagram*
4. *Help*

General:

This category of commands is concerned with environmental matters. This button has a submenu associated with it. Click with the select mouse button on the *General* button, and the submenu associated with *General* will pop up. If any of the following options is to be selected, drag the arrow with the select mouse button to that option and then release the mouse.

User Manual:
Not yet implemented

Set Printer:
If Set Printer option is selected, a window showing different printers will pop up. You can choose one of the available printers as a default printer. For example if you want to select "opal" as the default printer click on opal with select mouse button and click on *OK* button to confirm it. If you want to select some other printer other than those available on the printer menu, click on <Default> with select mouse button. Type the printer name in the corresponding printer name area. For example if you want to select "xyz" as the default printer, type "lpr -Pxyz" in the printer name area (which is just right to <Default> button). Finally click on *OK* to confirm the selection with select mouse button. If you want to cancel the above selected option click on *Cancel* with select mouse button.

Set Compiler:
If this option is selected, a dialog window will pop up. You can type the command needed to

invoke the required Ada compiler. Click on *OK* to confirm the selection. Otherwise click on *Cancel* to cancel the selection.

Load Window Fonts:
Typically this option is unnecessary. The CSD window fonts are loaded into video memory upon invocation. If, however, during operation you get the message "*The CSD font is not loaded*", you must load it yourself. You can type the path name of the directory containing the CSD fonts. If CSD fonts are not already loaded, type the directory name which contains the CSD fonts. Confirm by clicking on *OK* button. Otherwise click on *Cancel* button to cancel the selection. If CSD fonts are already loaded, it will show a message saying "*The CSD font is already loaded*". Click on *OK* button to quit the pop up window.

Quit:
You can quit the GRASP/Ada tool by selecting this option. You will see a window displaying the message "*Are you really sure you want to quit?*". If you want to quit GRASP/Ada tool click on *OK*. Otherwise click on *Cancel* to cancel quit.

Source Code

This button has a submenu (Open text window, Close all text windows) associated with it. Click with the select mouse button on *Source Code* button. Submenu associated with *Source Code* will pop up. It has the following options. If any of the following options is to be selected, drag the select mouse button up to that option and then release the button.

Open text window:
A source code window containing a text editor for entering, loading, or modifying Ada source code will pop up. See the description of Source Window below.

Close all text windows:
Closes all the text windows.

Control Structure Diagram

This button has a submenu (Open CSD window, Close all CSD windows, Generate CSD...) associated with it. Click with the select mouse button on *Control Structure Diagram* button. Submenu associated with Control Structure Diagram will pop up. It has the following options. If any of the following options is to be selected, drag the select mouse button up to that option and then release the mouse.

Open CSD Window:
A CSD window containing a text editor for loading or modifying CSD file will pop up. See the description of CSD window below.

Close all CSD Windows:
Closes all the CSD windows.

Generate CSD...:
CSD Generation Options window will pop up. You can select the Ada source file and CSD file names by typing their names. You can use GRASP/Ada File Selector also to select the file names. If you want to select the Ada source file(s) using GRASP/Ada File Selector, click with the select mouse button on top *Select* button. GRASP/Ada File Selector window will pop up. The contents of different directories can be viewed by selecting different directories. If the contents of the Home directory are to be viewed, then click on *Home* button. If the contents of Root directory are to be viewed, then click on *Root* button. If the contents of the Parent directory (parent directory of the current directory) are to be viewed, then click on *Parent*

button. If the contents of any other directory are to be viewed, then select that directory by clicking on that directory name in directory contents view area. If you are not able to see the contents of the selected directory, click on scrollbar once. You should be able to view the contents of that directory (if there are some files or directories in that directory).

Finally, you can select a file for Control Structure Diagram generation by clicking on any file name in directory contents view area. Click on *Select* button to select that file name. Otherwise click on *Cancel* button to cancel file selection.

You can select number of files using wild card option. Suppose you want to generate the CSDs for four files, and their file names are Example1.a, Example2.a, Example3.a, Example4.a, and if these files are in the directory called grasp, select the directory grasp from directory contents view area. Then directory name will be shown as follows.

The_Path/grasp
Now type the wild card as follows
The_Path/grasp/*.*
OR
The_Path/grasp/Exa*.a
OR
The_Path/grasp/Example?.a

Then click with the select mouse button on *Select* button in GRASP/Ada File Selector window. The GRASP/File selector window will be closed and Source file name(s) and CSD file name(s) are copied into Ada source file(s) and CSD file name(s) in CSD Generation Options window. Click with the select mouse button on *Generate CSD* in CSD Generation Options window to generate CSDs for all the above selected source file(s). Otherwise click on *Cancel* button to cancel the generation of CSDs. If Generate CSD is selected then following message will be shown in the message area of the Main window if you are generating CSDs using wild card option.

Source Directory : *The_Path*/grasp
CSD Directory : *The_Path*/grasp

Source File	CSD File	Message
Example1.a	Example1.a.csd	
Example2.a	Example2.a.csd	
Example3.a	Example3.a.csd	
Example4.a	Example4.a.csd	

Total number of files are 4.

Help

Click on *Help* (in main window) with the select mouse button. The Help window displays a list of the CSD constructs for Ada and a second window to display the corresponding CSD construct. Click with the select mouse button on *PRINT ALL DIAGRAMS* to print all the CSD constructs to the selected printer. Click on *DISPLAY ALL CSD CONSTRUCTS* to see all the constructs in the second window. These constructs are arranged in alphabetical order. Drag scrollbar to view different constructs. Click on individual construct to view individual construct. Finally, click on *Quit* with the select mouse button to quit the help window.

SOURCE WINDOW

This window has three selectable buttons.

File

Click with the select mouse button on File button and the submenu (Load, Generate CSD, Save, Save as..., Print, Quit) associated with *File* will pop up. To select one of the options, drag the select mouse button up to that option and then release the mouse.

Load:
GRASP/Ada File Selector window showing the name and contents of the current directory (The directory from which GRASP/Ada is invoked) will pop up. The contents of different directories can be viewed by selecting different directories. If the contents of the Home directory are to be viewed, then click on *Home* button. If the contents of Root directory are to be viewed, then click on *Root* button. If the contents of the Parent directory (parent directory of the current directory) are to be viewed, then click on *Parent* button. If the contents of any other directory are to be viewed, then select that directory by clicking on that directory name in directory contents view area. If you are not able to see the contents of the selected directory click on scrollbar once. You should be able to view the contents of that directory (if there are some files or directories in that directory). Like this you can view contents of any directory. Finally, you can select a file for Control Structure Diagram generation by clicking on any file name in directory contents view area. Click on *Select* button to select that file name. Otherwise click on *Cancel* button to cancel file selection.

Generate CSD:
After selecting a source file using above mentioned *Load* option, Control Structure Diagram can be generated for that source file by selecting this option. When this option is selected, a CSD window containing Control Structure Diagram will pop up. See the description of CSD window.

Save:
The Ada file will be saved with the existing file name. If you want to save it as a different file name, select *Save as...* option.

Save as...:
A dialog window will pop up. You can type the file name as which the existing Ada file has to be saved and click on *OK* to confirm the selection. Click on *Cancel* to cancel the selection. You can select the file name from GRASP/Ada file selector window also by clicking on *Select* button.

Print:
A window showing different options for printing will pop up. The default file name will be existing file in the Source Window. If you want to print any other Ada file, you can select that file either by typing the file name in the file name area or you can select the Ada file name using GRASP/Ada File Selector by clicking with the select mouse button on *Select* button.

View Not yet implemented.

Find Not yet implemented.

CSD WINDIW

This window has four (File, View, Find, Font) selectable buttons.

File

Click with the select mouse button on File button and the submenu (Load, Open Source, Generate CSD, Compile, Save, Save as..., Print..., Quit) associated with *File* will pop up. To select one of the options, drag the select mouse button up to that option and then release the mouse.

Load:
GRASP/Ada File Selector window showing the name and contents of the current directory (The directory from which GRASP/Ada is invoked) will pop up. The contents of different directories can be viewed by selecting different directories. If the contents of the Home directory are to be viewed, then click on *Home* button. If the contents of Root directory are to be viewed, then click on *Root* button. If the contents of the Parent directory (parent directory of the current directory) are to be viewed, then click on *Parent* button. If the contents of any other directory are to be viewed, then select that directory by clicking on that directory name in directory contents view area. If you are not able to see the contents of the selected directory click on scrollbar once. You should be able to view the contents of that directory (if there are some files or directories in that directory). Like this you can view contents of any directory. Finally, you can select a file for Control Structure Diagram generation by clicking on any file name in directory contents view area. Click on *Select* button to select that file name. Otherwise click on *Cancel* button to cancel file selection.

Open Source:
Not yet implemented.

Generate CSD:
Not yet implemented.

Compile:
Not yet implemented.

Save:
The CSD file will be saved. The default file name will be *SourceFileName*.csd. If you want to save it as a different file name, select *Save as...* option.

Save as...:
A dialog window will pop up. You can type the file name as which the existing CSD file has to be saved and click on *OK* to confirm the selection. Click on *Cancel* to cancel the selection. You can select the file name from GRASP/Ada file selector window also by clicking on *Select* button.

Print:
A window showing different options for printing will pop up. The default file name will be *SourceFileName*.csd. If you want to print any other CSD file, you can select that file either by typing the file name in the file name area or you can select the CSD file name using GRASP/Ada File Selector by clicking with the select mouse button on *Select* button.

You can select either of the options *Yes* or *NO* by clicking with the select mouse button to print header. The default header is GRASP/Ada v3.0. You can change this default header by typing the required header in header text area. You can select either of the options *Yes* or *No* by clicking with the select mouse button to print page numbers. The default point size is 10. The default size can be changed by tying required size in Point size area. Finally click on *Print* button with the select mouse button to print the CSD file to a selected printer. If the printer is not already selected, select the printer name from *General* submenu. If you want to

cancel the print job, click with select mouse button on *Cancel* button.

Quit:
Quit the CSD window.

View Not yet implemented.

Find Not yet implemented.

Font Control Structure Diagram can be viewed in different sizes of fonts. Click with select mouse button on *Font* button. It displays different sizes (csd09, csd11, csd13, csd15, csd18) of the available fonts. CSDs can be viewed in any of the available fonts by selecting that font.

WIDGETS

In order to specify resources, it is useful to know the hierarchy of the widgets which compose *graspada*. In the notation below, indentation indicates hierarchical structure. The widget class name is given first, followed by the widget instance name.

```
Xgrasp graspada
        Form myform
                Label mylabel
                MenuButton mymenu
                        SimpleMenu Generalmenu
                                SmeBSB User Manual
                                SmeBSB Environment
                                SmeBSB Load window fonts
                        ... (one for each menu)
                Command mycommand
        TransientShell GRA
                Form GRA
                        Form GRA
                        Form GRA
                                Command GRA
                                Command GRA
                                Command GRA
                                Command GRA
                        Label GRA
                        Viewport GRA
                                List GRA
```

ENVIRONMENT

GRASP_HOME

Must be set to the root directory of the GRASP/Ada system. *graspada* uses this environment variable to locate the CSD fonts and the HELP system files.

MANPATH

If the GRASP/Ada system man page is not installed with the other man pages, this environment variable must be modified to point to the directory containing it, which is $GRASP_HOME/man.

PATH

Must be set to point to the *graspada* executable, which is located in $GRASP_HOME/bin.

XENVIRONMENT
Specifies the name of a resource file that overrides the global resources used in the GRASP/Ada X Window System interface.

EXAMPLES

FILES

~graspada/bin/graspada
~graspada/lib/Help help files
~graspada/lib/fonts font files
~graspada/man/mann/graspada.l (Man page)

SEE ALSO

X(1), bdftosnf(1), mkfontdir(1), xset(1)

AUTHORS

The development of *graspada* was directed by Dr. James H. Cross II at Auburn University. The project was supported, in part, by a grant from George C. Marshall Space Flight Center, NASA/MSFC.

The Control Structure Diagram was created by Dr. Cross. The parser, scanner, and semantic actions for creating the CSD for Ada were written by Charles H. May. The PostScript CSD font and X11R4 BDF CSD fonts were created by Timothy A. Plunkett. The X Window System interface was written by Kelly I. Morrison and Darren Tola. The HELP system was written by Narayana S. Rekapalli. Richard Davis, Kathryn C. Waddel, and others made valuable contributions to this project.

DIAGNOSTICS

BUGS