THE DEVELOPMENT OF A PROGRAM ANALYSIS ENVIRONMENT FOR ADA

Technical Report CSE-90-01

James H. Cross, Richard A. Davis, Charles May
Kelly I. Morrison, Timothy Plunkett, Darren Tola

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

December 1989

GRASP/Ada

Graphical Representations of Algorithms, Structures, and Processes for Ada

The Development of a Program Analysis Environment for Ada

Reverse Engineering Tools For Ada

Task 2, Phase 2 Report

Contract Number NASA-NCC8-14

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

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

ACKNOWLEDGEMENTS

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

Faculty Investigator:

Dr. James H. Cross II, Principal Investigator

Graduate Research Assistants:

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

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

001, FMap, TMap are trademarks of Hamilton Technologies, Inc.

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

AdaGRAPH is a trademark of George W. Cherry.

IORL is a trademark of Teledyne-Brown Engineering.

PAMELA is a trademark of The Analytical Sciences Corporation.

Rational is a trademark of Rational, Inc.

UNIX is a trademark of AT&T.

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

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

Table of Contents

Appendices

A. GRASP/Ada Control Structure Diagram

1.0 INTRODUCTION

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

Reverse engineering normally includes the processing of source code to extract higher levels of abstraction for both data and processes. Our primary motivation for reverse engineering is increased support for software reusability and software maintenance, both 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. Alternatively, the user may specify the software in Ada or Ada/PDL and then automatically generate the graphical representations either dynamically as the code is entered or as a form of post-processing.

Figure 1 shows the project divided into three phases, each of which corresponds to one of the following broad categories of graphical representations: (1) algorithmic (PDL/Code), (2) architectural, and (3) system level diagrams. Each of these categories may contain overlapping entries that depict, for example, data structure, data flow, or other useful relationships. Phase 1 of GRASP/Ada has been completed and a new graphical notation, the Control Structure Diagram (CSD) for Ada and supporting software tool is now being prepared for evaluation [CRO89]. In Phase 2, the focus is on a subset of Architectural Diagrams that can be generated automatically from source code with the CSD included for completeness. These are described briefly in the order that they might be generated in a typical reverse engineering scenario.

1.1 Algorithmic Diagrams (PDL/Code)

As the complexity of software has increased, so has the utility of graphical representations for algorithms. The industry has progressed well beyond the simple constructs of sequence, selection and iteration promoted by the theory of structured programming in the 1970's. For example, Ada includes control constructs for concurrency (tasks and task rendezvous), exception handling, and loop exits, none of which fits well into the simple sequential control constructs of structured programming. Since the ANSI flowchart was introduced in the mid-50's, numerous notations have been proposed and utilized [MAR85, TRI89]. These notations typically include control constructs for sequence, selection, and iteration, and several include constructs for concurrency and exits; however, none explicitly contains all of the control constructs found in Ada.

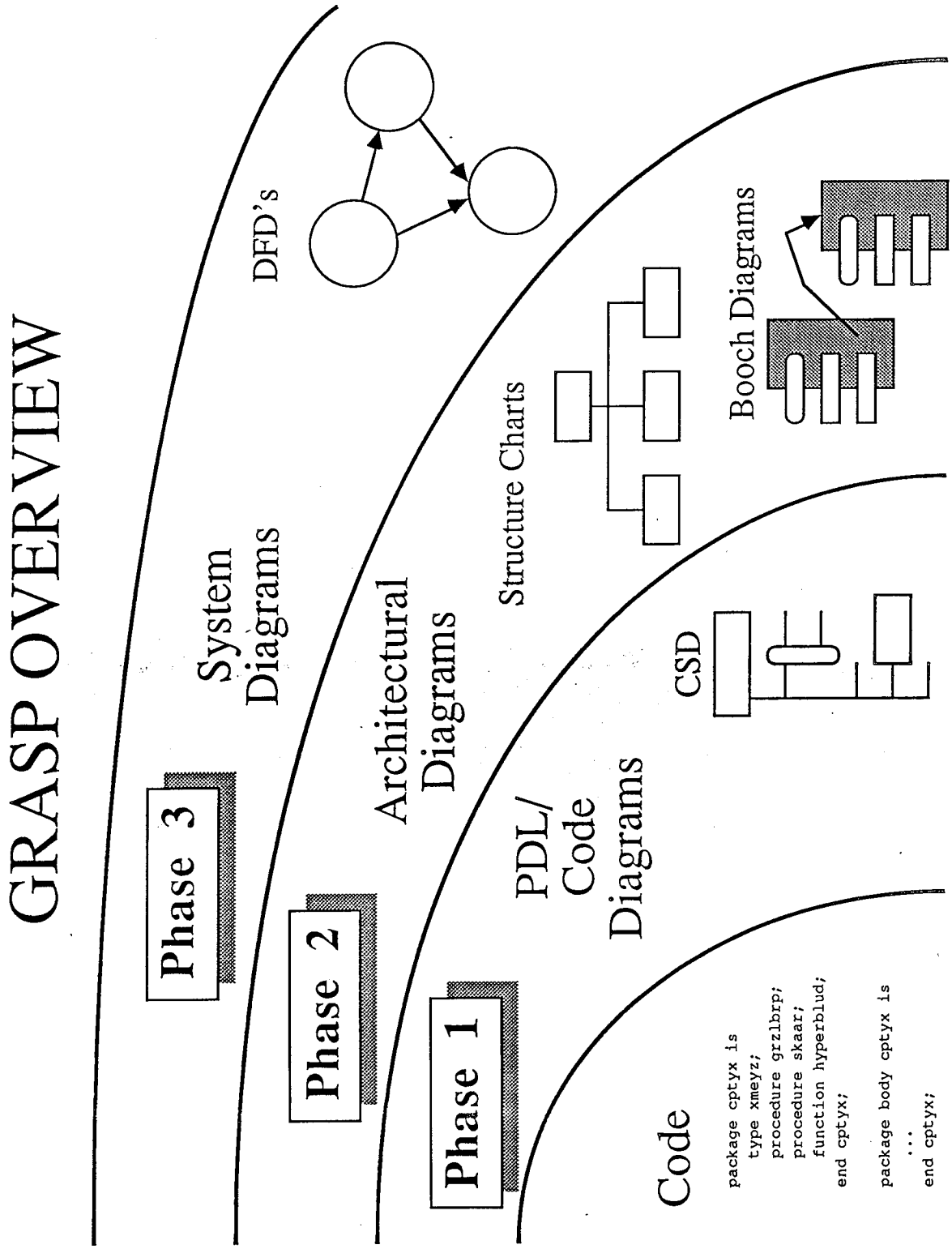


Figure 1. GRASP Overview

For the GRASP/Ada project, the control structure diagram (CSD) [CRO88] was selected as a basis for a graphical representation that maps directly to Ada control constructs. The CSD is a graphical notation intended to increase the comprehensibility of Ada PDL or source code by explicitly depicting control constructs and control flow. The traditional textual representation of PDL or source code has been extended with intuitive graphical constructs which are easily adaptable to editors and printers. The CSD has the attractive property that it can be overlaid directly on prettyprinted Ada code. In fact, a CSD generator may be perceived as a "graphical prettyprinter." The CSD graphical constructs for each of Ada's control constructs are in Appendix A.

1.2 Architectural Charts and Diagrams

The next level of diagrams in the reverse engineering process is a group commonly known as architectural diagrams. Structure charts, data structure diagrams, and entity-relationship diagrams are traditional examples of these. The object/package diagram is a relatively recent addition at this level. Structure charts, object/package diagrams, and a collapsed version of the control structure diagram have been targeted for prototyping in Phase 2. Structure charts and object/package are each discussed briefly below in the context of automatically generating the diagram from source code or PDL.

Structure charts are one of the oldest and potentially most useful diagramming notations available. We use the term here in the generic sense to refer to those charts and diagrams that depict the overall hierarchical organization of a software system without concern for the algorithmic details. IBM's HIPO, and Yourdon's structure chart are common examples in this category. The structure chart is simply an

invocation graph of functions and procedures, less redundant calls. Some versions indicate data items along the control lines between procedures to show data flow as well as detailed control flow information such as selection and iteration.

The structure chart offers the user a high-level solution-oriented view of the software. Although algorithmic details are suppressed, the user can still get a sense of what is going on from the perspective of solving the problem as well as a feel for the layers of procedures and functions involved. Unfortunately, structure charts generated during initial development of a system are rarely kept current without the aid of a CASE tool which links the diagram and corresponding code. A major role of reverse engineering in a CASE environment is to ensure the availability of an accurate set of structure charts as well as graphical representations for other software views.

Automatic generation of structure charts from source code is relatively straightforward. In the case of Ada, the abstract syntax tree built during the parse must be traversed, capturing procedure and function calls (a task rendezvous has the appearance of a procedure call). A call to a procedure or function results in the traversal of its abstract syntax tree. Redundant calls from a single procedure are normally captured but not displayed. Data items and their direction of flow are identified syntactically by their IN, OUT, or INOUT designation in the parameter list. Additional program analysis is required to determine references to non-local variables that are not formal parameters.

The *Object/package diagram* made popular by Booch [BOO83] is a recent architectural level diagram that is useful for object-oriented software. The object/package diagram shows all of the dependencies among packages and package

components. This is an important view of the software with respect to its construction or composition from parts. For example, an Ada package may be used for encapsulation of types and operations to form abstract data types. These packages can then be considered objects from an object-oriented development perspective. The object/package diagram is used to show the dependency relationships among the object components.

Object/package diagrams are generated from a syntactical analysis of the Ada source code. The basic dependencies are defined by the WITH clause. The actual package components that are utilized are determined by references to types, procedures and/or functions exported by the package. These objects or packages can be further graphically encoded by using icons, shading, and coloring.

Preliminary analysis has revealed that structure charts and object/package diagrams are complementary in nature and, furthermore, that in isolation each affords a somewhat incomplete view of the software. The hierarchical or layered structure chart is easily related to the software solution of the problem. That is, a reader can discern "what" is being done with respect to solving the problem or, from a reverse engineering perspective, which problem is being solved. The object/package diagram, on the other hand, offers a view of component packaging (e.g., how data and operations are packaged into objects). While Booch points out that the object/package diagram is much closer to the data flow diagram of the general specification of the problem (e.g., external entities and data stores become objects), it has been our experience that the dependencies shown in the object/package diagram provide little or no information regarding the interaction of the objects and operations. The structure chart and

ultimately the control structure diagram do supply the additional information necessary for complete comprehension of the solution.

The remainder of this report is oganized as follows. Section 2 discusses architectural diagrams that are currently in use and provides a summary of several general trends in visualizations in computing. Section 3 provides a discussion of the problem Phase 2 of the GRASP/Ada project is addressing. Section 4 provides the preliminary requirements for the prototype that will be developed to support the automatic generation graphical representations from Ada source code. These requirements include functional, interface, hardware, and system software. Section 5 provides the schedule for Phase 2.

2.0 ARCHITECTURAL DIAGRAMS IN CURRENT USE

In this section, the term "architectural diagram" and some related terms are defined. This is followed by a brief survey of recent as will as traditional architectural diagrams which have been used for Ada. The specific needs for architectural diagrams for Ada software are examined. This section concludes with a brief discussion of trends in visualization for computing in general.

2.1 Definitions

An *architectural diagram* (AD) may be defined as follows: a graphical representation of the logical components of a software system, the interfaces between such components, and the hierarchical relationship among the components.

Logical components of a software system are those structures which group statements and components into cohesive units. In Ada, these structures include the package, procedure, function, and task. Most well-designed logical components are functionally cohesive, each providing a single and specific service.

The *interfaces* between the logical components of a software system show the parameters which are passed between the components. The simplest case there may be no parameters passed between a given set of components. Often, however, these parameters consist of items of complex types and, in the case of Ada, may even include tasks.

The *hierarchical relationship* among the logical components of a software system is

shown as a utilization hierarchy. A connection between any two components represents a resource usage of one component by the other.

Two other terms that are of use when referring to hierarchical diagrams are *visibility* and *connectivity*. Each is a term referring to the scope of a given software component. Visibility refers to the set of components that may be invoked by a given component, regardless of whether the code actually specifies an invocation of such components. Connectivity refers to the set of components that are explicitly invoked by a given software component in the source program.

2.2 Graphical Representations for Architecture

In this section, several architectural diagrams currently in use are briefly discussed. This is followed by an examination of the Ada programming language with respect to its components that must be considered when developing architectural diagrams. Finally, several considerations pertaining to the Ada programming language are presented that must have a bearing on the development of any practical architectural diagram for Ada.

2.2.1 General Trends

Perhaps the best-known architectural diagram is the traditional *structure chart* made popular by Yourdon and Constantine (see Figure 2). This diagram represents the architecture of a system using a set of boxes representing functions and procedures connected by lines indicating invocation. Small arrows are arranged along the lines of invocation to depict the flow of data between the modules. Typically, data flows are

STRUCTURE CHART

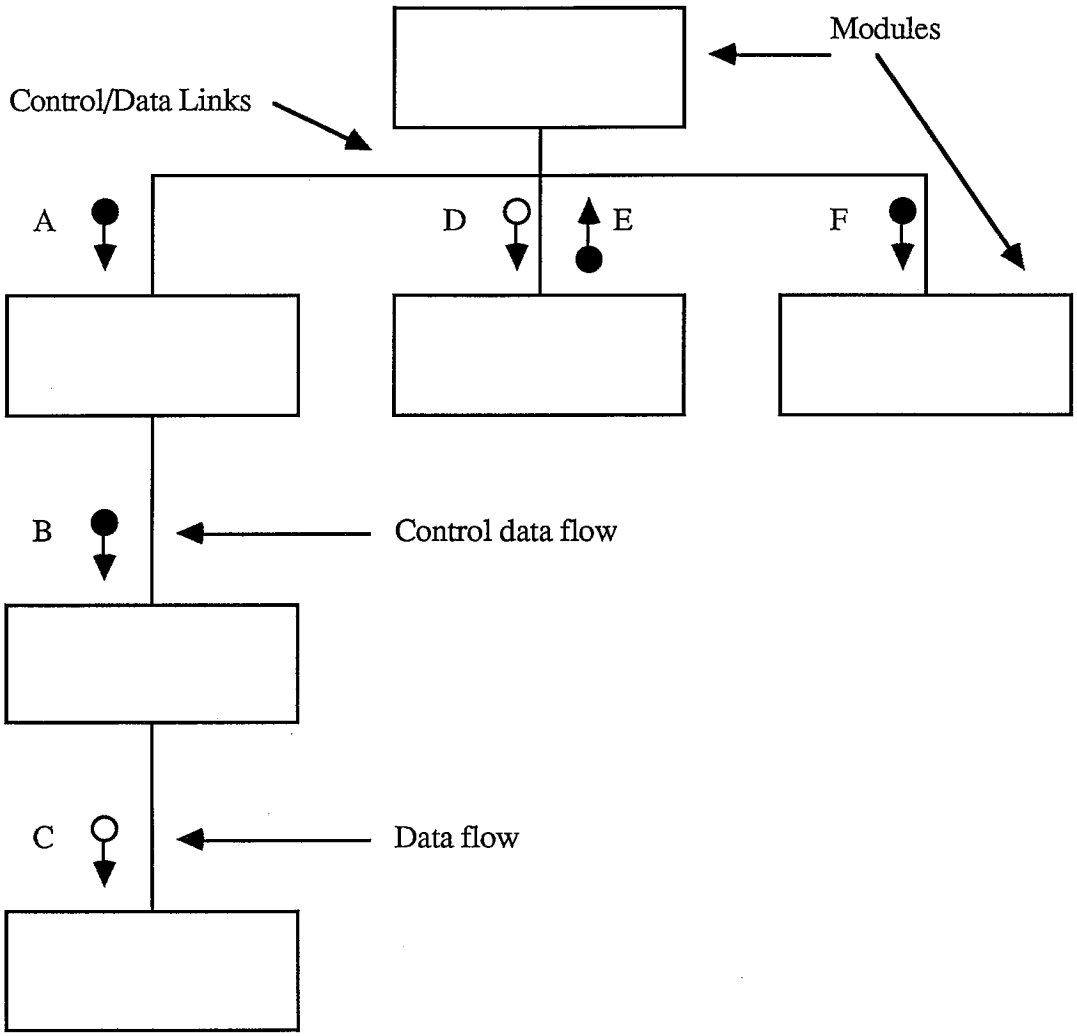


Figure 2. Structure Chart

of two types: data items flows, which may be either simple or complex data types, and control data items, which are used to determine the execution of the invoked procedure. Although the traditional structure chart is useful for depicting the architecture of systems written in simple languages such as Pascal, it lacks in its ability to represent advanced features found in Ada such as tasking and generic instantiation of procedures from templates.

CAEDE (Carleton Embedded System Design Environment) is a software CAD system developed at Carleton University by Buhr [BUH89] that uses modified *Buhr diagrams* to represent the architecture of an Ada program (see Figure 3). The structural CAEDE diagrams are block-oriented and include distinct symbols for tasks, packages, and procedures. Although the CAEDE system does include graphical representations for all of the Ada architectural components, it does not represent generics well. In addition, the nesting required to produce an accurate CAEDE diagram for a typical Ada program can become cumbersome. At this time, there is no existing tool for generating CAEDE diagrams from existing code.

OOSD (Object-Oriented Structured Design), developed by Wasserman [WAS89], is a method for designing the architecture of systems. The heart of OOSD is the *OOSD design chart*, a modified structure chart, that describes a set of architectural components, their invocation hierarchy, and the parameters passed among them (see Figure 4). At a lower level, information clusters provide an object-oriented description of the components depicted on the design chart. Because OOSD is designed to be language-independent, it does not correspond exactly to Ada, and therefore does not directly support all of Ada features, especially the tasking features. On the other hand, OOSD

CAEDE

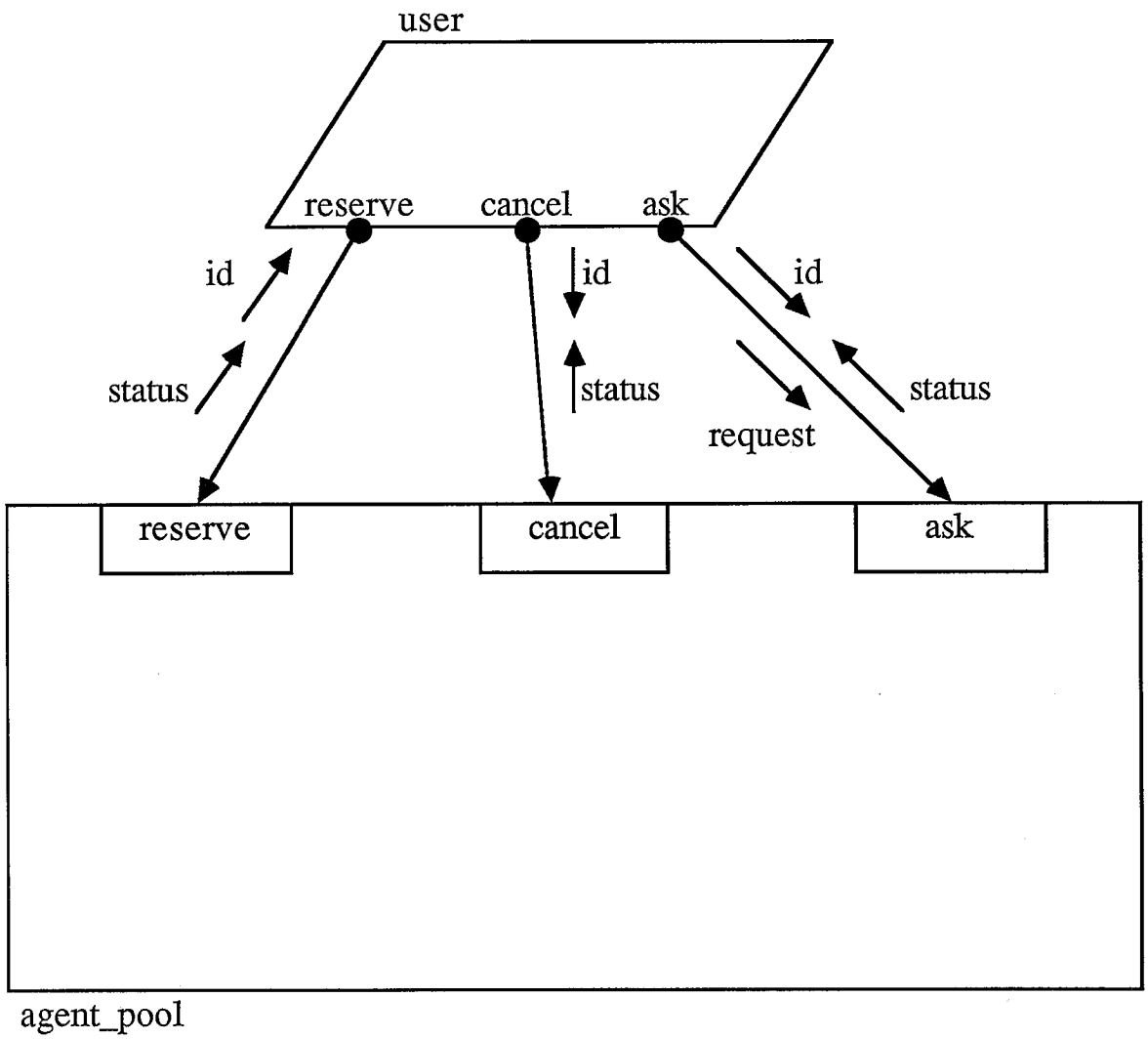
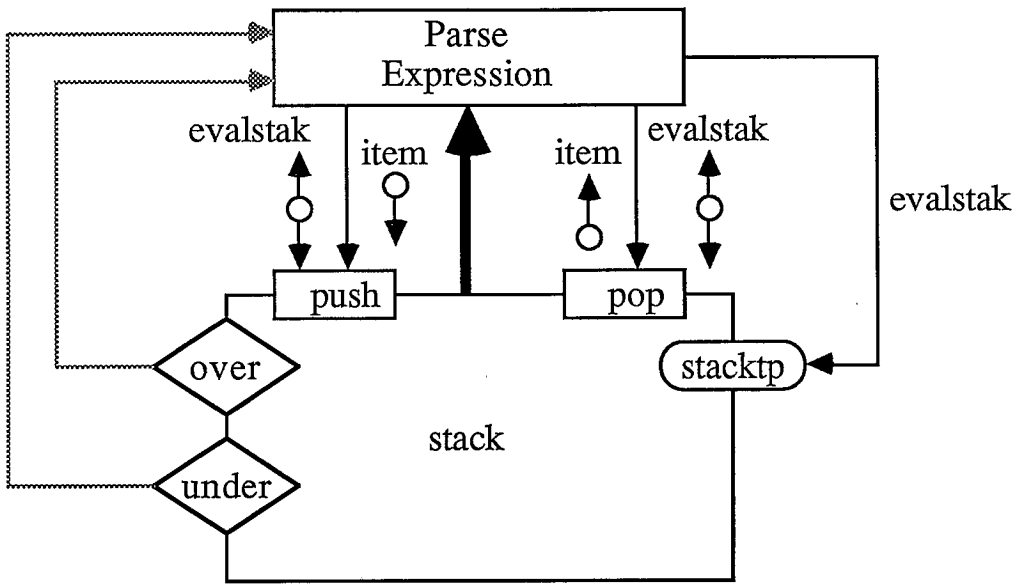


Figure 3. Buhr Chart

OOSD



Example of an Information Cluster in OOSD.

Figure 4. Example of an Information Cluster in OOSD

does allow the designer to utilize some features that Ada does not provide. At this time, there is no existing tool for generating OOSD diagrams from existing code.

Hamilton Technologies, Inc., has developed an integrated hierarchical, functional and object-oriented modeling approach collectively called 001 technology. The 001 technology is based, in part, on USE.IT developed by Higher Order Software (HOS) [HAM79]. In 001, a system is defined in terms of a single control map which integrates both *function control maps (FMaps)* and *type control maps (TMaps)*, where an FMap defines a hierarchy of functions and a TMap defines a hierarchy of abstract types. The underlying specification language for these maps is 001 AXES, which is based on a set of control axioms derived from empirical data gathered during the development and operation of the existence of a universal set of objects. The leaves of the maps represent primitives implemented in a language for a particular native computer environment. When a system specified in 001 AXES is processed by the "Resource Allocation Tool," the result is a complete system in the source language of the primitives.

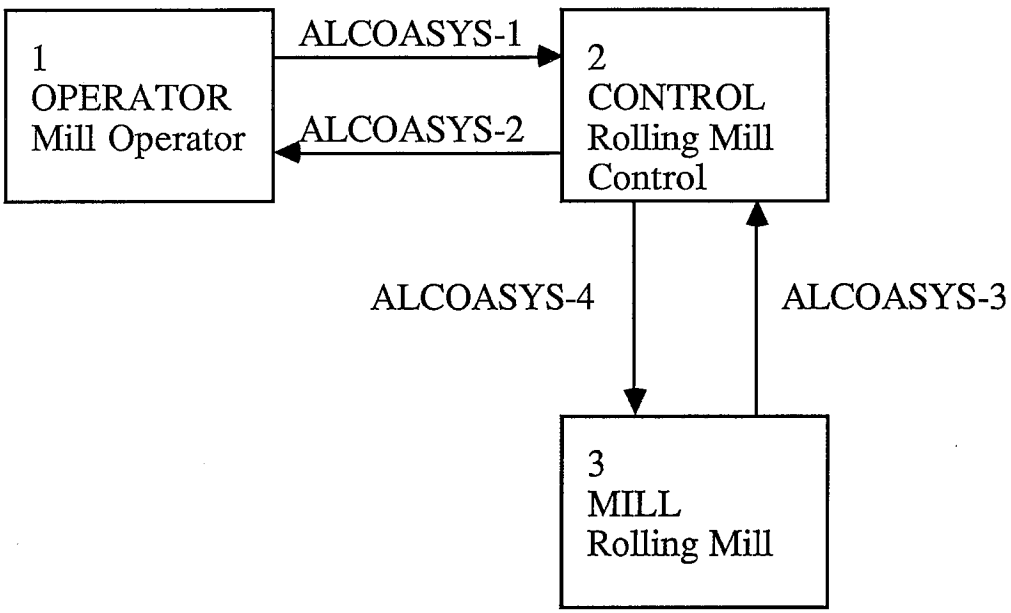
PAMELA (Process Abstraction Method for Embedded Large Applications) is a methodology developed by Cherry [CHE88] and supported by the *AdaGRAPH* environment on the IBM PC. A specification is written in PAMELA by first describing a system as a collection of flow diagrams. Next, the analyst is prompted to answer certain questions about each of the processes in the flow diagrams, resulting in corresponding annotations to the diagrams. Finally, the analyst completes the skeleton code generated from the flow diagrams and fills it in to form executable Ada programs. It is interesting to note that the "automatic code generation" provided by PAMELA falls

mainly into the area of providing correctly specified modules and communications between these modules. Generating procedural code is left to the analyst, although the AdaGRAPH environment does provide facilities for simplifying this.

IORL (Input/Output Requirements Language) is a high-level requirements language developed for the design of real-time embedded systems with the TAGS (Technology for the Automated Generation of Systems) methodology [SIE85]. TAGS embodies the hierarchical top-down development of a system, and relies upon graphical representations to present control flow within a process and data flow among different processes executing simultaneously (see Figure 5). A system may be viewed at any time from a number of levels: from a very high level showing an overview of the entire system, from a very low level showing the IORL primitives that make up a process, or from any level in between. The latest release of IORL utilizes an icon-oriented interface for the easy creation of IORL diagrams, and some errors from earlier versions have been corrected. Currently, Teledyne Brown Engineering is working on a "Simulation Compiler" which will significantly enhance the TAGS development environment.

Booch diagrams [BOO83] provide a graphical representation of the architectural components of Ada along with some dependency information (see Figure 6). Experience indicates that the graphical representation of large systems using Booch diagrams often leads to a network decomposition rather than a strict hierarchical control organization. In addition, at the present time, only primitive tools exist for the extraction of Booch diagrams from Ada source code.

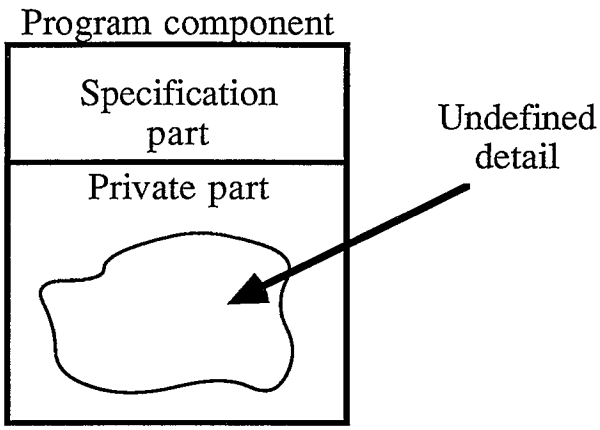
IORL



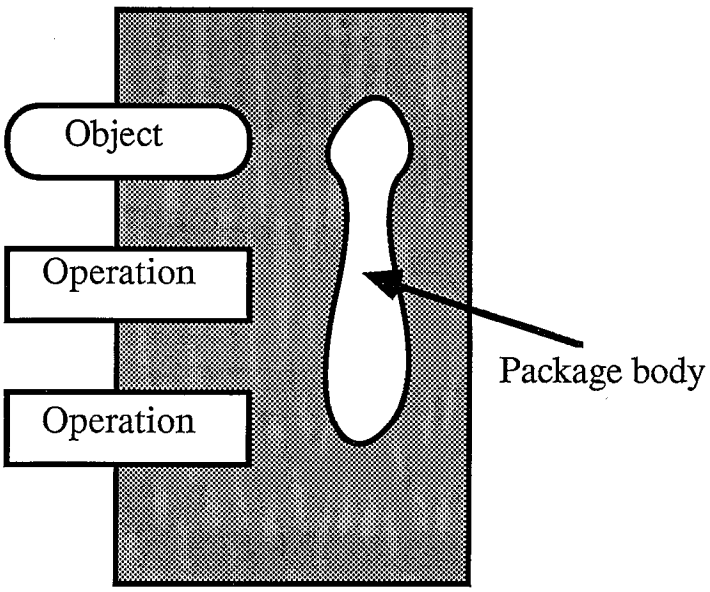
Example of a Schematic Block Diagram (SBD) in IORL.

Figure 5. Example of a Schematic Block Diagram (SBD) in IORL

BOOCH



Booch diagrams show the architecture of a system using a set of program components, each having a specification and a private part.



Booch diagrams support the object-oriented paradigm by allowing package to specify a number of OBJECTs and a number of OPERATIONs on those objects.

Figure 6. Examples of Booch Diagram Components

2.2.2 Architectural Components of Ada

Most high level programming languages have very few architectural components. For example, Pascal has only procedures, functions, and a single main program. However, Ada is much more complex, with constructs that are difficult to represent using traditional architectural diagrams. In this section, the architectural components of the Ada programming language are examined.

The architectural components of Ada may be subdivided into two categories: logical and physical. The logical components are those structures defined within the language that serve to group sets of logically related statements or components. The physical components are those components which serve more to assist the Ada compiler rather than the Ada programmer.

There are five logical components in the Ada programming language: packages, procedures, functions, tasks, and operators. Packages are structures which serve to group the other logical components into cohesive modules. Procedures, functions, and tasks are much alike in that they are small threads of executable code that generally provide a single specific service. Operators may be considered a special case of function that may take one or two arguments. Although operators are predefined in most programming languages, Ada allows them to be overloaded.

There are three physical components in the Ada programming language: library units, secondary units, and subunits. A library unit is a specification that defines a set of logical components and data declarations. A secondary unit is the body of code that implements each of the logical components defined in the corresponding library unit.

Finally, a subunit is a section of code that implements a logical component defined in a library unit but may be compiled separately.

In addition, the logical components may have properties associated with them. For example, a logical component may be a standard component, with all its data types explicitly defined. Or, it may be a generic component that may be instantiated for a given data type. Another property that logical components in Ada exhibit is that of visibility. A logical component may be visible, and accessible to any other component that refers to it, or it may be hidden, only accessible by other components in its package.

2.2.3 Architectural Diagrams for Ada

In this section, some of the special issues which must be addressed in the development of a set of architectural diagrams for Ada are discussed.

Representation of generics. The generic construct in the Ada language allows the definition of "templates" for software functions which describe a function's logic without making any commitments to data types. The generics may be easily instantiated to operate on any set of data types. In an architectural diagram, these functions would appear in many places as distinct functions, although they differ only in the data types on which they operate. Some method for capturing this similarity in the architectural diagram should be developed.

Representation of overloading. Ada allows a number of simple operators to be "overloaded." This is similar in respect to the notion of generic functions in that the only difference between functions is the set of data types on which they operate.

Representation of tasking. Architectural diagrams generally represent the invocation hierarchy among a set of procedures for a single thread of program execution. Ada introduces the concept of tasking, or simultaneous execution, whose graphical depiction has not been well investigated.

Representation of "static" vs. "dynamic" scope. In most high level languages, all of the components of a software system "exist" for the duration of the system's execution; this may be referred to as "static" scope. In Ada, however, components may exist only for portions of the system's lifetime, due to tasking and to the ability to embed components inside others; this may be referred to as "dynamic" scope. Some method for representing these on an architectural diagram must be developed.

Representation of scope of private functions and procedures. Ada allows packages to have private functions and procedures which are only callable by other functions and procedures in that package. Traditional architectural diagrams have no provision for showing this.

Representation of recursion. Ada, like most other high level procedural languages, supports both direct and indirect recursion. Although simple methods for depicting this on a structure chart have been developed in the past, a representation more suitable for Ada must be devised.

Representation of functions passed as parameters. Ada allows functions to be passed as parameters in the instantiation of generics. Traditional architectural diagrams have no means for showing components passed as parameters in an invocation.

Representation of embedded packages and tasks. Ada allows packages, procedures and tasks to be declared anywhere in a program that variables and data types may be

declared. As a result, procedures with a dynamic lifetime may be declared that are callable by the component in which they are embedded but only for the scope of their declaration. There is no convention for showing this on an architectural diagram.

Representation of physical components of software. Traditionally, architectural diagrams show only the logical architecture of software and ignore the physical architecture.

Representation of architecture using layers. As the needs of software systems become more and more complex, the size of such systems has grown dramatically, often beyond the point where a single person could readily understand the inner workings of the systems. To render these systems more presentable to the software engineer, it is necessary to develop some method for layering the architecture of the system so that it may be presented in successive degrees of abstraction.

Representation of all Ada-specific components. For an architectural diagram for Ada to be practical, it must represent all of the architectural components of the Ada programming language.

Representation of visibility and connectivity. To assist the maintenance programmer, visibility and connectivity must be represented on the architectural diagram.

2.3 Visual Computing Trends

In this section, current trends in visualization in computing are presented. While much of the discussion focuses on visual programming, the ideas are relevant to all phases or levels of graphical representations. Although relatively new to the automation environment, visual programing techniques provide an effective as well as

versatile means to perform a wide spectrum of analysis and design functions. It has been observed that the use of graphical representations to model, design, and evaluate complex programing processes greatly enhances the ability of the user to understand the process in question [SHU88, AMB89]. The concept of allowing a user to visualize information in an other than textual form is being utilized in numerous areas. The graphical representation of complicated or enormous quantities of information is currently being employed in the fields of data design, program design, program execution analysis, software engineering, visual programming languages as well as other applications.

The use of visual representations has evolved far beyond the simple mapping of textual data to that of a graphical representation. In fact, new developments in the field are leading to systems and environments that are graphically oriented by nature. Visual user interfaces modelled after Kay's paradigm of using overlapping windows, such as those found in Smalltalk, provide multiple views of a common internal database. Whenever any portion of the data is changed, all relevent views are updated to reflect that change. Graphically oriented language environments include Pecan, Cedar, and Software through Pictures [AMB89, FOR88].

Visual editing provides the user with the capability to modify existing programs or produce new ones through the use of templates that correctly reflect the language's syntax. Such current systems include the Cornell Program Synthesizer editor and the Aloe editor used in Gandalf. Several other graphical editors enforce logical consistency through the addition of rules regarding the structure of a program. Higher Order Software's Use.It and PegaSys are examples of systems that use this technique

[AMB89].

The utilization of visual technology to edit programs written in traditional languages has been joined by a new philosophy of programming paradigms under a category referred to as "naturally visual languages" [AMB89]. Under these language environments the basic language constructs are visual rather than textual. A variety of approaches are used in languages. The use of dataflow, constraints, form-based and program-by-demonstration paradigms serve as the bases for environment supported languages such as ThingLab, ThinkPad, and Rehearsal World [AMB89].

Somewhere between the visual programming language and the textual languages one finds Conic. This programming environment uses a combination of text and graphics to define "configurations" that collectively make up a program [KRA89]. It focuses on the functionality of processes, their control characteristics, and communication interaction.

Although much emphasis has been placed on the role visual programming plays in user interfaces, editors, and programming languages, its potential far exceeds this scope. As stated above, the use of graphical representations has showed itself to be extremely useful in any area that inherently has large quantities of complex information. Two such applications utilizing visual techniques as a means to better understand actual events include performance debugging, specifically in regard to multiprocessor systems, and concurrent computations [LEH89, ROM89].

Carnegie Mellon University has demonstrated the usefulness of visualization through its special software development environment known as the Parallel Programming and Instrumentation Environment or PIE. This system is designed to develop performance-

efficient parallel and sequential computations. This is accomplished by mapping parallel applications onto specific architectures, gathering data as they execute and producing graphical representations that reflect selected characteristics of the actual execution [LEH89].

The visualization of concurrent computations employs visual abstraction by "mapping from computational states to the states of graphical objects" [ROM89]. This approach has been used to insure the correctness of a process, consistency in execution and progress in the computation of a solution.

Visualization of programming has been demonstrated to be an effective means of representing complex processes, data structures, and computational events. The primary element that makes each of the systems examined above viable is its well defined utilization of graphical representations within the context of its application.

3.0 STATEMENT OF THE PROBLEM

In this section, the overall direction for the GRASP/Ada Phase 2 prototype is presented. First, the goals and objectives for the prototype are briefly discussed. Finally, the tentative architectural diagrams for Ada are introduced.

3.1 Overview

In Phase 1 of the GRASP/Ada project, the focus was on the algorithmic representation of Ada programs and the CSD (Control Structure Diagram) was developed to graphically depict Ada control constructs. In Phase 2, the focus shifts to the structural (or architectural) view of Ada, and new diagrams must be developed to represent this view. Although one diagram (the CSD) was sufficient to represent the algorithmic view of Ada, multiple diagrams are needed to adequately represent the structural view.

3.2 Introduction of Taxonomy

To assist in the development of a layered approach to the graphical depiction of Ada, a tentative taxonomy of graphical representations has been developed. This taxonomy defines five distinct views of Ada software: the code view, the algorithmic view, the connectivity view, the visibility view, and the logically related view (see Figure 7). Each of these views will be discussed in more detail below.

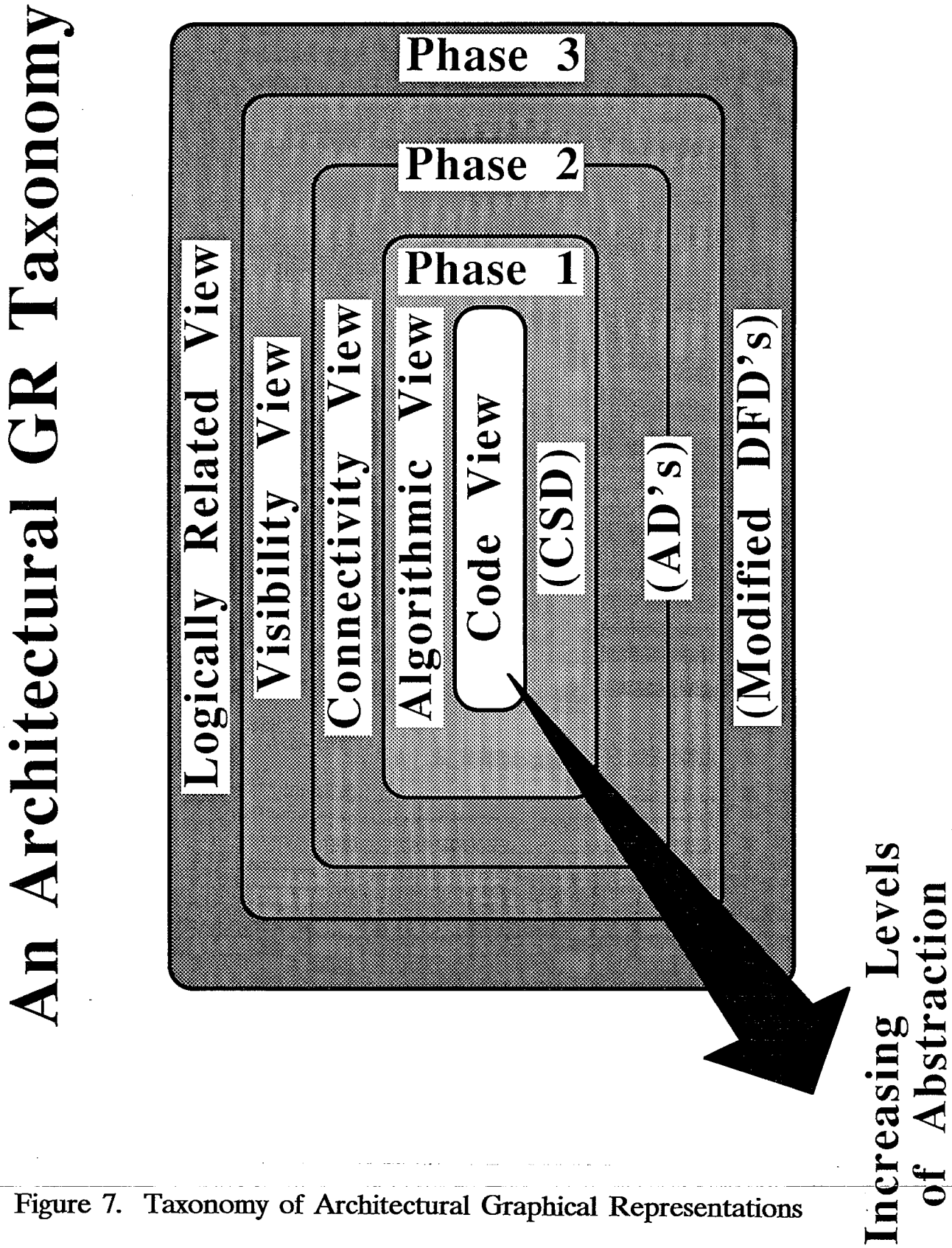


Figure 7. Taxonomy of Architectural Graphical Representations

The code view is the base view of Ada software, consisting of the source code itself. This code may be optionally augmented with some additional information such as line numbers, nesting data, and a cross-reference, but its low-level nature renders it difficult for the software engineer to quickly comprehend the code.

The algorithmic view of Ada is intended to enhance the code view by graphically representing control structures. The CSD developed in Phase 1 of the GRASP/Ada project serves this purpose by augmenting Ada code with small iconic representations of the various control structures. These graphics are embedded in the code in the area normally used for "white space", and thus coexist with the code without requiring significant spatial reorganization.

Phase 2 of the GRASP/Ada project is focused on the connectivity view and the visibility view of Ada. The connectivity view shows the architectural components of an Ada system with their invocation hierarchy and associated parameters. This view is most like the traditional structure chart, yet has been enhanced and represented by two distinct graphical representations in the GRASP/Ada system. The first is the Level 1 architectural diagram which consists of a "collapsed" CSD that shows the architectural components and the control logic that leads to the statements that show each of the components being invoked. The second graphical representation is the Level 2 architectural diagram that utilizes a traditional structure chart with appropriate modifications and extensions for Ada.

The visibility view of Ada represents a set of architectural components and their associated scopes, both static and dynamic. Whereas the connectivity view shows which

component are explicitly called (or invoked) by other components, the visibility view shows which components may be invoked by other components. This view also denotes the dependency relations among Ada software components, and will be graphically represented using modified Booch diagrams.

The logically related view of Ada will be the focus of the proposed Phase 3 of the GRASP/Ada project. This view shows the data flow among logically related groups of software architectural components, and may be considered an abstraction of the visibility view. Although the proposed GRASP/Ada graphical representations for this view have not yet been developed, it is believed that they will consist of a set of modified data flow diagrams and tasking diagrams.

3.3 Derivation of Base Set of Architectural Diagrams

In this section, the tentative base set of architectural diagrams for Phase 2 of the GRASP/Ada project are described. There are three proposed graphical representations for this phase: the Level 1 architectural diagram, the Level 2 architectural diagram, and the Level 3 architectural diagram. Each of these is discussed in more detail below.

3.3.1 Level 1 Architectural Diagram

The Level 1 architectural diagram bears a close resemblance to the CSD used for representing algorithmic details. This diagram is produced by taking a CSD for a given Ada program and removing all data declarations, data type and structures, and all "unimportant" code (see Figure 8). "Unimportant" code includes all simple statements that do not directly lead to the invocation of another software component. The Level

LEVEL 1 STACK EXAMPLE

```
package STACKS   is

   procedure CREATE_STACK (THE_STACK: out STACK);

   procedure PUSH (THE_STACK:  in out STACK; THE_ITEM:
      in NATURAL);

   function POP (THE_STACK:  in STACK) return NATURAL ;

   procedure PRINT_STACK (THE_STACK:  in STACK);

end STACKS;

procedure SAMPLE_PACKAGE  is

begin

   CREATE_STACK(S);

   for I in 1..100 loop
      if THE_ELEMENT(I) = UNCOMPUTED then

         PUSH(S,THE_ELEMENT(I));

      end if;
   end loop;

   PRINT_STACK(S);

end SAMPLE_PACKAGE;
```

Figure 8. Collapsed Control Structure Diagram

1 architectural diagram may be thought of as a CSD that has been "collapsed" to show the architectural components which it includes, the invocations of such components, and the control logic leading to those invocations.

The Level 1 architectural diagram may be obtained using the CSD generator developed in Phase 1 of the GRASP/Ada project. Some tentative research into the feasibility of this approach leads to the possibility of this diagram being produced in O(N) time, where N is the number of statements in the source code.

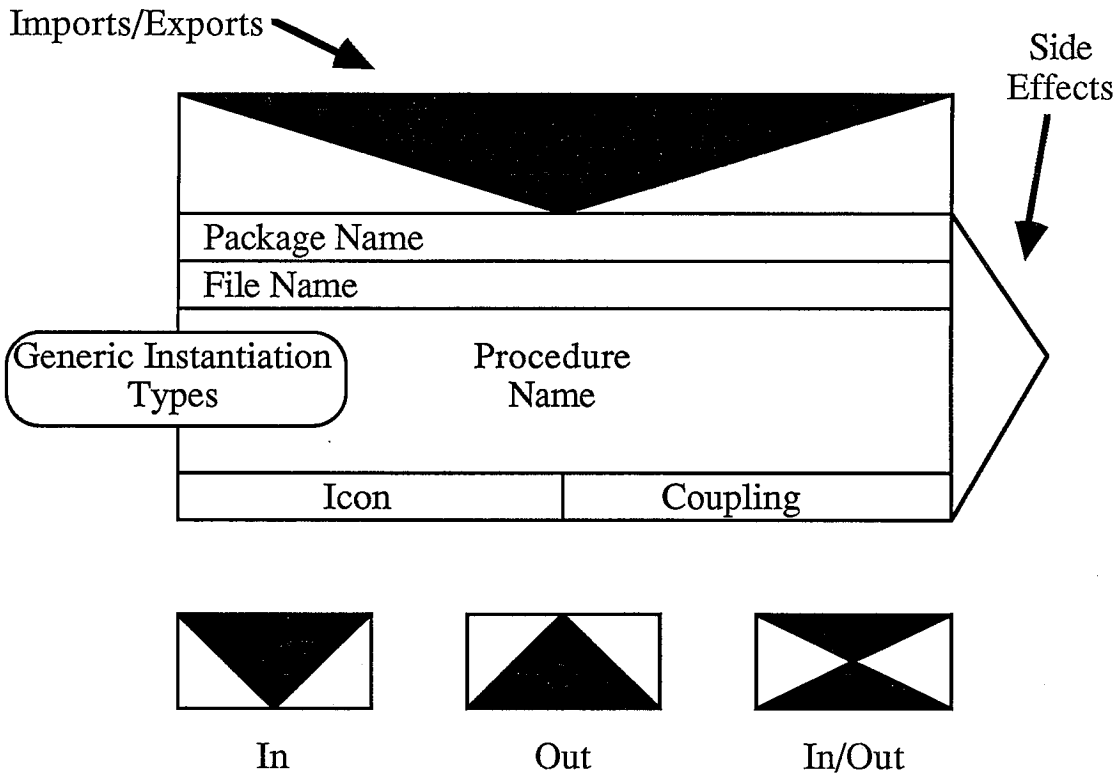
3.3.2 Level 2 Architectural Diagram

The Level 2 architectural diagram may be thought of as an extensively modified structure chart that has been customized for Ada. The diagram consists of two parts: a set of modules, which define Ada architectural components such as procedures and functions, and a set of control/data links, which define the invocation hierarchy among the components and the data passed among them (see Figure 9).

Modules are depicted using a compartmented box, with each Ada procedure, function, task and overloaded operator mapping into distinct boxes. The upper compartment is used to indicate the overall flow of imports and exports into and out of the module. An IN indicator shows that all of the parameters passed to the module are of type IN. An OUT indicator shows that all of the parameters passed to the module are of type OUT. An IN/OUT indicator shows that the parameters passed to the module may be of type IN, OUT, or IN/OUT. Finally, a null indicator shows that the module has no parameters. Note that the graphical nature of the indicator allows the software engineer to quickly determine the overall flow of data among a

Level 2 Architectural Diagrams (Tentative)

MODULES



CONTROL/DATA LINKS

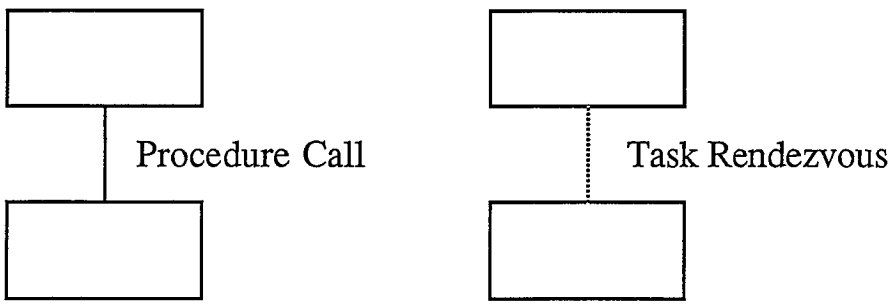


Figure 9. Tentative Graphical Construct for Level 2

program's architecture.

The second and third compartments in the modules indicate the logical and physical names associated with the module. The logical name shows the name of the logical structure (usually a package) in which the module is directly embedded, if such a structure exists. The physical name shows the name of the file containing the specification for the module. With these two pieces of information, the software engineer can easily determine where a particular module fits into the logical architecture of a system as well as find the code associated with the module.

The fourth compartment in the modules indicates the name of the software architectural component. This name may correspond to either a procedure, a function, a task, or an overloaded operator.

The data in the fifth compartment in the module will not be automatically generated, but will allow the software engineer to customize a reverse engineered system for ready visual reference. The engineer may define an icon for each package in a system that can be included in the architectural diagrams. For example, a stack icon might be created to visually set apart those modules which are part of a stack package.

The sixth compartment in the modules indicates the type of coupling that the module shares with the component that invoked it. Although determining formal coupling as defined by Myers is a difficult problem, there have been attempts at determining coupling using program metrics. It is this approach that the GRASP/Ada project will take in determining the degree of coupling among software architectural components.

The inclusion of an arrowhead on the right side of a module indicates that the module exhibits side effects. Typically, this pinpoints the use of a data item or data structure that was not declared within the module or passed to it. Although well-designed systems refrain from using this approach whenever possible, it does frequently occur in practice and can lead to frustration when trying to understand a complex system.

The last compartment in the modules is used to indicate a generic instantiation. If the module was instantiated from a generic template, the data types used to instantiate the module are listed along the left edge. In this way, identical modules that operate on distinct data types may be easily distinguished in the architectural diagram.

Control/data links are shown using a solid line in most cases. However, when one of the two components in an invocation is a task, a dashed line is used to indicate a rendezvous is in progress. This suggests that a task rendezvous is similar to a procedure call, which is a reasonable analogy. A procedure call might be thought of as a task rendezvous where the task that initiated the rendezvous suspends execution until the task with which it rendezvoused ceases its execution. An example of a Level 2 architectural diagram for a stack package is shown in Figure 10.

3.3.3 Level 3 Architectural Diagram

The Level 3 architectural diagrams will show the visibility view of Ada rather than the connectivity view exhibited by the Level 1 and 2 diagrams. Although the diagrams are still under development at this time, they will be based upon the Booch diagram and will convey the dependency information that the Booch diagrams exhibit, while

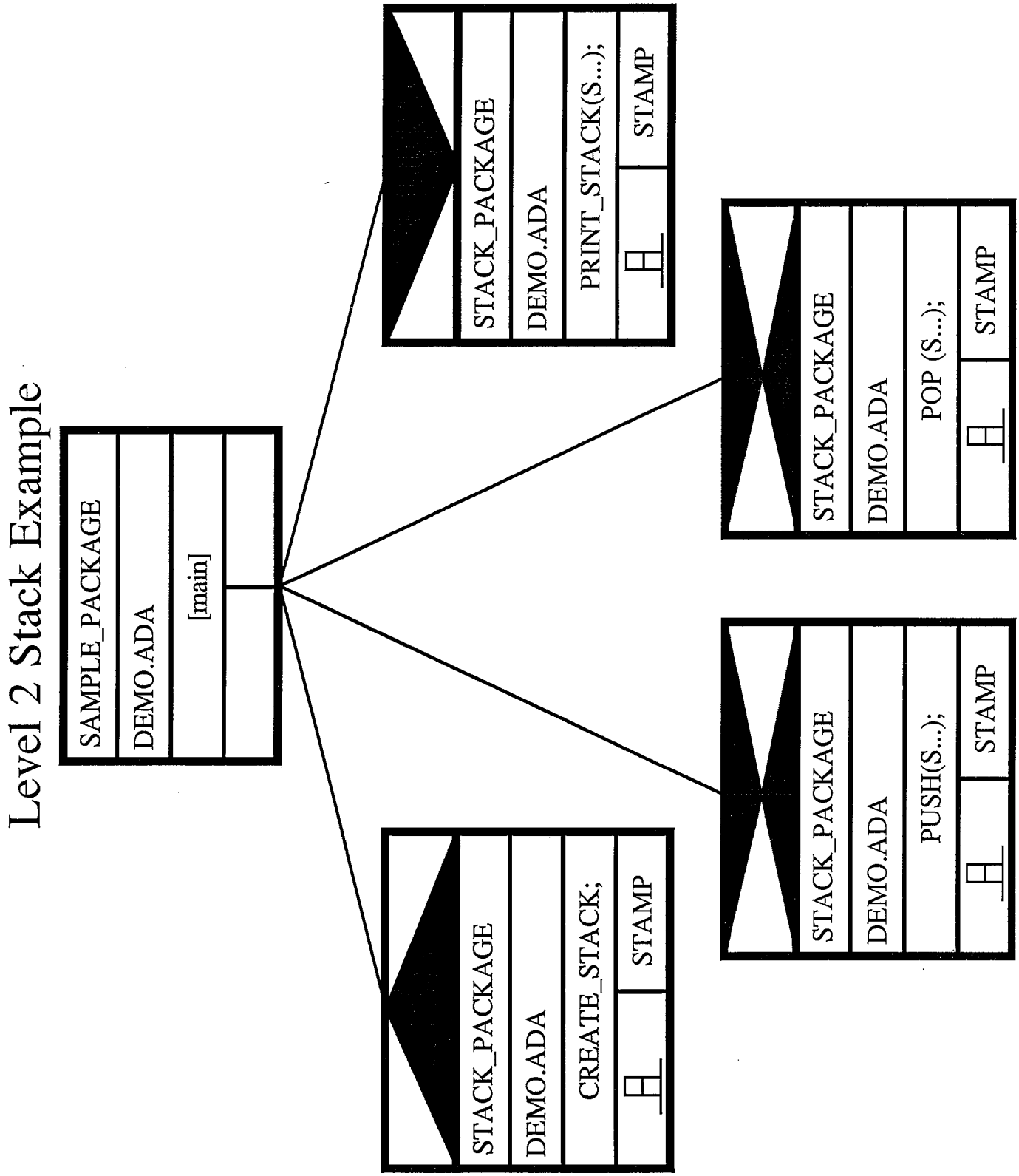


Figure 10. Level 2 Construct Example

extending the diagrams to more fully suit Ada and customizing them for inclusion in the GRASP/Ada system.

4.0 PRELIMINARY REQUIREMENTS

The prototype tool in Phase 2 of GRASP/Ada will be a reverse engineering tool for automatically deriving graphical representations of Ada source code. Graphical representations include the Control Structure Diagram for depicting control flow and various hierarchical diagrams. The hierarchical diagrams will include forms of:

-- Subprogram invocation graphs

-- Package/compilation unit dependency diagrams

The tool requirements are outlined below.

4.1 Functional Requirements

The following sections describe the requirements for the functionality of the tool. Discussed are the requirements for the input of source code to the tool, the processing of the code by the tool, and the display and printing of results by the tool.

4.1.1 Input Requirements

The user will have several modes of inputting Ada code to the tool. These alternatives are described below. For instance, input can proceed via an editor linked to the tool (e.g. **vi**). For the Phase 2 tool, editing capabilities will be text editing only, rather than syntax-directed editing. In addition, no incremental recompilation or reconstruction of diagrams will occur during the editing process. Finally, editor-based input must be done within the context of a "GRASP library". In other words, the

edited file must be associated with the current GRASP library, or else another GRASP library must be specified. See **4.4 System Software Requirements** for more on GRASP libraries.

A second input alternative involves the querying of an existing Ada library (for instance, a VADS library). Such a scheme seems feasible because an Ada library should contain all dependency information among units within a system. This option has been discarded, however, due to schedule constraints and because such an input scheme could become too dependent upon the format chosen by a compiler vendor for its library files.

A third alternative for input involves the direct entry of or selection of file names. The file names need not reflect the true compilation order, since one of the purposes of the tool is to determine that order.

4.1.2 Processing Requirements

This section will describe the general scenario of tool operation. Once the user has selected the Ada files to submit to the tool, he will invoke compilation of the selected files, in turn producing an intermediate form of the Ada code (e.g. DIANA nets) for each unit compiled, deriving dependency information among the units compiled (including noting deficiencies in the supplied compilation list), and creating appropriate entries in the selected GRASP library.

Once the needed information is derived and stored in the GRASP library, the user will select the diagrams that he wishes to generate. The tool will then generate the necessary graphical descriptions. Among the options open to the user are:

-- CSD:
The user will select units in compilation set for which he wishes to see CSDs;

-- Collapsed CSD

-- Subprogram invocation graph

-- Object/Package diagram:
The notation introduced by Booch [BOO87a, BOO87b, BOO86, BOO83] is the leading candidate for this. An object/package diagram will show the seniority relationship among units (as defined by **with** clauses) as well as the existence of separate specifications and bodies. If a package is composed of units unknown outside that package then a separate object/package diagram will be required to display the interrelationships among the composing units.

It is envisioned that the architectural diagrams will be divided into components corresponding to the subprograms or units that they represent. The display mechanism will bring together the components when needed.

Once the tool has created the desired diagrams, the user will select the particular diagrams that he wishes to display. See section **4.1.3 Display Requirements** for more details on display options.

The tool should maintain consistency of diagram (or diagram component) to code to the extent of noting when a diagram (or diagram component) has become obsolete through editing of corresponding code. The tool should have the ability to add new units or supply missing units and determine scope of subsequent recompilation.

The tool should detect incompleteness in the compilation list initially submitted. Also, inconsistency between specifications and bodies should be flagged. Finally, the tool should flag any error that would be found by a validated Ada compiler; however, the system should be capable of processing and displaying diagrams which represent incomplete programs.

4.1.3 Display Requirements

Once the tool has generated diagrams, the user will select which diagrams he wishes to display. He should choose from the four views available (i.e. CSD, Collapsed CSD, subprogram invocation graph, object/package diagram). Each view selected will have its own display window. Display layout should be improved by a rule base which specifies heuristics for icon placement and connection.

4.1.4 Output Requirements

To print an entire diagram requires reconstruction of the diagram in a work buffer (screen). It should be possible to print a single component of an architectural diagram (e.g. a completely specified component of an object/package diagram, showing applicable operations). All printing should be submitted to a PostScript printer.

4.2 User Interface Requirements

The user interface should be a window-based environment with diagrammatic views displayed in individual windows. Command selection should be based on pull-down/pop-up menus. Mouse selection of command options and of diagrams on which to exercise those options should be allowed. X Windows is currently the leading candidate; however, the final decision will be based on available tools of the Ada environment in which the prototype is built.

4.3 Hardware Requirements

The intended platform for development and distribution will be a Sun/SPARC workstation. The advanced graphics capability of this system was a primary consideration. Other options included the VAX 11-780 and a PC environment.

4.4 System Software Requirements

The system software includes a base operating system (which supports a windowing environment) and an Ada compiler/development system. Ideally, the GRASP/Ada tools require access to the intermediate form generated by the compiler. One such example of this intermediate form is DIANA which is described briefly below.

4.4.1 DIANA--An Intermediate Representation for Ada

DIANA, Descriptive Intermediate Attributed Notation for Ada, is an intermediate representation language for Ada source code. DIANA is called a "language" because its definition [GOO83] is described in a BNF-like notation known as Interface Description Language (IDL) [NES81, GOO83, McK86]; in reality, DIANA is an abstract data type whose model is that of an abstract syntax tree supplemented with semantic links, creating a DIANA net. A DIANA net consists of typed nodes decorated with four types of attributes: (1) syntactic (links to other nodes producing the tree), (2) semantic (producing a directed acyclic graph), (3) lexical, and (4) code generation-specific. An instance of DIANA with only lexical and syntactic attributes comes close to a comparable abstract syntax tree except that some similar nodes (e.g.

nodes referencing identifiers) are typed differently so that each type may contain different semantic attributes. In addition, a storable form of DIANA is defined to facilitate reuse of specific instances of the data type. [GOO83, ROS85]

Figure 11 partially illustrates the contents of a DIANA subnet corresponding to a segment of Ada code. Consider the following segment:

```
type MYFLOAT is digits 6 range -1.0..1.0;
subtype MYFLOAT2 is MYFLOAT digits 2;
X : MYFLOAT2;
```

The figure illustrates in part the concurring DIANA subnet. For convenience, the diagram is split into three sections paralleling the subnet for each line in the above code. These three subnets are part of a larger DIANA net for the enclosing unit. The subnet for the variable declaration has its basic abstract syntax tree form (syntactic attribute names prefixed by *as_*), supplemented by a semantic attribute (named *sm_type_struct*) pointing back to a subnet containing the subtype structure of MYFLOAT2. This subnet, in turn, has its own semantic attribute (again named *sm_type_struct*) pointing back to the underlying type structure. This figure, adapted from [GOO83], is incomplete in that many more semantic attributes exist which may point to distant subnets when evaluated.

DIANA was first developed in 1981 by the cooperative effort of teams from the University of Karlsruhe (West Germany), Carnegie Mellon University, Intermetrics, and Softech. The design was based on previous intermediate languages TCOL [BRO80, GOO83, McK86] and AIDA [DAU80, PER80, GOO83, McK86]. A revision effort headed by Arthur Evans, Jr. and Kenneth J. Butler at Tartan Laboratories under the auspices of the Ada Joint Program Office produced a revision of DIANA based on the

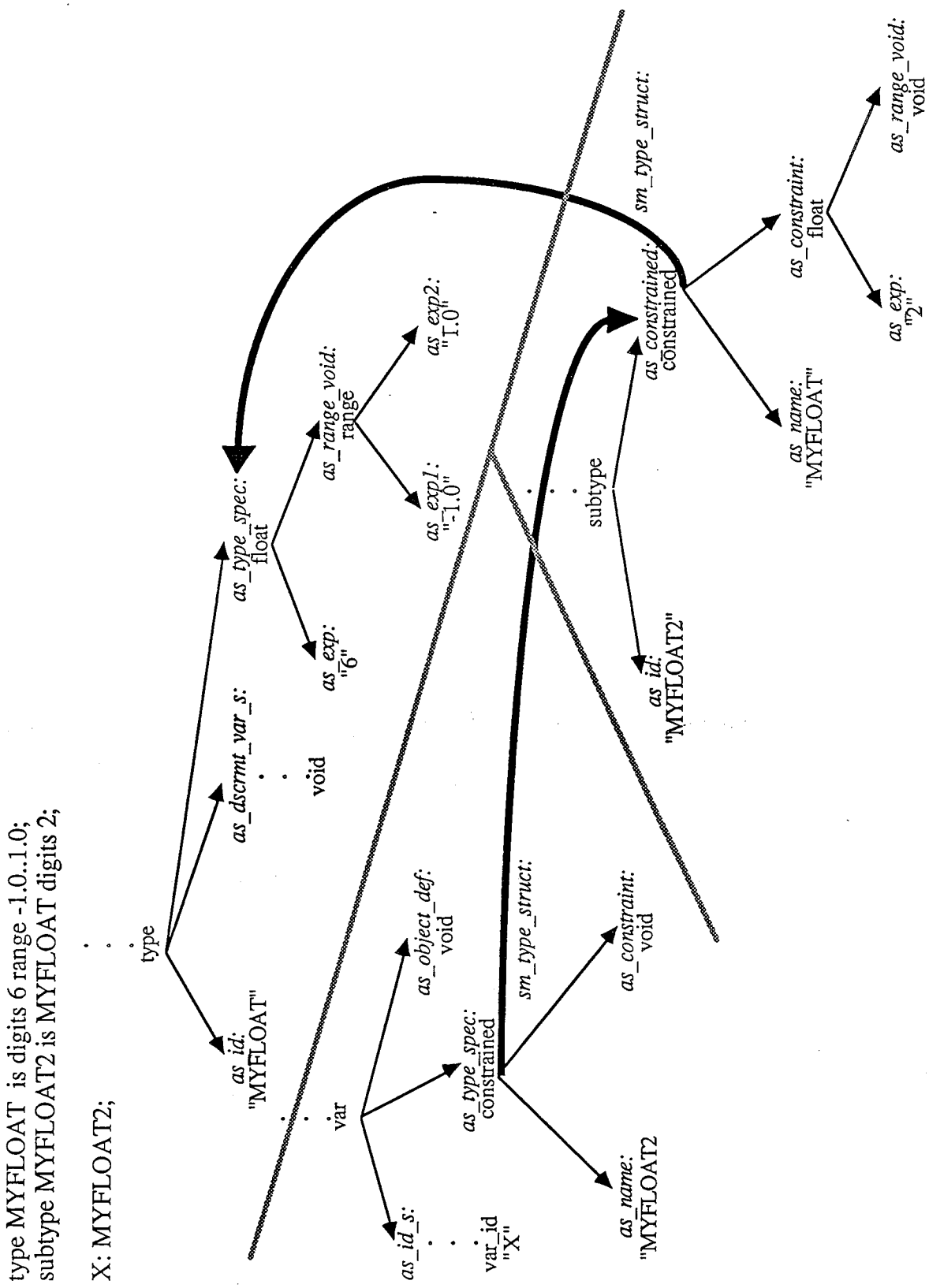


Figure 11. DIANA Subnet Example

1982 version of the Ada definition. This edition contained an Ada package specification for the DIANA data type. [GOO83] A third revision was drafted in 1986 by Carl F. Schaefer and Kathryn L. McKinley of Intermetrics for the Naval Research Laboratories; however, no example Ada package specification for the DIANA type was provided. [GOO83, McK86, SMI88] The MITRE Corporation derived two package specifications in its effort to evaluate the 1986 version of DIANA. [SMI88]

The original purpose of the DIANA data type was to serve as a basis for communication between early and late stages of compilers [GOO83]; in fact, [SMI88] mentions several compilers which are DIANA-based including VERDIX, Rational, and others. However, [GOO83] claims the suitability of DIANA for other tools as well. Several of these tools are mentioned below along with discussions of their DIANA implementations.

[ROS85] is concerned with the use of DIANA data type templates to create source "transformation tools". However, the article was useful in that it demonstrates the necessary contents of a DIANA support toolset. As described by Rosenblum, the necessary tools include a parser to translate Ada source into an abstract syntax tree, a "tree normalizer" to convert the AST to a full DIANA net, a pretty printer to revert the DIANA net to Ada source, a "tree dumper" to convert the internal DIANA to external (ASCII) DIANA, and a "tree reader" to perform the inverse function. The tools described in [ROS85] were based on the 1983 version of DIANA.

[SMI88] describes the MITRE effort in evaluating the 1986 version of DIANA. This involved the translation of the IDL specification for DIANA into a data type and structure specification plus operations on that type using the IDL Toolkit developed at

the University of North Carolina. [WAR85, SNO86, SMI88, SHA89] Also required were the development of a parser and a set of packages to connect the semantic links of the underlying DIANA tree.

[MEN89] describes the Stanford implementation of Anna, a superset language of Ada containing formal annotations. The manual describes the tools which comprise the Anna toolset and outlines scenarios for their use. Most of these tools work with DIANA nets in varying stages of development. The DIANA implementation is based on the 1983 version of DIANA and on the work described in [ROS85].

The major tool dealing with DIANA is in fact the package **ast_v.a** which provides the definition of the DIANA type, of constituent types, and of the operations on those types. In addition to the node types mentioned in [GOO83], there are node types which are specific to Anna and are not defined in standard DIANA. There is a parser which translates Anna source code (or presumably pure Ada code) into a DIANA abstract syntax tree with possible Anna-specific nodes. A semantic processor adds the semantic links, changing the tree into a directed acyclic graph. A transformer translates the Anna-specific subnets into pure Ada-based DIANA.

There are other support tools such as a DIANA reader/dumper, a DIANA-to-Anna (or Ada) pretty printer, and a parser generator complete with an Anna grammar.

An interesting problem which could have arisen with the use of this toolset would be the possible overhead resulting from the fact that the toolset implements a superset of Ada (e.g. the use of the transformer). Another problem which could have proven troublesome is the incompleteness in the implementation of Ada semantics.

Currently, the team is considering using the DIANA interface used by the VERDIX VADS compiler. It is unknown at this time exactly what facilities are provided by this interface and by the compiler as a whole (although sales literature has alluded to certain features such as automatic determination of compilation order). In fact, the feasibility of many of the tool requirements specified herein is contingent on the nature of the VADS facilities available to the project.

In general, DIANA would be useful to the project in that it provides a fairly standard persistent representation which will be needed to derive many of the graphical representations described herein. However, there are several deficiencies inherent to DIANA which would have to be addressed. One such deficiency is the lack of connectivity of DIANA nets corresponding to different units. For instance, a package body DIANA net will point back to the DIANA net corresponding to the package specification, but not vice versa. While this instance and other similar instances were conscious design decisions of the DIANA developers (i.e. not to allow "forward references") [GOO83, McK86], this constraint would impede the tool's ability to associate specifications and bodies bidirectionally. These deficiencies will have to be addressed in library management.

Another serious issue is the sheer size of a DIANA net in relation to corresponding code. If the external representation of a DIANA net were in ASCII form as described in [GOO83], then storage constraints could be rather confining. [SMI88]

4.4.2 Library Management

The purpose of a GRASP library is to maintain information on an Ada system needed to produce appropriate graphical representations. Among the necessary tasks of the library will be to maintain hierarchical relationships of various sorts among the program units in the system. It is envisioned that the library would act as a supplement to DIANA in the areas of deficiency mentioned earlier were DIANA chosen as the intermediate representation.

The entity-relationship database model is recommended for APSE databases [McD84, LYO86]; such a choice is quite appropriate given the variety of relationships among units of an Ada program. For each unit (whether such a unit is embedded within another or not), the library should contain, among other things, the name of the unit, its intermediate representation, a file name and position where the unit can be located, a timestamp, and any graphical representation heretofore created corresponding particularly to that unit. Each unit can be related by various forms of hierarchy, and this relationship will be reflected the library structure as well.

The library command structure will probably follow the command structure of the underlying system or of Stoneman requirements.

4.4.3 Graphics Tools Requirements

Tools will be required to produce icons appropriate for the diagrams expected to be produced by the GRASP tool. Since the X Windows system is the prime candidate as the interface construction tool, the X Windows graphics facilities are likewise the prime contender as the icon construction tool.

5.0 FUTURE WORK (December 1989 - May 1990)

The work planned for December 1989 through May 1990 is summarized in the GANT chart on the next page. The most critical decision that must be made in the immediate future concerns the selection of a commercial Ada environment in which to build the Phase 2 prototype. VERDIX Corporation and TeleSoft each have Ada environments that run on the Sun-4 platform. Negotiations are in progress regarding access to their respective intermediate representations of Ada. VERDIX uses a subset of DIANA and Telesoft uses a more efficient (time/space) representation called High Form. Either of these will provide the functionality needed for the Phase 2 prototype.

The formulation of the base set of architectural diagrams is expected to be ongoing. Three distinct diagrams have been identified for inclusion in the Phase 2 prototype: Collapsed Control Structure Diagram, Structure Chart, and Object/Package Diagram. The detailed requirements for the unparsing/display algorithms for each of these will be developed as the base set is established.

The system dictionary or GRASP library may be implemented directly by supplementing the DIANA software received from Stanford or indirectly by interfacing with the intermediate forms of Ada (DIANA or High Form) generated by the commercial Ada development environment. The latter is preferred since it would provide a complete integrated, operational environment in which to evaluate the GRASP/Ada reverse engineering tools.

BIBLIOGRAPHY

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

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

[BAR89] Barnes, J. G. P. *Programming in Ada.* (3rd Ed.) New York : Addison-Wesley. 1989.

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

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

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

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

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

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

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

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

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

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

[DEC85] *Developing Ada Programs on VAX/VMS.* Maynard, MA : Digital Equipment Corporation. 1985.

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

[FRE85] Freedman, Roy S. *Programming with APSE Software Tools.* Princeton, NJ: Petrocelli Books. 1985.

[GEH87] Gehani, Narain. *UNIX Ada Programming.* Englewood Cliffs, NJ : Prentice-Hall, Inc. 1987.

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

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

[HEI88] Heilbrunner, Stephan, ed. *Ada in Industry : Proceedings of the Ada-Europe International Conference (Munich 7-9 June 1988).* New York: Cambridge University Press. 1988.

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

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

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

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

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

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

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

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

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

[PRA84] Pratt, Terrence W. *Programming Languages : Design and Implementation.* Englewood Cliffs, NJ : Prentice-Hall, Inc. 1984.

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

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

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

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

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

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

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

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

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

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

Appendix A

GRASP/Ada Control Structure Diagram

GRASP/Ada Control Structure Diagrams

James H. Cross II
Computer Science and Engineering
Auburn University, AL 36849
(205) 844-4330

Overview

The GRASP/Ada (Graphical Representation of Algorithms, Structure, and Processes for Ada[1]) Project[2] at Auburn University is a research effort focused on design, analysis and reverse engineering of Ada software and, in particular, the creation of software tools for extraction and generation of graphical representations from Ada/PDL or source code. Our primary motivation for reverse engineering is increased support for software reusability and software maintenance, both 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. Alternatively, the user may specify the software in Ada or Ada/PDL and then automatically generate the graphical representations either incrementally as the code is entered or as a form of post-processing.

---

[1]"Ada" is a registered trademark of the the U.S. Government, Ada Joint Program Office.

[2]This project is funded, in part, by George C. Marshall Space Flight Center, NASA/MSFC, Alabama 35812.

Figure 1 shows the project divided into three phases, each of which corresponds to one of the following broad categories of graphical representations: (1) algorithmic (PDL/Code), (2) architectural, and (3) system level diagrams. Each of these categories may contain overlapping entries that depict, for example, data structure, data flow, or other useful relationships. Phase 1 of GRASP/Ada has been completed and a new PDL/code level graphical notation and supporting software tool is now available for evaluation. Phase 2 of the research is
currently underway.

Phase 1: The Control Structure Diagram For Ada

As the complexity of software has increased, so has the utility of graphical representations for algorithms. The industry has progressed well beyond the simple constructs of sequence, selection and iteration promoted by the theory of structured programming in the 1970's. For example, Ada includes control constructs for concurrency (tasks and task rendezvous), exception handling, and loop exits, none of which fits well into the simple sequential control constructs of structured programming. Since the ANSI flowchart was introduced in the mid-50's, numerous notations have been proposed and utilized. These notations typically include control constructs for sequence, selection, and iteration, and several include constructs for concurrency and exits; however, none explicitly contains all of the control constructs found in Ada.

For the GRASP/Ada project, we selected the Control Structure Diagram (CSD)[2] as a basis for a graphical representation that maps directly to Ada control constructs. The CSD is a graphical notation intended to increase the comprehensibility of Ada PDL or

source code by explicitly depicting control constructs and control flow. The traditional textual representation of PDL or source code has been extended with intuitive graphical constructs which are easily adaptable to editors and printers. The CSD has the attractive property that it can be overlaid directly on prettyprinted Ada code. In fact, a CSD generator may be perceived as a "graphical prettyprinter." Figure 2(a) contains an Ada task body adapted from Barnes[3] which continually loops through a priority list until it receives a request for a rendezvous or alternatively reaches the end of the list. In either case, the scan of the list is restarted at the head of the list to service higher priorities first. Figure 2(b) shows the corresponding CSD which highlights the control paths of the nested loops, the select and the task rendezvous. The improved readability provided by the CSD reduces the time required to understand the code (i.e., misinterpretations are reduced as well as all of the time lost as a result). This is especially important during code or PDL reviews and maintenance when readers are not intimate with the code. Figure 3 contains each of the individual CSD constructs for Ada.

The creation of a prototype software tool to support the generation of the CSD from source code was relatively straightforward. We used a scanner and parser generator which accepted as input a grammar for Ada with embedded calls to the routines which generate the graphical constructs. The CSD offered a simple presentation format to the screen and printer in that the diagrams can be composed of characters from a custom font thus allowing the diagram and code to co-exist in the same ASCII file.

A user-friendly and relatively portable interface provides the user with the capability to specify options quickly without having to learn cumbersome command languages. The user has the option to choose from a variety of line spacings, font styles, printers, etc.

All options are visible onscreen and can be selected and modified using only the terminal cursor keys and the RETURN key. The user may preview the CSD before printing with a specially modified version of the VAX EVE editor. The editor allows the user to suppress the CSD to display conventional prettyprinted code. Version 2 of the prototype will support (1) collapsing the diagram based on the constructs of Ada, (2) editing the diagram directly, and (3) updating the diagram incrementally.

Phase 2: Architectural Diagrams For Ada

The Phase 2 prototype, which includes a collapsed version of the CSD as well versions popular diagrams such as Yourdon's structure chart and Booch's Object/package notation, is currently under development. It is loosely integrated with a commercially available Ada environment through an interface to the intermediate representation of Ada generated from the compiler. The Phase 2 prototype is expected to be available for initial evaluation in Fall 1990.

References

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

2. 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*, January 6-8, 1988, pp. 446-454.

3. J. G. P. Barnes, *Programming in Ada*, (3rd Ed.) Addison-Wesley, New York, 1989.

GRASP OVERVIEW

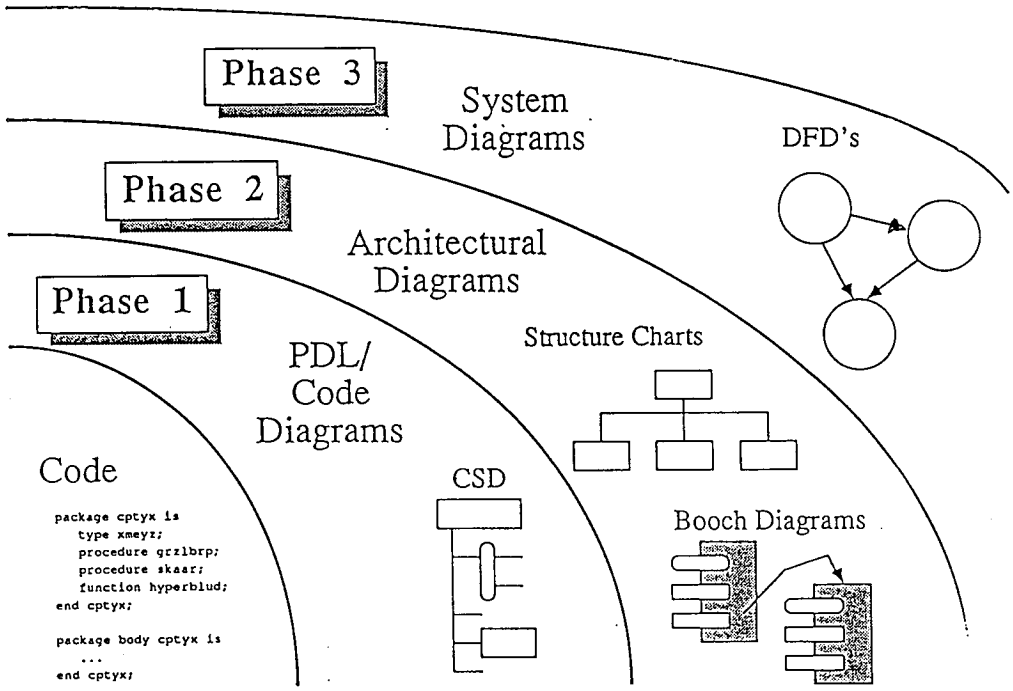


Figure 1. GRASP Overview

```
task body CONTROLLER is
begin
   loop
      for P in PRIORITY loop
         select
            accept REQUEST(P) (D:DATA) do
               ACTION(D);
            end;
            exit;
         else
            null;
         end select;
      end loop;
   end loop;
end CONTROLLER;
```

Figure 2(a). Sample Ada Source Code.

from Barnes, J.G.P., 1984, Programming in Ada, 2nd Edition, Addison-Wesley Publishers Limited, Reading, Massachusetts.

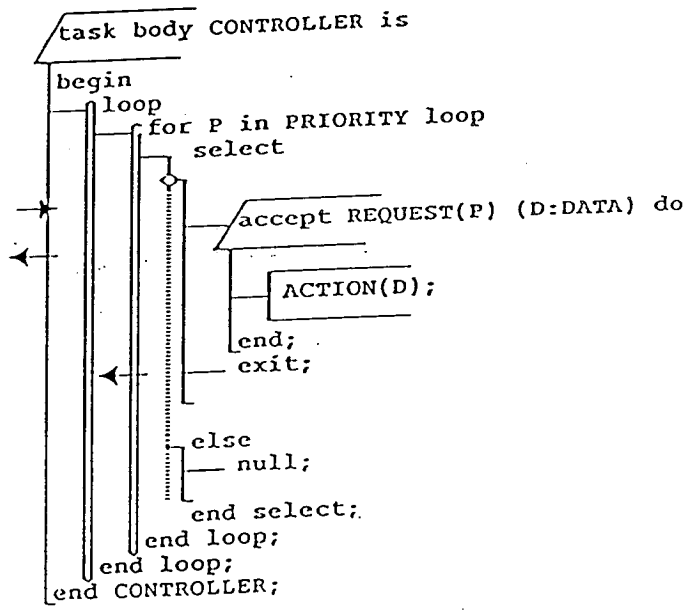


Figure 2(b). Sample Ada Source Code Overlaid with Control Structure Diagram.

```
-- PROCEDURE
procedure X is
begin
   S;
   S;
   S;
   S;
end X;

-- PACKAGE
package Y   is
   procedure Z;
   function Z return Boolean ;
end Y;

-- SEQUENCE
S;
S;
S;
S;

-- SELECTION
S;
if C then
   S;
   S;
else
   S;
   S;
end if;
S;

-- CASE
S;
case D is
   when C1 =>
      S;
   when C2 =>
      S;
end case;
S;

-- FOR
S;
for F in R loop
   S;
   S;
   S;
end loop;
S;

-- WHILE
S;
while C loop
   S;
   S;
   S;
end loop;
S;
```

Figure 3. Control Structure Diagram (CSD) Constructs For Ada

```
-- INFINITE LOOP
   S;
   loop
      S;
      S;
      S;
   end loop;
   S;
```

```
-- GO TO
   S;
   <<L>>
   S;
   S;
   goto L;
```

```
-- LOOP EXIT
   S;
   loop
      S;
      exit when C;
      S;
   end loop;
   S;
```

```
-- RAISE
   S;
   S;
   raise Err;
```

```
-- EXCEPTION HANDLER
   S;
   S;
   S;
   exception
      when Err1 =>
         S;
      when Err2 =>
         S;
      when Err3 =>
         S;
   end;
```

```
-- BLOCK
   S;
   begin
      S;
      S;
      S;
   end;
   S;
```

```
-- BLOCK WITH DECLARATIONS
   S;
   declare
      C : INTEGER;
   begin
      S;
      S;
      S;
   end;
   S;
```

Figure 3 (Continued). Control Structure Diagram (CSD) Constructs For Ada

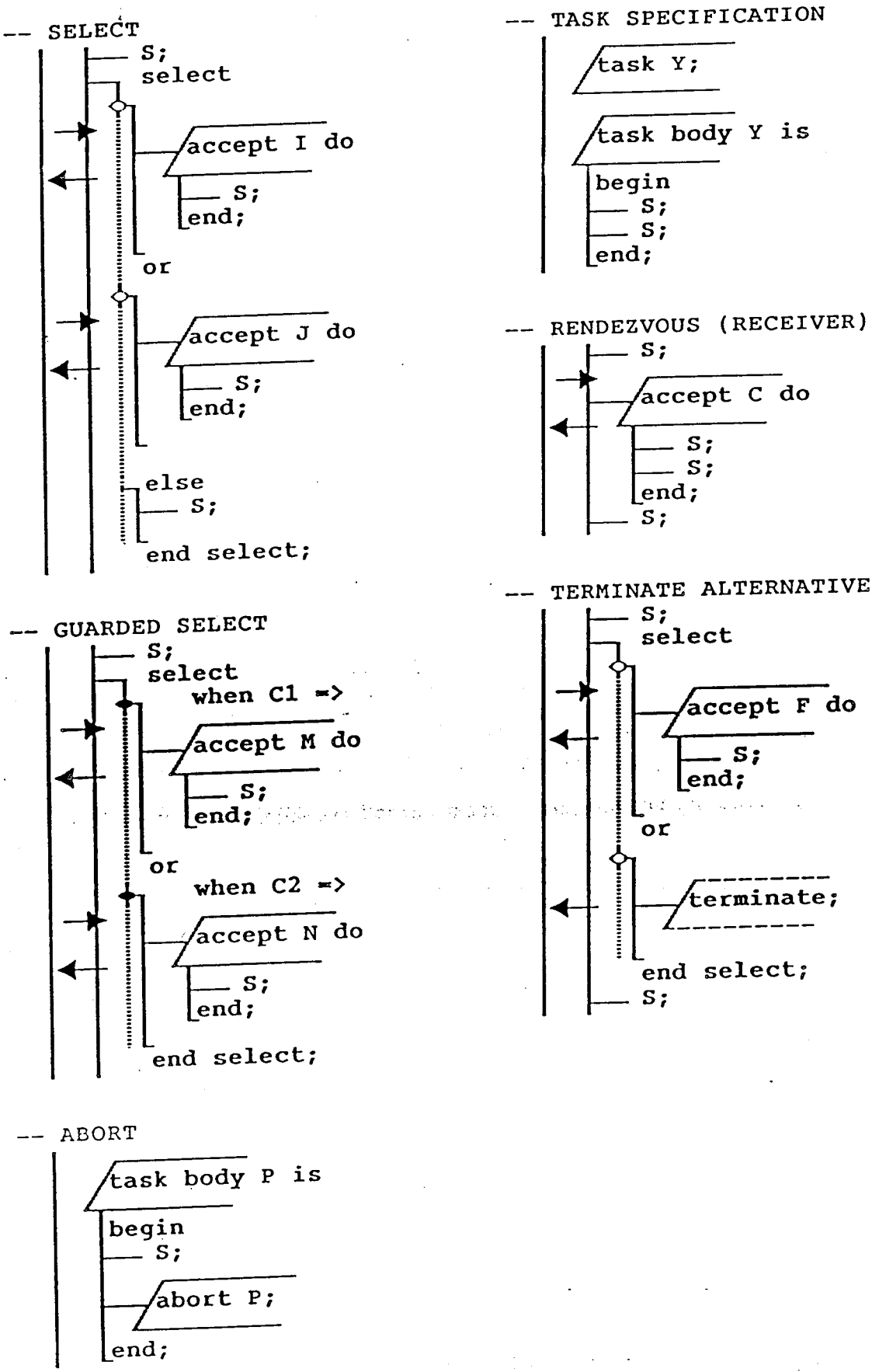


Figure 3 (Continued). Control Structure Diagram (CSD) Constructs For Ada