DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

COLLEGE OF ENGINEERING &
ENGINEERING EXPERIMENT STATION

AUBURN ENGINEERING

AUBURN UNIVERSITY

AUBURN, ALABAMA 36849

EW41 / Doris Hipp

GRASP/Ada

Graphical Representations of Algorithms, Structures, and Processes for **Ada**

GRASP/Ada
Reverse Engineering Tools For Ada
Final Report

for

Delivery Order No. 30
Basic NASA Contract No. NAS8-39131

Technical Report 95-09
June 21, 1995

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

Contact: James H. Cross II, Ph.D.
Principal Investigator
(334) 844-6315
cross@eng.auburn.edu

GRASP/Ada

Graphical Representations of Algorithms, Structures, and Processes for **Ada**

Reverse Engineering Tools For Ada
Final Report

for

Delivery Order No. 21

Basic NASA Contract No. NAS8-39131

Technical Report 95-07

June 21, 1995

James H. Cross II, Ph.D.

Principal Investigator

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, as a result, improve reliability and reduce costs. The emphasis has been on the automatic generation of the CSD from Ada PDL or source code to support reverse engineering and maintenance. The CSD has the potential to replace traditional prettyprinted Ada source code. The current update has focused on the design and implementation of a new Motif compliant user interface, and a new CSD generator consisting of a *tagger* and *renderer*. The current Version 4.0 prototype provides the capability for the user to generate CSDs from Ada PDL or source code in a reverse engineering as well as forward engineering mode with a level of flexibility suitable for practical application. This report provides an overview of the GRASP/Ada project with an emphasis on the current update.

ACKNOWLEDGEMENTS

We appreciate the assistance provided by NASA personnel, especially Mr. Robert Stevens, Kathy White, Amy Cardno, and Mr. Keith Shackelford. This work was also supported, in part, by a grant from ARPA, which focused on the utilization of GRASP/Ada in Computer Science and Engineering courses at Auburn University and preparation of GRASP/Ada for distribution to other universities (May 3, 1994 - October 2,1995).

The following is an alphabetical listing of the team members who have participated in various phases of the project.

Principal Investigator: Dr. James H. Cross II, Associate Professor

Current Graduate Research Assistants: Dean Hendrix, Brian Randles, Mark Sadler

Past Graduate Research Assistants: Richard A. Davis, Charles H. May, Kelly I. Morrison, Timothy A. Plunkett, Narayana S. Rekapalli, Darren Tola

TABLE OF CONTENTS

LIST OF FIGURES

Page

1.0 Introduction

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 [SHU88, AOY89, SCA89]. 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 specific task has focused on *reverse engineering* of control structure diagrams from Ada PDL or source code.

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. 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.

The GRASP/Ada project was divided into three primary development phases followed by three update phases: Update'92, Update'93, and Update'94-95, the current phase. Each of these phases is briefly described below.

1.1 Phase 1 - The Control Structure Diagram For Ada

Phase 1 focused on a survey of graphical notations for software with concentration on detailed level diagrams such as those found in [MAR85, TRI89], 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. The CSD is described in Section 2.0.

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 diagrams for Ada were surveyed and the Booch's module notation [BOO94] and the OOSD notation [WAS89] were identified as having good potential for accurately representing many

of the varied architectural features of an Ada software system. Phase 2 also included the preliminary design for an architectural CSD [DAV90]. The aspects of architectural CSD are expected to be integrated into the fully operational GRASP/Ada prototype during a future phase of the project.

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

Phase 3 has had two major thrusts: (1) completion 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. The User Interface was reworked based on the Athena widget set. CSDgen was installed and demonstrated on a Sun workstation at Marshall Space Flight Center, Alabama.

The development of a preliminary prototype for generating architectural object diagrams (ODgen) for Ada source/PDL was 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.

1.4 Update'92 - Preliminary Evaluation and User Interface Enhancements

Following Phase 3, the Version 3.1 prototype was used in several software engineering classes at Auburn University, evaluated, and enhanced to create Version 3.2. A preliminary analysis was done on data collected from students, and then changes were made to the User Interface to reflect the indicated usage patterns. The UNIX man-page was drafted to provide online documentation, and the installation guide was drafted to provide for limited distribution of the tool. And finally, the prototype was modified so that it could be invoked from IDE's CASE tool StP with a pspec or PDL file (see Appendix B).

1.5 Update'93 of the GRASP/Ada

Update'93 focused on the statistical analysis of the data collected in the previous update and preparation of Version 3.4 of the prototype for limited distribution to facilitate further evaluation. Since Update'92, the Version 3.2 prototype has undergone continual upgrades which have resulted in Version 3.4 of the prototype. The CSD evaluation data collected in the previous update was formally analyzed during Update'93 and reported in *Final Report for Update'93*. Statistical analysis indicated highly significant differences among five graphical notations when compared with respect to eleven performance

characteristics. There was a clear preference for the CSD for seven of the eleven performance characteristics.

1.6 Update'94-95 of the GRASP/Ada - The Current Phase

Update'94-95 is the most recent phase of the GRASP/Ada project, and the one described in the remainder of this report. Since Update'93, the Version 3.4 prototype has been completely redesigned resulting in Version 4.0 of the prototype. In particular, the GRASP/Ada tool was evaluated, the User Interface was completely reengineered to be Motif compliant, and a new CSD generator was created as two components, the *Tagger* and the *Renderer,* to provide a significant degree of language independence. Each of these tasks is briefly described below.

(1) **The GRASP/Ada Tool Evaluation.** As part of the ongoing evaluation of GRASP/Ada, Version 3.4 was used in CSE 422 (Introduction to Software Engineering) by approximately 45 junior computer science and engineering students in conjunction with the design, and implementation of their software project. Problems encountered by the students and suggestions for improving the User Interface to facilitate better utilization were factored into the design of the new Motif compliant user interface in Version 4.0.

(2) **Motif Compliant User Interface for the GRASP/Ada tool.** A new Motif compliant User Interface was developed for Version 4.0. Several GUI builders were evaluated during the course of the project to determine their appropriateness for GRASP/Ada. The new User Interface and the results of the evaluation of the GUI builder tools are described in Section 4.

(4) **CSD Generator (Tagger and Renderer) for the GRASP/Ada tool.** A new CSD generator was developed based on two components, a Tagger and a Renderer. The Tagger parses the Ada source file and tags the control constructs and other useful elements using a new markup language, called GRASP-ML, which is based on Standard Generalized Markup language (SGML). The Renderer then inputs the tagged source code and produces the CSD. The primary goals for using the tagger/renderer approach were to provide a language independent renderer and to improve the overall structure of the CSD generator to reduce future maintenance effort. The new CSD generator is in Section 5.

(3) **Installation and Online User Manual.** Version 4.0 is currently a combination of C and Ada 83. The User Interface, which requires access to Motif libraries during compilation, and the Tagger are written in C, and compiled using Sun C. The GRASP/Ada main program and the Renderer are written in Ada 83 and compiled using Rational Apex. The Version 4.0 executable is created by linking the all C and Ada components using Rational Apex. The current version of GRASP/Ada can be downloaded via *ftp* as follows:

ftp ftp.eng.auburn.edu

user_id: anonymous
password: (your email address)

The files are located in *pub/graspada/version4.0*. The README file describes the system requirements, installation procedure, and getting started. In a future version of the tool, after the software is installed, an online User Manual is available via the HELP button on the GRASP Control Panel or the CSD Window. The User Manual will be hyperlinked to allow the user to view individual topics.

(4) **Investigation of Object Diagrams.** Existing CASE tools that utilize object diagrams in one form or another were reviewed. The results of the review indicate that the GRASP/Ada tool can play an important role as a natural extension to these existing object diagrams and their supporting CASE tools. The control structure diagram provides a detailed algorithmic level graphical representation which has statistically significant advantages over other graphical notations, PDL, and source code. The entire spectrum of re-engineering oriented CASE tools with respect to functionality and availability has recently been well-documented in [OLS93, SIT92] .

(5) **Presentation of Update'94-95.** Specifications for this update and the expected results were presented at the 2nd Working Conference on Reverse Engineering (WCRE'95), July 14-16, 1995, in Toronto, Ontario [CRO95].

The following sections describe the control structure diagram, the GRASP/Ada system model, the user interface, the control structure diagram generator, and future requirements. The overall rationale for the development of the CSD is described in [CRO90], which was written during Phase 1. A taxonomy and extensive literature review of reverse engineering can be found in [CHI90, CRO92], which were written during Phases 2 and 3.

2.0 The Control Structure Diagram

Advances in hardware and software, particularly high-density bit-mapped monitors and window-based user interfaces, have led to a renewed interest in graphical representation of software. Although 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, the complex nature of the control constructs and control flow defined by programming languages such as Ada and C and their associated PDLs, makes source code and detailed design specifications attractive candidates for graphical representation. In particular, source code should benefit from the use of an appropriate graphical notation since it must be read many times during the course of initial development, testing and maintenance. The control structure diagram (CSD) is a notation intended specifically for the graphical representation of algorithms in 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. The CSD is a natural extension to existing architectural graphical representations such as data flow diagrams, structure charts, and object diagrams.

The CSD, which was initially created for Pascal/PDL [CRO88], 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 should 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 diagraming with those of well-indented PDL or source code.

2.1 Background

Graphical representations have 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 life cycle. In particular, 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.

Since the flowchart was introduced in the mid-50's, 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 [MAR85, TRI88, SHN77]. Tripp, for example, describes 18 distinct notations that have been introduced since 1977 and Aoyama et.al. describes 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 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 U.S. in favor of non-graphical PDL. A lack of automated support and the results of several studies conducted in the seventies which found no significant difference in the comprehension of algorithms represented by flowcharts and pseudo-code [SHN77] have been a major factors in this underutilization. However, automation is now available in the form of numerous CASE tools and recent empirical studies reported by Aoyami [AOY89] and Scanlan [SCA89] 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 pseudo-code, were read by intermediate-level students. Scores for the flowchart were significantly higher than those of the PDL. The statistical studies reported by Aoyami et.al. involved several tree-structured diagrams (e.g., PAD, YACC II, and SPD) widely used in Japan which, 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 productivity over current non-graphical approaches.

2.2 The Control Structure Diagram Illustrated

Two examples are presented below to illustrate the CSD. The first shows the basic control constructs of sequence, selection and iteration in Ada. These three control constructs are common to all structured procedural languages such as Ada, C, and Pascal. The second example illustrates a more complex control construct, the task rendezvous in Ada.

Figure 1 contains an Ada procedure called SearchArray that searches an array A of elements and counts the number of elements above, below, and/or equal to a specified element. Figure 2 contains the CSD for SearchArray which includes the three basic control constructs sequence, selection, and iteration. Although this is a very simple example, the CSD clearly indicates the levels of control inherent in the nesting of control statements. For example, at level 1 there are four statements executed in sequence - the three assignment statements and the *for* loop. The *for* loop defines a second level of control which contains a single statement, the *if* statement, which in turn defines three separate level 3 sequences, each of which contains one assignment statement. It is noteworthy that even the CSDs for most production strength procedures rarely contain more than ten statements at level 1 or in any of the subsequences defined by control constructs for selection and iteration. This graphical chunking on the basis of functionality and level of control appears to have a

```
procedure SearchArray (A :  in ArrayType;
    Element:  in ElementType;
    Above,Below, EqualTo: out integer)   is

begin
   Above := 0;
   Below := 0;
   EqualTo := 0;
   for index in A'first..A'last loop
      if Element > A(index) then
         Below := Below + 1;

      elsif Element < A(index) then
         Above := Above + 1;

      else
         EqualTo := EqualTo + 1;

      end if;
   end loop;
end SearchArray;
```

Figure 1. Ada Source for SearchArray.

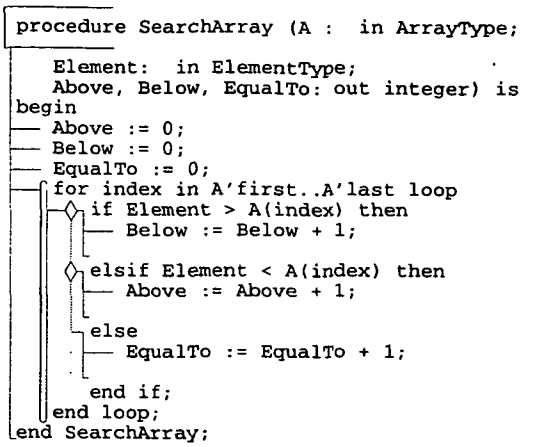

Figure 2. CSD for SearchArray

substantial positive effect on detailed comprehension of the software.

Figures 3 and 4 contain an Ada task body CONTROLLER adapted from [BAR84], which 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

```
task body TASK_NAME is

begin
   loop
      for p in PRIOITY loop
         select


            accept REQUEST(p) (D: DATA) do


               ACTION(D);

            end;
            exit;


         else
            null;

         end select;
      end loop;
   end loop;
end TASK_NAME;
```

Figure 3. Ada Source for Task Body Controller.

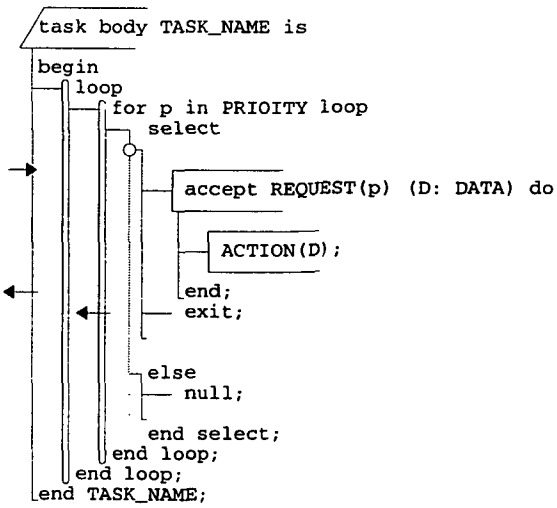

Figure 4. CSD for Ada Task Body Controller.

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.

The CSD in Figure 4 uses intuitive graphical constructs to 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 3, the control constructs and control paths are much less visible although the same structural and control information is available. With additional levels of nesting and increased physical separation of sequential components, 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 5. Each of the CSD constructs should be relatively self-explanatory since the CSD is designed to supplement the semantics of the underlying Ada.

2.3 Observations

The control structure diagram is a new graphical tool which maps directly to Ada and Ada PDL. The CSD offers advantages over previously available diagrams in that it is 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, object diagrams, and structure charts.

The GASP/Ada prototype, described in Sections 4 and 5, provides for the automatic generation of the CSD from Ada or Ada PDL. A preliminary statistical evaluation of the CSD is presented in [CRO93].

2.4 Control Structure Diagram - Future Directions

The CSD constructs shown in Figure 5 are expected to continue to evolve, especially with Ada 95 on the horizon. Suggestions for improvements to the individual CSD graphical constructs are continually solicited from users. While most of these suggested changes appear to be minor when considered individually, their aggregate implementation in the current prototype represents a major rework. Theoretically, the CSD and its individual constructs are a separate issue from the automatic generation of the diagrams in a production environment. However, in practice unless CSDs (or any other diagrams) can be automatically generated, they will not be utilized. Additional future considerations concerning the overall system model, the user interface, and the automatic generation of the CSD can be found at the end of Sections 3, 4, and 5 respectively.

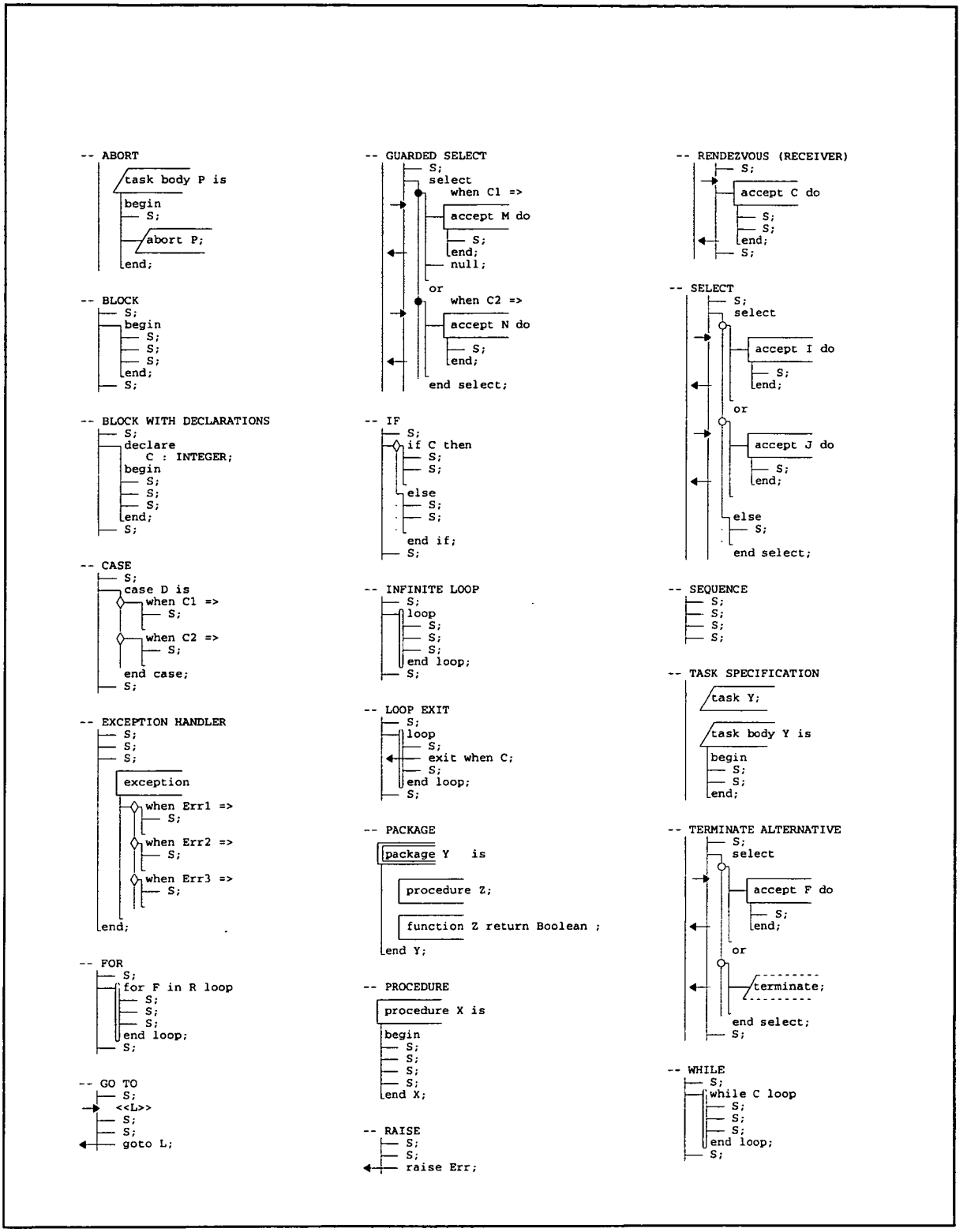

Figure 5. Control Structure Diagram Constructs for Ada.

3.0 The GRASP/Ada System Model

3.1 Overview

The major system components of the GRASP/Ada system are shown in the block diagram in Figure 6. Currently, the Version 4 prototype is implemented in a combination of the C and Ada 83. The **User Interface** was built using a GUI builder for Motif and the X Window System. The user interface includes a special CSD window (modified text editor) and provides general control and coordination among the other components.

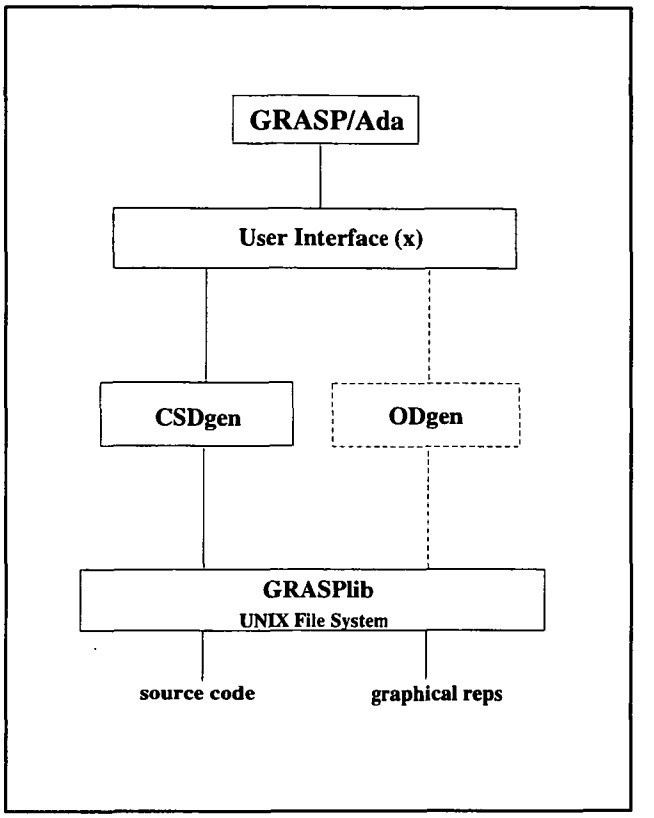

Figure 6. GRASP/Ada System Block Diagram.

The control structure diagram generator, **CSDgen**, has a separate *tagger* and *renderer*. The tagger has its own parser/scanner built using LEX and YACC. The tagger utilizes GRASP-ML [CRO95], a markup language based on SGML, to tag all necessary language elements required for input to the respective renderer. The tagger inputs Ada PDL or source code and produces a tagged file. The CSD renderer, built using AFLEX, the Ada version of LEX, then interprets the tagged file and displays the resulting diagram in the CSD window. When changes are made to the Ada PDL or source code in the CSD window, the file is reparsed to produce an updated CSD. A CSD editor, 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 dashed lines indicate future 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. For example, the Verdix Ada development system (VADS) and DIANA interface could be used 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. *ObjectMaker* by Mark V Systems has also been reviewed and is a strong contender as a basis for generating and displaying object diagrams. ObjectMaker was recently ported to UNIX from a PC platform, and unfortunately it appeared to be somewhat unstable in its current state.

The GRASP/Ada library component, **GRASPlib**, allows for coordination of all generated items with their associated source code. The current file organization uses standard

UNIX directory conventions as well as default naming conventions. For example, all Ada source files end in *.a*, 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. Its purpose is to facilitate navigation among the diagrams and the production of sets of diagrams.

3.2 System Model - Future Directions

The GRASP/Ada tool was conceived to be self-sufficient reverse engineering tool that would generate control structure diagrams primarily, and then architectural level diagrams (e.g., object diagrams) secondarily, all from Ada source code or PDL. An alternative to the model presented in Figure 6 would be to concentrate on issues that improve the integration capabilities of GRASP/Ada with commercially available CASE tools and programming environments. The following are potential tasks.

(1) A CSD editor/generator should include full syntax checking with appropriate error messages. Appropriate hooks to and from the compiler and debugger should be considered.

(2) Appropriate hooks to/from the GRASP/Ada tool are necessary to facilitate integration with commercially available CASE tools and programming environments.

(3) GRASP/Ada system model should include the capability of running in a stand-alone mode in which several CSD windows are to be coordinated are coordinated by a main window, similar to the current Version 4.3 prototype.

(4) GRASP/Ada system model should include the capability of running in a single window mode, similar to opening an X Windows textedit application.

(5) Finally, GRASP/Ada system model should include the capability of running in a single window mode, as an extension of a commercial CASE tool. For example, when clicking on an object or module in an architectural diagram, a CSD window could be opened with the PDL or source code represented in a CSD rather than simply text. Many commercial tools are competing in this market. One of particular interest is Rational's Ada programming development environment for UNIX (*Apex*), which has recently been made available to universities. Apex is a state-of-the-art environment which supports the Ada Semantic Interface Specification (ASIS) to facilitate tool integration. The GRASP/Ada tool with a CSD editor/generator could play an integral role in software development, maintenance, and reengineering when integrated with an environment such as *Apex*.

4.0 User Interface

This section describes the approach taken to design a Motif interface for GRASP, and serves to document the interface itself. The redesigned interface includes all of the functionality of the existing GRASP system, while adding significant improvements. Graphical User Interface (GUI) development tools were used during this project to enhance code production, consistency, quality, and maintainability. Additionally, new PostScript® Type1 fonts were created for GRASP to provide portability between a UNIX™ version and a future Windows™ version of the tool. The new interface conforms to Open Software Foundation's *OSF/Motif Style Guide*. Existing GRASP screens and code (X Window System™) were referenced to ensure retention of all required functionality.

The user interface incorporates the following improvements identified in the GRASP UPDATE '93 specification: ease of use (e.g., "Pinnable" menus and Shortcut keys) and color (e.g., default frame, background and foreground). The interface incorporates additional functionality though the routines that are called to provide such functionality are not yet written. These additional features have interface callback "stubs" created and so will not require additional redesign of the interface. An example of such additional functionality is the option to select a language other than Ada to parse and render. Another future enhancement is the ability to select font attributes (type, size and color) for reserved words, identifiers and comments.

Interface Development and Conversion are addressed in Section 4.1, which describes the efforts to produce a Motif Compliant interface in Ada. Several GUI builders were tried and the problems associated with each tool are described. Section 4.2 describes font development efforts and includes the new GRASP font sets, with and without the program unit symbols. Section 4.3 provides a description of the User Interface, in which each window and option are displayed, identified and discussed. "Stub" interface items are identified and the effort to complete them is estimated. Section 4.5 provides a summary and some ideas for the future.

4.1 Interface Development and Conversion

The development of the GRASP user interface was complicated by the desire to use a GUI design tool that would enhance the process. The theory is that such a tool enhances the production process by generating good quality code from screens developed within the tool. Additionally, maintainability of the code would be enhanced by segregating the user interface design from the code packages incorporating the functionality of the program (parsing, rendering, printing, etc.). Future changes to the interface could be made with little impact on the functional packages. Development of the GRASP interface was performed using OpenWindows Developer's Guide™. ScreenMachine™, ezX® and Builder Xcessory™ were utilized to generate Ada code. The ActivAda™ GUI Builder was used to develop a prototype Microsoft® Windows version of GRASP. The use of each of these tools is addressed in the following sections.

4.1.1 OpenWindows Developer's Guide™

Sun Microsystems' OpenWindows Developer's Guide™ was used as a rapid prototyping tool to design new GRASP screen images. DevGuide's ease of use allowed for fast production of screen images. The tool provided for the interconnection (callbacks) of the developed widgets that in turn allowed the design to be tested and validated. Unfortunately, DevGuide supports neither Motif nor Ada, but Open Look™ and C. Once the screen design was validated we began the task of selecting a tool for implementing these new GRASP screens in Motif and Ada.

4.1.2 ScreenMachine™

Objective Interface Systems' ScreenMachine™ 1.4 was originally recommended as the Ada GUI builder of choice by Air Force personnel at the Standard Systems Center and by programmers at Harris Data Services Corporation. It was also recommended for use by Rational with their Apex™ Ada development tools. Auburn acquired ScreenMachine from OIS as a grant-in-kind to the University.

ScreenMachine lacked several important features which immediately impacted our development process. First, the callbacks of windows, pop-up dialogs, and widgets were not incorporated in the tool as had been the case with DevGuide. These callbacks had to be added manually to the generated code. Second, regeneration of the code resulted in the overwriting of any such added code. The actual callback functionality was segregated in a separate package, but the callbacks themselves were not protected. Third, ScreenMachine did not include Motif bindings, only allowing access to the widget set provided. The provided widgets are a subset of the Motif set and do not include several critical widgets such as FileSelectionDialog. We built our own FileSelectionDialog but only connected the FileSelectionTextField and the OK Button. Fourth, geometry management is not supported since managed forms are not provided. This resulted in either non-resizable windows or windows that were unsightly after resizing. This included the CSD window and its text area. We deemed these defects and missing functionality sufficient to dismiss the use of this version of ScreenMachine. The new version of ScreenMachine (version 2.0), which addresses at least three of these four concerns, was not available in time to complete this project. When version 2.0 is available, we will evaluate it as well. As a result, we decided to evaluate another Motif/Ada GUI builder.

4.1.3 EzX®

Sunrise Software International's ezX® 3.3 became available to the University through the office of Logicon, Inc. in Montgomery. Having maintained contact with Air Force personnel at the Standard Systems Center, we learned ezX had been included in the Integrated Computer Aided Software Engineering (I-CASE) contract. I-CASE is a six hundred million-dollar project to standardize software development for the Department of Defense. Logicon is the prime contractor for the project and graciously invited us to use their demonstration system in Montgomery.

Our experience with ezX was less than satisfactory. Although the tool was relatively easy to use and included the full Motif widget set, it lacked stability and consistency. For

example: selecting a widget on the design screen and pressing the delete key on the keyboard resulted in a program termination during any ensuing operation. Additionally, callbacks did not behave properly. Even the provided callback examples behaved inconsistently. In one particular example, which demonstrated the callback to pass the value in a text widget, the callback worked as long as there was no text in the text widget. As soon as text was entered into the widget, the callback ceased to function. A third defect in the tool is that on occasion the Control Structure Diagram window's menu would collapse for no apparent reason. This was disturbing as it was unpredictable and the only solution was to exit without saving. ezX does not directly include callback code, but incorporates a "script" which allows the user to specify the name of a callback function. ezX will then generate a function shell where the user can place the callback code.

Geometry management is supported, resulting in the ability to resize screens without unsightly widget behavior. This tool brought us to the point where we felt we had a Motif/Ada version that was comparable to the original DevGuide design.

4.1.4 Builder Xcessory™

Integrated Computer Solutions Incorporated's (ICS) Builder Xcessory™ 3.1 was acquired by the University for a thirty-day evaluation period. It directly supports the Motif 1.2 widget set, associated resources, and Xt Intrinsics. Geometry management is supported, as is the ability to import DevGuide (.G) data files and User Interface Language (UIL) files.

Of all of the UNIX-based GUI builders evaluated, Builder Xcessory provided the most promise for creating a GUI in Ada. It included an Ada/Motif binding developed by Systems Engineering Research Corporation (SERC). However, problems were encountered with the UIL2Ada conversion routine and the SERC Ada/Motif bindings. UIL2Ada generates Ada source code from the UIL file created by Builder Xcessory. Apparently, these Ada specific routines had not been updated to work with Rational Apex 1.6.1C. After four weeks of attempting to build a GUI in Ada with Builder Xcessory, a decision was made to delay the Ada implementation until ICS and SERC solved the problems.

The development of the GUI began to move forward again toward a C implementation. Builder Xcessory was used to generate C (with direct calls to Motif) rather than Ada. The generated C code then had to be completed with detailed, hand-written callback routines written in C, not in Ada as planned. An Ada main program was required so that Ada and C programs could be integrated. The main program (Ada) calls the GUI (C) which, in turn, calls the routine Generate_CSD (Ada). This routine calls the Filter (Ada), the Tagger (C), and the Renderer (Ada).

4.1.5 ActivAda™

Thomson Software Products' ActivAda™ 1.2 is being used to develop a Microsoft Windows prototype for GRASP. Although we experienced several problems with it, the ActivAda GUI builder is a more robust tool than most of the previously mentioned Unix tools. This tool implements the Microsoft® Windows win32's which are widgets and operations utilizing them. The win32 API includes, besides the open file dialog, a print setup, font select and color select dialog.

Several problems were noted during the use of this tool as well. Once several windows are created with this GUI builder, it becomes unstable. The names of windows tend

to change as a result of maneuvering through them. This resulted in the same name being applied to two (or more) different dialogs. The only solution is to look at the dialog names and rename them if necessary, before saving the file. Additionally, the GUI builder does not support Ada strings very well. The Ada concatenate symbol "&" is not accepted by the tool and so truncates strings at the "&" symbol.

This version of GRASP, though a prototype, is actually more functional than any of the Unix versions. Some of the most important functionality yet to incorporate is the Multiple Document Interface, printing and controlling screen font size. The print and screen font selection should be implemented by the end of the quarter.

4.2 Font Development

In Version 3, screen display is facilitated by sending the CSD file to a CSD window opened under an X Window manager. The default CSD screen font was a bitmap 13 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 9 to 24 point are user selectable. These fonts were later converted to OpenWindows fonts which has since become the version supported in the distribution tar file.

Printing in Version 3 was accomplished by converting the CSD file to a PostScript file and then sending it to a printer. 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.

The upgrade from Athena to Motif widgets encouraged the use of a new Unix compatible PostScript® Type1 font. Altsys' Fontographer® 3.5 was selected as the font building tool. Fontographer is a Microsoft® Windows™ program with which one can draw each character of a font set. Once a font is created many different font files can be generated, including PostScript Type1 and TrueType.®

Each of the GRASP 3.0 characters (Figure 7) was individually re-created (Figure 8) using Fontographer. During this font re-drawing process we decided it would be a significant improvement to represent Ada program units (packages, subprograms, tasks and generics) as individual character symbols rather than the old GRASP box drawings. These new program unit symbols can be seen in the lower part of Figure 8. This new representation uses less space and is visually more meaningful. Additionally, the CSD can now differentiate graphically between the specification and body of a program unit.

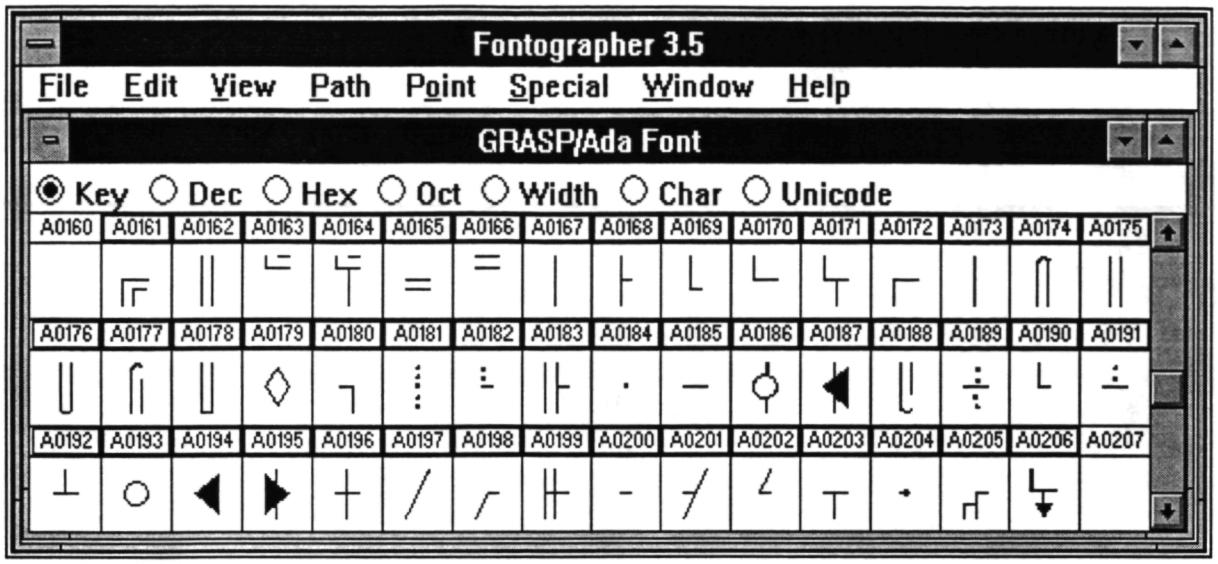

Figure 7. GRASP Characters without Program Unit Symbols

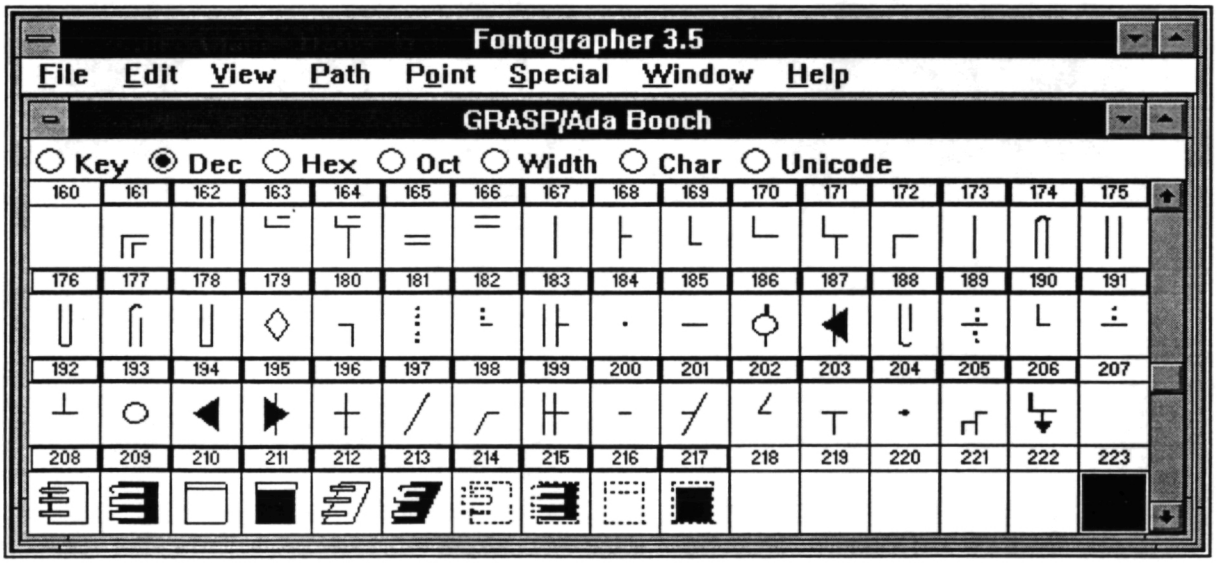

Figure 8. GRASP Characters with Program Unit Symbols

4.3 User Interface Description

The ***Control Panel***, shown in Figure 9, gives the user capabilities for creating or viewing a Control Structure Diagram (CSD) in a CSD window, shown in Figure 10. A future version of the tool will allow multiple CSD windows to be opened to access several CSDs at once.

Figure 9. Control Panel

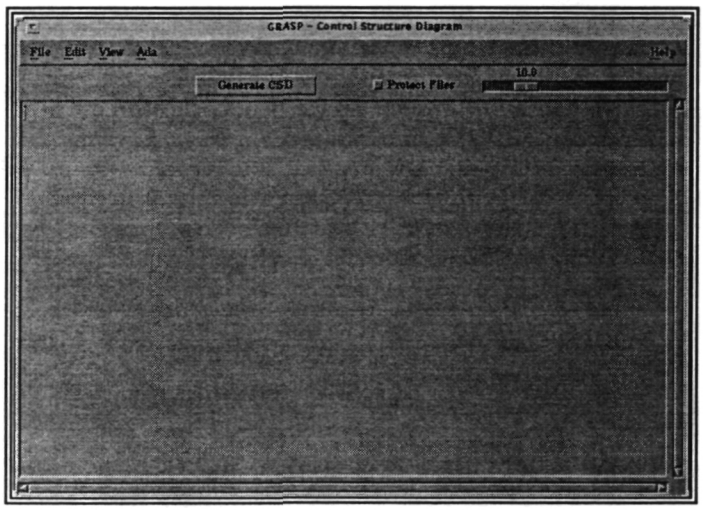

Figure 10. Control Structure Diagram

Control Panel Options. The Control Panel window options are described below.

File - Allows the user to select from the following menu items shown in Figure 11.

Create CSD/ODG - User may open an empty CSD (default) or ODG window.

Create CSD - a CSD window is opened.
Create ODG - ODG window option is for future use.
Open - Displays an Open File Dialog so the user may select a file to load into the CSD/ODG window as it is created.
Save All Files - Saves each open CSD/ODG file with its current name.
Exit - Closes the Control Panel and exits GRASP.

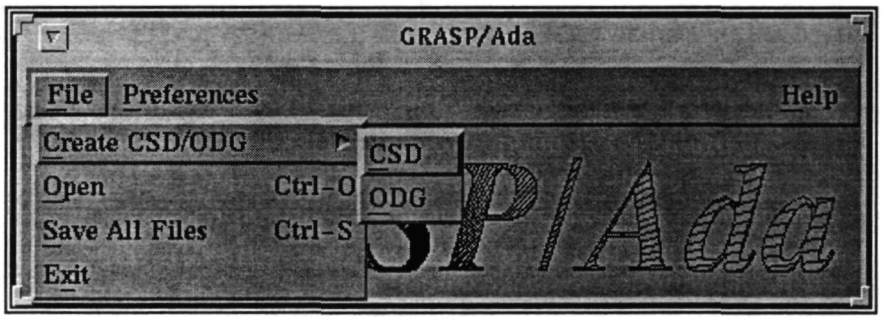

Figure 11. Control Panel - File Menu

Preferences - Allows the user to set the preferences using the menu shown in Figure 12. Future

Backup Options - User may indicate a desire to keep backup copies of data files and how they are to be named.

Location of Files - User may select a location for backup files.

Code Attributes - User may set font preferences for elements of the code.

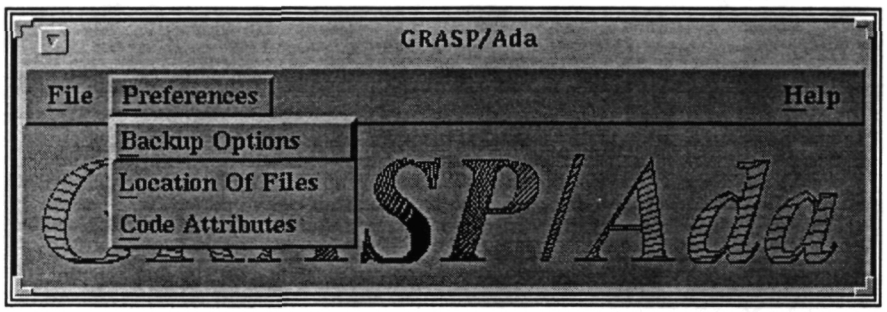

Figure 12. Control Panel - Preferences Menu

Help - Allows the user to obtain help on the following items from the menu shown in Figure 13.

On Context - Context sensitive help. Future.

On Help - User may obtain help on using the help facility. Future.

On Keys - Help on keyboard key functions. Future.

On Version - Display version menu as shown in Figure 14.

Credits - Information on the writers of GRASP (see Figure 15).

Feedback - To provide feedback on GRASP (see Figure 16).

Version Info - GRASP version number (see Figure 17)

On Window - User may obtain help on this window. Future.

Index - A complete index of the help facility. Future.

Search - User may search the help facility. Future.

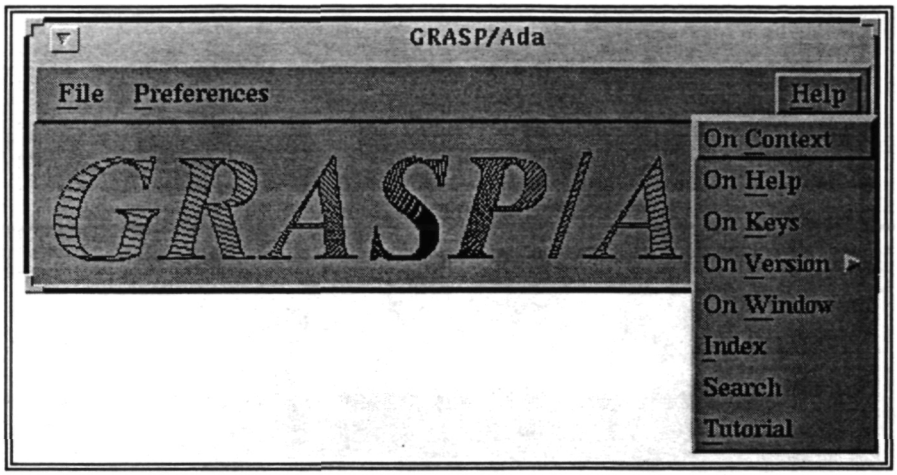

Figure 13. Control Panel - Help Menu

Tutorial - A GRASP tutorial. Future.

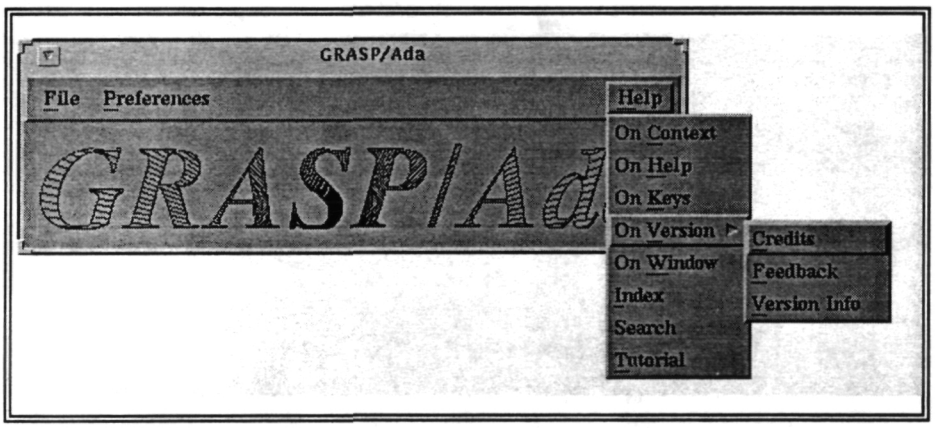

Figure 14. Control Panel - Help Version Menu

Figure 15. Credits Dialog

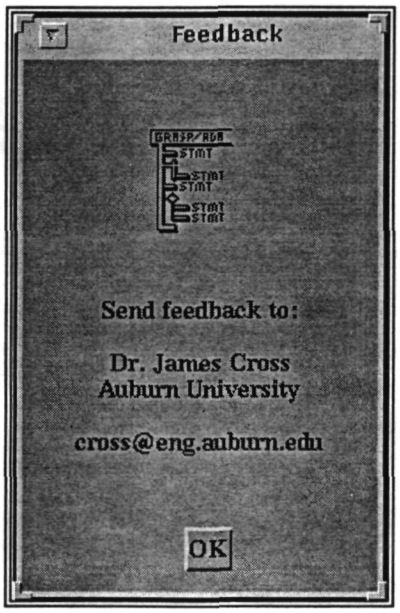

Figure 16. Feedback Dialog

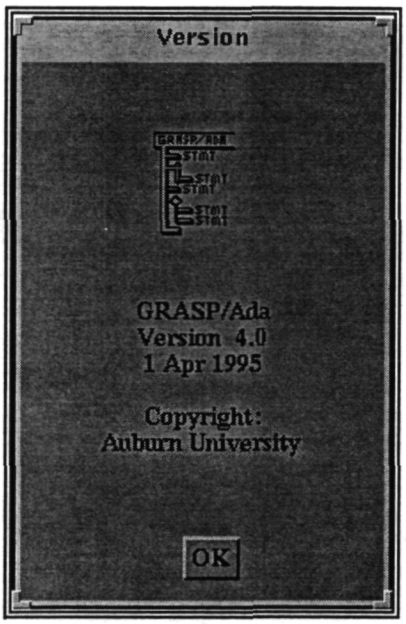

Figure 17. Version Dialog

Control Structure Diagram Window. The ***Control Structure Diagram*** window, shown in Figures 16, provides the user with capabilities for creating, editing or viewing CSDs. The CSD window options are described below.

File - Allows the user to select from the following menu items as seen in Figure 18.

> **Clear Window** - User may delete the contents of the text area. A warning dialog appears to allow the user to verify (see Figure 19).
>
> **Open** - Displays an Open File Dialog so the user may select a file to load into the CSD.
>
> **Save** - Saves the open CSD file with its given name.
>
> **Save As** - Allows the user to rename the open file using the Save As Dialog (see Figure 20) .
>
> **Print** - Allows the user to select from the following.
>
> > **Express Print** - Prints the loaded file without accessing the print dialog.
> >
> > **Print Setup** - Allows the user to go to the print dialog.
>
> **Exit** - Closes the CSD window.

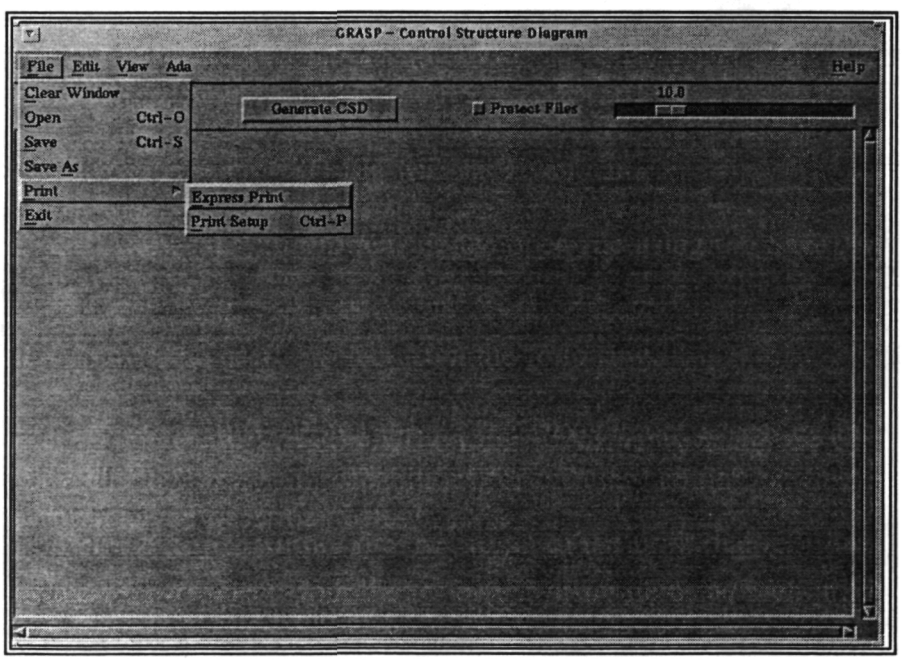

Figure 18. Control Structure Diagram - File Menu

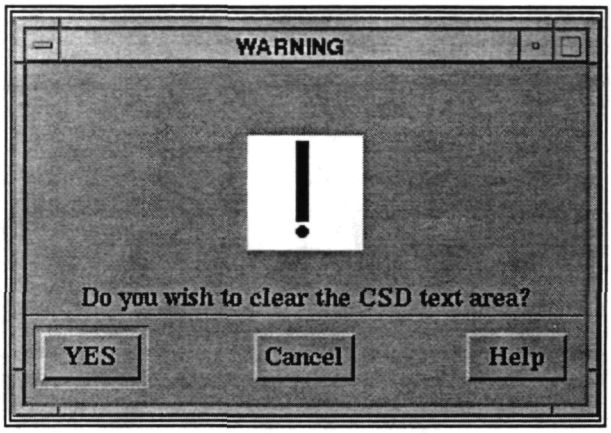

Figure 19. Warning Dialog

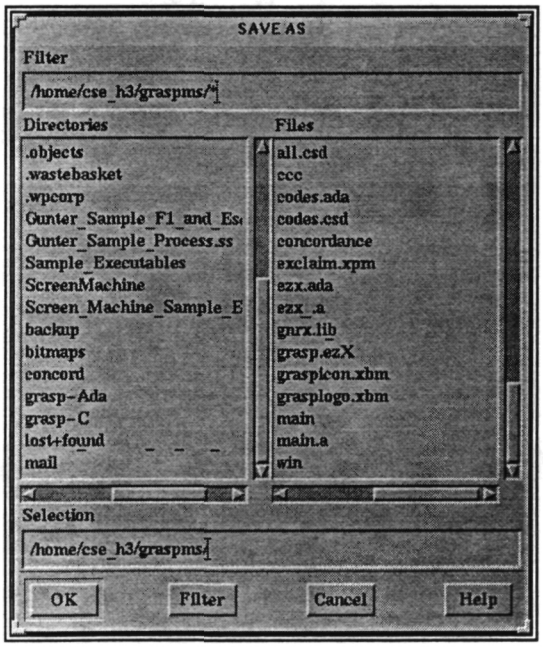

Figure 20. Save As Dialog

Edit - Allows the user to select from the following menu items as seen in Figure 21.

Undo - Allows the user to reverse the last edit operation.

Cut - User may remove a pre-selected section of text. The removed text is placed in the clipboard.

Copy - User may copy a pre-selected section of text. The copied text is placed in the clipboard.

Paste - Data from the clipboard is inserted into the CSD text area at the insertion point.

Search/Replace - Opens the Search/Replace Dialog (See Figure 29). Text strings may then be searched for and replaced as desired. Future.

Go To Line - Allows the user to specify, in the Go To Line dialog (see Figure 22), a particular line in the file to view. Future.

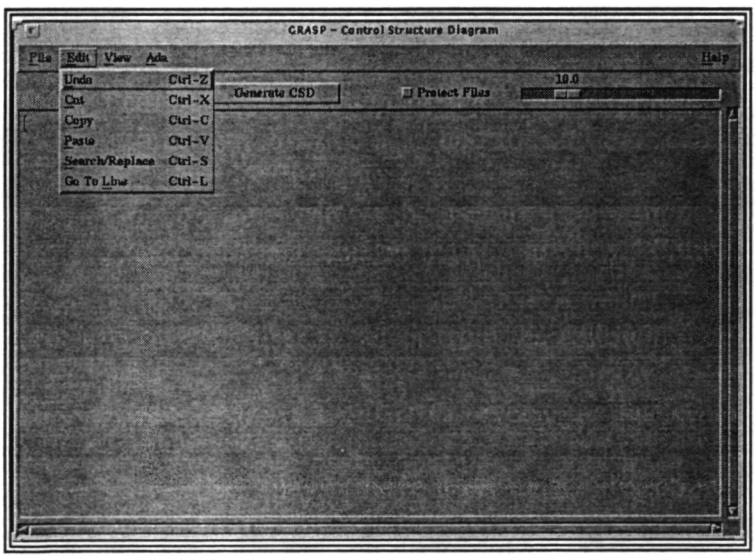

Figure 21. Control Structure Diagram - Edit Menu

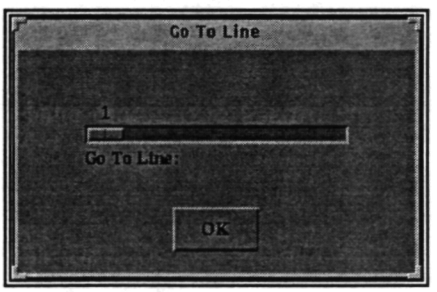

Figure 22. Go To Line Dialog

View -Allows the user to select from the following menu items as seen in Figure 23.

Graphics - Allows the user to turn the GRASP diagram characters on and off. Future.

Line Numbers - Allows the user to turn line numbering on and off. Future.

Line Spacing - Allows the user to select line spacing for the CSD. Future.

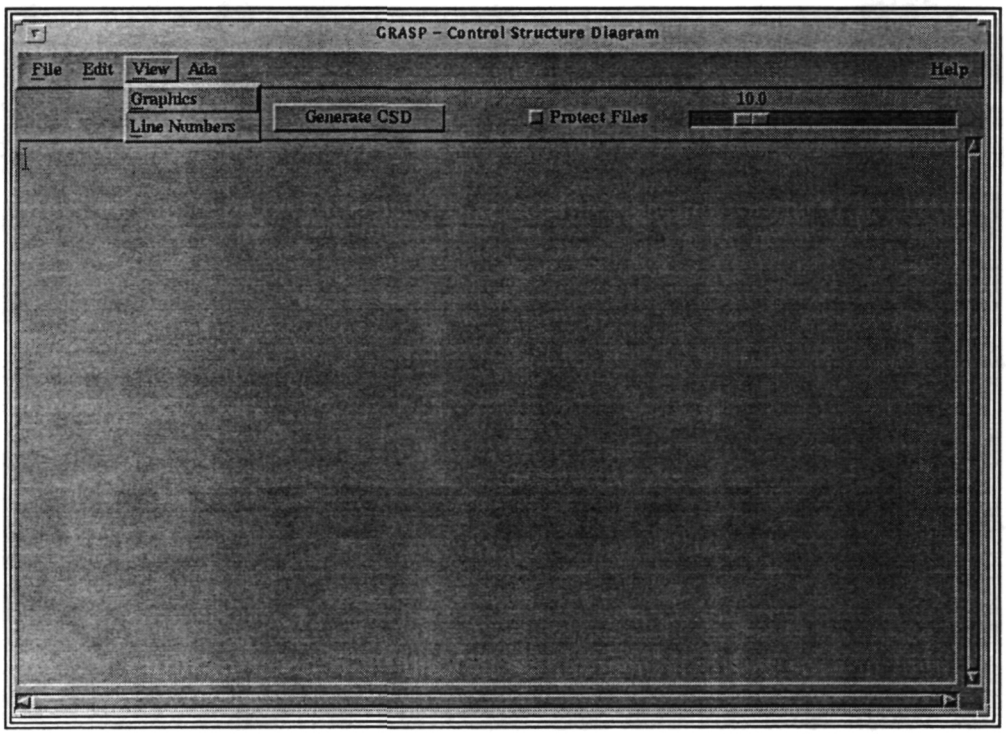

Figure 23. Control Structure Diagram - View Menu

Ada - These menu items allow the user to insert syntactically correct blocks of Ada code into the CSD text area at the location of the curser. This menu item is "pinnable" (see Figures 24 - 26). Several options are available for inserting text from a menu. A straight menu (as shown) could be used, or a cascading menu (e.g. only one "Loop" menu item which, when selected, cascades into the three different loop constructs) could be utilized to reduce the size of the pinned menu (see Figure 26). The Ada blocks include the major Ada constructs as well as two user defined custom blocks.

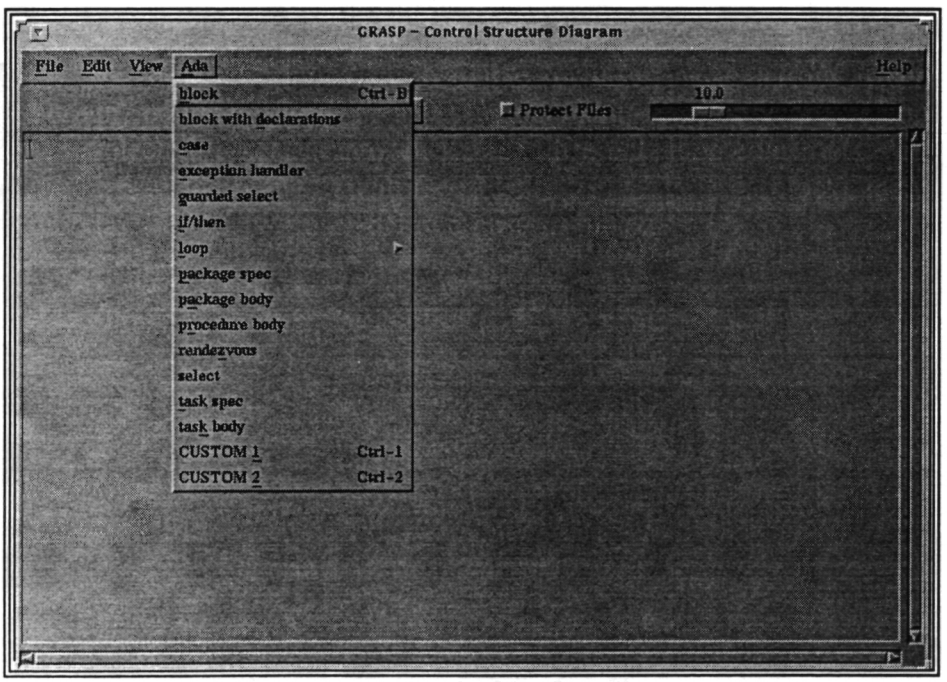

Figure 24. Control Structure Diagram - Ada Menu

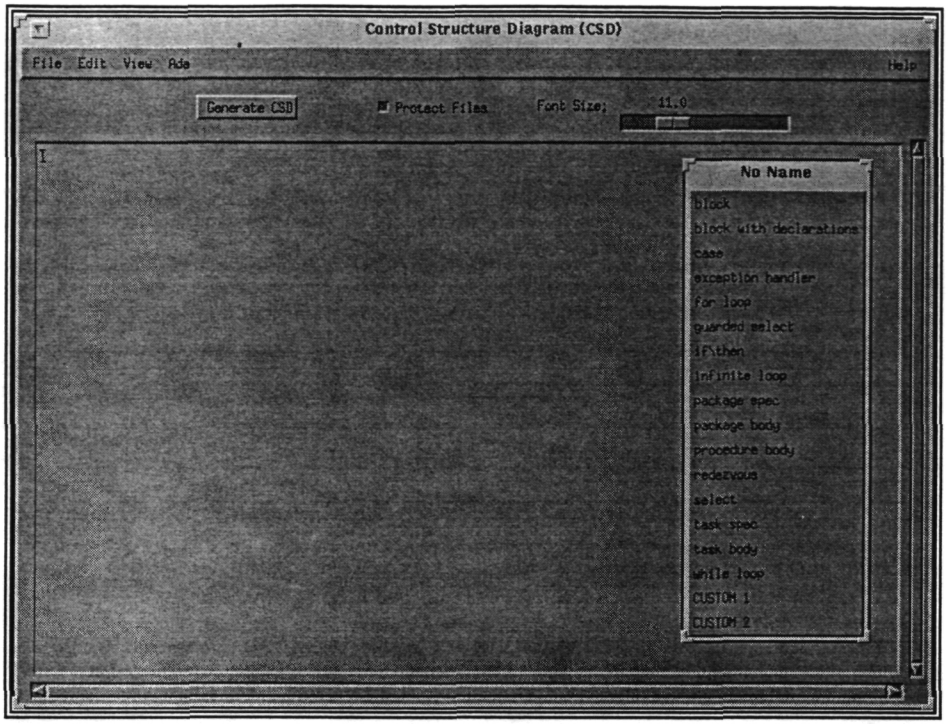

Figure 25. Control Structure Diagram - Ada Menu "Tear Off"

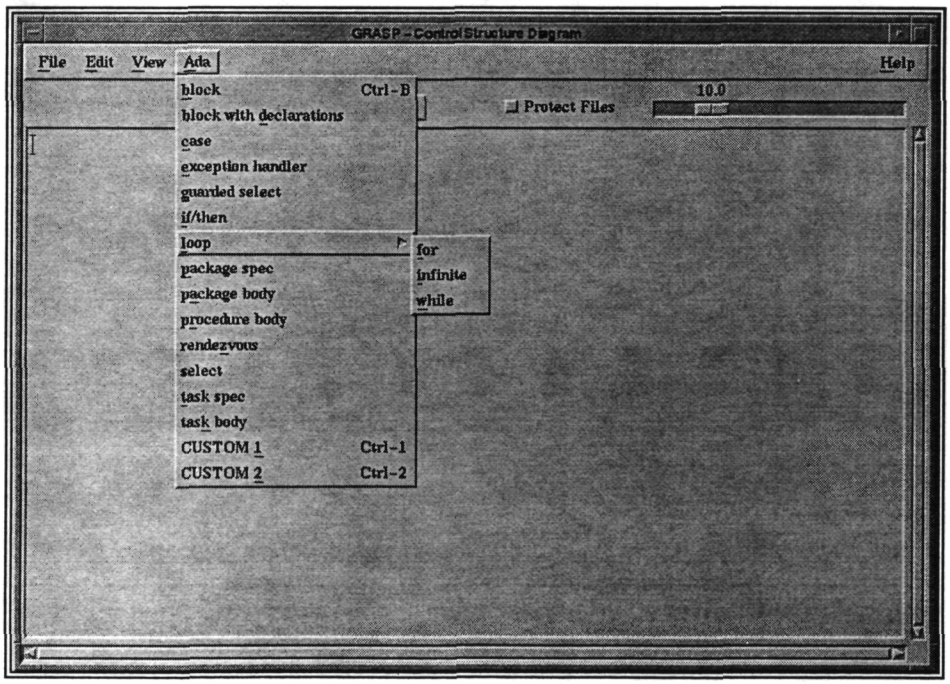

Figure 26. Control Structure Diagram - Ada Menu - Cascading Loop

Protect Files - Allows the user to set an option to prevent overwriting existing files without confirmation.

Generate CSD - This button generates a CSD from source code and/or regenerates a CSD after modification. When the CSD window is opened and loaded with a source file without a .csd extension, a separate CSD window is automatically opened to display the CSD when it is generated. Note that CSD graphics characters, if any, are filtered prior to the parse or reparse.

Font Size - Allows the user to select font sizes ranging from four points to one hundred points. The default font size for the CSD window is 14 points.

Language - Currently, a language selector is not displayed on the Unix screens. Additional consideration must be given to the location of this selector. If displayed on the CSD window, the user may attempt to change to a different language after a file is loaded. Future.

Help - See the help section above for the **Control Panel**.

The ***Code Attributes*** window, shown below in Figure 27, provides a prototype screen for setting text attributes in the CSD text area. The user may select the type (comment, graphics, identifiers, pragmas or reserved words) and then the desired appearance. The OK button would apply the settings to the text displayed in the CSD. Future.

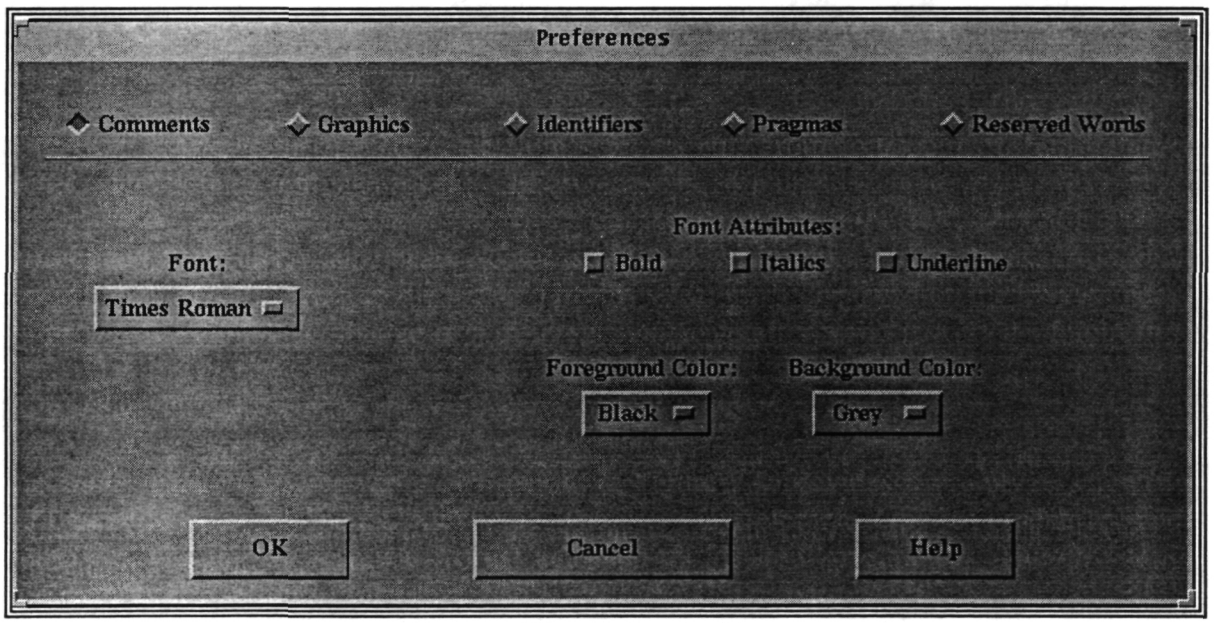

Figure 27. Code Attributes

Print Window. The ***Print*** window, shown below in Figure 28, provides the user with capabilities for selecting a printer, toggling page numbering, page header information, and selecting a file to print.

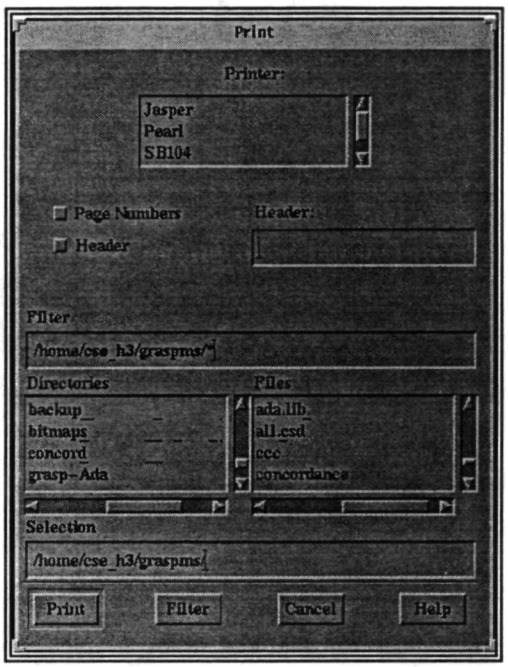

Figure 28. Print Dialog

Search/Replace Window. The ***Search/Replace*** window, shown below in Figure 29, allows the user to search for a text string entered into the upper text area and replace matching string(s), if desired, with a text string entered into the lower text area.

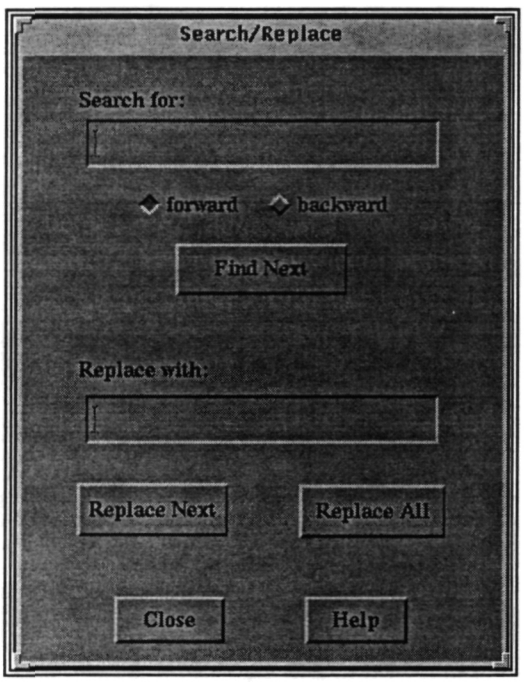

Figure 29. Search/Replace Dialog

4.4 User Interface - Future Directions

The GRASP interface will require minor modifications as additional components are connected. Further adjustments are anticipated as the tool is beta tested by Auburn CSE students. The Version 4 GUI is a major enhancement over the previous version.

The attempt to find and utilize a state-of-the-art Ada/Motif GUI builder has been frustrating. In the long run, however, the right tool should result in lower maintenance effort and "better" code. The following are essential for a GUI builder tool: (1) the tool must be capable of accessing all of the Motif widgets and their associated functions, (2) the tool must be capable of administering all callback code within the tool, and (3) the tool must be capable of managing multiple source files (e.g. one dialog per file). Once these criteria are met, Ada/Motif GUI builders will have come of age.

The following three items would yield a product that would greatly enhance the comprehension of software and allow the use of this tool by a large percentage of the software engineering community.

1. A Windows version is a must. When GRASP is able to run on a X86 class laptop in Windows, users will have the convenience and ease-of-use they have come to expect from professional quality software products.

2. Hypertext in applications like GRASP will soon be the norm. For example, double-clicking on a package specification symbol in an object diagram and having a CSD window open to display the CSD of the package body.

3. Support for "collapsing" diagrams should be developed. Constructs such as case statements and loops could collapse to a single symbol.

5.0 Control Structure Diagram Generator

The GRASP/Ada control structure diagram generator (CSDgen) is described in this section from a technical and developmental perspective. Since display mechanisms for both the screen and printer are an integral part of the CSDgen application, these are included in this section as well. A more complete history and rationale for the development of the CSD is contained in [CRO93]. The graphical constructs produced by CSDgen are summarized in Figure 5 (Section 2).

5.1 Generating the CSD

The primary function of CSDgen is to produce a CSD for a corresponding Ada source or PDL file. 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 and error reporting are attempted when a syntax error is encountered. The diagram is simply generated down to the point of the error. In the case of Ada PDL, non-Ada statements must be valid Ada identifiers so that they are treated like procedure calls. For example, the PDL for "search array for largest element" could be represented as "Search_array_for _largest_element" so that the phrase becomes a single identifier.

In the new version, the source code is first parsed/tagged using the GRASP Markup Language (GML), and then the tagged source code is rendered into a CSD. With a well-developed scanner for the Renderer, the additional overhead of the two-step process should prove negligible and language portability will be greatly enhanced. Using a markup language also removes some of the deficiencies and constraints of Version 3. For example, the Version 4 renderer maintains the location of user line feeds and comments, and attempts to output a more compact CSD. Figure 30 shows a detailed view of the CSD Generator (CSDgen).

5.1.1 The Tagger and Markup Language Concept

As indicated in Section 3, the tagger has its own parser/scanner built using LEX and YACC. The tagger utilizes GRASP-ML [CRO95], a markup language based on SGML, to tag all necessary language elements required for input to the respective renderer. The tagger inputs Ada PDL or source code and produces a tagged file.

Markup is defined as the process of embedding special marks (tags) in the content of document to explicitly indicate structure, formatting or other information. Several software packages use markup but the "tags" are usually transparent to the user. WordPerfect, for example, uses markup to control all aspects of its presentation of text and graphics. The WordPerfect markup tags can be seen using the Reveal Codes feature but they can not be entered directly by the user. The tag set that has been developed for Version 4 is based on Standard Generalized Markup Language (SGML), an ISO standard meta-language for specifying grammars that has been used for a variety of software tools [CRO95].

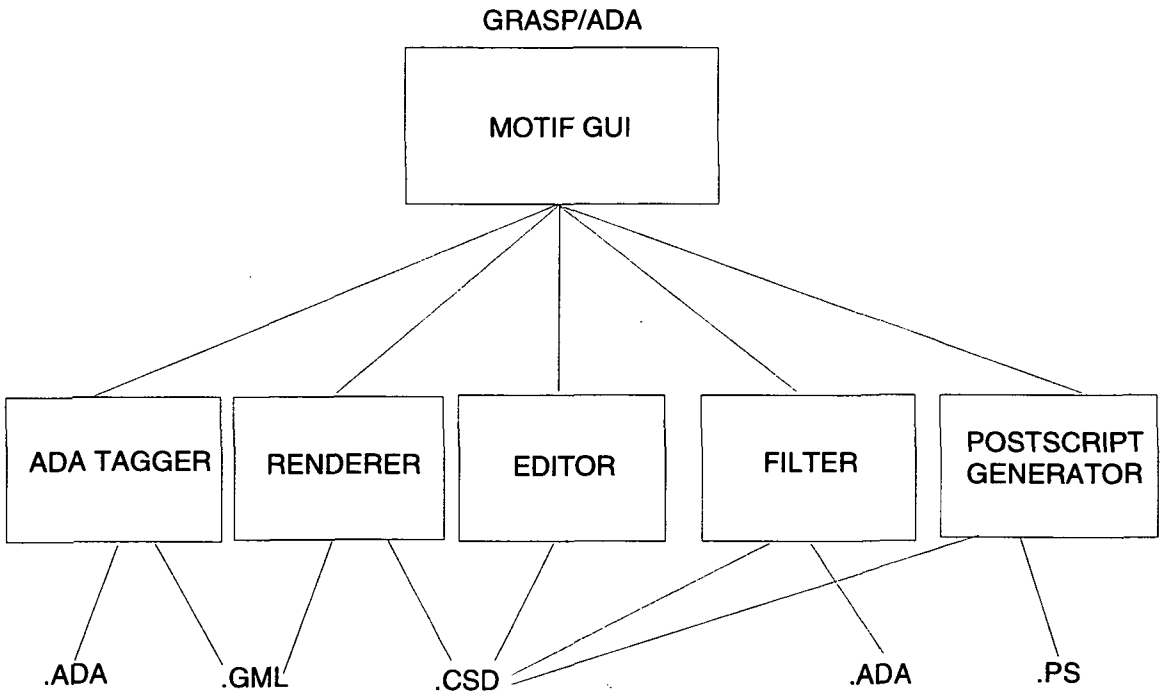

Figure 30. CSD Generator (CSDgen) for GRASP Version 4.

The GRASP Markup Language (GML) tag set was developed to promote language independence for the generation of graphical representations that has been previously restricted to Ada [HEN95]. If a tag set is used as a basis for rendering, several different language-specific taggers can be developed to work with the same language independent renderer. Having been carefully crafted so as to parallel grammar rules which describe program structures, the GML tags are used to further facilitate language independent CSD rendering.

5.1.2 The Renderer

The initial CSD generator, developed for GRASP/Ada 3.0, was implemented in the C programming language [MAY92]. The new Version 4 iteration is being developed in the Ada programming language in the Rational Apex environment. Version 4 uses the GRASP markup language, GML, to tag the source code during the parsing part of the CSD generation. The Renderer processes a GML tagged file to produce the CSD file. Many of the same construct ideas that appear in the C version are used in the new Ada version. However, there are some fundamental differences that had to be addressed. The architecture of the Renderer is shown in the Ada module diagram in Figure 31. Details on the implementation of the Renderer can be found in [RAN95].

The main procedure body of the new renderer was implemented using AFLEX. AFLEX was created by John Self of the Arcadia Project at the Department of Information and Computer Science at the University of California, Irvine. AFLEX was developed to be an Ada alternative to FLEX, the popular GNU lexical analyzer for C. It operates in much the same manner as FLEX and uses many of the same "conventions." AFLEX is used to build an efficient scanner that can read files and search for particular regular expressions. Once these regular expressions are matched, specific program fragments can be executed [SEL90]. The scanner built for the Renderer in GRASP Version 4 reads in source code tagged with GML and generates the CSD by calling appropriate action routines.

The Renderer uses a doubly linked list of records as an efficient way to build the CSD prefixes. This list can be thought of conceptually as going from left to right. Most of the actions performed on the list occur on the right side of the list. The Renderer produces the **.CSD** output file one line at a time. To accomplish this, both the CSD prefix and the program text must be buffered. A record with two fields is used to accomplish this. One field holds the prefix buffer and the other holds the text buffer. The text buffer simply receives text straight from the AFLEX generated scanner. The prefix buffer is more complicated. The prefix buffer is built by reading the prefix list starting from left to right. At each node of the list, the Rule field and the Code fields are read. When the prefix is output in procedure Output_Csd_Buffer, the ASCII characters in the prefix buffer are offset by 96 to put them in their appropriate positions between 160 and 206.

Error Handling. The Tagger will tag an error when it occurs and then continue tagging a file. The Renderer handles the error by capturing the error text and transferring it to a file along with the line number that the error occurred on. This error information will be used by the interface to show the user where the error occurred. The following is an example of how an error is tagged:

```
end loop</_loop_><_error_>*missing semicolon*</_error_>
```

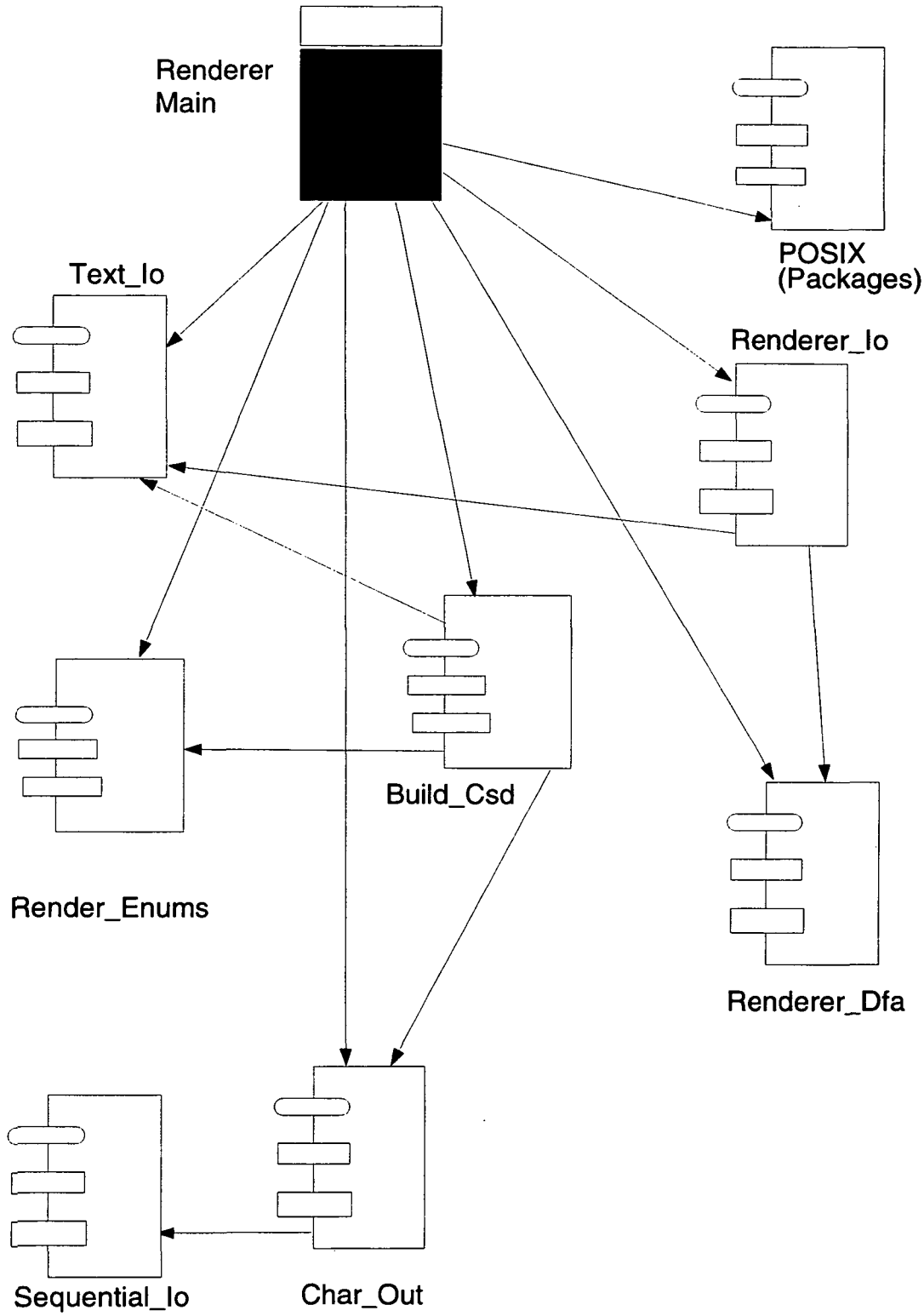

Figure 31. GRASP/Ada Version 4 Renderer Ada Module Diagram.

In the GRASP renderer most tags must have a tag to "begin" a construct and a tag to "end" the construct. When an error occurs, there is no guarantee that every start construct tag will have a corresponding end tag. When such a tag imbalance occurs, an Ada CONSTRAINT ERROR (NULL ACCESS) is raised. The Renderer handles this particular error by ignoring it and allowing rendering to continue. In such a case, the CSD may look grossly incorrect, which provides a visual indication to the programmer that an error has occurred.

Creating A .CSD File. The CSD prefix codes are displayed and/or printed using a custom font. The characters used to generate the prefix are mapped into positions 160 to 206, which supplement the standard ASCII characters in positions 0 to 127. Some Ada 83 compilers will allow Text_Io to handle characters beyond position 127. For Example, as part of the migration to Ada 95, Rational's Apex compiler, will allow character positions 0 to 255. However, in order to enable portability between all Ada 83 and Ada 95 compilers, a different way of handling characters for CSDs was needed. One approach considered was to implement an Ada user defined character set. The problem with this method was that all the operations for the new character set would also have to be defined. Another approach, and the one selected, was to represent the characters as a subset of Integer with a range of 0 to 255. An Ada representation clause was also used to ensure that the new integer type was constrained to eight bits. Instead of using Text_Io to output the characters to a file, the Ada Sequential_Io package was used. This method seems to work well and appears to be compatible on all compilers. This Ada Text_Io problem will be resolved with the release of Ada 95 compilers which support ASCII characters beyond 127.

5.1.3 Changes From Version 3 for Generating CSDs

Preserving Line Feeds. While GRASP 3.0 removes all user line feeds then inserts line feeds only where it is necessary to draw the CSD, the new GRASP 4.0 maintains all user line feeds. In addition. the renderer inserts the necessary line feeds for drawing the control structure diagram. All user entered Line Feeds are captured at the scanner level. Some of these line feeds are output immediately while others are stored temporarily and output as soon as possible (in their appropriate position). The reason for this temporary storage of line feeds is that occasionally the CSD Buffer must be changed before it can be output. For example, a named loop requires changing the CSD Buffer. As described earlier, the <_stmt_> tag is used to surround an assignment statement:

```
<_stmt_> A := A + 1 </_stmt_>
```

This same tag, however is also used to surround a named loop:

```
<_stmt_>A_LOOP:
<_pretest_loop_> for i in 1..25 loop
                    ....
                end loop </_loop_></_stmt_>
```

This created a problem since the beginning of the CSD prefix for an assignment statement is different from the CSD prefix for the beginning of the named loop. The problem occurred when a user line feed was entered between the <_stmt_> and the <_pretest_loop_>

tags. If this line feed was not buffered, the CSD buffer would be output at a point before the <_pretest_loop_> tag was ever reached. The solution to this problem was to temporarily store line feeds after the <_stmt_> tag and then change the CSD Buffer whenever <_pretest_loop_> is processed. When <_pretest_loop_> is encountered, Handler uses a Loop_Block_Tag "tag kind." Handler checks to see if Simple_Stmt is in the rightmost position in the list. If so, it changes the buffer before adding the loop construct.

Even though user line feeds are preserved, not all user white space is rendered. All user tabs and spaces at the beginning of a line are removed in order to maintain control structure diagram consistency. However, All white space before a comment is preserved to provide proper formatting as described in the next section.

User Comments. GRASP Version 3 does not handle user comments very gracefully when rendering CSDs. Comments are completely removed from where they are located and put on either the preceding or the following line. This is not considered appropriate because the user may have intended the comment to remain on the same line. This would be important if the user was commenting an enumerated type or the identifiers within a statement as in the example below.

(ORIGINAL TEXT)

```
A := A      -- This is very important
   + B;     -- This is also important
```

(VERSION 3)

```
-- This is very important
-- This is also important
A := A + B;
```

(VERSION 4)

```
A := A      -- This is very important
+ B;        -- This is also important
```

(ORIGINAL TEXT)

```
X :Array(1..3) of Integer:= (1,  -- First
                             2,  -- Second
                             3); -- Third
```

(VERSION 3)

```
-- First
-- Second
-- Third
X :Array(1..3) of Integer:= (1,2,3);
```

(VERSION 4)

```
X :Array(1..3) of Integer:= (1,  -- First
2,  -- Second
3); -- Third
```

The ability to maintain user comments in their current location is an important improvement made by the latest Renderer.

Construct Changes. In the new GRASP Renderer every attempt was made to keep the generated CSDs as compact as possible. One new addition is the short end hook on the end of a nested compound statement. In GRASP Version 3, long end hooks were inserted on a new line. This spreads out the code unnecessarily and is not particularly aesthetically pleasing. The line feed buffering technique that was mentioned previously was used to put in the short end hooks in the necessary places. Boxes around procedure calls were also eliminated to facilitate code compactness and to make tagging procedure calls easier for the parser. Figure 32 and Figure 33 illustrate the difference between compactness of code in Version 3 and Version 4.

A guarded select is rendered differently in the new version. Figure 34 shows a Version 3 CSD of the guarded select. Since the guard represents a conditional, it made sense semantically to display a guard in a similar manner to a conditional. The rendering of the guard on the select now resembles the when condition of a case statement. Figure 35 shows a Version 4 CSD of the guarded select. Figure 36 shows the corresponding GML file. This change was necessary because when the parser was looking at a select, it could not look far enough ahead to determine that a guard was coming. There was no way to tag it so that the Version 3 blackened circle could be output.

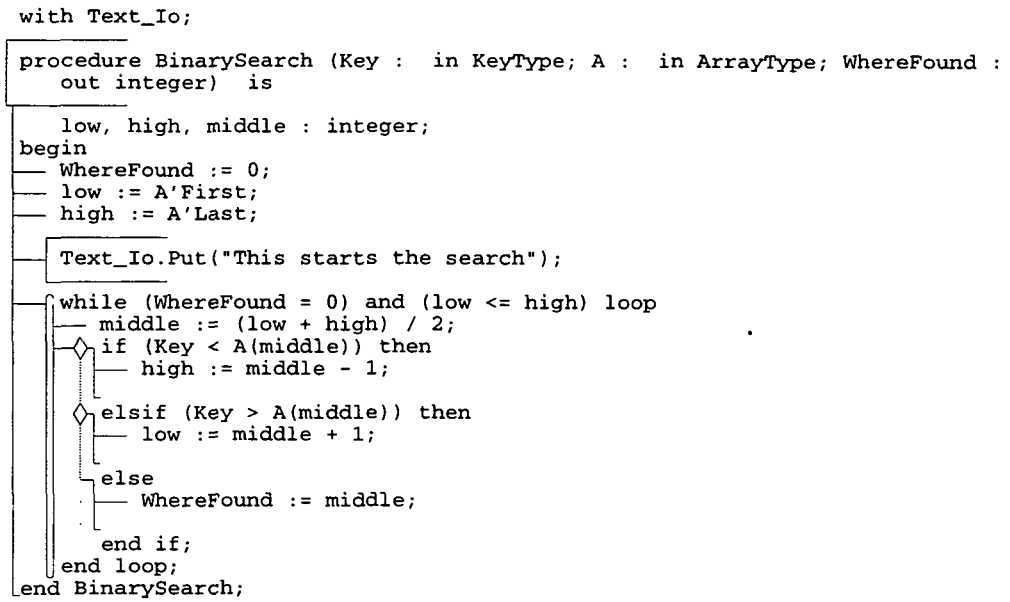

Figure 32. GRASP/Ada Version 3 CSD Compactness Example

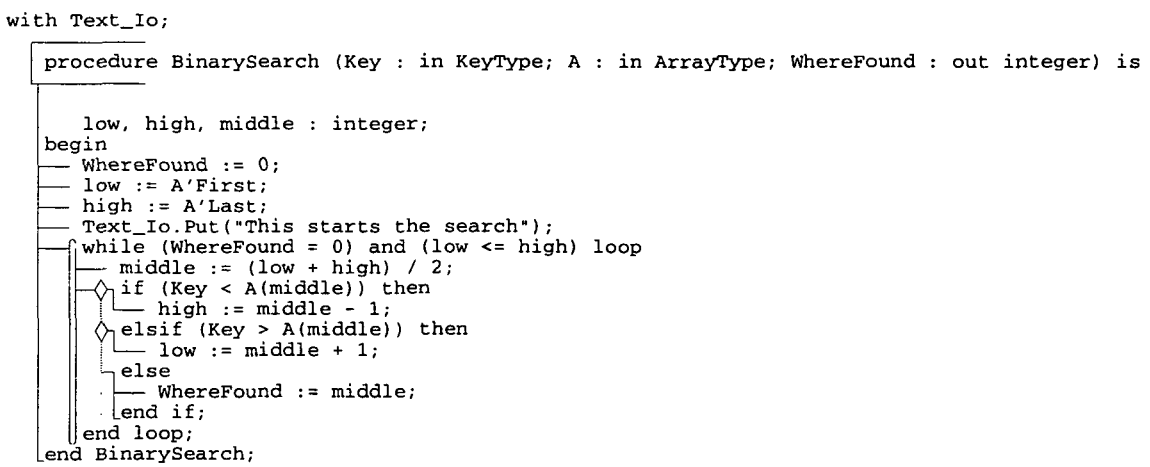

Figure 33. GRASP/Ada Version 4 CSD Compactness Example

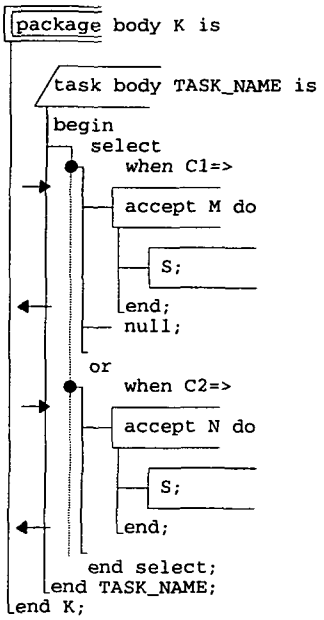

Figure 34. GRASP/Ada Version 3 Guarded Select Example.

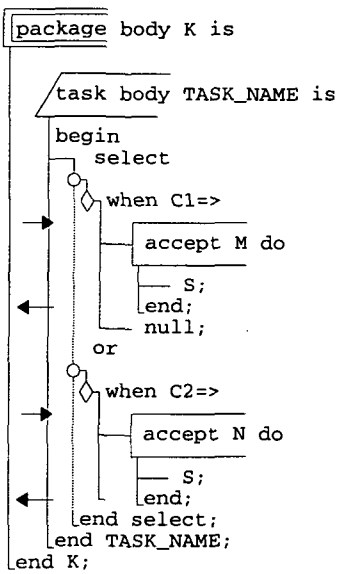

Figure 35. GRASP/Ada Version 4 Guarded Select Example.

```
<_unit_><_packbody_><_packheading_>package body K is</_packheading_>
<_declaration_><_decl_>
<_taskbody_><_taskheading_>task body TASK_NAME
is</_taskheading_><_taskstmtpart_>begin
<_select_>select
<_selection_><_guard_>when C1=></_guard_>
<_accept_><_accept_heading_>accept M do</_accept_do_heading_>
<_stmt_>S;</_stmt_>
<_end_>end;</_accept_>
<_stmt_>null;</_stmt_>
</_selection_></_selection_><_or_>or</_or_>
<_selection_><_guard_>when C2=></_guard_>
<_accept_><_accept_heading_>accept N do</_accept_do_heading_>
<_stmt_>S;</_stmt_>
<_end_>end;</_accept_>
</_selection_></_selection_><_end_>end select;</_select_>
<_end_>end TASK_NAME;</_taskstmtpart_></_taskbody_></_decl_>
</_declaration_><_end_>end K;</_packbody_></_unit_>
```

Figure 36. Guarded Select GML File.

5.2 CSD Generator - Future Considerations

As indicated above, the actual generation of the CSD and the subsequent display of it on the screen or printer are in some ways inseparable. For example, margins, line spacing, and indentation of the CSD constructs could be part of the actual generation of the CSD or they could be a function of the display mechanism tightly coupled with a CSD editor. Hence, there is some degree of overlap in the discussion below regarding generating and displaying the CSD.

5.2.1 Generating the CSD - Future Considerations

Ada 95. The Ada 95 specification will impact the CSD generator in at least two important ways.

(1) **New 95 control constructs** - Additional CSD graphical constructs must be created as appropriate. Addressing the new 95 control constructs should be relatively straightforward. Numerous examples will need to be diagrammed and evaluated for comprehensibility and ease of implementation and integration into the current set of constructs.

(2) **The Ada 95 specification allows all 256 ASCII character codes** - These are allowed to facilitate international character sets. The use of all 256 character codes presents somewhat more of a challenge. In the present prototype, ASCII codes above 128 have been used for the CSD graphics characters. Several operations in the user interface have freely filtered these character codes during the regeneration process as well as the "save as Ada" operation. Obviously, this approach will not be acceptable if the source code itself contains character codes in this range.

Internal Representation of the CSD - Alternatives. Several alternatives were considered for the internal representation of the CSD in the Version 4 prototype. Each has its own merits with respect to processing and storage efficiency and is briefly described below. These alternatives remain under consideration for future versions.

(1) Single ASCII File with CSD Characters and Text Combined. This is the most direct approach and is currently used in the Version 4 prototype for the **.csd** file. The primary advantage of this approach is that combining the CSD characters with text in a single file eliminates the need for elaborate transformation. The major disadvantages of this approach are that it does not lend itself to dynamic incremental changes during editing and the CSD characters have to be filtered if the user wants to regenerate the CSD or send the file to a compiler.

(2) 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.

(3) Single or Intermediate ASCII File Without Hard-coded CSD Characters. The **.gml** file in Version 4 is the intermediate file in this approach, which represents a compromise between the previous two. The file contains "begin construct" and "end construct" tags for each CSD graphical construct rather than all of the CSD graphics characters that compose the diagram. These are generated by the renderer for screen display and printing in the current **.csd** file. A single .gml file approach would be most beneficial in conjunction with a dynamic editor since the diagram could grow and shrink automatically as additional text/source code is inserted into the diagram (i.e., between the appropriate tags). The extent of required modifications to text edit windows must be considered with this alternative.

Ada Coding Standards. Future CSD generators should provide the user with the capability to generate CSDs (together with prettyprinted source code) according to a prescribed standard. This may include conventions for keywords, identifiers, indentation, layout for compound statements, placement of comments, etc.

Direct Generation Using the Ada Semantic Interface Specification (ASIS). If tight coupling and integration with a commercial Ada development system such as *Rational Apex* or *Verdix VADS* is desired, then the ASIS may provide for the direct generation of the CSD from its underlying intermediate representation produced as a result of compilation. This would require a layer of software which uses ASIS to call the appropriate CSD primitives as control nodes are encountered. This approach may eliminate the possibility of directly editing the CSD since the ASIS interface may not support modifying the underlying intermediate representation, only reading it. In practice, it may prove more efficient to allow the CSD generator to simply reparse the entire compilable unit being edited.

Dynamic Changes to the CSD. In the present prototype, there is no capability for incrementally modifying the CSD. When the CSD or source code is modified in the CSD window, the CSD must be regenerated by reparsing the entire file. While this has been sufficient for prototyping, especially for small programs, editing capabilities with incremental modification of the CSD may be desirable in an operational setting. A **CSD editor** would function similar to the WordPerfect editor in that it will be able to read and interpret GML tags. It will probably be implemented using X-Windows Motif and X-Windows libraries. A CSD editor could also implement several prettyprinting options such as bolding reserved words. Currently, it is somewhat dubious as to which part of GRASP should handle prettyprinting. It could be done by the Tagger, the Renderer, or a separate routine. The new GML-based CSD editor would disambiguate this problem. The ASIS cited above may offer a bridge for these incremental changes to the CSD.

Additional Languages. The current Renderer will, of course, be used to render CSDs for other languages as soon as other taggers are developed. Languages that will likely be implemented for GRASP after Ada 95 are C and C++.

5.2.2 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 4 CSDgen prototype, horizontal and vertical spacing are not user selectable. They are a part of the CSD file generation and are defaulted to that of the input file. 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 are expected to be handled by a new CSD editor, 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. 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. 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 4 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 fully integrated with a commercial CASE tool, the document preparation system provided by the CASE tool may be utilized for this function.

Navigating among CSDs, Module Diagrams, and Object Diagrams. A GRASP library is required to provide 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 via the ASIS.

Alternatives for implementing the GRASP library include the following.

(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 Rational Apex library system along with ASIS.

Of these alternatives, the first one may be the most direct approach since it would be the easiest to control. The ASIS library approach may be more useful with the addition of object diagram generation and also with future integration of GRASP with commercial CASE tools.

New CSD Fonts with Program Unit Symbols. The current Renderer does not utilize the new CSD fonts with program unit symbols. A future version will use the new notation to represent subprogram, package, task, and generic bodies and specifications. It will be implemented using a variation of the Version 4 Renderer. Figure 37 shows a CSD using the new proposed notation.

Hypertext Links. Hypertext links could be used to visit a procedure from the location it is called in the body of another procedure. This navigational feature would aid in program development and in program understanding. It would probably be implemented in part by using Rational ASIS to extract a the required information from Ada programs.

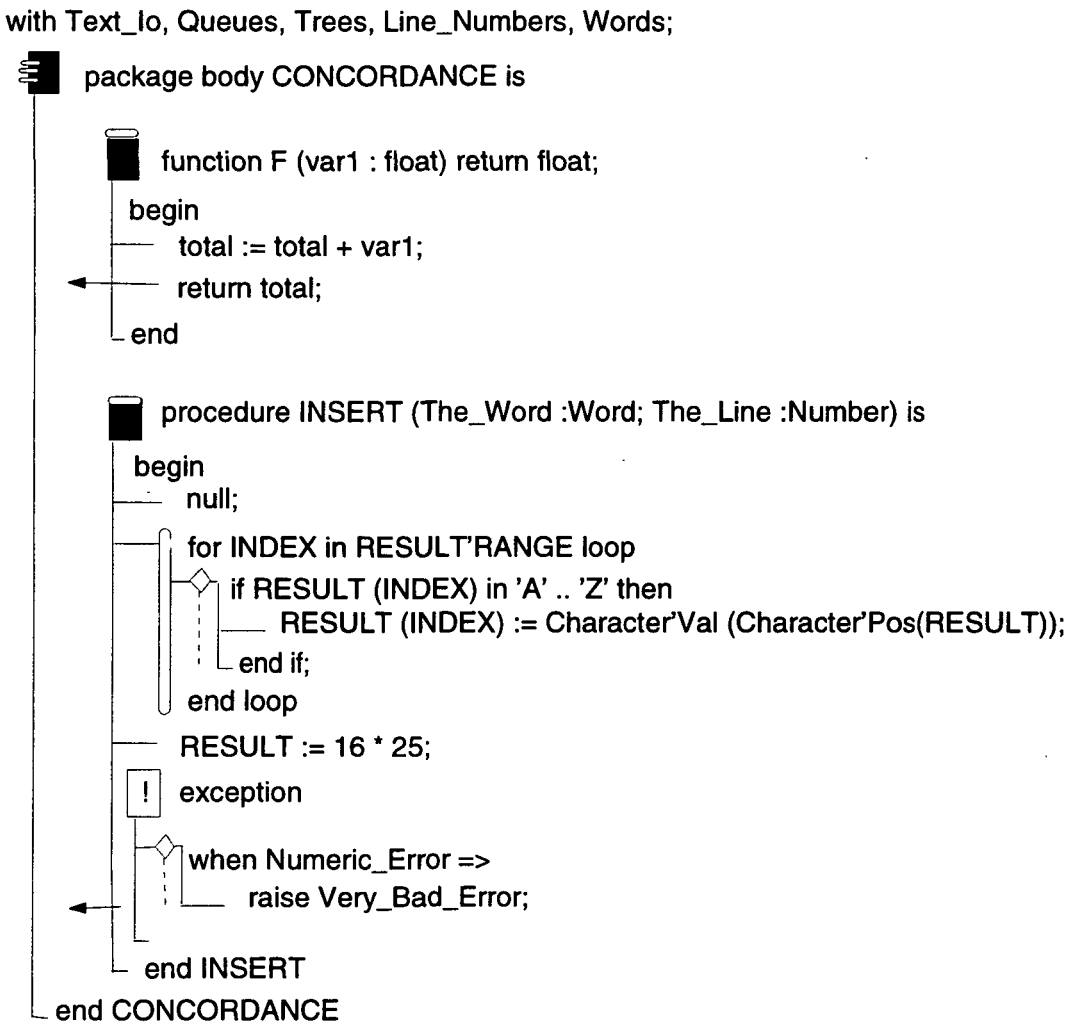

Figure 37. Example of New CSD Notation [SAD95].

6.0 Conclusions

The GRASP/Ada project has provided a strong foundation for the automatic generation of graphical representations from existing Ada software. The current prototype provides the capability for the user to generate the Control Structure Diagram (CSD) from syntactically correct Ada PDL or source code in a reverse engineering mode with a level of flexibility suitable for practical application. The prototype is being used in two software engineering courses at Auburn University on student projects in conjunction with other CASE tools. The feedback provided by the students has been very useful, especially with respect to the user interface. The prototype has been prepared for limited distribution (GRASP/Ada Version 4).

An important issue for all software tools in general, and graphical representations in particular, is evaluation. An evaluation based on preference was conducted to provide information on user perceptions of the CSD. An experiment was designed and data was collected from software engineering students. Statistical analysis indicated highly significant differences among five graphical notations when compared with respect to eleven performance characteristics. There was a clear preference for the CSD for seven of the eleven performance characteristics. Experience indicates that empirical evaluation of the comprehensibility (rather than preference) of graphical notations such as data flow diagrams, object diagrams, structure charts, and flowgraphs is difficult. However, such an evaluation for the CSD and GRASP/Ada tool would provide further insight into the role that graphical notations play in the comprehension of software and, as a result, their potential impact on the overall cost of software.

The primary impact of reverse engineering graphical representations will be improved comprehension of software in the form of visual verification and validation (V&V). To move the results of this research in the direction of visualizations to facilitate the processes of V&V, numerous additional capabilities must be explored and developed. A set of graphical representations that directly support V&V of software at the architectural and system levels of abstraction must be formulated. For example, the Object Diagram generator (ODgen) prototype described earlier is one the components of the GRASP/Ada project which would address architectural and system levels of abstraction. This task must 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 must be investigated and classified. Prototype software tools which generate visualizations at various levels of abstraction from source code and PDL, as well as other intermediate representations, must be designed and implemented. Graphically-oriented editors must provide capabilities for dynamic reconstruction of the diagrams as changes are made to other diagrams at various levels. These graphical representations should provide immediate visual feedback to the user in an incremental fashion as individual structural and control constructs are completed. Future directions and specific tasks for the GRASP/Ada project have been described at the end of each of the sections above.

The current prototype of the CSD generator, while only one of set of required visualization tools, has clearly indicated the utility of the CSD. Future enhancements will only increase its effectiveness as a tool for improving the comprehensibility of software.

REFERENCES

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.

AOY89 M. Aoyama, et.al., "Design Specification in Japan: Tree-Structured Charts," *IEEE Software*, Mar. 1989, 31-37.

BAR84 J. G. P. Barnes, *Programming in Ada*, Second Edition, Addison-Wesley Publishing Co., Menlo Park, CA, 1984.

BOO94 Grady Booch and Doug Bryan, *Software Engineering with Ada, 3rd Ed.*, Benjamin Cummings Publishing Co. Inc., NY, NY, 1994 p. 46-49

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

CON80 W. J. Conover, Practical Nonparametric Statistics, Joh Wiley and Sons, New York, 1980.

CRO88 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, D. C., 1988, Vol. 2, pp. 446-454.

CRO90 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, 199-254.

CRO93 J. H. Cross, "Update of GRASP/Ada Reverse Engineering Tools For Ada," *Final Report*, G. C. Marshall Space Flight Center, NASA/MSFC, AL 35821 (Delivery Order No. 21, NAS8-39131), December 14, 1993, 41 pages + Appendices.

CRO95 J. H. Cross and D. Hendrix, "Using Generalized Markup and SGML for Reverse Engineering Gaphical Representations of Software," *Proceedings of 2nd Working Conference on Reverse Engineering* (Toronto, Ontario, July 14-16, 1995), 2-6.

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

HEN95 D. Hendrix and J. Cross, "Language Independent Generation of Graphical Representations of Source Code,", *Proceedings of ACM Computer Science Conference* February 28-March 2, Nashville, TN, p. 66-72

MAY92 Charles H. May ,II, "A Tool For The Generation And Manipulation of Control Structure Diagrams", RBD Library: Auburn University, 1992 p.54-61

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

OLS93 M. R. Olsem and C. Sittenauer, "Reengineeing Technology Report, Volume 1," Software Technology Support Center, Hill Air Force Base, UT 840556, August 1993.

RAN95 B. Randles and J. Cross, "A Language Independent Renderer For Control Structure Diagrams," Department of Computer Science and Engineering, Auburn University, AU-CSE TR-95-07, May 23, 1995.

SAD95 Mark L. Sadler and J. Cross, "A Motif Compliant Interface for Graphical Representation of Algorithms, Structures and Processes", AU-CSE TR-95-08, Department of Computer Science and Engineering, Auburn University, May 23, 1995.

SCA89 D. A. Scanlan, "Structured Flowcharts Outperform Pseudocode: An Experimental Comparison," IEEE Software, Sep. 1989, 28-36.

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.

SEL90 Self, John, "Aflex - An Ada Lexical Analyzer Generator, Version 1.o", Arcadia Environment Research Project, Department of Information and Computer Sciences, University of California, Irvine, 1990.

SHN77 B. Shneiderman, et. al., "Experimental Investigations of the Utility of Detailed Flowcharts in Programming," *Communications of the ACM*, No. 20 (1977), pp. 373-381.

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

SIT92 C. Sittenauer and M. R. Olsem, "Re-engineeing Tools Report," Software Technology Support Center, Hill Air Force Base, UT 840556, July 1992.

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

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

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

NASA
National Aeronautics & Space Administration

Report Documentation Page

1. REPORT NO.	2. GOVERNMENT ACCESSION NO.	3. RECIPIENTS CATALOG NO.

4. TITLE AND SUBTITLE

Software Engineering Capability For Ada
(GRASP/Ada Tool)

5. REASON DATE

6. PERFROMING ORGANIZATION CODE:
Auburn University
0010090000

7. AUTHORS
Dr. James H. Cross II
Principal Investigator

8. PERFORMING ORGANIZATION REPORT NO.
4

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Computer Science and Engineering
Auburn Univeristy

10. WORK UNIT NO.
Delivery Order No. 30

11. CONTRACT OR GRANT NO.
NAS8-39131

12. SPONSORING AGENCY NAME AND ADDRESS

NASA/MSFC

13. TYPE OF REPORT AND PERIOD COVERED
Final Report
June 21, 1995
Period Covered:
June 22, 1995 - June 21, 1995

14. SPONSORING AGENCY CODE

15. SUPPLEMENTAL NOTES

NONE

16. 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, as a result, improve reliability and reduce costs. The emphasis has been on the automatic generation of the CSD from Ada PDL or source code to support reverse engineering and maintenance. The CSD has the potential to replace traditional prettyprinted Ada source code. A new Motif compliant graphical user interface has been developed for the GRASP/Ada prototype.

17. KEY WORDS (SUGGESTED BY AUTHORS)	18. DISTRIBUTION STATEMENT
Ada, reverse engineering, software engineering	Unlimited

19. SECURITY CLASSIFICATION (OF THIS REPORT)	20. SECURITY CLASSIFICATION (OF THIS PAGE)	21. NO. PAGES	22. PRICE
None	None	50	N/A

NASA FORM 1626 OCT 86 ALT