A Graphically Oriented Specification Language for Automatic Code Generation

GRASP/Ada – A *G*raphical *R*epresentation of *A*lgorithms, *S*tructure, and *P*rocesses for *Ada* **(Phase I).**

Final Report
NASA-NCC8-13 (SUB 88-224)

Prepared by

James H. Cross II, Principal Investigator

Kelly I. Morrison, Graduate Research Assistant
Charles H. May, Jr., Graduate Research Assistant
Kathryn C. Waddel, Graduate Research Assistant

Department of Computer Science and Engineering
Auburn University, AL 36849-5347
(205) 826-4330

Prepared for

The University of Alabama at Huntsville
Huntsville, AL 35899

George C. Marshall Space Flight Center
NASA/MSFC, AL 35812

September 1989

Table of Contents

1.0. Introduction 1

2.0. Literature Review 7
2.1. Automatic Code Generation 7
2.1.1. Non-Graphical Specification Languages 10
2.1.2. Graphical Specification Languages 17
2.2. Design Methods for Ada 20
2.2.1. Data Flow-Oriented Design 20
2.2.2. Data Structure-Oriented Design 22
2.2.3. Object-Oriented Design 24
2.2.4. Applicability to Ada 27
2.3. Graphical Representations for Algorithms 28
2.3.1. Specific Notations 28
2.3.2. Empirical Studies 33
2.4. Graphical Representations for Architecture 36
2.5. Reverse Engineering 37
2.6. Conclusions 41

3.0. Requirements 43
3.1. Introduction 43
3.2. Prototype Requirements 46
3.2.1. Environment Requirements 47
3.2.2. Scanner/Parser Requirements and Tradeoffs 47
3.2.3. CSD Generator Detailed Requirements 50
3.3. CSD Constructs for Ada 51
3.4. Empirical Evaluation 55
3.4.1. Objectives of Empirical Evaluation 55
3.4.2. Overview of Procedure 56
3.4.3. Implementation Plan 57

4.0. Prototype Design and Implementation 59
4.1. Introduction 59
4.2. Parser/Scanner 60
4.3. Graphical Prettyprinting Routines 61
4.4. User Interface 69
4.5. EVE editor enhancements 72
4.6. Software tools 72

5.0. Examples of output 75
5.1. Example 1 – Complex Numbers 75
5.2. Example 2 – User Interface 77
5.3. Example 3 – Buffers 78

6.0. Future Directions 89

Bibliography 91

Appendices

A. An Empirical Evaluation of Graphical Representations for Algorithms

B. Publications

B1. GRASP/Ada – A Graphical Representation of Algorithms, Structure, and Processes for Ada (Phase I)

B2. An Instrument for Empirical Evaluation of Graphical Representations for Algorithms.

B3. CASE '89 Report: Reverse Engineering and Maintenance.

1.0 Introduction

Automatic code generation may be defined as the creation of code from a higher level specification [BAL85]. The specification should have the property of being easier to create, understand, and maintain than the code generated from it. Ideally, the specification should be non-procedural, resemble documentation rather than detailed logic, and be comprehensible by both the customer and developer. Graphical specifications of systems are more quickly understood than their corresponding textual specifications, and many of the recent approaches to automatic code generation are based, in part, on graphical presentation. Most of these approaches use variants of data flow diagrams and hierarchical charts made popular by Yourdon, Constantine, Gane, and Sarson (e.g. IORL [SIE85] and PAMELA [CRA86]). Graphical representations (GRs) of software represent a major thrust in computer-aided software engineering (CASE) tools in general. While the benefits of CASE tools are still being debated, there is solid evidence of a move in the direction of these graphically oriented tools.

The research reported herein describes the first phase of a three-phase effort to develop a new graphically oriented specification language which will facilitate the reverse engineering of Ada source code into GRs as well as the automatic generation of Ada source code. Figure 1 shows a simplified view of the three phases of GRASP/Ada (*G*raphical *R*epresentations for *A*lgorithms, *S*tructure, and *P*rocesses for *Ada*) with respect to three basic classes of GRs. Phase I concentrated on the derivation of an algorithmic diagram, the control structure diagram (CSD) [CRO88a] from Ada source code or Ada PDL. Phase II includes the generation of architectural and system level diagrams such as structure charts and data flow diagrams and should result in a requirements specification for a graphically oriented language able to support automatic code generation. Phase III will concentrate on the development of a prototype to demonstrate the feasibility of this new specification language.

Figure 1. The Planned Three Year GRASP/Ada Research and Development Schedule.

While generic structure charts and data flow diagrams are widely used graphical tools, the CSD is representative of a new group of graphical representations for algorithms that can co-exist with source code or PDL. Figure 2(a) contains an example of an Ada task body and Figure 2(b) shows the corresponding CSD. CSD constructs are more fully described in Section 3.3.

Phase I of *GRASP/Ada* was intended to provide a theoretical, as well as practical, foundation for the project. It included a survey of previous and current work in the area of automatic code generation, a survey of current methodologies for the design of Ada software, and a survey of graphical representations for systems and algorithms. Phase I was focused on the general problem of graphical representation of several integrated views of algorithms, structure, and processes.

It was mutually agreed between NASA representatives and the researchers that the first phase should concentrate on the complementary problem of generating graphical representations from Ada source code. The justification for this approach was multifaceted. The primary reason is that addressing the generation of GRs from Ada source code provided key insights into the problem of generating code from graphically oriented specifications, the overall goal of the project. Furthermore, since Ada has the potential to become a widely accepted and utilized standard, it provides a firm base from which abstract graphical models can be synthesized.

```
package CHAPTERONE is

   task CONTROLLER is
      entry REQUEST(PRIORITY) (D:DATA);
   end;

end CHAPTERONE;

package body CHAPTERONE is

   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;

end CHAPTERONE;
```

Figure 2(a). Sample Ada Source Code.

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

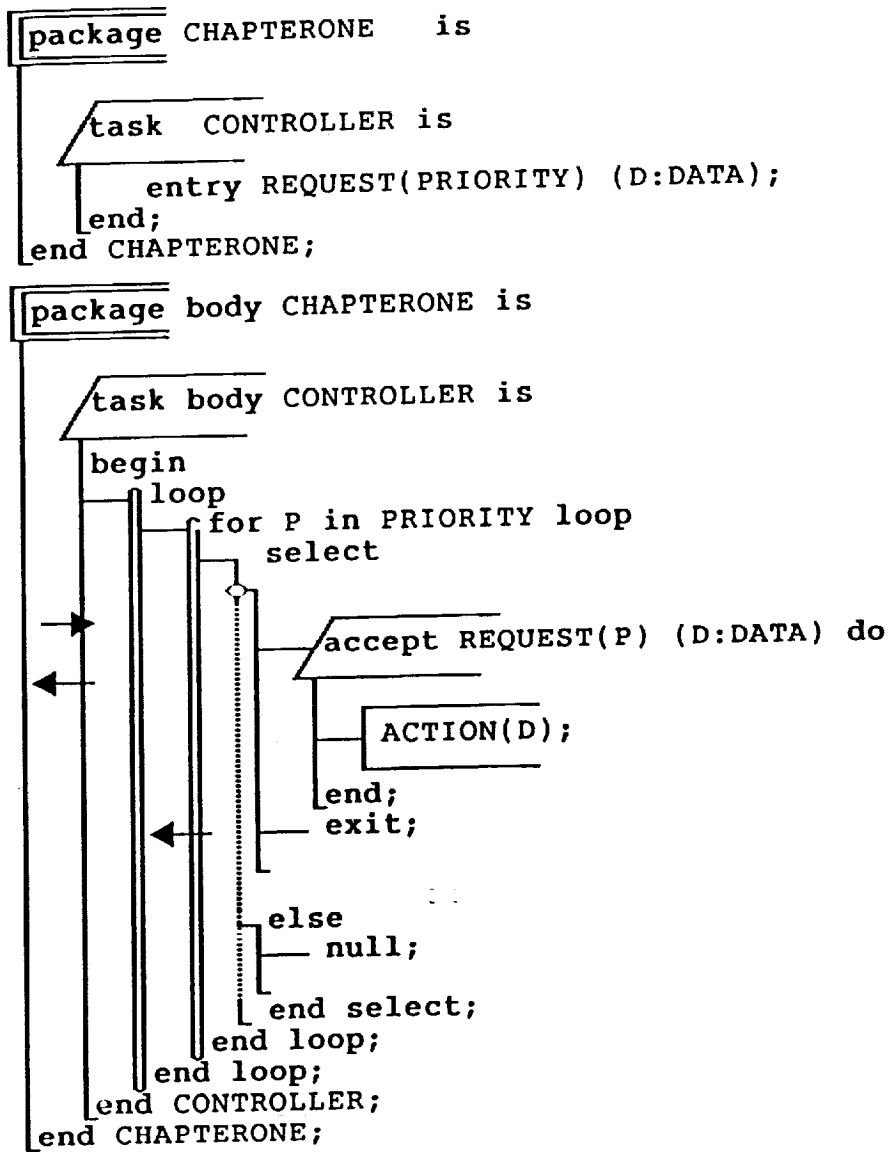


Figure 2(b). Sample Ada Source Code Overlaid with Control Structure Diagram.

Second, the GRASP/Ada CSD generator has the potential to increase the comprehensibility of Ada source code and/or Ada PDL, which may have wide ranging implications for the design, implementation, and maintenance of software written in Ada. In particular, many designers and implementors will be working with Ada or Ada PDL and thus can utilize the tool to provide GRs which are more easily understood than textual equivalents. Understanding between customer and designer, designer and implementor, as well as among individual members of each group, is critical to the success of any project. Maintenance personnel tend to deal with large amounts of foreign code which must be read and understood prior to any modification. Graphical aids which can increase the efficiency of this understanding can reduce the overall cost of maintenance.

Finally, software verification, which is essential throughout design, implementation, and maintenance, can benefit from any useful aid to code reading. Code reading has been found to provide the greatest error detection capability at the lowest cost as compared to functional testing and structural testing [NAS88]. While the actual increased efficiency of understanding (i.e. fewer errors, reduced time) afforded by GRs seems intuitive, this project will also address the empirical evaluation of the proposed tool set.

The remainder of this report is organized as follows. Section 2 provides a survey of the literature in the areas of automatic code generation, design methods for Ada, graphical representation of algorithms, and reverse engineering. Section 3 describes the requirements for Phase I of GRASP/Ada. Section 4 describes the design and implementation of the prototype tool. Section 5 presents some examples of Ada source code that have been processed by the CSD generator. Finally, Section 6 describes future directions for this research. The appendices includes the results of a preliminary empirical evaluation of graphical representations of algorithms and copies of publications produced from the research.

2.0 Literature Review

Several areas of computing were identified as relevant to the current research. The results obtained in automatic code generation were reviewed. Current design methods were explored to identify the many ways in which software engineers specify software, and to see the mechanisms by which these specifications are converted into working source level software. Procedural and architectural graphical representations were examined to see how large software programs may be viewed graphically. Finally, the topic of reverse engineering was explored to see how others are approaching the problem of converting source code into higher level specifications, both graphical and textual. A complete list of the software engineering tools and environments surveyed is provided in Figure 3.

2.1 Automatic Code Generation

The term "automatic code generation" has numerous meanings in the literature. Balzer [BAL85], in his survey of the work done in the field of automatic programming, reiterates the traditional definition:

> "Automatic programming has traditionally been viewed as a compilation problem in which a formal specification is compiled into an implementation."

He then goes on to provide two elaborations of these definitions. The first involves

> "...the addition of an optimization that can be automatically compiled and the creation of a specification language which allows the corresponding implementation issue to be suppressed from specifications."

Surveyed Tools

Name	Class	Graphical?	Generates	Date	Reference
PSL/PSA	SD	NO		1977	Teichroew, et.al.
REVS/RSL	SD	YES		1977	Alford
SA	SD	YES		1977	Ross
ARGUS	SD	YES		1983	Stucki
TRIAD	SD	NO		1983	Kuo,et.al.
HIDOC	RE,M	YES		1984	Harada, et.al.
SLAN-4	SL	NO		1984	Beichter, et.al.
ANNA	SL	NO	Ada	1985	Luckham,et.al.
Descartes	SL	NO		1985	Urban, et.al.
Gandalf	SD	NO		1985	Habermann, et.al.
GIST	SD	NO		1985	Balzer
IORL/TAGS	SD	YES	Ada	1985	Sievert, Mizell
KBEmacs	SD	NO		1985	Waters
Larch Family	SL	NO	*	1985	Guttag, et.al.
PhiNIX	SD	NO		1985	Barstow
PROMPTER	RE	NO		1985	Fukunaga
TSL	SL	NO	Ada	1985	Helmbold, et.al.
PAISLey	SL	NO		1986	Zave, Schell
PAMELA/AdaGRAPH	SD	YES	Ada	1986	Crawford, et.al.
SPC/SCHEMACODE	SD,SL	YES	FORTRAN, C, Pascal, dBASE III, COBOL	1986	Robillard
Transformation Schema	SD	YES		1986	Ward
GRASP/GT	SL	YES	Ada	1987	Morrison
WLISP	RE	YES		1987	Fischer, et.al.
D*	RE	YES		1988	Blaze, Cameron
GETS	SD	YES		1988	Arthur
GRAPES/86 & GRAPES	SL	YES		1988	Wagner
KDA	EV	NO		1988	Sharp
TOMALOGIC	RE	NO		1988	Lerner
VIC	SD,M	YES	C	1988	Rajlich,et.al.

Key:
SD - Software Development, RE - Reverse Engineering, SL - Specification Language, M - Maintenance, EV - Evaluation

Figure 3. Surveyed Software Engineering Tools and Environments.

In the second definition

> "... a desired specification language is adopted, and the gap between it and the level that can be automatically compiled is bridged interactively."

Balzer views these approaches as complementary, with the second approach elaborating on the concepts set forth in the first. He believes that automatic programming is not entirely possible, but will involve an interactive step in which the program generator resolves ambiguities and patches incomplete specifications by interrogating the user.

Rich and Waters [RIC88] set forth what they term the "cocktail party" definition for automatic programming:

> "There will be no more programming. The end user, who only needs to know about the application domain, will write a brief requirement for what is wanted. The automatic programming system, which only needs to know about programming, will produce an efficient program satisfying the requirement. Automatic programming systems will have three key features: They will be end-user oriented, communicating directly with end users; they will be general purpose, working as well in one domain as in another; and they will be fully automatic, requiring no human assistance."

They then proceed to point out several problems with this definition. First, they argue that automatic programming systems cannot be domain-independent, but must have some knowledge about the particular field of programs they are expected to generate. Second, they argue that fully automatic programming is not possible, because it would require that the automatic programming system have a knowledge base for *every* application domain. Third, they argue that requirements cannot possibly be fully specified, and that some degree of interactivity is necessary for automated code generation.

Rich and Waters note that current automatic programming methods fall into four categories: (1) procedural methods, which typically use high level and very high level languages; (2) deductive methods, which create programs after first finding "a constructive proof of the (program) specification's satisfiability"; (3) transformational methods, which

take very high-level language specifications and translate them into working programs via successions of transformations; and (4) inspection methods, which detect "motifs" or "cliches" in a problem and match them to existing implementations or implementation templates.

An interesting observation made by Rich and Waters states that "(t)o date, essentially all commercialization of automatic programming research has been via the very high level language approach. However, we will soon begin to see the first commercialization of research on the assistant approach."

Barstow [BRS85] discusses "automatic programming systems" and, in particular, his PhiNIX project for automatically generating programs for use in application areas involving oil well logging. He defines such a system as:

> ". . . allow(ing) a computationally naive user to describe problems using the natural terms and concepts of a domain with informality, imprecision, and omission of details. An automatic programming system produces programs that run on real data to effect useful computations and that are reliable and efficient enough for routine use."

2.1.1 Non-Graphical Specification Languages

A popular method of achieving automated code generation is through the use of a specification language. A specification language is a "formal way[s] of representing [a] specification[s] with high precision" [MAR86], that "provides facilities for explaining a program" [LUC85]. Beichter, Herzog, and Petzsch [BEI84] state that "the objective of these languages is to prevent design errors. . . at an early stage of software development." Jones [JON80] states that "it is the role of a specification to record precisely what the function of a system is." Abrial [ABR80] agrees, saying "the formal specification of a problem is provided by a strict statement of its contents written in a non-natural language." Meyer [MEY85] expounds on these definitions, saying that "their underlying concepts are,

for the most part, well-known mathematical notions like sets, functions, and sequences." Kemmerer [KEM85] agrees, stating that a high level formal specification of a system "gives a precise mathematical description of the behavior of the system omitting all implementation details," accompanied by "zero or more less abstract specifications which implement the next higher level specification with a more detailed level of specification." However, not everyone agrees that specifications should be isolated from their implementations. Indeed, Guttag, Horning, and Wing [GUT85] have done research on a two-tiered approach to software specification in which the lower tier is tailored to specific programming languages. Luckham and Henke [LUC85] consider high level languages that have been extended with proper annotations to be specification languages; certainly these cannot be independent of implementation.

Luckham and Henke also state that there are two different approaches to be taken in designing specification languages. One is the "fresh start," where the language is designed from scratch and based on a sound mathematical background. The other is "the evolutionary approach, whereby an existing high-level programming language is extended."

Alford [ALF77] reiterated ten desirable properties of a software specification that were summarized by Bell and Thayer:

- Completeness
- Correctness
- Unambiguity
- Traceability
- Modularity
- Consistancy
- Testability
- Design Freedom
- Communicability
- Automatability

Sievert and Mizell [SIE85] identified several goals that were desired in IORL (Input/Output Requirements Language), including:

- enforcement of a rigorous methodology for system development
- applicability to all systems, not just computer systems
- ease of use (systems should be difficult to misuse)
- the capability to express system performance characteristics and algorithms using common mathematical notation
- the use of graphical symbols derived from general systems theory

Guttag, Horning, and Wing [GUT85] pointed out several desirable features that are embodied in their Larch family of specification languages. Some of these are:

- Composability
- Emphasis on presentation
- Suitability for integrated interactive tools
- Semantic checking
- Localized programming language dependencies

Meyer [MEY85], who assisted in the creation of an unnamed specification language [ABR80], addresses the issue of software reusability as an important consideration: "An essential requirement of a good specification is that it should favor reuse of previously written elements of specifications."

Luckham and Henke [LUC85], the creators of ANNA (a specification language for Ada) stated that their system:

- should be easy for an Ada programmer to learn and use
- should give the programmer the freedom to specify and annotate as much or as little as he wants and needs
- should encourage the development of new applications of formal specifications

Martin [MAR85a] listed a large number of desirable properties of a specification language. He believes that a good specification language:

- improves conceptual clarity
- should be easy to learn and use
- should be computable
- should be rigorous and mathematically based
- should use graphic techniques that are easy to draw and remember
- should employ a user-friendly computerized graphics tool for building, changing, and inspecting the design
- should employ an integrated top-down or bottom-up design approach
- should indicate when a specification is complete
- should employ an evolving library of subroutines, programs, and all the constructs the language employs
- should link automatically to data-base tools, including a dictionary
- should guarantee interface consistency
- should be easy to change

Meyer [MEY85] stated seven problem areas, which he termed the "seven sins of the specifier," that should be addressed by a specification language. These are:

- Noise
- Silence
- Overspecification
- Contradiction
- Ambiguity
- Forward reference
- Wishful thinking

Balzer [BAL83] identifies several features which should be provided by support environments for specification languages. A support environment should allow the software engineer to enter a specification concisely, because "the amount of information that must be specified for the system to correctly process the problem must be reduced." Balzer also states that "a mechanism is required for the modification of specifications that have been previously entered." Finally, Balzer says that a support environment, in addition

to generating a source program, should provide "a mechanism for transforming it into an efficient one."

Case [CAS85] identifies a set of tools that could be provided by support environments for specification languages. Some of these tools are:

- an interactive, "friendly" user-interface
- graphics/word processing editors
- project management tools
- design dictionaries and design analyzers

One of the most rigorous forms of specification language is the formal specification language. Formal specification languages have precise semantics and are based upon established mathematical principles [JON80, MEY85]. These languages are used to describe what software should do, and not how it is to be done. In fact, Jones suggests that formal specification languages should not be extended to handle algorithmic specification [JON80]. Formal (implicit) specifications are generally developed as a set of axioms and a set of functions. The functions are described using a type clause, which shows the data types of the inputs and outputs, a pre-condition, which specifies any assumptions which must hold on the input, and a post-condition, which specifies the required relation between the input and the output. The functions are used to define operations which carry a program from one state to another. The chief advantage of formal specification languages is that they are very precise and lend themselves well to formal proofs and verification.

One approach to formal specification is given by Jones [JON80]: the "rigorous approach." Jones approaches the problem of formal specification by using strict mathematical notation to define a kernel of operations which can be used to define the functions to be performed by the software.

SLAN-4 is a formal specification language which bears more resemblance to conventional programming languages than to mathematics. Developed by Beichter, et. al. [BEI84], it introduces the concept of modules (analogous to the functions used by Jones [JON80]) and classes (which are collections of modules accompanied by some declarations

common to the modules). Abstract data types are described algebraically, separating their specification from implementation details. However, SLAN-4 does allow pseudocode to be used to specify low-level design details.

A Software Blueprint is a formalized program specification developed by Chu [CHU82] of the University of Maryland. The typical software blueprint consists of three components: a level A document, which describes a modular decomposition of the system; a level B document, which sketches the control and data flow in each of the modules; and a level C document, which details precisely how to implement the program. The blueprints are written using a combination of SDL-1 (Chu's Software Design Language) and natural language for the level A and B documents, and SDL-1 alone for the level C documents. It is interesting to note that SDL incorporates features such as data structures (trees, queues, lists, etc.) and timing structures (semaphores and switches) as part of the language.

ANNA (ANNotated Ada) is a specification language designed by Luckham and Henke [LUC85] to be used as an extension to Ada. The extensions, called annotations, are embedded in the Ada program as comments and are distinguished from ordinary comments (which begin with "--" in Ada) by the addition of a third character ("--|" , or "--:"). Thus, an ANNA specification is simply an Ada program with formalized comments. Quantified expressions are made available to simplify the writing of specifications, and axioms may be described using an Ada-like notation. In addition, package annotations are used to introduce the concepts of package states, which are modified by the operations contained in the package.

GIST, a specification language which formalizes the constructs used in natural language, has been used with some success by Balzer [BAL85]. The language was employed in developing several real applications and has been chosen as the basis for a software engineering environment being developed at USC. One problem that has been noted is the poor readability of a final GIST specification. USC and TRW are currently working on a paraphraser program to translate GIST specifications into natural language.

PSL/PSA (Problem Statement Language and Problem Statement Analyzer) is a specification language and accompanying requirements analyzer developed by Teichroew and Hershey [TEI77]. System specifications in PSL have eight major components:

- System input/output flow
- System structure
- Data structure
- Data derivation
- System size and volume
- System dynamics
- System properties
- Project management

These components are filled in by the analyst using a predefined format so that the PSA can syntactically analyze the specification. The specification information is collected in a database, from which various analytical reports can be produced. When all of the requirements have been entered, the system gathers the information and produces final specification documents for the system.

Hevis [HEV88] describes a subset of specification languages known as executable specification languages. He defines an executable specification language as "a language which has a natural language syntax with pictorial representation, and the added capability of 3GL code generation." Hevis identifies four important objectives for an executable specification language:

- "to provide systems designers or domain experts which have no programming experience, with the means to write a formal and complete specification of their problem with a minimum of training on the language itself."
- "to be able to develop a system, with a minimum knowledge of the target software and hardware platforms."
- to be able to define problems easily by using visual representations.
- "to be able to execute and test those specifications at the design stage, with an incomplete definition of the problem."

PAISLey is an executable specification language for describing concurrent digital systems [ZAV86]. It uses the technique of functional decomposition, and describes any system as a set of asynchronous processes. "Exchange functions" are used to specify the

interactions between processes. One of the more interesting features of PAISLey is that it can always execute a specification, whether it is complete or incomplete.

Urban, Urban and Dominick [URB85] used the Descartes executable specification language to describe the MADAM information and storage retrieval system at the University of Southwestern Louisiana. Descartes, based upon Hoare's data structuring methods, utilizes operations such as direct product and recursion to break a program's input into parts and then construct an output from those parts. In this respect, Descartes bears a striking resemblance to the data structure-oriented approaches of Warnier [WRN74, WRN81] and Jackson [JAC83].

2.1.2. Graphical Specification Languages

The Structured Analysis and Design Technique (SADT), developed by Ross, et. al. [ROS77], is a graphical language for the specification of systems. Using SADT, a system is decomposed into a set of processes, each represented as text inside a box. Inputs and outputs to the process are shown as labeled arrows entering and leaving the box on the left and right sides, respectively. Control data is shown using a labeled arrow entering the top of the process box. The algorithmic mechanism controlling the process is labeled on an arrow entering the bottom of the process box. Typically, the process boxes are connected to form a "waterfall" configuration. Each SADT diagram is accompanied by an information sheet for project managers.

SREM (Software Requirements Engineering Methodology) was developed by Alford [ALF77] for the specification of large, real-time systems. It utilizes a Requirements Statement Language (RSL), and a Requirements Engineering and Validation System (REVS) which analyzes the RSL statements. SREM centers on the concept of a requirements network (R-Net), a structure useful in describing the responses to a given input or stimulus. Processes on the R-Net can be described using predefined RSL elements, or new RSL elements can be created by the analyst. The SREM methodology is

notable as being one of the few to be applied to large, practical problems.

Many recent specification languages are developed concurrently with specific support environments which often make use of graphical representations of specifications and query users for additional information during development. Four of these languages and environments are described here: USE.IT and 001 with their environments on the DEC VAX; PAMELA with the AdaGRAPH environment on the IBM PC; IORL with the TAGS environment on the Apollo Workstation; and GRASP/GT with its GRASP environment on the Apple Macintosh.

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 takes the code generated from the flow diagrams and fills it in to form completed 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. 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.

In true Ada form, the acronym GRASP has been "overloaded." GRASP/GT (GRaphical Approach to the Specification of Programs/Graphics and Text) is an executable specification language designed by Kelly Morrison of Auburn University for specifying Ada programs employing tasking [MOR87a, MOR87b, MOR88]. A GRASP/GT specification may be viewed in two ways: as a graphical GRASP/G document utilizing both architectural and procedural graphical representations, or as a textual GRASP/T document which outlines the specification in a PDL-like listing. The GRASP/G diagrams for architectural specification are derived from the data flow diagrams promoted by Yourdon [YOU78], and Gane and Sarson [GAN79]. The GRASP/G diagrams for procedural specification are based on the Warnier-Orr diagrams established by J. D. Warnier [WRN74, WRN81] and modified by others [ORR77, BRN84]. GRASP/GT currently runs on the Apple Macintosh, although the GRASP/T translator is portable and is currently available for the DEC VAX and IBM PC.

2.2. Design Methods for Ada

Three categories of design method are presented in this section: (1) data flow-oriented design, (2) data structure-oriented design, and (3) object-oriented design. Each category has its particular area of emphasis in what Pressman [PRE87] calls the "information domain" and also in the type of design (i.e. architectural as opposed to procedural) each undertakes. Several of the design methods discussed herein are also parts of larger life-cycle methods which encompass complementary requirements analysis methods. The following is a brief discussion of several design methods in the three categories along with a comparison of the three general approaches with respect to suitability for Ada-based software. Pressman [PRE87] provides a comprehensive overview of several of the design methods presented.

2.2.1. Data Flow-Oriented Design

Data flow-oriented design was developed through the efforts of Yourdon, Constantine, [YOU75], DeMarco [DEM79], and others [STE74, MYE78, YOU78, GAN82] and is based on analysis of system data flow characteristics, aided by the inclusion of the data flow diagram.

The data flow-oriented design espoused by Yourdon, Constantine, and DeMarco, called *Structured Design*, is primarily an architectural method, converting data flow specifications into structure charts. *Structured Design* offers no unique tools for procedural design, although DeMarco [DEM79] does present a pseudocode-like notation for process specification in the analysis stage. The construction of the structure chart is accomplished by partitioning the data flow diagram and applying a mapping procedure to each of the partitions. The partitioning is accomplished by analysis of the characteristics of the overall data flow. Two types of flow are recognized. In *transform flow*, the overall data flow follows a pattern of large input flow into a general transform area producing large output

flow. In contrast, data flow may exhibit characteristics of a *transaction*, where one particular data item determines the flow path subsequently followed. The topology of the architectural structure differs according to the type of flow exhibited by the data flow diagram. It is possible to have both types of flow in different areas of the same diagram.

In areas where transform flow is dominant, the mapping of such areas to an architectural specification begins by defining the input flow areas, the general transformation areas, and the output flow areas. For each of these areas, a control process subordinate to a system controller is added to the structure chart. Subsequently, the processes within the areas are added to their respective structure chart branches as modules. As a rule, input and output processes closer to the transform area boundary have control over those processes further from the transform.

In an area dominated by a transaction, however, the partitioning of the data flow diagram is based on different criteria. Instead of a transform center, the heart of the partition is the *transaction center*, a single process from which the different flow path alternatives emanate. Also identified is the flow path through which the discriminating data item arrives at the transaction center. The resulting structure chart has a branch corresponding to the arrival path and also a *dispatch* branch, the latter controlling the branches for each of the alternative paths. Note that the alternative paths and the arrival path will have to be analyzed and structured individually as they will have distinctive flow characteristics of their own.

The derivation of the complete system structure chart is followed by its refinement to improve the strength of the modules comprising the chart. This refinement is the work of the human designer and is based more on experience and intuition than on any mechanical algorithm. Following refinement, each module in the final structure chart can be specified using any number of detailed design techniques.

2.2.2. Data Structure-Oriented Design

Data structure-oriented design is based on the premise that the composition of software is directly related to the structure of the data with which the software is concerned. Presented are two development methods with design techniques based on this premise: Jackson System Development and Data Structured Systems Development.

Jackson System Development (JSD) is a method concerned with the modeling of real-world situations. It is a comprehensive method, covering the life cycle from requirements analysis to implementation. Jackson [JAC83] divides the method into two phases: *specification* and *implementation*. In JSD, there is no definitive design phase; instead, design issues (especially pragmatic issues such as processor allotment and data base construction) are handled in the implementation phase. Much of the JSD specification phase, however, resembles the typical design phase as it determines a logical architecture for processes and also a pseudocode-like description of the processes.

Cameron [CAM86] provides an overview of JSD specification. System specification begins with the identification of relevant entities and the actions that may befall them. From this set, a series of *model processes* are derived. Each model process is a description of an entity in terms of the actions that befall it and the order in which such actions occur; in other words, a description of the life cycle of a particular entity. The model processes are depicted with *Jackson diagrams*, tree-like structures having added notation to represent selection between alternate branches or repetition of a branch, as well as for sequencing among sibling branches.

The set of model processes constitutes the heart of the system specification. In order to communicate with the real world, utility processes for such tasks as input and output must be developed and linked with the model processes. Cameron [CAM86] describes the development of a JSD specification as being "middle-out", that is, starting with the model processes (which do not communcate with each other), adding the utility processes to the periphery, and linking with the model to produce a network specification.

The linkage can occur in one of two modes: *data stream communication* and *state vector inspection*. A data stream connection consists of a conceptually boundless queue of messages from one process to another. A state vector is simply the collection of variables local to a model process which relate the state of the modeled entity. This vector may be examined (but not altered) by the utility processes. The final result of this phase of specification is a series of independent processes (more precisely, process types) connected via data stream queues or state vectors. Each process may be elaborated by its Jackson diagram.

From the network specification, a structure chart may be derived. Cameron [CAM86] describes a "knitting needle" technique for developing such a chart. In a network specification, data streams can be directly connected to the outside environment in order to supply utility processes with needed input. The technique involves conceptually threading a needle through such data streams with the resulting topology representing the architecture of the system (the needle itself may be considered the main process). Allowances may be made for loops within the system and for timing requirements which call for buffering.

Another method in this category is one developed by Orr [ORR81], based on the work of Warnier [WRN74, WRN81]. This method, known as *D*ata *S*tructured *S*ystems *D*evelopment (*DSSD*), is premised on the concept of "output-oriented" design; in other words, the system should be developed solely on the basis of the required outputs. Like JSD, DSSD encompasses requirements analysis as well as design. Like JSD, DSSD also defines the functions and begins procedural studies of those functions in requirements analysis. Many of the notations used in DSSD are based on the Warnier diagram (see [WRN74, WRN81]) and its successor, the Warnier/Orr diagram [ORR81].

DSSD begins requirements analysis with definition of the *application context* which defines the scope of the system in relation to the real-world environment in which it will operate. The application context is determined through the use of *entity diagrams* which show information flow among the relevant players in an organization. From these diagrams, the entities comprising the actual system are determined, and in this way the

domain of the system is bounded. System *objectives* are determined by examining and ordering the data flow that crosses the newly defined system boundary. The ordering of objectives is more fully defined through the use of the *assembly-line diagram*, a notation based on the Warnier/Orr diagram altered to show distinct threads of data flow. From this basis, an analysis of the procedural specifications of each of the functions defined from the assembly-line diagram is conducted using the Warnier/Orr notation. After functional requirements have been determined, the *application results*, or the outputs which justify the system, are examined in detail. Eventually, this study will produce Warnier/Orr representations of the system outputs; these representations will be the input to the design phase of DSSD.

The objective of the design phase, according to Hansen (see [HAN83]) is to produce a *logical process structure* from the Warnier/Orr representations of outputs, otherwise known as *logical output structure*. The mapping from the LOS to the LPS is usually quite direct.

2.2.3. Object-Oriented Design

Object-oriented design is a design philosophy which has been seriously studied only in the past few years. Booch [BOO86] provides an overview of the fundamental concepts in this relatively new area. The most fundamental, of course, is that of the *object*, which is simply a software manifestation of some real-world entity. A software system designed in light of this philosophy will consist of several such objects, corresponding to the actual objects in the problem domain. With each object is associated a group of *operations* , or *methods*, which are performed on the object. In addition, software objects have *attributes*, which serve as modifiers (adjectives) for the objects. As a real world object can be a member of a larger grouping which has attributes common to each member, so also can a software object be a member of a *class* and *inherit* the attributes and operations from the more general class of which it is a member.

An object (or object class) can be viewed from two perspectives. First, an object has a *implementation* which contains the details of the object and its operations and yet shields such details from the object's users, and (2) a *specification*, or the interface used by other processes to invoke the operations provided by the object (and to create objects belonging to the class). Note that in true object-oriented design the operations provided by objects define the extent of what may be done with the object. Since the detailed structure of the object is unknown to outside processes, such processes cannot exploit the object's internal data structure in any way other than that allowed by the given operations. It is this characteristic of object-oriented design which makes the objects, and the systems to which they belong, more amenable to change.

An early OOD method was devised by Booch [BOO83] based on a technique proposed by Abbott [ABB83]. The basis for any OOD method is the identification of relevant objects and operations from software requirements documentation. In this method, objects, operations, and attributes are identified from an English description of the proposed solution plan, known as an *informal strategy*. Next, the designer associates each of the operations to exactly one object, based on which object's internal structure was required for the operation to proceed. Dependency among the objects then is established; object A depends on object B if object A uses any of the data types or operations in object B's interface. The resulting overall dependency relationship constitutes the architectural view of the system. The dependencies among the objects plus the interfaces of the objects can be demonstrated using graphical notations specially created for OOD. Once the system structure is established, the implementation details of each object and its operations are defined. If the detailed design of any particular object reveals an underlying system of constituent objects, then the entire method can be applied recursively to the solution description for that object.

Pressman [PRE87] illustrates another method, developed by Cox and others [COX86], which utilizes the OOD principles of class and inheritance, whereas Booch's early method did not. In this method, object classes inherit operations and attributes, called

instance variables,"from their more general ancestors. In addition, the more specific classes have the ability to provide operations and attributes unique to them and even to override operations and attributes inherited from the ancestors.

Another method, devised by Seidewitz and others at Goddard Space Flight Center [SEI87], attempts to address some of the perceived inadequacies of Booch's early method. The major disadvantage of the method seen by these researchers is that it did not offer any special design notation for larger software systems. To alleviate this need, Seidewitz and his team drew from previous work [RAJ85] to develop two hierarchical representations for object-based software systems. The *parent-child* hierarchy (called *composition hierarchy* in a later article [SEI88]) relates how an object can be composed of subordinate objects which are unknown outside the domain of the encompassing object (this structure was indeed recognized in [BOO83] although it was not explicitly named). The *seniority hierarchy*, on the other hand, configures the system as layers of *virtual machines* [DIJ68] consisting of objects; the objects of each virtual machine layer may invoke the resources of objects within their layers or from subordinate layers. This hierarchy differs from the parent-child hierarchy in that subordinate objects may be known and directly exploited by multiple superiors.

The development scheme of this method starts with a data flow diagram and the identification of a *central entity* and support entities in a process known as *abstraction analysis* [STA86]. Seidewitz and Stark [SEI87] adopt this approach in lieu of Booch and Abbott's informal strategy. From this is devised a static diagram showing the entities and the known control relationships between them. The entities and relationships in this diagram are then translated into objects and dependencies, and virtual machine layers are more firmly established. Later developments [SEI88] have included the analysis of a complete entity-relationship diagram, and have dubbed the method *GOOD*, for *G*eneral *O*bject-*O*riented *D*esign.

In his later article, Booch [BOO86] adapts his method somewhat and appears to address some of the inadequacies noted in [SEI87]. These alterations are duly noted in

[SEI88]. Instead of using an English description of the problem, Booch, like Seidewitz and Stark, derives the objects and operations from a data flow diagram, although not in the exact manner as Seidewitz and Stark. Booch [BOO86] now describes the operations associated with each object as being "suffered by" the object; the objects which would invoke those operations are now spoken of as "requir[ing]" said operations. Objects are now classified as (1) *actors*, objects which do not offer any operations and hence do not suffer operations, (2) *servers*, objects which do not invoke the operations of other objects but simply suffer invocation from others, and (3) *agents*, objects which both suffer operations of their own and inflict invocations on other objects. With the information about the relationships between operations and objects, dependency among the objects is determined. Provisions are made for a layering approach with the addition of a notation for *subsystems*, corresponding to the virtual machines of Dijkstra, Seidewitz, and Stark.

2.2.4. Applicability to Ada

A task confronting project managers is choosing a design approach suitable for the applications which they must oversee. Increasingly, applications have grown in sheer magnitude and complexity; hence, the desire and the need to control complexity is growing ever more acute. In addition, concurrency in the application domain is now seen as a quality to exploit directly rather than simply to simulate or even to avoid. Ada was created to serve these ends.

Presumably, all of the above methods can be applied to Ada-based software as the language provides all of the necessary constructs for each of the approaches to succeed. However, Ada provides a number of unique constructs which render object-oriented approaches even more applicable. The *package* construct is the basis for objects in most of the OOD methods discussed. [BOO83, BOO86] [SEI87, SEI88] The package *specification* parallels the object specification in that it provides necessary data types and invocation mechanisms for operations. The package *body* contains the details of the operations and

the types and shields that information from the package user. The use of *private types* aids in the achievement of information hiding in that it allows hiding of the details of the type and prevents illicit exploitation of those details. The *task* construct facilitates the construction of concurrent systems and also can represent actor objects as described by Booch [BOO86]. Inheritance is somewhat more difficult to establish in Ada, although Booch [BOO86] suggests some means of accomplishing this. The parent-child hierarchy and the seniority hierarchy can be implemented via the *separate* clause and the *with/use* context clauses, respectively [RAJ85].

2.3. Graphical Representations for Algorithms

Up to this point, GR's have been addressed in conjunction with the specification languages and methodologies which they support. These diagrams are for the most part at the system and architectural levels. Block diagrams, data flow diagrams, and structure charts fall into one or more of these categories. A discussion of GR's of software would not be complete without a review of those notations specifically intended to represent algorithms. In Section 2.3.1, many specific GRA's are cited. In Section 2.3.2, the literature survey of empirical studies of GRA's is summarized.

2.3.1. Specific Notations

Since the ANSI flowchart was introduced in the mid-50's, numerous notations have been proposed and utilized. Several authors have published notable books and papers that address the details of many of these [MAR85b, TRI88]. Tripp, for example, describes 18 distinct notations that have been introduced since 1977. Figure 4 contains a chronological list of traditional as well as lesser known notations. In general, these diagrams have been strongly influenced by structured programming and thus contain

Diagram Name	Contributor/Date
Flowchart	Von Neumann (mid-50's)
Doran Chart	Doran and Tate (1972)
Warnier-Orr	Warnier (1974), Orr (1977)
Dimensional Flowchart	Witty (1977)
Lindsey Chart	Lindsey (1977)
Flowblocks	Grouse (1977)
Ferstl Chart	Ferstl (1978)
Schematic Logic	Jensen and Tonies (1979)
SPDM Diagram	Marca (1979)
UPC Diagram	Harel, Norvig, Rood, To (1979)
Compact Chart	Hanata and Satoch (1980)
GREENPRINT	Belady, Evangelististi, Power (1980)
SSD Diagram	Kanada and Sugimoto (1980)
Schematic Pseudocode	Robillard (1981)
Problem Analysis Diagram (PAD)	Futamura, Kawai, Horikoshi, Tsutsumi (1981)
Rothon Diagrams	Brown (1983)
Structure Chart	Chyou (1984)
Action Diagrams	Martin & McClure (1985)
FPL	Taylor, Cunnift, Uchiyama (1986)
Control Structure Diagram (CSD)	Cross (1986)
Box Chart	Johnson (1987)
FP Diagrams	Pagan (1987)

Figure 4. Graphical Notations for Program Design.

control constructs for sequence, selection, and iteration. In addition, several contain an explicit EXIT structure [LIN77, FER78, JEN79, MAR85b, CRO88a, CRO88b] as well as a parallel control construct [LIN77, FER78, HAR79, MAR85b, CRO88a, CRO88b]. However, none of the diagrams cited above explicitly contain all of the control constructs found in Ada.

Several diagrams were found to be particularly relevant to the GRASP/Ada project, including the Nassi-Shneiderman diagram, the Warnier-Orr diagram, the action diagram, the schematic pseudocode diagram, and the control structure diagram. These diagrams are functionally similar in that they each have constructs for sequence, selection, and iteration. However, the symbols or icons and the spatial arrangement used for these individual constructs are distinct. Each of these diagrams is illustrated in Figure 5 and briefly described below.

The Nassi-Shneiderman diagram [NAS73] was developed as an alternative to the flowchart. The control structures in Nassi-Shneiderman diagrams are represented using detailed boxes that fully delimit the scope of the structure. Control enters the structures from the top of the box and leaves at the bottom. Nested structures are realized by nesting the appropriate construct boxes. A completed Nassi-Shneiderman diagram consists of a labelled box containing nested boxes. Nassi-Shneiderman diagrams are very clear and simple to follow, although they can be difficult to draw and edit manually.

Warnier diagrams [WRN74, WRN81] use a simple symbology consisting of braces, pseudocode, and logic symbols, and are employed to analyze systems in a top-down fashion. The diagrams are easy to read and understand, even by laymen, which is convenient when communicating with end users. The most important property of the diagrams is that they show information in a hierarchical structure while preserving information from level to level. Any given level is a complete synthesis of all its sublevels, and all of the sublevels belonging to a given level comprise a complete analysis of that level. In fact, each level in the diagram may be thought of as a set, and each sublevel may be thought of as a subset. Orr [ORR77] has taken some of Warnier's concepts and

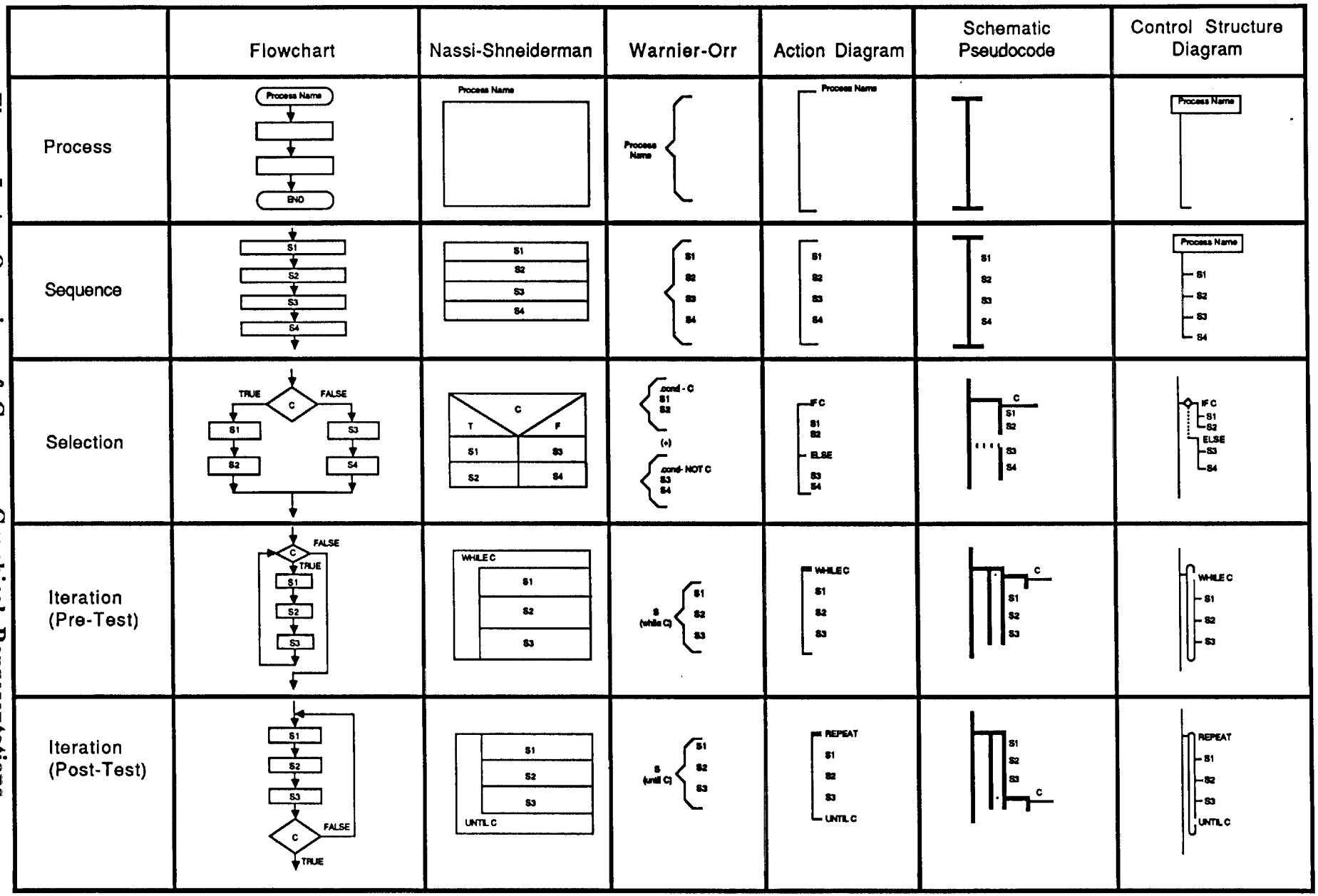


Figure 5. An Overview of Common Graphical Representations.

integrated them with other concepts taken from sources such as HIPO. The resultant diagrams are commonly known as Warnier-Orr diagrams.

The action diagram [MAR85] is a graphical representation that can be considered as a graphical overlay to source code. It consists of a series of structures, most resembling a detailed bracket, that are drawn to the left side of the source code in the space generally unused because of tabbing and indentation. The action diagram is simple to draw and edit, and shows structure nesting well. However, it can be difficult in a heavily nested diagram to tell what structures are nested, as the details which differentiate most action diagram constructs are generally confined to the top and bottom of the bracket.

Robillard [ROB86] has identified two existing problems with conventional source code documentation. First, source code is not generally documented systematically, but is often done rather haphazardly after the coding. Since the documentation is not an integral part of the language itself, it tends to vary widely from practitioner to practitioner, as each programmer generally has his own documentation style. Second, documentation is often done in a bottom-up style as the programmer scans through modules adding comments here and there. Robillard's Schematic Pseudocode (SPC) is a graphical representation for documentation which purports to solve both of these problems. It resembles an action diagram in that it uses lines and brackets on the left side of source code to represent control flow. An interesting aspect of SPC is that it may be represented by an LL(1) grammar. Because of this, Robillard was able to construct a software environment (SCHEMACODE) for editing SPC diagrams and for automatically compiling SPC documents into code.

The control structure diagram (CSD) [CRO86] was designed to improve the readability of algorithms by highlighting their control structure. In addition, the CSD attempts to clearly depict the individual control paths defined by the constructs. And, as was the case with the action diagram, the CSD can be conceptually drawn or overlayed onto source code and thus may be considered a graphical extension of it. The CSD is more fully described in Section 3.3.

2.3.2. Empirical Studies

Designing and automating graphical notations is an important research area in computer-aided software engineering. A critical but often overlooked component of this process is that of empirical evaluation of these notations. One of the major purposes of GRA's is to increase comprehension efficiency (less time, fewer errors). Thus, while a GRA may be intuitively preferred on the basis of increased comprehension, it should be evaluated formally to determine any actual increases and their significance. This section begins with a summary of the literature on general program comprehension and concludes with a brief discussion of empirical studies that dealt with GRA's.

There were numerous articles that dealt with programmer behavior and general program comprehension. Although these articles do not address the subject of GRA comprehension, they are important because they indirectly support the use of GRA's. Three articles (BAS86, BRK80, CUR86) emphasize some important points about evaluating programmer behavior in empirical research. All recognize that programmer behavior is a relatively new but important topic. But evaluating programmer behavior is akin to evaluation of any other kind of human behavior and requires strict adherence to the methodologies followed by the psychological and educational realm of human behavioral observation. These three articles offer suggestions for attaining these goals.

The most comprehensive body of programmer comprehension theory is summarized in an article by Brooks (BRK83). His theories are supported by a number of empirical studies. Brooks explains that comprehension is a top-down approach involving the recognition of "beacons", or key parts in the programming language (WIE86). This recognition, along with the programmer's expectations (SOL84), leads to the formation of hypotheses about the function of the program (GUG86). The programmer validates or changes the hypotheses in an iterative process of spotting more beacons and formulating inquiries about the program's activity (LET86). The ability to recognize beacons and formulate hypotheses depends on programmer experience and knowledge: novices possess

underdeveloped skills in these areas.

Five other articles support these theories with studies on the effects of program structure on comprehension. Four support the use of meaningful comments, indentation, and white space to show structure of the program (MAP86, SHE79, SHN77a, SHN76). A fifth (SIM73) went further by concluding that some constructs used in programming are more comprehensible than others; the study found that the nestable IF-THEN-ELSE was more easily understood by nonprogrammers than the simple JUMP-to-a-label statement.

All these studies support visual chunking or blocking of related parts of the algorithm, indentation, and meaningful comments, characteristics which are prominent features in most GRAs.

What makes a good diagrammatic notation is the key point in an article by Fitter and Green (FIT79). A picture is worth a thousand words, but the best picture will have the following: relevance, restriction, redundant recoding, revelation and responsiveness, and revisability. They close by commenting that it is important for the computer engineering community to support the behavioral sciences in their research to find the most suitable GRA.

An attempt was made early in the research effort to find empirical research on the four oldest GRAs: the Nassi-Shneiderman diagram chart, the flowchart, the action diagram, and the Warnier-Orr diagram. Unfortunately, the only research that could be found related to the oldest GRA, the flowchart.

There were four studies which did not support the flowchart for use in programming applications. All four tested subjects using the flowchart in various programming tasks: either program creation, debugging, modification, or general comprehension. All four concluded that the flowchart was no better than the source listing (SHN82a, SHN7b, BRO80a) and that it may even inhibit understanding in some cases (MAY75).

Nine articles supported the flowchart over a listing or PDL. It is noteworthy that only two of the nine (CUN87, SHE81) tested subjects in use of the flowchart in

programming tasks. Of the remaining seven, two (SCA87, SCA88) were preference surveys (the flowchart was more preferred than pseudocode for a number of tasks), and were not empirical comprehension studies. The remaining five studies (WRI73, BLA73, BLA74, KAM75, KRO83) tested the use of the flowchart in nonprogramming applications, such as the use of a correct flowchart in procedural problem-solving tasks (such as finding your way out of a maze). One could conclude that perhaps the utility of the flowchart depends on the manner in which it is being used.

This conjecture is substantiated by two more studies (BRO80b, GIL84). The first found that, when subjects searched for a bug in a linear fashion, the flowchart was better than the listing, but if the search was nonlinear, as so many programmer tasks are, use of the listing was more accurate in spotting the bug. The second article found that utility of the flowchart depends on the nature of the task and the strategy the programmer uses to employ these tasks. Thus, the issue of the utility of the flowchart is not a clear-cut one. In addition, much more research is needed on the other GRAs.

Two articles looked at the use of the data-flow diagram, a graphical representation of program architecture commonly used in software engineering applications. One showed the advantage of using data-flow diagrams in library use over other standard library methods (CAR86). The second (NOS86) was an empirical study demonstrating the comprehensibility of the task-oriented downward cascading menu representation over the DFD.

The last group of articles related to software engineering methods and tools. The first (YAU86) surveys various techniques used to design software. They describe the stages for software development and the methods for validating and verifying the correctness of software. Shneiderman (SHN82b) discusses ways for making a system more amenable to human use. The last two articles (RAE85, BRW85) discusses automated software development tools which utilize various graphical representations such as PDL, flowcharts, and Nassi-Shneiderman diagrams.

2.4. Graphical Representations for Architecture

Data flow diagrams are used to produce a graphical representation of a system. The diagrams resemble connected graphs with elaborated nodes. The nodes of the graph represent processes or data stores, and the links between nodes represented data flowing between them. Currently, there are two popular varieties of data flow diagram: the Yourdon data flow diagram [YOU78] and the Gane and Sarson data flow diagram [GAN79]. Each uses a slightly different symbology to achieve the same end results. Although the Yourdon data flow diagram appears to be the more common variety, the Gane and Sarson rendition is more finely developed and is better suited for automation. Batini, Nardelli, and Tamassia [BAT86] have developed algorithms for the automatic layout of these data flow diagrams.

The structure chart is a graphical representation of a system's architecture that exhibits the various modules within the system and their invocation hierarchy. Modules are represented by small boxes which contain the name of the module. Invocation of one module by another is denoted by an undirected line from the calling module to the called module. The driver module is commonly placed at the top of the diagram, and modules which are invoked by the driver are arranged horizontally below it.

Two types of data items are normally found on a structure diagram: control flow information and data flow information. Control flow items are data items passed from one module to another that affect the flow of control within the called module. These items are represented on the structure chart with an arrow that has a hollow circle at the end (the name of the control item is normally placed near the shaft of the arrow). The control arrow is placed near the invocation line that connects the boxes representing the two modules. Data flow items represent all other data items that may be passed from one module to another. These items are similarly represented using an arrow that has a filled circle at the end.

For complex systems, structure charts may grow rather unwieldy in size. To combat this problem, structure charts are generally layered, so that individual pieces of the

structure chart may each fit on a normal typed page. For systems which pass a large number of parameters from module to module, the names of the parameters are often replaced with a reference number, and the chart is accompanied by a table which shows the list of parameters associated with each reference number.

2.5. Reverse Engineering

Acly [ACL88] defines reverse engineering as "an emerging term used to describe a procedure and a set of tools which make it easier to maintain and update old application code. Reverse engineering extracts the specifications from existing systems and translates these specifications into the more abstract specifications used for design and analysis." He points out that users typically need more help with maintenance programming than with software design and development: in his opinion, automated reverse engineering is the "missing link" that can bridge the gap between this maintenance of "old", existing code and development of "new" code.

Acly lists several benefits and drawbacks to the process of reverse engineering. Some of the benefits he lists follow:

• Existing code for large systems can find new life if they can be deciphered via reverse engineering tools.

• Specifications and documentation will be up-to-date and will match the actual program.

• Maintenance (both corrective and perfective) can be performed at a higher level by modifying the specifications, rather than the source code.

Some of the drawbacks Acly mentioned include the following:

• The reverse engineering process cannot be fully automated. Some human interaction will always be required to extract a meaningful specification from existing source code written by conventional means.

• There must be some way to prevent bad programs from being converted into bad specifications. Poorly written programs should be modified or restructured in order to produce an accurate and meaningful specification.

With these thoughts in mind, let us consider some case studies in reverse engineering.

Blaze and Cameron [BLZ88] have created D*, an automatic documentation system for IC* programs. IC* is a project under way at Bell Communications Research that will attempt to provide an environment for designing and developing complex systems for networking and communications. Two languages (C&E and L.0) are used to implement IC* systems, and D* is the automatic documentation tool for these languages.

D* programs depict the program variables and the relationships between them. The documentation takes the form of a grid of boxes, with each box representing a program variable. Lines are drawn between the boxes to denote relationships between the variables. Information hiding is supported: groups of boxes can be tucked inside a "parent" box as an abstract representation.

Blaze and Cameron believe that the D* system produces good documentation quickly, which leads them to believe that custom documentation systems for other languages are feasible. They have proposed the possibility of using D* documentation for other, more conventional languages.

Fukunaga of the Science Institute of IBM Japan, Ltd. [FUK85], has created PROMPTER: a system for annotating programs written in assembly language. He considered using a rule-based approach, which means that knowledge rules detailing how specifications are to be extracted from the source code must be supplied. However, he found that it was difficult to isolate the different kinds of knowledge needed to produce meaningful annotations. He therefore pursued an object-based approach to program annotation.

PROMPTER considers registers, data storage, and program instructions to be "objects" which may be manipulated by "messages." The system consists of four parts: (a)

a symbolic simulator which looks at each instruction and simulates it to determine the data transfer needed; (b) an abstraction part, which extracts a conceptual meaning from an assembly instruction; (c) a high level annotator, which combines the low level concepts determined by the abstraction part and creates more high level annotations for the program; and (d) a controller, which passes control back and forth among the other components as needed.

Fukunaga feels that PROMPTER is quite successful for providing low-level annotation of existing programs written in assembly code. He plans to do further experimentation with providing higher-level automated program documentation.

CARE (Computer-Aided Reverse Engineering) is a project being carried out by Wagner [WAG88] that will investigate the possibility of maintaining and redesigning programs using a set of tools interacting with a data dictionary. The first CARE prototype is designed for the development of COBOL systems, and the tools available will include a parser for deconstructing programs and storing them in the CARE data dictionary, a restructuring tool that replaces unstructured program constructs with more easily maintainable structured versions, an architectural viewing tool that allows the modular hiearachy of the system to be studied, a query system, and a tool for software tracing. CARE will eventually support a graphical design language known as GRAPES which may be used to develop software.

Harada and Sakashita [HAR83] have developed HIDOC, another tool for providing graphical representations of COBOL programs. HIDOC automatically produces four distinct types of program documentation. The HIDOC Process Chart is a quick reference that shows the environment the target program deals with, including any external entities and files. The Hierarchy Chart is a standard structure chart which shows program modules and their interrelationships. The Data Chart graphically presents data structures and file record formats. Finally, the Source Listing Cross Reference annotates the program with a simple graphic akin to the action diagram that allows logic flow within the program to be traced more easily.

VIC (Visual Interactive C) is an environment for supporting the maintenance and development of C programs. Designed by Rajlich et. al. [RAJ88], VIC allows a program to be represented in two forms: in its normal form as code, or in a visual, iconic form. The visual form allows C programs to be seen as an entity-relationship graph (ER-graph). Because VIC contains four distinct groups of operations that allow conversion of code to and from ER-graphs, it becomes useful as a reverse engineering tool for providing a graphical representation of large C programs.

Fischer et. al. [FIS87] have developed WLISP, a system which contains object-oriented tools for building and reusing user interfaces. The interfaces from existing programs developed using WLISP may be modified and used as a starting point for developing interfaces for new applications.

TOMALOGIC is a reverse engineering tool developed by Lerner [LER88] that constructs system matrices from program code. A program is decomposed into nodes, small program chunks that have one entrance and one well-defined exit. TOMALOGIC then builds a matrix of these nodes, showing the possible transitions from node to another. A primary application of TOMALOGIC is the decomposition of "spaghetti" code so that it may be structured.

Grau and Gilroy [GRA87] have examined the feasibility of mapping Ada programs into the DOD-STD-2167 documentation structure. DOD-STD-2167 is a software development standard created by the Department of Defense that "defines a consistent design structure for system and software development projects." Grau and Gilroy examined the Ada language to determine the entities which compose an Ada program, and then looked for corresponding elements in the DOD-STD-2167 structure. After considering several approaches, they determined that a "simple, compliant one-to-one mapping of all Ada programs to DOD-STD-2167 does not exist." However, they do feel that the related, many-to-one mapping rule is sufficient for mapping Ada programs into DOD-STD-2167.

2.6. *Conclusions*

The major findings of the literature survey that are considered most relevant to GRASP/Ada have been collected and summarized below.

• Automated programming is feasible, especially for limited domains. Several authors have demonstrated this for particular domains [BAL85, BRS85, CRA86, HAM79, MOR88, SIE85, URB85].

• Specification languages are a promising method for realizing automated code generation. There is considerable interest in the use of specification languages for achieving automated code generation: indeed, this appears to be the preferred method [ABR80, BAL85, BEI84, CRA86, GUT85, HEV88, LUC85, MOR88, SIE85, URB85, ZAV86].

• Specification languages should be sufficiently rigorous to promote correctness. This is a general consensus among several authors [ABR80, ALF77, BAL85, BEI84, GUT85, JON80, KEM85, LUC85, MAR85a, MEY85, MOR88, SIE85].

• Specification languages should utilize graphics where possible, and specifically those graphics commonly used in software engineering. Many recent software engineering development environments make use of graphics for specifying programs [ART88, CRA86, HAM79, HEV88, MOR88, ROB86, SIE85].

• Specification languages should be accompanied by an integrated environment suitable for use throughout the entire life cycle. This may be an influence due to Ada, because Ada programmers are strongly suggested to use an APSE (Ada Programmers Support Environment). Several specification languages have accompanying support environments [CRA86, HAM79, MOR88, SIE85].

• Object-oriented design appears to be the leading design method candidate for designing systems in Ada. Data flow oriented methods are currently being used as "front end" methods for object-oriented design.

• Graphical representations are useful in understanding programs. This has been suggested in studies done by Gilmore and Smith, and by Brooke and Duncan [BRO80b, GIL84]. The empirical study which composes part of the GRASP/Ada project is expected to confirm this hypothesis.

• An important part of reverse engineering is the understanding of existing programs, especially through the use of graphics. Many systems which attempt to reverse engineer existing programs produce graphical documentation as a result [BLZ88, HAR83, RAJ88, WAG88].

3.0 Baseline Requirements

In this section, the requirements for the Phase I GRASP/Ada CSD generator prototype are presented. First, the goals and objectives for the generator are briefly discussed. This is followed by a statement of general requirements for the CSD generator prototype and justification for several of the tradeoffs encountered during requirements analysis. Finally, the CSD constructs for Ada are introduced.

3.1 Introduction

The overall structure of the GRASP/Ada project may be seen in the data flow diagrams (DFD's) presented in Figures 6 and 7. In these diagrams, the major functions to be provided by the system, the data upon which they operate, and the data which are produced may be seen in context. The system DFD in Figure 6 shows the manner in which the user interacts with the system. The user provides Ada source code to the GRASP/Ada system which in turn parses the code. The primary output of the Phase I prototype is the procedural GR (CSD).

Figure 7 shows the second level of the GRASP/Ada system. The first process, P1, is the scanner/parser which takes the Ada source code and breaks it down into syntactical and semantic units. As the program is parsed, certain information is passed to the other processes, specifically P2 which produces procedural GRAs, and P3 which records program information. P2 looks at the Ada control constructs and prints the appropriate CSD representation for those constructs.

Although the immediate result of the implementation of the CSD generator will be a tool to aid the programmer in code comprehension and maintenance, there is another, more far-reaching goal in mind. Foremost in the GRASP/Ada project is the promise of gaining

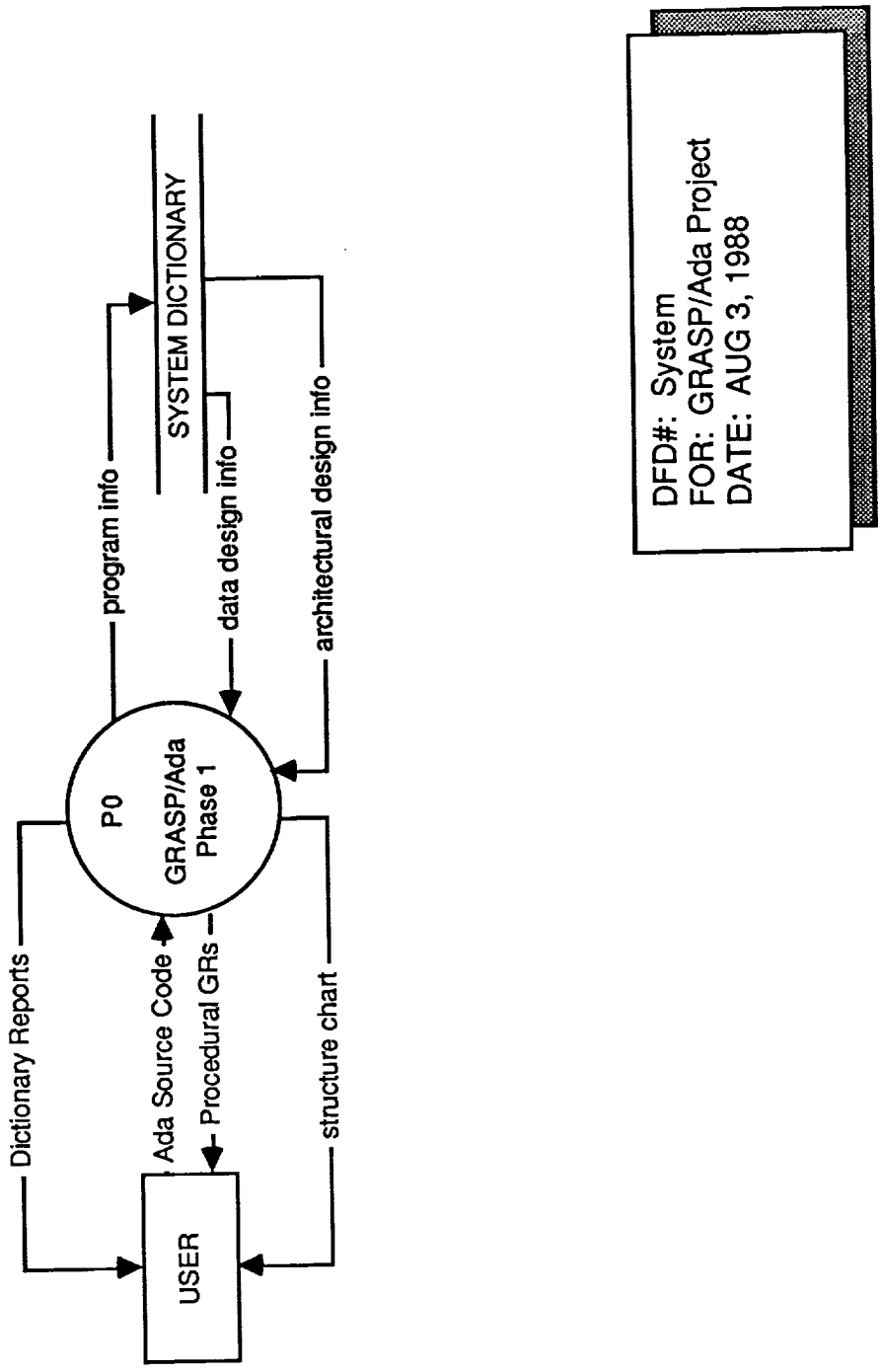


Figure 6. The System Data Flow Diagram (DFD) for GRASP/Ada (Phase I).

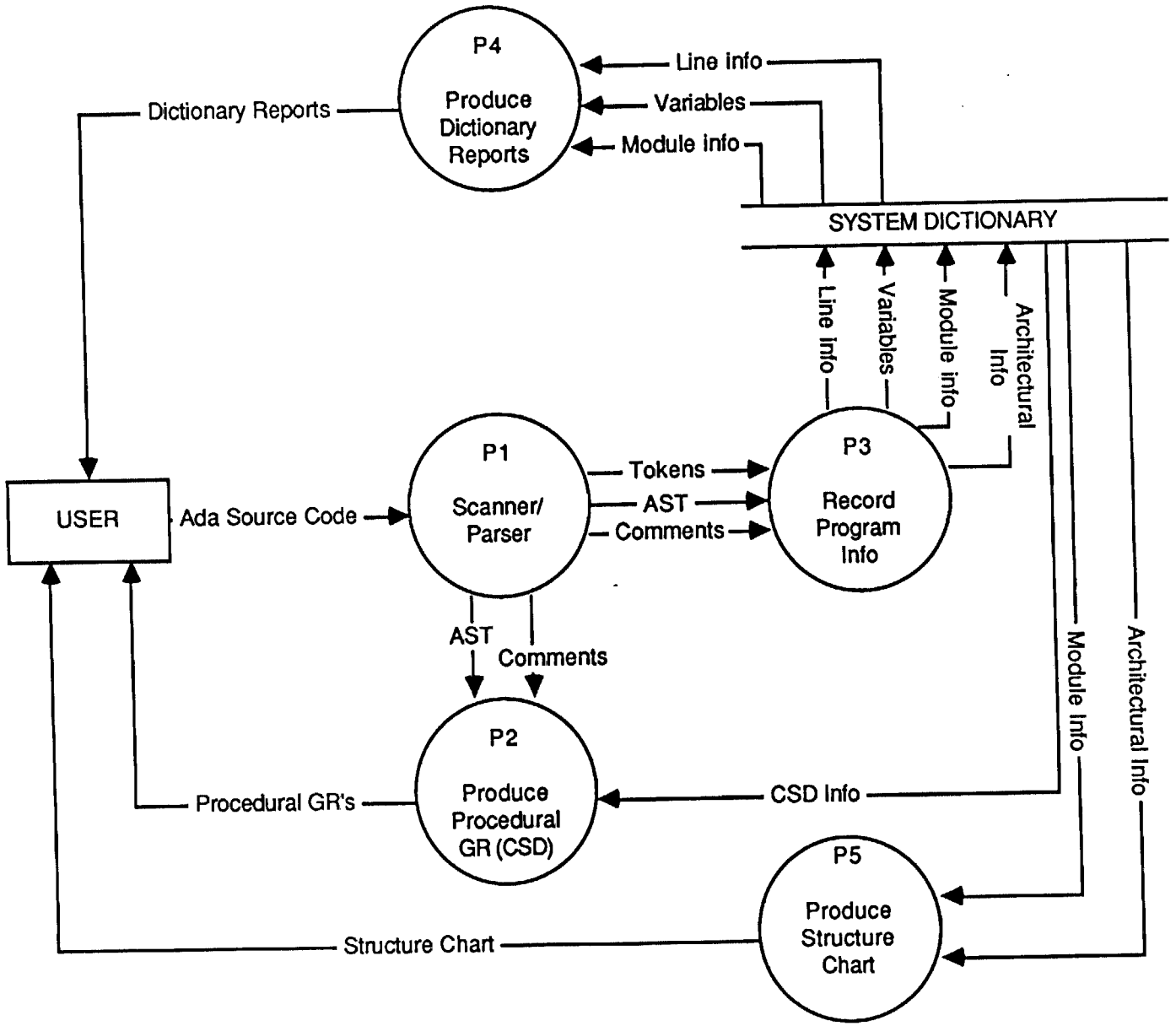


Figure 7. First Breakdown of the GRASP/Ada System (Phase I).

insight into the problem of automatic code generation. Most research efforts in this area have tackled the problem by designing systems from high-level design specifications and attempting to have automated tools generate correct implementations. Unfortunately, it is difficult to know exactly what should be provided in a good design specification. For that reason, the GRASP/Ada project is approaching the problem from the reverse side. By giving programmers incrementally more and more abstract tools to display code and its meaning, the tools can be examined to determine their effectiveness at each step. Observing the effect of graphical representations on "real" source code instead of trivial examples should prove most beneficial in the design of tools that adequately abstract meaning from implementation. In the GRASP/Ada project, procedural tools (the CSD) will be introduced first and applied to "real" code. Next, architectural tools (the structure chart and the DFD) will be devised and examined. It is hoped that by working up from a ground level, practical and useful tools will evolve: once these tools have been proven, the problem of automatically generating code from them can be addressed.

The primary contributions that were gathered from this particular phase of the project (Phase I) are twofold. First was the evolution of an improved CSD with constructs for concurrency and other features specifically adapted for Ada. Second was the creation of a a tool for automatically documenting existing Ada programs with the CSD. This tool may be used to examine the effectiveness of the CSD in practical situations.

3.2 Prototype Requirements

In this section, the requirements for the CSD generator prototype are discussed. Section 3.2.1 reviews several potential implementation environments, Section 3.2.2 examines the various tools that were considered for use in developing the scanner and parser for the prototype, and Section 3.2.3 briefly discusses the detailed requirements for the CSD generator.

3.2.1 Environment Requirements

Many of the requirements for the CSD generator prototype have been purposefully generalized in order to fit a variety of implementation environments. For implementation of a prototype, the VAX 11/780 computing environment was selected. Since the CSD generator produces graphics, and since graphics are one of the least transportable features among different computing environments, the selection of an environment was a key issue. It is often easier to rewrite applications for specific computers than to attempt transporting the graphics routines. For example, the Apple Macintosh treats text and graphics equally, whereas the IBM PC has separate text and graphics modes. Larger computers such as the DEC VAX use graphics packages such as Regis and CURSES. Each of these environments assumes different hardware and software configurations, as well.

The VAX 11/780 was considered the best available environment for implementation of the prototype for several reasons. It was readily available, and all project members had ready access to a terminal. It has C, Pascal, and Ada compilers installed, so the prototype could be coded and tested without any up/downloading. The VAX is connected to Auburn University's MicroVAX, which has compiler development tools such as LEX and YACC. Thus, the VAX was identified to be the most attractive development environment presently available for producing the CSD generator prototype.

3.2.2 Scanner/Parser Requirements and Tradeoffs

During the fall quarter, the implementation of the scanner and parser for Ada was initiated. During the early portion of this research contract there was much discussion about the method through which these programs would be created. Several alternatives were available and are discussed briefly below.

Manual coding. The first possibility was to code the scanner and parser manually, but this was quickly discarded for several reasons. The Ada grammar, compared to other third generation languages such as Pascal and C, is an extremely large grammar.

Constructing parse tables for even these less complex languages is a very difficult task, and errors could take an exorbitant amount of time to detect and remove. Given the size and complexity of Ada, this approach was abandoned in favor of more reliable automated methods of scanner/parser generation.

The next possibility was to find a suitable grammatical description of the Ada language and use it as input for an automated scanner/parser generator tool. Scanner/parser generator tools are available commercially as well as through university research programs, and several of these were considered. Two such tools, developed here at Auburn University, were considered for use in the GRASP/Ada project.

CODE$IT. The first, designed by Dr. Mel Phillips of Auburn, was CODE$IT, an LL parser generator for the IBM 360/370 that generates PL/I scanners and parsers. CODE$IT has been used in the compiler construction course at Auburn University for several quarters and has been proven to be fairly reliable. However, the use of CODE$IT for the research was rejected for several reasons. The primary factor in the decision was that CODE$IT produces parsers and scanners written in PL/I, a language that is either not available or not suitable for use on the target machines being considered.

LL parser generator. The second locally available tool was an unnamed LL parser generator for the Apple Macintosh written by Greg Whitfield, a master's student at Auburn University. Its use was discounted for similar reasons as CODE$IT. It is a relatively unproven tool in that Mr. Whitfield has tested it with grammars as large as 100 productions, but Ada has roughly four times that many. In addition, the tool requires large amounts of memory in order to run properly. Mr. Whitfield reports that small grammars of approximately 100 productions require about one megabyte of internal memory in order to produce a parser/scanner. It was expected that producing a parser/scanner for Ada would require a minimum of two megabytes of memory, and probably more. The Macintoshes available at Auburn all have one megabyte of available memory, although one of the research assistants has access to a Macintosh that has two megabytes of installed memory. The possibility of modifying the parser generator to run in less memory was briefly considered, but this would have required too much time for what was a fairly unimportant part of the

research effort.

LALR 3.0. The first tool that showed promise was the LALR(1) parser generator, LALR 3.0 from LALR Research. It runs on the IBM PC, and creates parsers in C (although the documentation claims that the program may be changed to produce parsers in other languages, such as Pascal or Ada). Auburn University purchased a copy of this program, and it was utilized on the GRASP/Ada research project until several problems were encountered. First, we examined the Ada grammar provided with the LALR tool and found that it was not an accurate Ada grammar. We found at least one instance of a perfectly legal Ada program that the LALR grammar would not accept. Second, because LALR 3.0 runs on the IBM PC and development of the CSD generator is proceeding on the VAX 11/780, the up/downloading effort was getting tiresome. Keeping all of the development tools on the same computing system soon became a high priority for the GRASP/Ada project. Third, although the LALR tool came with an Ada grammar, it did not come with a lexical specification for Ada. This meant that a lexical analyzer for Ada would need to be specially written for the project. The LALR tool did come with an example scanner for the C programming language, but the scanner utilized many "tricks" in order to be more efficient in the scanning of C programs. Adapting this scanner for Ada would mean restructuring and rewriting the program substantially, and after some experimentation, this approach was abandoned.

LEX/YACC. LEX, a lexical analyzer generator, and YACC, an LR parser generator originated at Bell Labs and are now in the public domain and are highly regarded in the literature. Although these tools would have been excellent for use in the research project, we encountered some problems that discouraged this approach. However, we did select "improved" versions of LEX/YACC.

FLEX/BISON. We found an updated lexical analyzer generator (FLEX) from a software library at Purdue University (J.CC.PURDUE). Next, we received a public domain version of YACC, called BISON, from a software repository at MIT (PREP.AI.MIT). BISON runs on the IBM PC and can handle larger grammars than the version of YACC on Auburn's MicroVAX. Finally, we located an accurate LALR(1) grammar and lexical

description for Ada from a public domain exchange for Ada software at SIMTEL-20.ARMY.

With the proper tools in hand, generating the scanner and parser for Ada was relatively straightforward. The scanner/parser was tested on several Ada programs written by a doctoral student at Auburn (Mr. Wenkai Chung).

Because all of the compiler generator tools produce C code, including FLEX and BISON, it was decided to do all of the prototype coding in C. To choose another language would have meant translating the scanner and parser each time they were altered, which would have added up to a prohibitively large coding effort. The only other language considered was Pascal, because the Macintosh graphics routines use a Pascal interface. Had the Macintosh been chosen for development, Pascal may have proved to be a better choice of language.

3.2.3 CSD Generator Detailed Requirements

The CSD may be perceived as "graphical prettyprinting." Therefore, the next step after producing a scanner and parser was to produce a prettyprinter for Ada source code. Tentative requirements were developed for this tool after considering the following possibilities. A software switch for placing Ada keywords in boldface is desirable, as well as a software switch for automatically converting Ada keywords to lower case, as per the Ada standard. Double and triple spacing certain portions of Ada code was determined to be necessary in order to represent properly certain CSD constructs such as the procedure header and package header.

Once the CSD constructs have been formalized, grammatical representations will be written for each. These grammatical representations could be embedded in the Ada grammar itself, so that the CSD for Ada actually becomes a superset of the Ada language. This embedding of graphical documentation in the grammar rules of a language is fairly new: the only notable examples include SchemaCode [ROB86] and GRASP/GT [MOR87a, MOR87b, MOR88].

One of the problems involved with creating grammatical representations for CSD

constructs was that of determining how far to break up the graphical constructs. It is possible that the CSD symbols might be useful in reconstructing the parse stack at any point in a program given only the CSD symbols for that line in the program. Such a feature could provide a basis for the design of an interactive CSD workstation environment. Instead of having to reparse a program from the first line, only the line being edited would need to be read in order to regenerate the parse stack and ensure the correctness of the modified construct.

3.3 CSD Constructs for Ada

In this section, the new constructs created to map the CSD to Ada are presented. It will be noted that most of these constructs were introduced to handle the problem of representing Ada tasking.

The control and tasking constructs of Ada were examined and tentative CSD representations for each of these constructs were created and then iteratively refined. The Ada constructs have been divided into three groups. Group I (see Figure 8) consists of the basic CSD constructs that are found in almost every third generation programming language: procedures, packages (modules), sequences, selections, cases, for loops, and while loops. Group II (see Figure 9) contains the control constructs that are specific to the Ada programming language, including: infinite loops, loop exits, blocks, blocks with declarations, go to statements, exception handlers, and exception raises. Finally, Group III (see Figure 10) has all of the Ada constructs related to tasking and parallel processing: task specifications, rendezvouses (both calls and receives), select statements, guarded and unguarded alternatives within select statements, aborts, and terminations.

The following criteria were considered as these constructs were refined: readability, consistency with other graphical representations, consistency with other CSD representations, and ease of implementation. With respect to implementation, several assumptions were made (or not made). First, it could not be assumed for what type of environment the CSD would eventually be implemented. Although newer generation

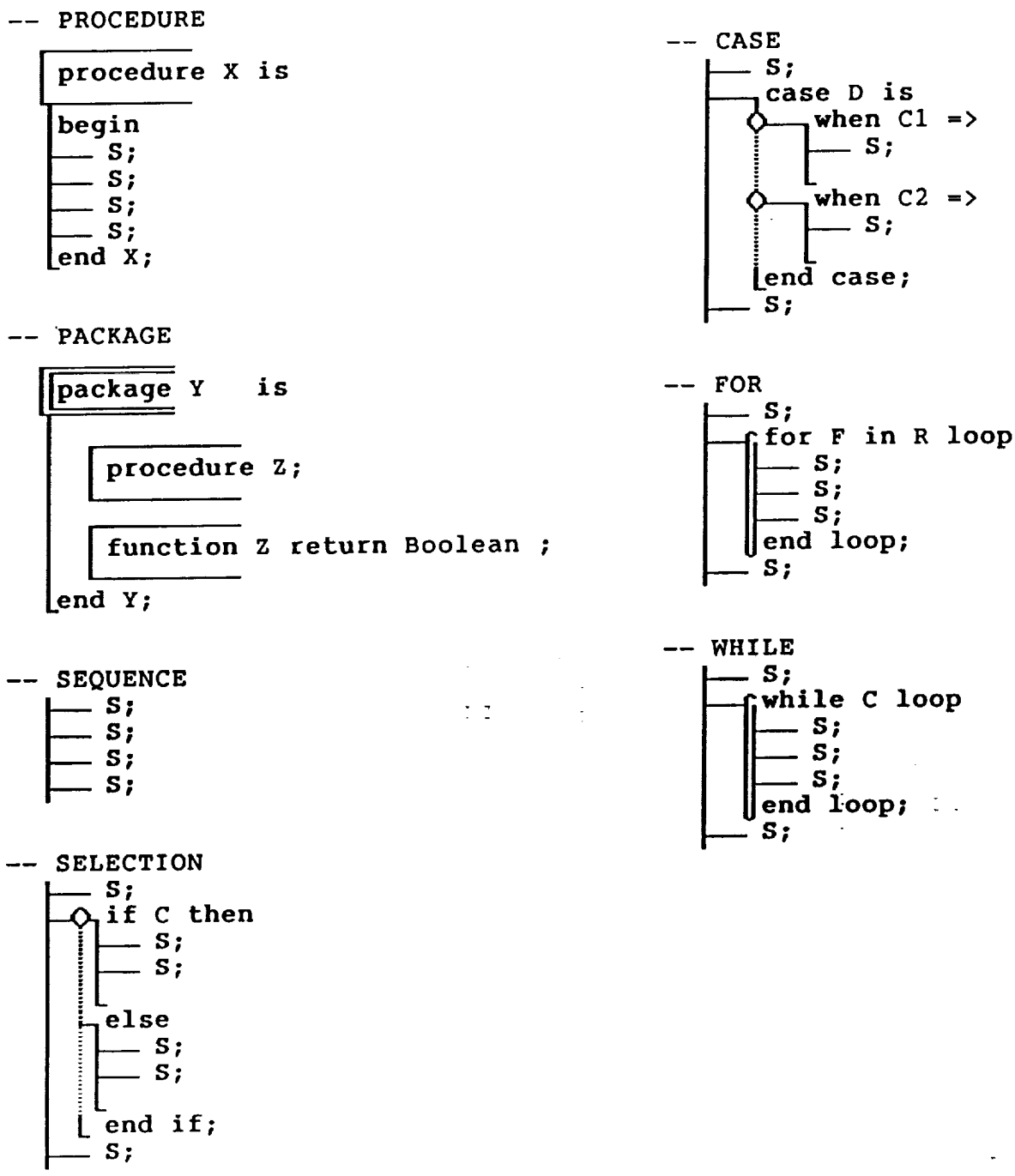


Figure 8. Group I CSD Constructs for Ada.

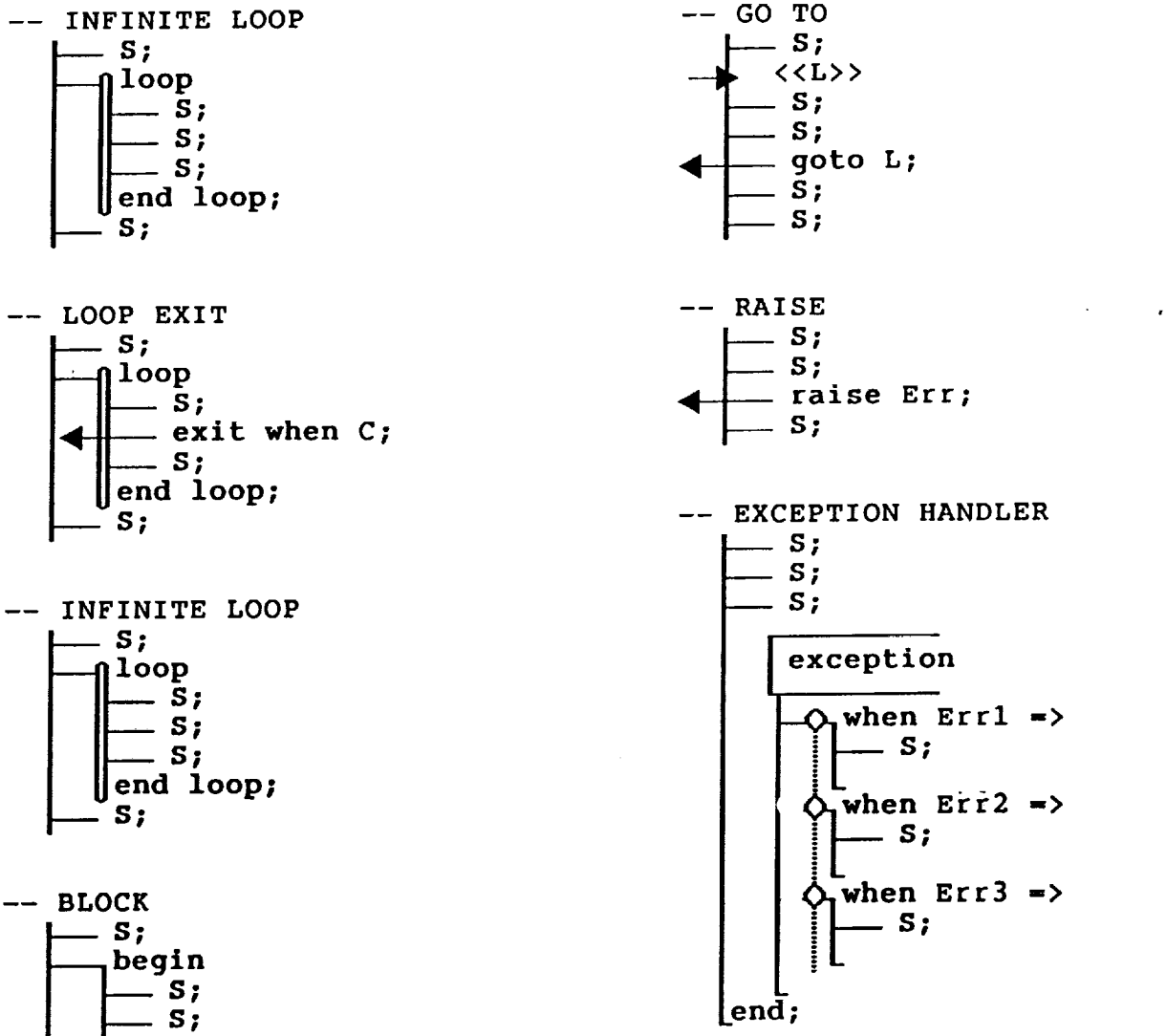


Figure 9. Group II CSD Constructs for Ada.

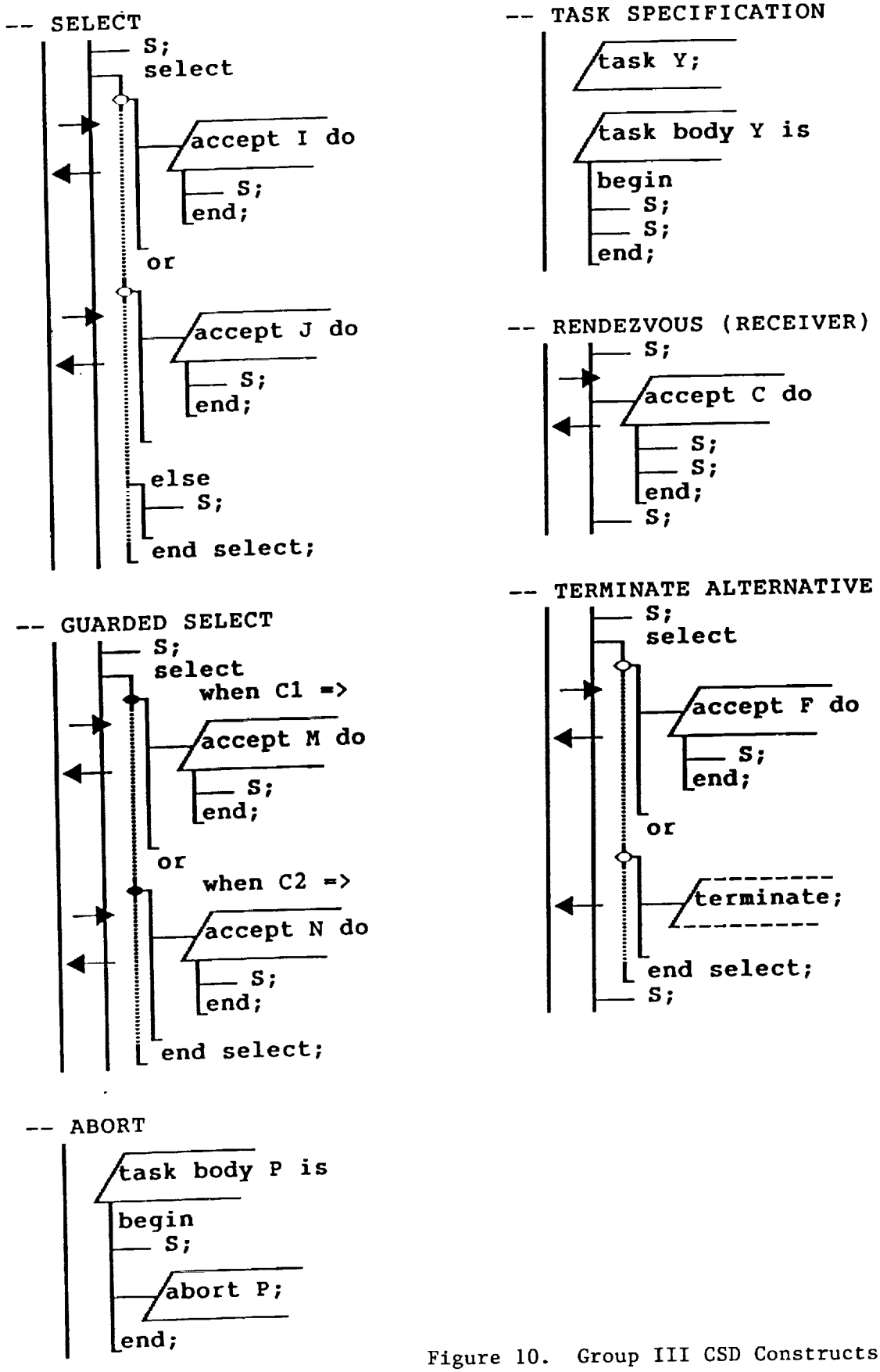


Figure 10. Group III CSD Constructs for Ada.

computing environments such as the Apple Macintosh and NEXT workstations provide graphical routines and hardcopy devices that would support almost any graphical representation that could be conceived, implementing the CSD should be feasible on older systems such as VAX 11/780s and IBM PC's. For this reason, the CSD constructs were simplified as much as possible so that they can be constructed with the "graphical characters" specifically designed for the CSD. The CSD generator prototype was targeted for the VAX 11/780 environment, since it was readily available, has good software in the form of the DEC C and Ada compilers, and has appropriate hardware such as an LN101 laser printer and VT220 graphics terminals. Earlier in the research effort, microcomputer environments such as the Macintosh II or IBM PS/2 were also considered for the prototype, but none were readily available.

3.4 Empirical Evaluation

An important element of this research is determining the utility of graphical representations of software. In particular, an empirical evaluation of graphical representations for algorithms was defined and conducted in parallel with the other tasks associated with the GRASP/Ada project. Since the population of available subjects had limited, if any, Ada experience, it was decided to conduct the experiment based on simple control constructs found in Pascal. In this section, the objectives of the study, and a brief overview of the procedure, the implementation plan are presented. Appendix A contains a detailed description of the experiment and an analysis of the results.

3.4.1 Objectives of Empirical Evaluation

There were four basic objectives and associated tasks in the empirical study. The first was to determine which, if any, of the conventional flowchart, the control structure diagram, or pseudocode is most easily comprehended and useful as a debugging aid. Two measures were observed: efficiency and accuracy. These notations were tested using subjects

from three experience categories: novice (course(s) in Pascal or PL/I), intermediate (courses in data structures, algorithms, and software engineering) and experienced programmers (seniors in a computer science or engineering curriculum).

The second objective was to determine if there are differences among the three experience groups in levels of comprehension of a particular notation. Novice programmers have not yet attained the necessary skills needed to modify or debug efficiently a program. It would be valuable to determine the difference in the error rate between the novice, experienced and expert programmer groups and note the difference across all notational formats. Perhaps one representation is more easily comprehended by experienced and expert programmers but may be a hindrance to novices.

The next objective was to find the preferred diagrammatical notation between the novice, experienced, and professional groups, via a preference survey. It is valuable to determine which notation is the most preferred, since this notation would be the one most readily used by programmers. Preference will be measured in terms of the task for which the notation is to be used, as subjects may prefer to use one notation for one purpose, but another for a different purpose.

Finally, the preference data and the accuracy/efficiency data were compared.

3.4.2 Overview of Procedure

Each subject from a sample of students and professional programmers was given a brief automated summary on the use of one of the proposed notations, followed by an automated test (approximately 50 minutes). This test contained three algorithms, each representing three difficulty levels: easy, moderate, and difficult. Each test represented the three algorithms in one of the following formats: the conventional flowchart, the control structure diagram, or pseudocode. Thus, a given subject would observe the three algorithms in one graphical format. Each algorithm had a number of bugs seeded in it; the subject was required to determine the exact nature and location of the bugs. The subject was also asked questions about flow of control. The questions were multiple-choice with five candidate

responses each. Each question/response was timed. Following the comprehension test, there was a short preference survey. After a brief explanation of all the notations and how they show sequence, selection, and iteration, the subject was asked to select and rate which of the notations he or she would use in a number of programming situations. Subsequently, statistical tests were conducted on all the data to determine the significance of the results.

3.4.3 Implementation Plan

The following five tasks were defined to accomplish the procedures described above.

1. Develop the instrument on paper.

For each experience group (novice, experienced or expert), the tests given were the same. Each subject observed three algorithms (easy, moderate, and difficult) represented by one graphical format. Within each format group, the order in which the algorithms were presented was randomized, for six possible combinations. All algorithms contained constructs for sequence, selection, and iteration. The easy algorithm contained simple variables, the moderate algorithm dealt with arrays and simple records, and the difficult algorithm consisted of pointers and more complex records. Examples of valid input data (but not output) will be provided.

A preference test was also created. This survey asked subjects to rate each of the three notations on a scale of 1 to 5, where 1 is not useful and 5 is very useful, given a hypothetical task for which the notations are to be used. This survey was included at the end of each comprehension test.

2. Develop the automated instruments.

Once all algorithms had been determined and the questions about them developed, they were transferred onto an IBM PC-compatible computer. Then the user interface was produced. This interface first took the user through a discussion of the notation and the way

it represents selection, iteration, and sequence. When the subject was ready to begin the test, the computer gave instructions about the content of the test and how the questions were to be answered. The user went through each algorithm, answering questions about where each bug was located and about general control flow. The interface allowed the subject to answer only one question at a time; once answered, the subject could not go back to a previous question. The user was allowed to view the algorithm as long as required. Each question was timed. As an answer was given, the computer stored the question, the response, and the elapsed time on a record that was made for each subject. When the user was finished with the test, the interface displayed a "sign-off" message and prepared for the next subject, who was given a test with the algorithms in a different order and in a different graphical format.

3. Testing and evaluation of the instruments.

Using a small sample of students the instruments were tested to make sure there were no problems and that they tested what was intended. The presentation of the test was also checked.

Once this small sample was taken, it was statistically analyzed. Based upon the variance obtained, the sample size needed for the entire experiment was determined.

4. Implementation of the instruments and collection of the data.

After the instruments were developed and tested, they were implemented at Auburn University.

5. Statistics and evaluations.

Once all the data was collected and stored, the IBM 360/370 version of SAS was used to analyze and determine the results. A more detailed explanation of the statistics and an analysis of the collected data are contained in Appendix A.

4.0 Prototype Design and Implementation

In this section, the design and implementation of the prototype GRASP/Ada CSD generator are discussed. First, the major software components which comprise the CSD generator are outlined. This is followed by in-depth examinations of each component.

4.1. Introduction

The heart of the GRASP/Ada CSD generator very much resembles the structure of an Ada compiler. A parser and scanner for the Ada source language were constructed using an Ada grammar and the software tools FLEX and BISON. But, where a compiler normally includes action routines that transform high-level grammar constructs into machine code, GRASP/Ada includes action routines that construct and manipulate the CSD prefixes that begin each line of source code. The implementation of these routines was inspired by the concept of object-oriented design. A *prefix* data object which manipulates lower-level CSD graphics characters was designed, with accompanying functions which create, manipulate, and print the CSD. These functions are called by the parser and scanner when appropriate as the Ada source code is read from the input file. The prefix and Ada code are then combined, prettyprinted and buffered for output.

A user-friendly and relatively transportable interface was designed to drive the GRASP/Ada CSD generator. The interface provides the user with the capability to specify options for the CSD generator quickly without having to learn cumbersome command languages and option formats. The user has the freedom to choose from a variety of line spacings and font styles. All options are visible onscreen and can be selected and modified using only the terminal cursor keys and the RETURN key.

For previewing the CSD before submitting the GRASP/Ada output to a laser printer, the GRASP/Ada system allows the user to transfer to a specially modified version of

Digital's EVE editor. This editor allows the user not only to view the CSD onscreen, but to manipulate it spatially in selected ways. Options are provided for temporarily cloaking the CSD so that the user is presented a traditional textual view of the Ada source code, then redisplaying the CSD to assist the user in interpreting the code. A future option still under development will allow the user to "collapse" portions of the Ada program so that the overall program structure and control flow can be more readily seen.

Finally, the construction of the GRASP/Ada system necessitated the prior implementation of several software tools for creating fonts and "massaging" Ada code to render it more suitable for the CSD generator. These tools were designed so that they could be quickly reconfigured to generate software components for versions of GRASP/Ada that run on computer systems other than the VAX 11/780.

In the following sections, each of these software components (parser/scanner, prettyprinting routines, user interface, CSD viewer, and software tools) will be examined in more detail. Where appropriate, the tradeoffs and design decisions which dictated the implementation of each component are discussed, and each component is analyzed in terms of adaptability and transportability to new hardware configurations. A separate *CSD Generator User Manual* has been drafted which contains additional details and examples.

4.2. Parser/Scanner

As described earlier, the CSD generator is largely grammar-based. The backbone of the generator is a combination of a FLEX-generated scanner and a BISON-generated LALR(1) parser. The lexical specification and the grammar input to these tools have been supplemented with calls to routines to effect the graphical or nongraphical prettyprinting of the code as it is parsed. These routines will be explained in more detail in the next section.

The placement of prettyprinting routine calls in the middle of grammar rules has been an important part of the project and also the source of much grief. Normally, a semantic action associated with a grammar rule is placed at the end of the rule; when all the right-hand symbols of a grammar rule have been recognized and when the value of the lookahead token

determines that a reduction is in order, the rule is reduced, the right-hand symbols on the stack are replaced with the left-hand symbol, and any block of actions at the end of the rule is executed. However, when an action is placed in the middle of a rule, BISON is forced to substitute a self-generated nonterminal for the action block and then produce a new rule whose right side consists solely of the action block. This simulates the end-rule scenario.

A problem has appeared with this extensive use of mid-rule actions, however. Conflicts of both the shift/reduce and reduce/reduce variety have occurred when actions have been placed inappropriately. At times, the conflicts have required slight alterations of the original grammar.

Another problem involves the use of the scanner for much of the work of prettyprinting. At first, the work of filling the buffer containing prettyprinted Ada code (the buffer will be explained in the next section) was performed by the scanner; for example, when the scanner recognized a certain keyword, the scanner placed that keyword into the buffer and then returned the corresponding token to the parser. However, at times the parser would call on the scanner to produce a lookahead token in order to determine its next action. This token would not be shifted onto the stack right away; in many cases, the token would not even be part of the construct under consideration. Even so, the prettyprinting routine called for on recognition of that token would be executed, often placing the lexeme on the line just previous to its proper position. This problem was solved for keywords, literals, and identifiers by adding the appropriate actions directly to the grammar instead of the lexical specification, although the problem still exists in part with regard to whitespace and comments.

4.3. Graphical Prettyprinting Routines

The architecture of the CSD generator tool with respect to the graphical prettyprinting routines is shown in Figures 11 and 12. Figure 11 is an adaptation of component objects in the system. The colored boxes represent object classes; listed inside these boxes are the operations associated with each object. Although the collection of

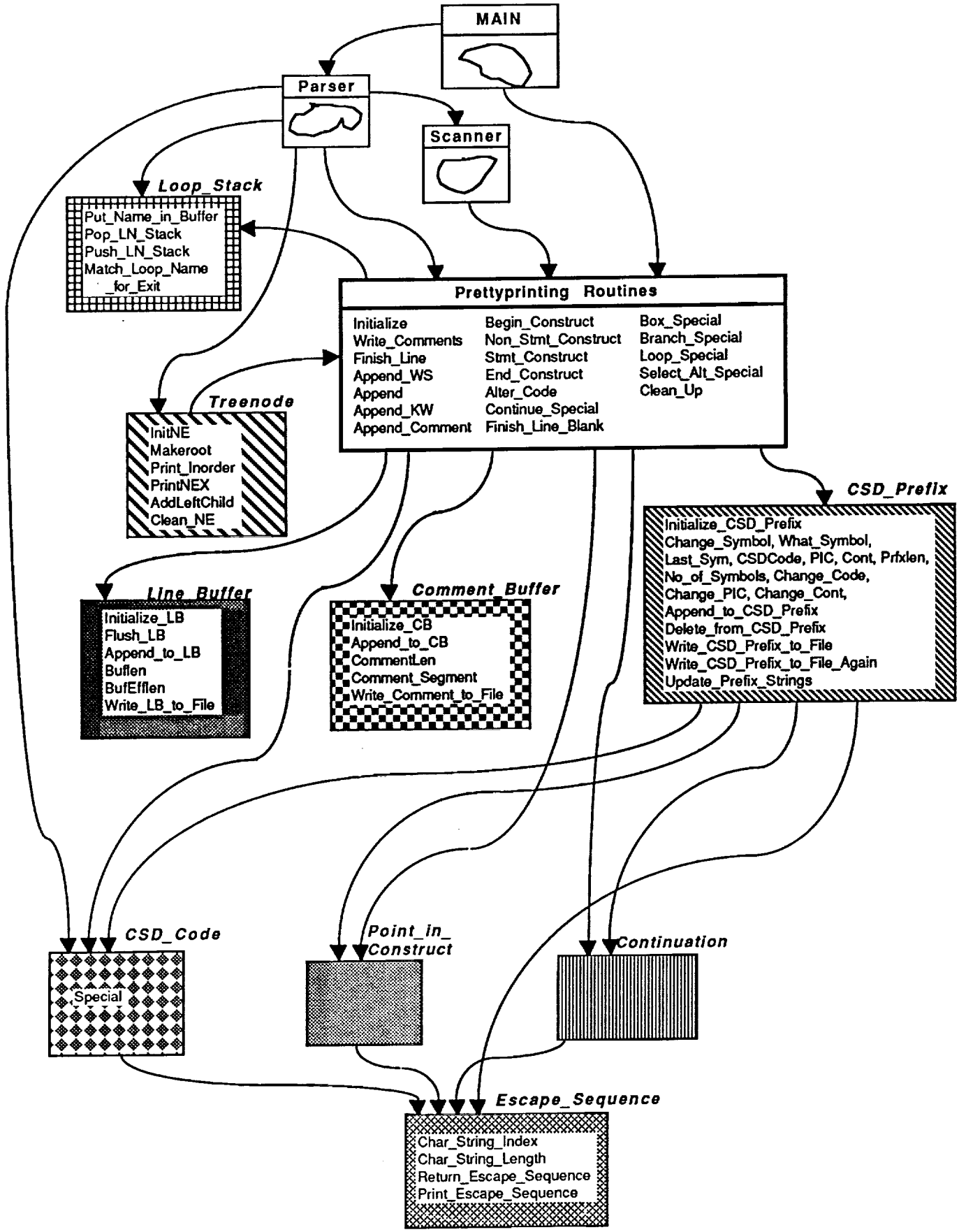


Figure 11. GRASP/Ada Component Objects.

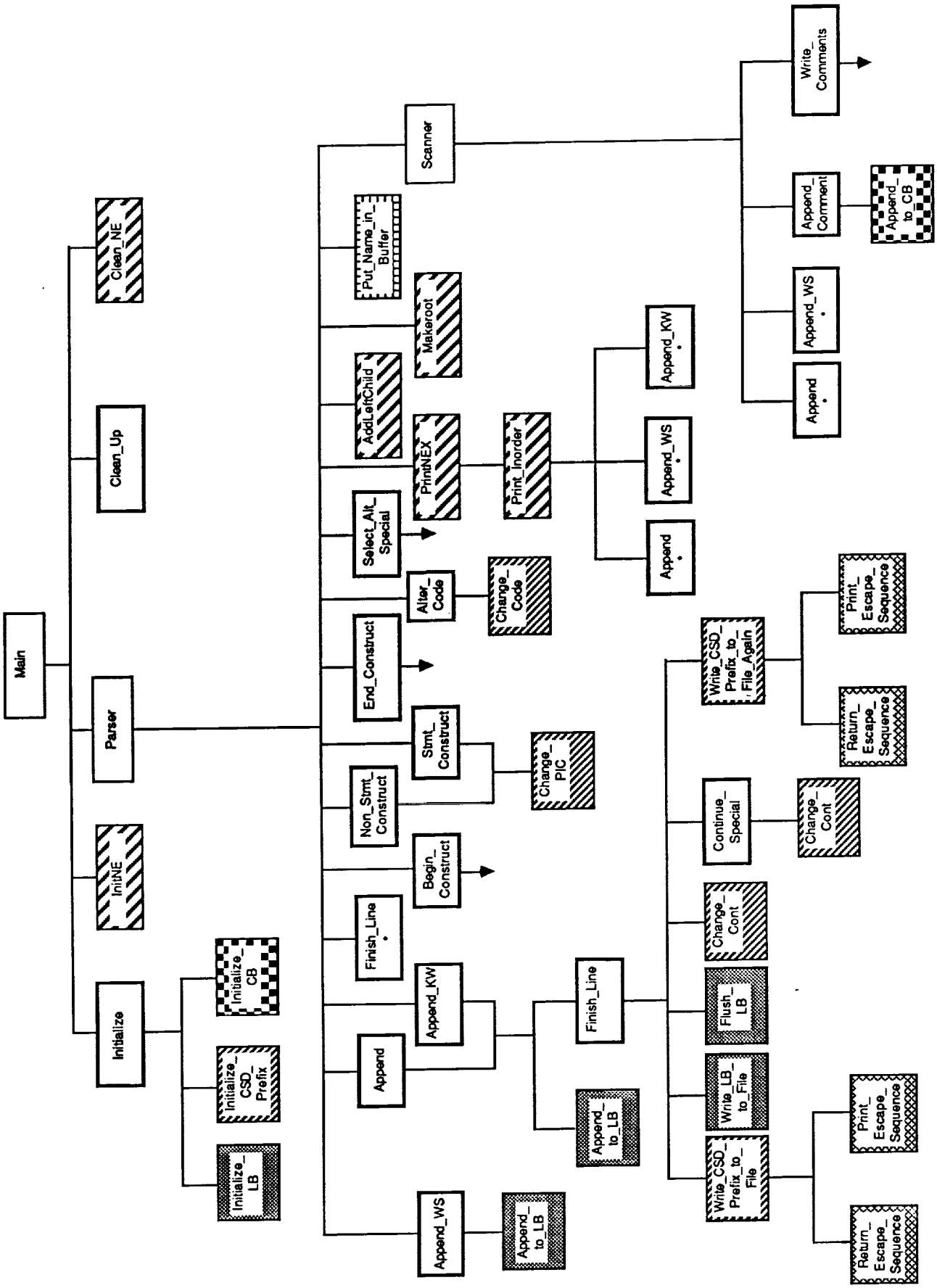


Figure 12(a). GRASP/Ada Hierarchical Structure.

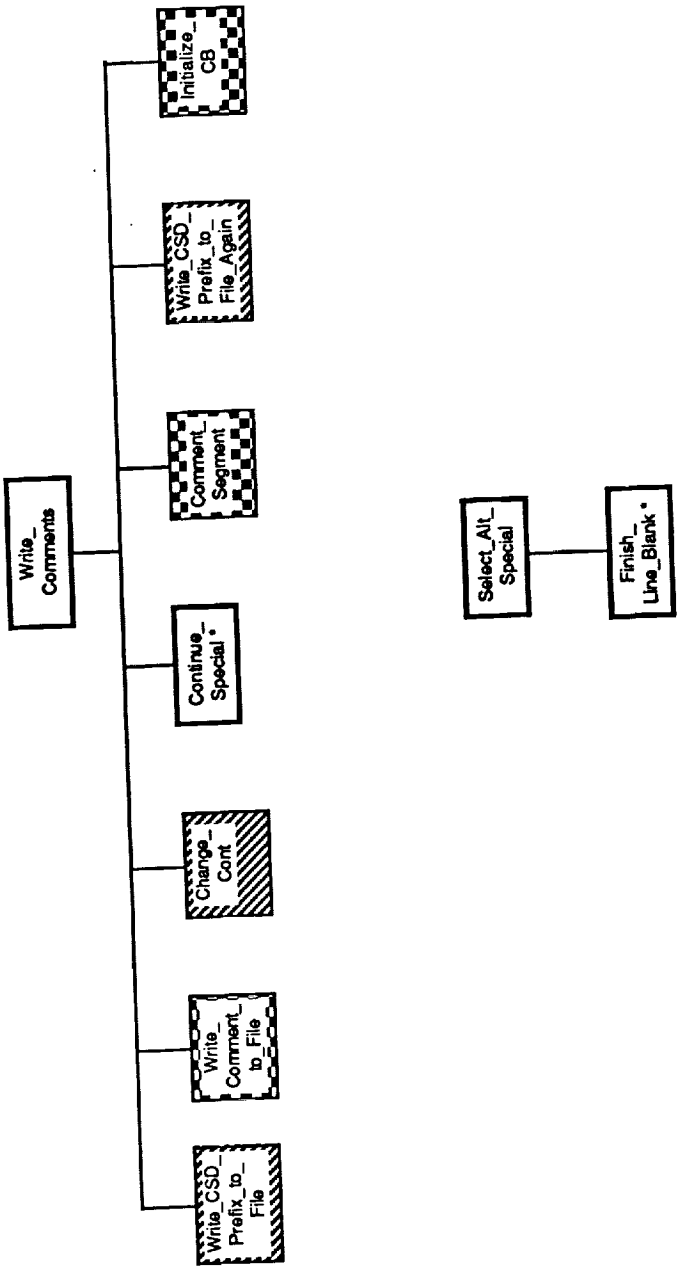


Figure 12(b). GRASP/Ada Hierarchical Structure.

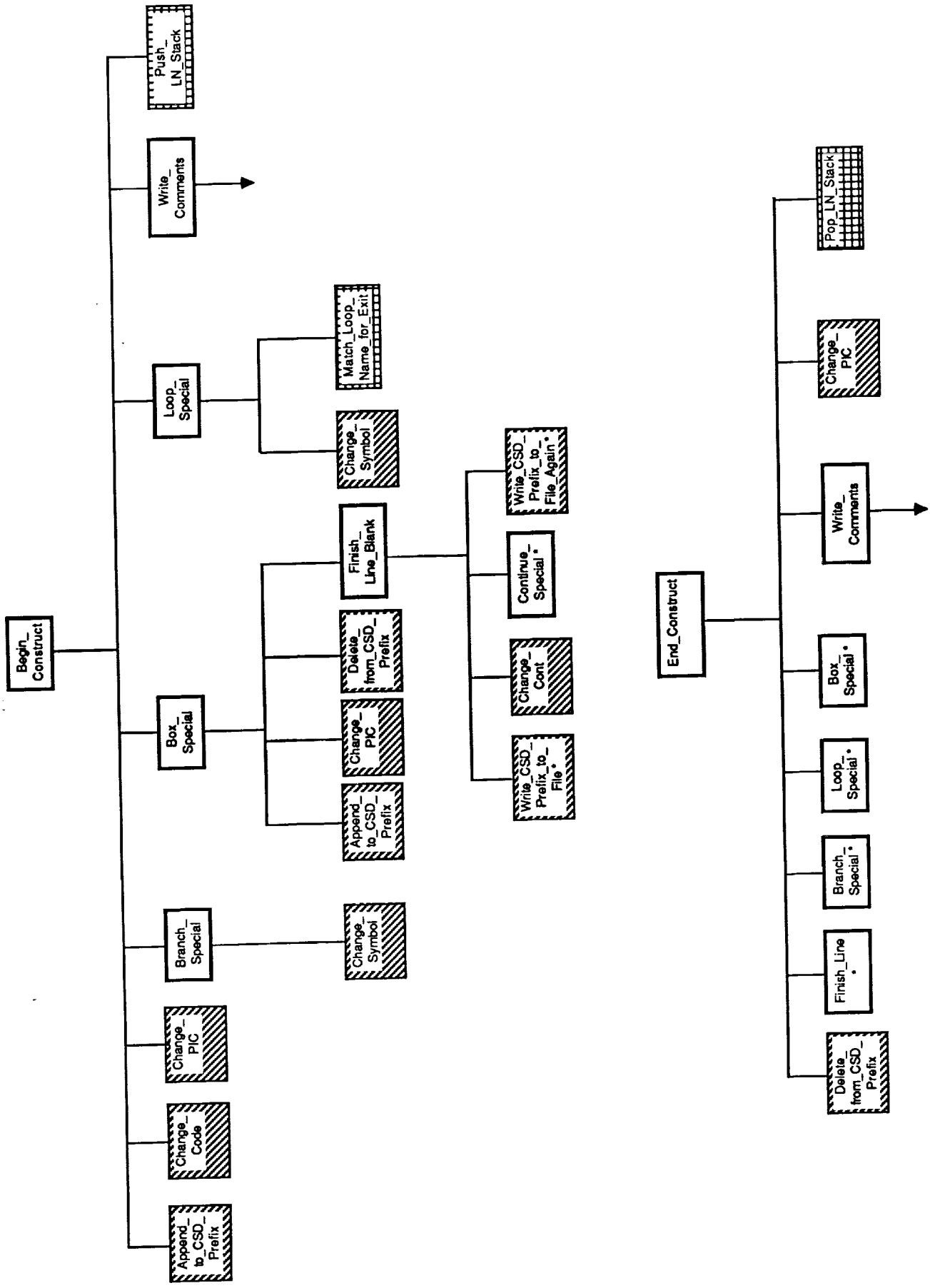


Figure 12(c). GRASP/Ada Hierarchical Structure.

prettyprinting routines (represented by the clear (or uncolored) box) is not an object according to the definition, the collection holds a central position in the object architecture. The arrows in the diagram show dependency among the objects and "unfettered" procedures where the entity at the tail of the arrow depends on the resources of the entity at the head of the arrow. Figures 12(a), 12(b), and 12(c) show the hierarchical structure of the procedures and object operations. These objects and operations, and their interaction with the parser are discussed in more detail below.

Over sixty graphical prettyprinting routines were developed for this project. Most of these routines are based upon two central object classes, *Line_Buffer* and *CSD_Prefix*.

The CSD generator is line-oriented in that it manages only one line of prettyprinted code at one time; previous lines are developed, output, and forgotten. Line_Buffer is fundamentally a character buffer of fixed size which holds processed Ada lexemes and whitespace until it is deemed appropriate to output the line. Line_Buffer then is cleared and is ready to hold another line of prettyprinted code. A prettyprinted line of code is output (CSD and all) by the routine *Finish_Line()*. A prettyprinted line of code can be terminated when the Line_Buffer would surpass its capacity with the addition of another lexeme, when the addition of a lexeme would cause wraparound, when the end of a construct is reached, or at other times called for by formatting conventions for Ada (e.g. a line beginning a FOR loop description is ended when the word **loop** is encountered). The last case is handled by the direct insertion of Finish_Line() into the grammar.

The CSD_Prefix contains coded information pertaining to what CSD characters would be output if the Line_Buffer were output at that moment. The CSD_Prefix is an array of structures known as *CSD_Symbols*. Each CSD_Symbol represents and holds the place of a sequence of CSD characters which are to be output when the Line_Buffer is emptied. Three fields contained in each CSD_Symbol determine which character sequence is appropriate: *CSD_Code*, *Point_in_Construct*, and *Continuation*.

The CSD_Code determines which of CSD construct should be output and, accordingly, corresponds roughly to constructs in Ada. Over seventy codes represent major constructs such as the FOR loop and minor constructs such as TYPE declarations.

Point_in_Construct declares at what point in the CSD diagram the generator is currently engaged. For instance, if the CSD_Code field for a particular CSD_Symbol calls for a LOOP construct, the Point_in_Construct field will help clarify whether the character sequence to be printed will contain the rounded top of the loop or the parallel lines of the loop body.

Each (*CSD_Code*, *Point_in_Construct*) tuple is associated with two possible CSD character sequences. The first sequence is what is output the first time the tuple is translated. The second sequence is what is output on subsequent lines. The second sequence is useful for times when the Line_Buffer is filled and output before the (CSD_Code, Point_in_Construct) tuple changes. The field Continuation designates which sequence is appropriate. This field is set to YES only after a prettyprinted line is output.

The tuple (*CSD_Code*, *Point_in_Construct*, *Continuation*) is used to index a table whose entries reference another table of normal character sequences. For each CSD_Symbol in the CSD_Prefix, the appropriate normal character sequences are concatenated and then translated by the font routines to produce the CSD characters for that line.

Based on these two objects, it was hoped to minimize the coupling between the lexical and grammatical description of the language and the prettyprinting routines themselves. This hope was undermined by a number of factors, however. The structure of the grammar itself made it necessary at times to write routines to alter the CSD_Prefix in some less than subtle way and insert these actions directly into the grammar. Also, there were quite a number of exceptional conditions which, while not directly dealt with in the grammar, caused the size of some of the routines and the total number of routines to grow.

Special sets of routines have been written to handle exceptional situations, such as printing of boxes for Ada subroutine and entry calls. Further, a stack of loop names exists to maintain loop nesting in order to determine which loop construct to override in the event of an EXIT statement. One other exceptional condition has been the handling of comments. Since comments are entirely lexical entities and are not even mentioned in the grammar, prettyprinting of comments has fallen prey to the problems involved in synchronizing grammatical and lexical prettyprinting routines mentioned at the end of the last section. A

compromise has been reached whereby comments are collected into a special buffer by the scanner until such time as is deemed appropriate to output them. These times are represented by calls to output the comment buffer within the bodies of some of the major prettyprinting routines. Original positioning of comments is lost in this process, but at least they are maintained in some order.

Another set of special routines involves the handling of names and expressions. In the Ada grammar that is available, subroutine and entry calls are recognized as names. Names can be exceedingly long and complex and can involve complex expressions as well. The problem was that the components of a name were appended to the Line_Buffer as they were recognized; if the name were exceedingly long, parts of it could be output before the entire name was recognized by the parser. This caused a major problem for handling both subroutine calls, entry calls, and assignment statements in that part of the text had already been output before the proper adjustments to the CSD_Prefix had been made (e.g. code for the top of the subroutine call box). This problem has been solved by routines which store name and expression components in ever-increasingly deep binary trees instead of appending them to the Line_Buffer. When the name is recognized by the parser, the adjustments are made to the CSD_Prefix and only then is the name output to the Line_Buffer.

There are certain limitations to this tool of which the user should be aware, most involving the sizes of the data structures. Dynamic memory allocation was considered for some of these structures; however, it seemed unconscionable to invoke memory allocation routines for small lexemes. The size of the CSD_Prefix is fixed at 40 CSD_Symbols, the Line_Buffer at 160 bytes plus space for an integer value, and the stack of loop names is fixed at 10 pointers to character strings. These sizes are hopefully large enough for any reasonable application.

Another limitation to this tool is in the distinction between subprogram calls and entry calls. Originally, different CSD diagrams had been planned for each, with the subprogram call being enclosed in a rectangular box and the entry call being enclosed in a slanted box. However, the grammar used did not distinguish the two in the general case (although a distinct entry call production did exist with respect to the selective wait

construct). Such discrimination would require the maintenance of a symbol table (in reality, a system of symbol tables due to the separate and previous compilations of other library units) which is simply not present in the current tool.

4.4. User Interface

A major requirement specified for the CSD Generator was that it be easily transportable to a variety of hardware environments. Upon reflection, this appears to be difficult to achieve because the routines needed to display graphics are non-standard among modern computer systems. Graphic routines range from the simple graphics supplied as part of the original IBM PC character set, to the bit-mapped graphics of the Apple Macintosh, to the PostScript graphic descriptions becoming popular among most laser printers and newer computers such as the NEXT.

To solve this problem, GRASP/Ada includes a set of simple graphics primitives that are necessary and sufficient to implement a simple graphical user interface. These primitives include routines for producing lines, boxes, and other simple graphics on an 80 column, 24 line display device. The internal detail of each of these primitives must be rewritten for each new environment, but the higher-level routines which call these primitives (and comprise the bulk of the code) remain the same. Because the implementation language chosen for GRASP/Ada is C, the alternative approach of using conditional compilation was briefly considered, but since the coding of graphics routines for various computer systems varies to such a great extent, the resulting code would have been more difficult to implement and maintain. To achieve portability, it is much more efficient to factor out the hardware-dependent operations and maintain separate graphics kernals for each.

One of the drawbacks to this approach is the increased overhead in drawing each of the graphic symbols. Because many of the GRASP/Ada primitives duplicate library routines provided by compilers on each of the host environments, the time required for on-screen display of the CSD symbols is increased. However, this overhead is practically indistinguishable in most cases, and the transportation benefits realized by taking this

approach are felt to outweigh vastly this minor degradation of performance. Should better performance ever become necessary, the GRASP/Ada primitives could be optimized for a specific installation to remove this overhead.

The graphics primitives needed for the GRASP/Ada user interface include routines to:

- Draw boxes
- Clear the screen
- Draw a window (making use of the box drawing and clear screen primitives)
- Clear a window
- Erase a window
- Get a keystroke (not a true graphics primitive, but needed for many I/O routines)
- Highlight text on the screen
- Dehighlight text on the screen

These routines were coded for the VAX 11/780 using the CURSES screen management package. CURSES is a library of screen I/O routines originally developed during the implementation of the *vi* editor. It consists of a simple set of graphics primitives that work on a variety of terminals and display devices, including the VAX 11/780 and the IBM PC. This means that the bulk of the GRASP/Ada system may be quickly ported to any system that supports CURSES.

The kernal of CURSES routines was used in the definition of a user interface for the GRASP/Ada system. The interface was crafted to be intuitive and simple. Two windows form the basis for most of the GRASP/Ada screens. The first is the working window, in which most of the user I/O occurs. This window is used to present menus and data entry mats, and it is in this window that the user is allowed to choose between various option settings. The second window is the help/status window, which is used to provide additional information about the currently selected object in the working window. For example, the user may be viewing an option for selecting how keywords are printed in the CSD. The help/status window would then tell all of the options available to the user.

A simple I/O package was developed for the display of windows, buttons, menus,

and text editing boxes. In the VAX 11/780 implementation, a window is actually implemented as a CURSES window, with some enhancement. GRASP/Ada routines provide the window with a custom border and name. Text editing boxes are framed strings which the user may edit using the cursor keys and alphanumeric characters. Buttons are implemented as text editing boxes with a default string which may not be edited. Menus are a group of buttons with some added properties. Finally, a variation of the button class was developed in which each button has a number of text strings associated with it. The user may toggle between these text strings with the left and right cursor keys. This class is used to present concisely options which have a finite and preset number of possible settings.

To remove the content of the GRASP/Ada menus and data entry mats from the mechanisms which manipulate them, a menu data object was developed. A menu is an array of Items which may be static (buttons, menu items) or editable (text edit strings). Each Item has several parameters associated with it: the type (static or editable), the screen coordinates, the number of alternates for the item (used to implement the button class for setting options), the currently selected alternate, and a pointer to the list of alternates. Each menu also has an associated HelpList, which contains a text string of helpful information for each of the Items in the menu. The menus and HelpLists are maintained separately from the main program and may be edited without requiring the recompilation of the main program. This separation of code and data allows the user interface to be modified rapidly without massive recompilation, a technique learned from the Apple Macintosh and OS/2 operating system environments.

Two useful routines are included for dealing with menus and mats. The first is the *DrawMenu()* routine, which prints a given menu or data entry mat in the working window. The second is the *DoMenuKey()* routine, which automates the menus and data entry mats. This routine is called repeatedly in the main program. When the routine is invoked, it gets a keystroke from the user and then determines if the keystroke is one that updates the menu or data entry mat. DoMenuKey() moves the cursor from button to button, highlighting and dehighlighting them as necessary. If DoMenuKey() moves the cursor into an editable text box, it places the text editing cursor in the last known position and calls the *ReadString()* routine in the STRINGS package for doing simple text editing. The end result is that the user

may do a fairly complex work of I/O with very little code, just an initialization and a loop.

4.5. EVE editor enhancements

DEC supplies the source code for a text editor for the VAX 11/780 called EVE (Extensible Vax Editor) that may be customized for end user applications. We decided to utilize this editor for viewing the CSD on the VAX. To accomplish this, we added an initialization module written in TPU (Text Processing Utility, the language in which EVE is written) that calls a C procedure for switching out the default screen font and replacing it with the CSD font. We also added commands (implemented in TPU) which "cloak" and "decloak" the CSD font so that the user may view the Ada code with or without the CSD. We plan to add a collapse/expand utility that would allow the user to collapse an Ada program to a small skeleton and then expand various sections as he follows the program's control flow. This utility would be of great use in a reverse engineering and maintenance situation. The equivalence of the CSD to a partial parse tree enables this utility to be feasible without requiring reparsing of the source code. This feature has not yet been implemented.

4.6. Software tools

The physical output of the CSD to the screen and printer was best accomplished using a specially-created font with characters corresponding to "slices" of the CSD constructs. In creating the font, there was a tradeoff to be considered involving the type of font to be created. Most commerical fonts fall into one of two categories: the bit-mapped font, in which the characters are represented as an array of bits; and the outline-font, in which the characters are represented using mathematical descriptions that comprise the character outlines.

Bit-mapped fonts are generally easier to create for a given output device because of their relative simplicity. In addition, bit-mapped fonts are generally optimal for use on personal computers and graphics terminals because the routines for displaying them are

highly optimized. On the other hand, a bit-mapped font is by definition created for use at one size: rescaling the font to smaller or larger sizes is difficult and often leads to an undesirable "jagginess" in the characters.

Outline fonts include the PostScript fonts popularized by Adobe and the new fonts created by Apple using its own proprietary outline font technology. These fonts are generally mathematical descriptions composed of Bezier splines which are translated by a PostScript (or similar) interpreter. These fonts may be rescaled to very large or very small sizes without losing any degree of resolution. However, the fonts are much harder to create, and typically are much slower for use in screen displays (witness the relative lack of Display Postscript devices).

For the CSD generator, the bit-mapped font was favored for several reasons. First, the equipment available at Auburn University does not include any screen devices that are capable of utilizing outline fonts, so going this route would have required the purchase or lease of additional hardware. Second, because bit-mapped fonts are relatively easy to generate for any given hardware environment, it was decided that it would not be difficult to produce the CSD for a variety of hardware environments. To this end, a "shell" font generator was created which could be quickly customized to produce fonts for any machine.

Although there are commercial font generators available for various machines (such as the IBM PC and the Apple Macintosh), we decided to create our own. By so doing, we gained a tool that could be quickly modified and customized for our own purposes, and perhaps later, even integrated into the CSD generator environment. In addition, creating a custom font generator that is reconfigurable for varying hardware environments enables us to present a consistent user interface for the software engineer porting the CSD.

The "shell" font generator was written for the Apple Macintosh in Turbo Pascal. Although the font generator is not a complete Macintosh application (i.e., resizable windows, cut & paste are not supported), it does present a graphical tool for the quick and efficient generation of custom fonts. The user is greeted with a rectangular grid (the dimensions of which may be quickly modified) corresponding to the dimensions of the desired font. By pointing and clicking on selected rectangles, a new character may be easily defined. Options

are included for saving the characters as they are created in two formats: a graphical format, for later editing; and a textual format, typically one or more lines of code that describe how to download the character into the targeted terminal or PC. The program is modular, and a font generator may be defined by simply specifying the dimensions of the desired font and writing a small code generation routine (often less than 100 lines of code).

At the present time, we have created font generators for the VT220 series of terminals, the LN03S laser printer, and the HP LaserJet laser printer. A font generator for the Panasonic KX-P1093 dot matrix printer is planned, but not yet implemented. We have used these font generators to create complete fonts for each of these output devices. During the daily use of these generators, it was found that an accessory program was desired to facilitate the collection of the font characters for upload to the VAX environment. A small "make" program was written, again in Turbo Pascal, which reads in a file of character names and loads the appropriate code fragments, along with appropriate device control sequences to create a complete font file.

5.0. Examples of Output

In this chapter, we present some examples of Ada source code that have been processed with the GRASP/Ada CSD generator. These examples demonstrate how the CSD constructs appear in "real" source code. The three examples in this chapter were taken as is from code written by Mr. Wenkai Chung of Auburn University in partial fulfillment of his Ph. D. degree in Computer Science and Engineering. The examples were chosen to reflect the Group I, II,and III CSD constructs, respectively.

The first example, *COMPLEX_NUMBER_PACKAGE.LN3*, is a package of Ada routines for handling complex numbers. The example is fairly object-oriented, using a complex number type and providing a set of functions for manipulating objects of that type. This example is used to demonstrate the Group I CSD constructs.

The second example, *INTERFACE.LN3*, is an Ada task that provides a simple user interface to a program that parses Prolog programs and builds a symbol table in preparation for later use by an AI program. This example primarily demonstrates the Group II CSD constructs for exception handling.

The third example, *BUFFERS.LN3*, is an Ada package that provides a simple buffering mechanism. The example was adapted from *Programming in Ada* by J.G.P. Barnes. This example demonstrates several of the Group III CSD constructs for tasking.

In the following sections, each of these examples is presented in greater detail with emphasis on individual CSD constructs. The source code listings for the examples have been concatenated to provide unique line numbers for the discussion of the three examples.

5.1. Example 1 – Complex Numbers

The Ada compilation unit *COMPLEX_NUMBER_PACKAGE.LN3* demonstrates a number of functions and procedures for manipulating complex numbers. This example demonstrates several Group I CSD constructs, including the procedure, package, sequence, and selection constructs. Let us examine these constructs in greater detail:

- **Procedure**

Lines 5-37 demonstrate how the CSD constructs clearly identify 8 functions and procedures in the package. Each function header is surrounded by a half box that provides a very easily seen visual directory to the contents of the package.

Lines 51-73 show the scope of a private function belonging to the package. The CSD control line leading down from the function header serves as a graphic aid in quickly determining the scope of the package. Although this function has a single entry point, it has two exits which are clearly indicated by arrows.

Lines 74-83 show the body of one of a number of the functions declared in the package specification. Notice how the CSD visually breaks the code in lines 74-167 into aesthetically pleasing and readily seen components.

- **Package**

Lines 1-3 and lines 48-50 show the CSD constructs that mark an Ada package. The CSD symbol for a package is a procedure box embedded in another box; this reinforces the image that a package is a collection of procedures and functions. The scope of the package may be easily followed using the CSD main control line that extends from the bottom of the package header.

- **Sequence**

Lines 77-83 show an example of a CSD sequence construct. The main control line extending from lines 77 to 83 shows the scope of the sequence. The horizontal control lines in lines 80, 81, and 82 show individual statements within the sequence.

- **Iteration**

In lines 61-66, an example of a sequence of statements in a loop is presented. The CSD clearly points out the various statements in the sequence, and although the code is spread over two pages, the CSD clearly depicts the boundaries of the loop.

- **Selection**

Lines 60-72 show an example of a CSD selection construct. The condition associated with the construct is shown in line 60, as noted by the diamond on the sequence control line. Because the diamond is the symbol widely used in flowcharting for showing conditions, it has been retained in the CSD for immediate recognition. The TRUE branch is shown by the solid CSD control line extending from the right of the diamond downward. Additional horizontal control lines denote the statements that will be executed when the condition is true. The FALSE (or ELSE) branch is shown by the dotted line extending from the bottom of the diamond downward. If there are no statements to be executed when the condition is false, the line simply ends. If there is an ELSE clause to the selection statement, a solid vertical CSD control line and appropriate horizontal control lines are used to indicate the ELSE body.

5.2. Example 2 – User Interface

INTERFACE.LN3 is an Ada unit containing a task that provides a simple user interface to a system written by Wenkai Chung of Auburn. This interface demonstrates the Group II CSD constructs for raising exceptions and for exception handling.

- **Raise**

Line 356 shows an example of an exception being raised. Notice how this is marked by an arrow breaking through the main CSD control line, indicating that the program is exiting its normal flow of control.

- **Exception handler**

Lines 413-444 show the CSD constuct for an exception handler. The handler is represented similarly to a procedure or function, the idea being that the exception handler represents an "emergency" procedure that is invoked when the executing program finds itself

in a situation that it cannot properly handle. The exceptions that may be handled are marked by diamonds, as seen in lines 416 and 439. And after an exception is handled, control passes out of the procedure which contains the handler as seen in lines 443-445.

5.3. Example 3 – Buffers

BUFFERS.LN3 is an Ada package found in *Programming in Ada* by J.G.P. Barnes and modified by Wenkai Chung. The package presents a simple buffering mechanism that may be used by other programs, and will be used to demonstrate several Group III constructs, including the task specification, rendezvous, terminate, and select.

- **Task specification**

A task type is declared in lines 465-470. Notice that the task type header is delimited using a half box with a slanted side. It is similar to the construct used for procedures, with the slanted side suggesting that the task is dynamic rather than static and may only exist for a portion of the lifetime of the program. Lines 476-512 show the CSD construct for a task body.

- **Rendezvous**

Task rendezvouses are shown in lines 486-488 and lines 496-498. Rendezvouses may be considered as communication with code outside the scope of the task, so they are represented in the CSD using arrows that cross the major control line of the task. This signifies that the flow of control effectively enters the task (as shown by the right arrow, lines 486 and 496) and then leaves the task after executing the statements in the accept statement (as shown by the left arrow, lines 488 and 498).

- **Terminate**

The terminate CSD construct is shown in line 507. A terminate command ends the task in which it is embedded, so this may be thought of as a termination of the flow of

control of the task. This is visualized in the CSD by an arrow leaving the major control line of the task.

- **Select**

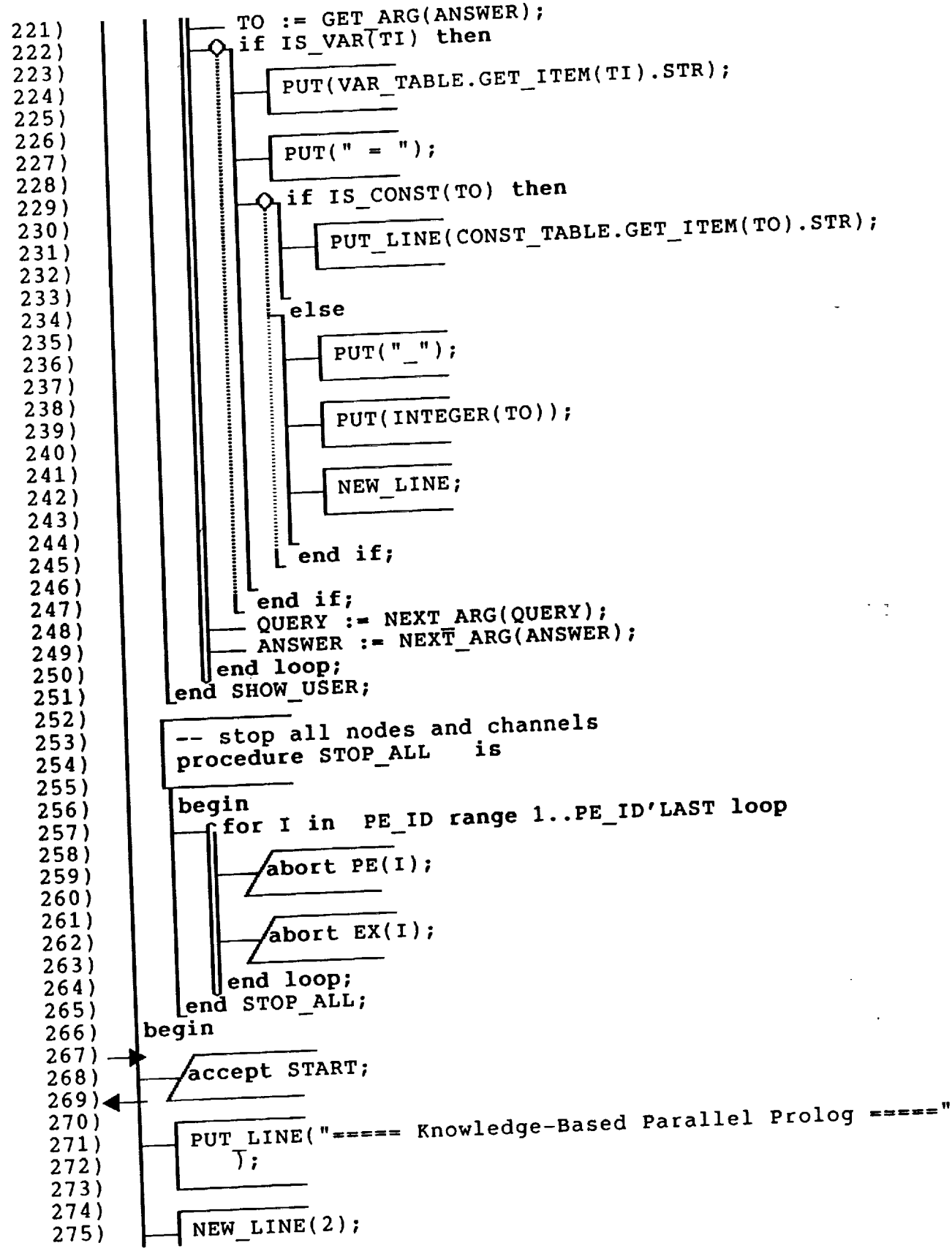
A select construct is shown in lines 484-510. The various alternatives that may be selected are shown by circles on the main control line of the select construct. Filled circles indicate guarded alternatives (lines 485 and 495) and hollow circles indicate unguarded alternatives (line 505).

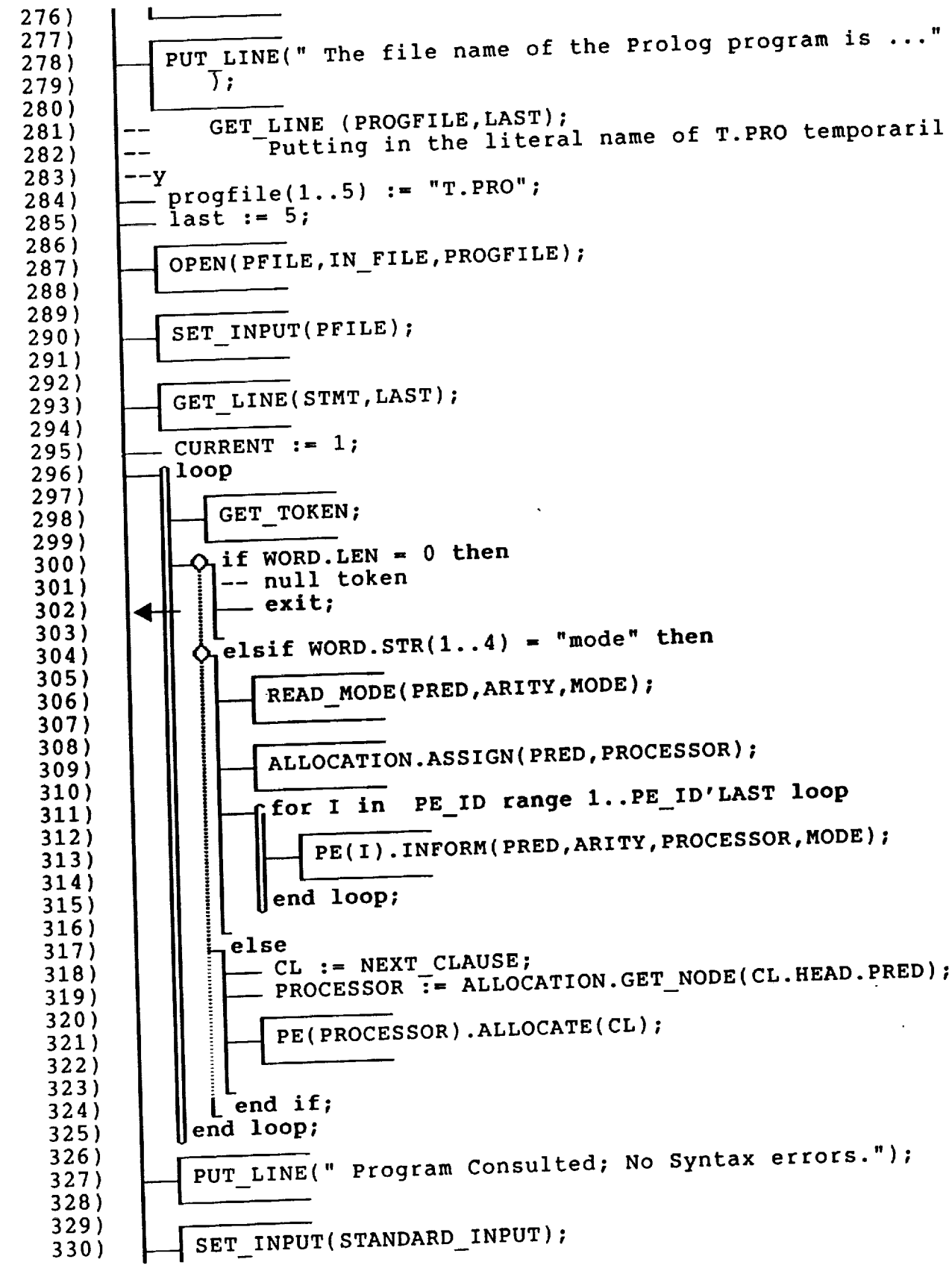
```
 1)
 2)  package complex_number_package   is
 3)
 4)     type complex_number_type  is private ;
 5)
 6)     function "-" (right :complex_number_type) return
 7)        complex_number_type ;
 8)
 9)     function "+" (left, right:complex_number_type) return
10)        complex_number_type ;
11)
12)     function "-" (left, right:complex_number_type) return
13)        complex_number_type ;
14)
15)     function "*" (left, right:complex_number_type) return
16)        complex_number_type ;
17)
18)     function "/" (left, right:complex_number_type) return
19)        complex_number_type ;
20)
21)     function "abs" (right :complex_number_type) return
22)        float ;
23)
24)     function to_complex(real_part, imaginary_part:float)
25)        return complex_number_type ;
26)
27)     procedure to_float(complex_number: in
28)        complex_number_type;  real_part, imaginary_part:out
29)        float) ;
30)
31)  private
32)     type complex_number_type  is
33)        record
34)           real_number : float;
35)           imaginary_number : float;
36)        end record ;
37)  end complex_number_package ;
38)
39)
40)
41)  package body complex_number_package is
42)
43)     function square_root (square:float) return float  is
44)
45)        guess:float := square / 2.0;
46)        previous_proximity:float :=  abs (square - guess**2
47)
48)
49)
50)
51)
52)
53)
54)
55)
```

```
56)          );
57)       proximity:float;
58)       imaginary_root_error :  exception;
59)    begin
60)       if square >= 0.0 then
61)          loop
62)             guess := (guess / square / guess) / 2.0;
63)             proximity :=  abs (square - guess**2);
64)             exit when proximity >= previous_proximity;
65)             previous_proximity := proximity;
66)          end loop ;
67)          return guess;
68)
69)       else
70)          raise imaginary_root_error;
71)
72)       end if ;
73)    end square_root ;
74)
75)    function "-" (right:complex_number_type) return
76)       complex_number_type  is
77)
78)       t : complex_number_type;
79)    begin
80)       t.real_number :=   -right.real_number;
81)       t.imaginary_number :=   -right.imaginary_number;
82)       return t;
83)    end "-" ;
84)
85)    function "+" (left, right:complex_number_type) return
86)       complex_number_type  is
87)
88)       t : complex_number_type;
89)    begin
90)       t.real_number := left.real_number + right.
91)          real_number;
92)       t.imaginary_number := left.imaginary_number + right
93)          .imaginary_number;
94)       return t;
95)    end "+" ;
96)
97)    function "-" (left, right:complex_number_type) return
98)       complex_number_type  is
99)
100)      t : complex_number_type;
101)   begin
102)      t.real_number := left.real_number - right.
103)         real_number;
104)      t.imaginary_number := left.imaginary_number - right
105)         .imaginary_number;
106)      return t;
107)   end "-" ;
108)
109)   function "*" (left, right:complex_number_type) return
110)      complex_number_type  is
```

```
111)
112)      t : complex_number_type;
113)   begin
114)      t.real_number := left.real_number * right.
115)         real_number * left.imaginary_number * right.
116)         imaginary_number;
117)      t.imaginary_number := left.real_number * right.
118)         imaginary_number * left.imaginary_number * right
119)         .real_number;
120)      return t;
121)   end "*" ;
122)
123)   function "/" (left, right:complex_number_type) return
124)      complex_number_type  is
125)
126)      t : complex_number_type;
127)      divide_constant : float :=  right.real_number**2 +
128)         right.imaginary_number**2;
129)   begin
130)      t.real_number := (left.real_number * right.
131)         real_number * left.imaginary_number * right.
132)         imaginary_number) / divide_constant;
133)      t.imaginary_number := (left.real_number * right.
134)         imaginary_number * left.imaginary_number * right
135)         .real_number) / divide_constant;
136)      return t;
137)   end "/" ;
138)
139)   function "abs" (right:complex_number_type) return
140)      float  is
141)
142)      t : float;
143)   begin
144)      t := square_root(right.real_number**2 + right.
145)         imaginary_number**2);
146)      return t;
147)   end "abs" ;
148)
149)   function to_complex(real_part, imaginary_part:float)
150)      return complex_number_type  is
151)
152)      t : complex_number_type;
153)   begin
154)      t.real_number := real_part;
155)      t.imaginary_number := imaginary_part;
156)      return t;
157)   end to_complex ;
158)
159)   procedure to_float(complex_number: in
160)      complex_number_type; real_part, imaginary_part:out
161)      float)  is
162)
163)   begin
164)      real_part := complex_number.real_number;
165)      imaginary_part := complex_number.imaginary_number;
```

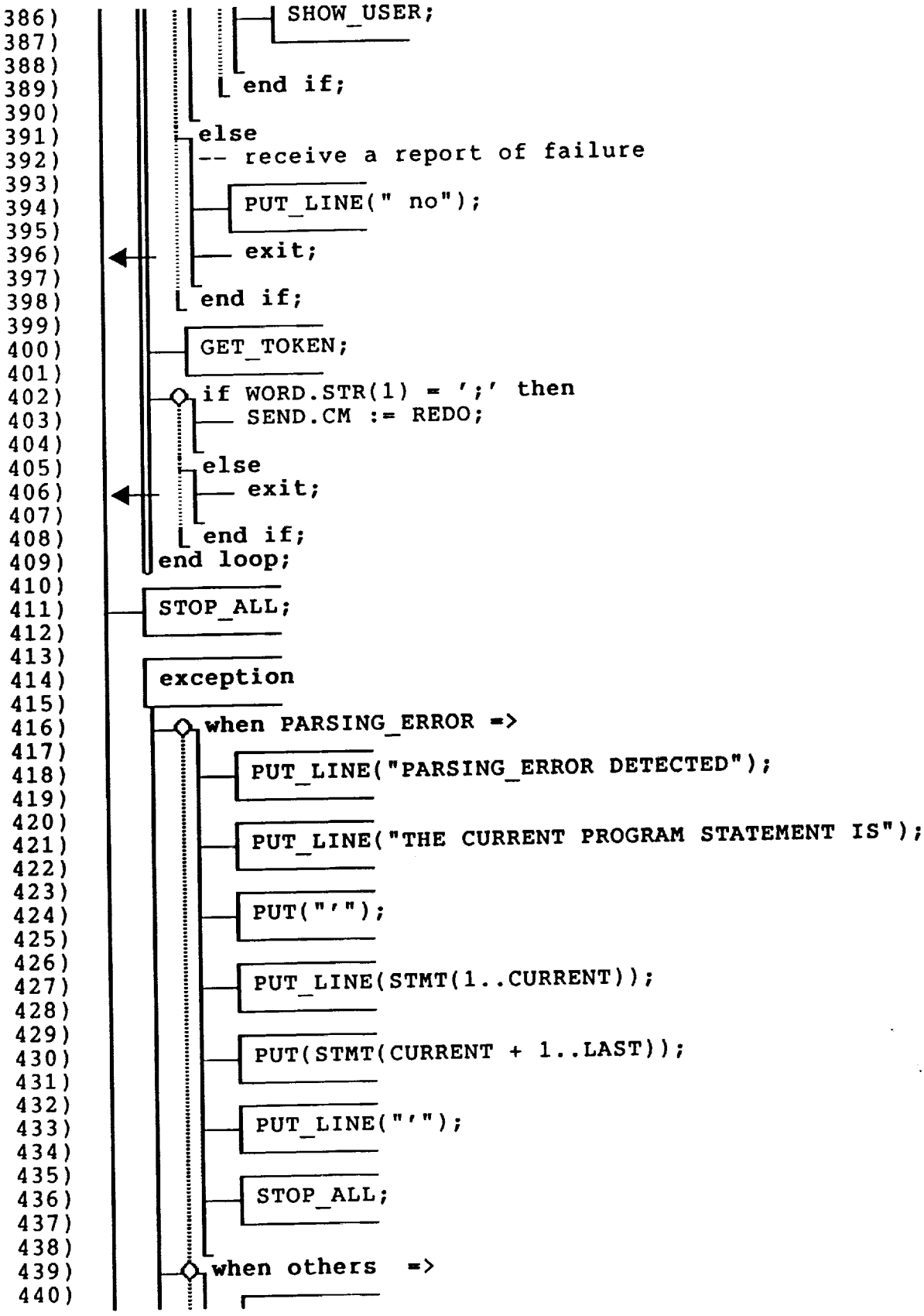
```
166)     end to_float ;
167)   end complex_number_package ;
168)
169)
170)
171)--     task INTERFACE is
172)--         entry START;
173)--         entry REPORT    (M: MESSAGE);
174)--     end INTERFACE;
175)     with TEXT_IO, INTEGER_TEXT_IO;
176)     use TEXT_IO, INTEGER_TEXT_IO;
177)     separate(MAIN)
178)
179)   task  body INTERFACE is
180)
181)       PFILE: FILE_TYPE;
182)       PROGFILE: STRING(1..32);
183)       PARSING_ERROR: exception;
184)   -- GET_TOKEN uses the following:
185)       STMT: STRING(1..72);
186)   -- current program statement
187)       CURRENT, LAST: NATURAL;
188)   -- indexes on STMT
189)       WORD: TOKEN;
190)       PRED: PRED_ID;
191)       ARITY: NATURAL;
192)       MODE: MODE_PT;
193)       CL: CLAUSE;
194)       PROCESSOR: PE_ID;
195)       GOAL: ATOM;
196)       SEND, RECEIVE: MESSAGE;
197)
198)       procedure GET_TOKEN    is separate;
199)
200)
201)       procedure READ_MODE(PRED: out PRED_ID; ARITY: out
202)          NATURAL; MODE: out MODE_PT)    is separate;
203)
204)
205)       function NEXT_ATOM return ATOM    is separate;
206)
207)
208)       function NEXT_CLAUSE return CLAUSE    is separate;
209)
210)
211)       -- show user the variable bindings
212)       procedure SHOW_USER    is
213)
214)          QUERY, ANSWER: ARG_PT;
215)          TI, TO: TERM;
216)       begin
217)          QUERY := SEND.GOAL.ARGS;
218)          ANSWER := RECEIVE.GOAL.ARGS;
219)          while QUERY /= NULL_ARG_PT loop
220)             TI := GET_ARG(QUERY);
```


TO := GET_ARG(ANSWER);
if IS_VAR(TI) then
PUT(VAR_TABLE.GET_ITEM(TI).STR);
PUT(" = ");
if IS_CONST(TO) then
PUT_LINE(CONST_TABLE.GET_ITEM(TO).STR);
else
PUT("_");
PUT(INTEGER(TO));
NEW_LINE;
end if;
end if;
QUERY := NEXT_ARG(QUERY);
ANSWER := NEXT_ARG(ANSWER);
end loop;
end SHOW_USER;
-- stop all nodes and channels
procedure STOP_ALL is
begin
for I in PE_ID range 1..PE_ID'LAST loop
abort PE(I);
abort EX(I);
end loop;
end STOP_ALL;
begin
accept START;
PUT_LINE("===== Knowledge-Based Parallel Prolog =====");
NEW_LINE(2);

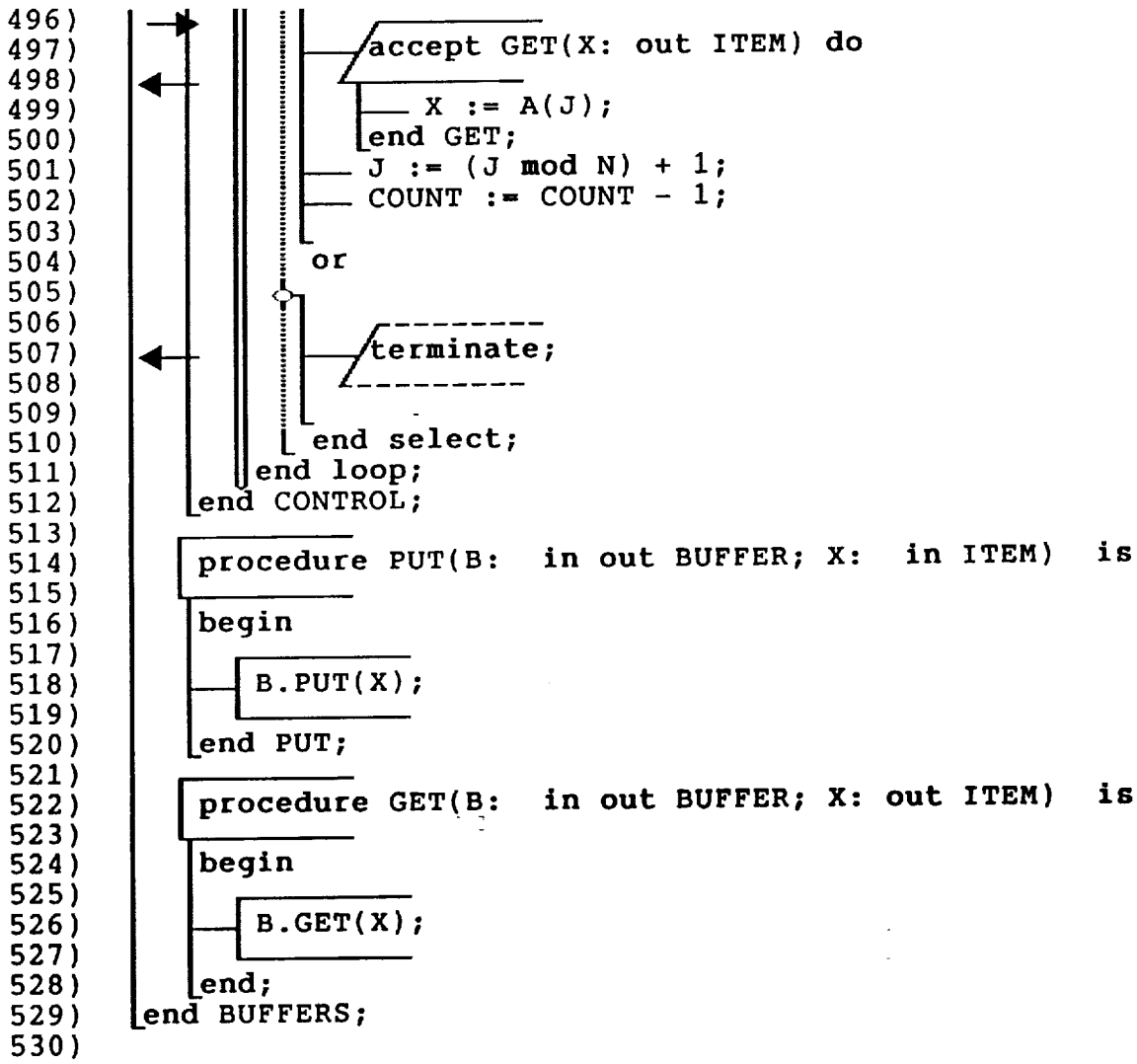


```
276)
277)
278)    PUT_LINE(" The file name of the Prolog program is ..."
279)        );
280)
281)  --     GET_LINE (PROGFILE,LAST);
282)  --         Putting in the literal name of T.PRO temporaril
283)  --y
284)    progfile(1..5) := "T.PRO";
285)    last := 5;
286)
287)    OPEN(PFILE,IN_FILE,PROGFILE);
288)
289)
290)    SET_INPUT(PFILE);
291)
292)
293)    GET_LINE(STMT,LAST);
294)
295)    CURRENT := 1;
296)    loop
297)
298)      GET_TOKEN;
299)
300)      if WORD.LEN = 0 then
301)        -- null token
302)        exit;
303)
304)      elsif WORD.STR(1..4) = "mode" then
305)
306)        READ_MODE(PRED,ARITY,MODE);
307)
308)
309)        ALLOCATION.ASSIGN(PRED,PROCESSOR);
310)
311)        for I in  PE_ID range 1..PE_ID'LAST loop
312)
313)          PE(I).INFORM(PRED,ARITY,PROCESSOR,MODE);
314)
315)        end loop;
316)
317)      else
318)        CL := NEXT_CLAUSE;
319)        PROCESSOR := ALLOCATION.GET_NODE(CL.HEAD.PRED);
320)
321)        PE(PROCESSOR).ALLOCATE(CL);
322)
323)
324)      end if;
325)    end loop;
326)
327)    PUT_LINE(" Program Consulted; No Syntax errors.");
328)
329)
330)    SET_INPUT(STANDARD_INPUT);
```

```
331)
332)
333)      NEW_LINE;
334)
335)
336)      PUT_LINE("===== Parallel Query Execution Starts =====")
337)          );
338)
339)
340)      NEW_LINE;
341)
342)
343)      PUT_LINE(" Type CNTL/Z to quit.");
344)
345)
346)      PUT("?-");
347)
348)
349)      GET_TOKEN;
350)
351)      GOAL := NEXT_ATOM;
352)
353)      GET_TOKEN;
354)
355)      if WORD.STR(1) /= '.' then
356)          raise PARSING_ERROR;
357)
358)      end if;
359)      PROCESSOR := ALLOCATION.GET_NODE(GOAL.PRED);
360)      SEND.PE := 0;
361)      SEND.ID := 1;
362)      SEND.CM := REQUEST;
363)      SEND.GOAL := GOAL;
364)      SEND.NO_VAR := VAR_TABLE.NUMBERS;
365)      loop
366)
367)          PE(PROCESSOR).INTERACT(SEND);
368)
369)
370)          accept REPORT(M: MESSAGE) do
371)
372)              RECEIVE := M;
373)
374)              COPY(M.GOAL,RECEIVE.GOAL);
375)
376)          end REPORT;
377)          if RECEIVE.CM = SUCCESS then
378)              if SEND.NO_VAR = 0 then
379)
380)                  PUT_LINE(" yes");
381)
382)                  exit;
383)
384)              else
385)
```


SHOW_USER;
end if;
else
-- receive a report of failure
PUT_LINE(" no");
exit;
end if;
GET_TOKEN;
if WORD.STR(1) = ';' then
SEND.CM := REDO;
else
exit;
end if;
end loop;
STOP_ALL;
exception
when PARSING_ERROR =>
PUT_LINE("PARSING_ERROR DETECTED");
PUT_LINE("THE CURRENT PROGRAM STATEMENT IS");
PUT("'");
PUT_LINE(STMT(1..CURRENT));
PUT(STMT(CURRENT + 1..LAST));
PUT_LINE("'");
STOP_ALL;
when others =>

```
441)       STOP_ALL;
442)
443)
444)
445)   end INTERFACE;
446)
447)
448)
449)   generic
450)   -- From: J.G.P. Barnes, "Programming in Ada"
451)      N: POSITIVE;
452)   --    pp. 249-250, with slight modification.
453)      type ITEM  is private;
454)
455)   package  BUFFERS  is
456)
457)      type BUFFER  is limited private;
458)
459)      procedure PUT(B:  in out BUFFER; X:  in ITEM);
460)
461)
462)      procedure GET(B:  in out BUFFER; X: out ITEM);
463)
464)   private
465)
466)      task  type CONTROL is
467)
468)         entry PUT(X:  in ITEM);
469)         entry GET(X: out ITEM);
470)      end CONTROL;
471)      type BUFFER  is new CONTROL;
472)   end BUFFERS;
473)
474)   package body BUFFERS is
475)
476)
477)      task body CONTROL is
478)
479)         A: array(1..N) of ITEM;
480)         I,J: INTEGER range 1..N := 1;
481)         COUNT: INTEGER range 0..N := 0;
482)      begin
483)         loop
484)            select
485)               when COUNT < N=>
486)
487)               accept PUT(X:  in ITEM) do
488)
489)                  A(I) := X;
490)               end PUT;
491)               I := (I mod N) + 1;
492)               COUNT := COUNT + 1;
493)
494)            or
495)               when COUNT > 0=>
```


accept GET(X: out ITEM) do
X := A(J);
end GET;
J := (J mod N) + 1;
COUNT := COUNT - 1;
or
terminate;
end select;
end loop;
end CONTROL;
procedure PUT(B: in out BUFFER; X: in ITEM) is
begin
B.PUT(X);
end PUT;
procedure GET(B: in out BUFFER; X: out ITEM) is
begin
B.GET(X);
end;
end BUFFERS;

6.0 Future Directions

The previous sections have presented the first phase of the GRASP/Ada research project. In this section, we briefly describe the future directions in which this project may evolve. The original motivation for the GRASP/Ada research project was to develop a graphical specification for Ada that would be useful at each level of program development: the process level (system diagrams), the structural level (structure charts) and the algorithmic level (control structure diagrams). The direction taken in the research was to approach the problem as one of reverse engineering. Beginning with Ada source code, algorithmic diagrams (the CSD) were proposed and modified in such a way that they could be derived automatically from the code with no intervention from the user. In the next phases of the GRASP/Ada project, this approach is to be taken a step farther, leading to the automated production of structure charts and system diagrams from the source code.

To achieve this, a set of graphical representations that support Ada at the system and architectural levels in much the same way that the CSD supports Ada at the procedural level must be developed and or adapted from existing diagrams. Natural candidates for these graphical representations are the data flow diagram for the system level and the structure chart and object diagram for the architectural level. Although these diagrams have been heavily discussed in the literature, each is generally too informal for reverse engineering, and must be developed and formalized to be of use. As the diagrams are formalized, the feasibility of automatically generating them from source code will be evaluated.

Once automatic generation of these graphical representations is determined to be feasible, a software tool for generating and displaying them will be designed and implemented. In Phase I, a software tool for producing the CSD was designed and implemented in the form of a prototype. Similar tools would be developed for the data flow diagram and structure chart, perhaps with a driver that automates the production and layout of the generated design documentation.

Since the GRASP/Ada research project has focused specifically on the Ada programming language, extensions to the graphical representations that deal with Ada-specific constructs such as tasking and exception handling will be addressed. A graphical representation of the package/object view of Ada software may prove useful for illustrating the data types and operations of an Ada package as well as for depicting the dependencies among various packages. This is especially true in view of the current trend toward object oriented design.

Finally, the application of artificial intelligence (AI) and expert systems to software engineering will be investigated with respect to the GRASP/Ada research project. Expert systems may prove invaluable in creating the structure and system representations, particularly for classifying the Ada source code into groups that correspond to the components of these graphical representations and for laying out these diagrams in a clear and meaningful fashion.

BIBLIOGRAPHY

ABB83 Abbott, R. J. 1983. "Program Design by Informal English Description," *Communications of the ACM*, Vol. 26, No. 11, pp. 882-894.

ABR80 Abrial, Jean-Raymond, Schuman, S. A., and Meyer, B. 1980. "A Specification Language," in *On the Construction of Programs*, R. McNaughten and R.C. McKeag, eds., Cambridge University Press.

ACL88 Acly, E. 1988, "Initiatives in Reverse Engineering: The Impossible, the Essential, and the Disappointing," *2nd International Conference on Computer-Aided Software Engineering*, pp. 15-3 to 15-8.

ALF77 Alford, Mack W. 1977. "A Requirements Engineering Methodology for Real-Time Processing Requirements," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January, pp. 60-69.

ART88 Arthur, J. D. 1988. "GETS: A Graphical Environment for Task Specification," *Proceedings of the 7th Annual International Phoenix Conference on Computers and Communications*, pp. 269-273.

BAL83 Balzer, Robert M. 1983. "A Global View of Automatic Programming," in *Proc. Third Int. Joint Conf. Artif. Intell.*, August, pp. 494-499.

BAL85 Balzer, Robert M. 1985. "A 15 Year Perspective on Automatic Programming," *IEEE Transactions on Software Engineering*, Volume SE-11, No. 11, November, pp. 1257-1268.

BAL81 Balzer, Robert M. 1981. "Transformational Implementation: An Example," *IEEE Transactions on Software Engineering*, Vol. SE-7, No. 1, January, pp. 3-14.

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

BRS85 Barstow, D. R. 1985. "Domain-Specific Automatic Programming," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11, November, pp. 1321-1336.

BAS86 Basili, V. R. 1986. "A plan for empirical studies of programmers," *Empirical Studies of Programmers: First Workshop*, pp. 252-255.

BAT86 Batini, Carlo, Nardelli, E., and Tamassia, R. 1986. "A Layout Algorithm for Data Flow Diagrams," *IEEE Transactions on Software Engineering*, Volume SE-12, No. 4, April, pp. 538-546.

BEI84 Beichter, Friedrich W., Herzog, O., and Petzsch, H. 1984. "SLAN-4 - A Software Specification and Design Language," *IEEE Transactions on Software Engineering*, Volume SE-10, No. 2, March, pp. 155-162.

BLA73 Blaiwes, A. S. 1973. "Some training factors related to procedural performance," *Journal of Applied Psychology*, No. 58, pp. 214-218.

BLA74 Blaiwes, A. S. 1974. "Formats for presenting procedural instructions," *Journal of Applied Psychology*, No. 59, pp. 683-686.

BLZ88 Blaze, M. and Cameron, E. J. 1988. "D* - - An Automatic Documentation System for IC* Programs," *2nd International Conference on Computer-Aided Software Engineering*, pp. 15-9 to 15-12.

BOE84 Boehm, B. W. 1984. "Verifying and Validating Software Requirements and Design Specifications," *IEEE Software*, January, pp. 75-88.

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

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

BRO80a Brooke, J. B. and Duncan, K. D. 1980. "An experimental study of flowcharts as an aid to identification of procedural faults," *Ergonomics*, No. 23, pp. 387-399.

BRO80b Brooke, J. B. and Duncan, K. D. 1980. "Experimental studies of flowchart use at different stages of program debugging," *Ergonomics*, No. 23, pp. 1057-1091.

BRK80 Brooks, R. 1980. "Studying programmer behavior experimentally: the problems of proper methodology," *Communications of the ACM*, No. 23(4), pp. 207-213.

BRK83 Brooks, R. 1983. "Towards a theory of the comprehension of computer programs," *International Journal of Man-Machine Studies*, No. 18, pp. 543-554.

BRN84 Brown, David B. and Herbanek, J. A. 1984. *Systems Analysis for Applications Software Design*, Holden-Day Inc., Oakland, California.

BRW85 Brown, G. P., Carling, R. T., Herot, C. F., Kramlich, D. A., and Souza, P. 1985. "Program visualization: graphical support for software development," *IEEE Computer*, No. 18, pp. 27-35.

BUH84 Buhr, R. J. A. 1984. *System Design with Ada*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

BUR85 Burns, R. N. and Dennis, A. R. 1985. "Selecting the Appropriate Application Development Methodology," *DATA BASE*, Fall, pp. 19-23.

CAM86 Cameron, J. R. 1986. "An Overview of JSD," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February, pp. 222-240.

CAR86 Carlson, D. H. 1986. "Structured analysis and the data flow diagram: tools for library analysis," *Information Technologies and Libraries*, No. 6, pp. 121-128.

CAS85 Case, Albert F., Jr. 1985. "Computer-Aided Software Engineering (CASE): Technology for Improving Software Development Productivity," *DATA BASE*, Fall, pp. 35-23.

CHE88 Chen, T. L. C. and Sutton, M. M. 1988. "Object-Oriented Design: Is It Enough For Large Ada Systems?", *Proceedings of the 1988 ACM Sixteenth Annual Computer Science Conference*, pp. 529-534.

CHU82 Chu, Yaohan. 1982. *Software Blueprint and Examples*, Lexington Books, Lexington, Massachusetts.

COX86 Cox, B. 1986. *Object-Oriented Programming*, Addison-Wesley.

CRA86 Crawford, Bard S. and Jazwinski, Andrew H. 1986. "The AdaGraph™ Tool for Enhanced Ada Productivity," *Proceedings of the IEEE National Aerospace and Electronics Conference - NAECON*, Dayton, Ohio, May 19-23, pp. 664-670.

CRO86 Cross, J. H. 1986. *The Control Structure Diagram: An Automated Graphical Stepwise Refinement Tool With Control Constructs*, Dissertation, Texas A&M University, College Station, TX.

CRO88a Cross, J. H. and Sheppard, S. V. 1988. "The Control Structure Diagram: An Automated Graphical Representation For Software," *Proceedings of the 21st Hawaii International Conference on Systems Sciences*, January 5-8.

CRO88b Cross, J. H. and Sheppard, S. V. 1988. "Graphical Extensions for Pseudo-Code, PDLs, and Source Code," *Proceedings of the 1988 ACM Sixteenth Annual Computer Science Conference*, pp. 520-528.

CUN87 Cunniff, N. and Taylor, R. 1987. "Graphical vs. textual representation: an empirical study of novices' program comprehension," *Empirical Studies of Programmers: Second Workshop*, pp. 114-131.

CUR86 Curtis, B. 1986. "By the way, did anyone study any real programmers?" *Empirical Studies of Programmers: First Workshop*, pp. 256-262.

DEM79 DeMarco, Tom. 1979. *Structured Analysis and System Specification*, Prentice-Hall, Inc. Englewood Cliffs, New Jersey.

DIJ68 Dijkstra, E. W. 1968. "The Structure of the 'THE'-Multiprogramming System," *Communications of the ACM*, Vol. 11, No. 5, pp. 341-346.

DOD80 Department of Defense. 1980. *Requirements for Ada Programming Support Environments*, February.

FER78 Ferstl, O. 1978. "Flowcharting by Stepwise Refinement," *SIGPLAN Notices*, Vol. 13, No. 1, pp. 34-42.

FIC85 Fickas, Stephen F. 1985. "Automating the Transformational Development of Software," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11, November, pp. 1268-1277.

FIS87 Fischer, G., Lemke, A. C., and Rathke, C. 1987. "From Design to Redesign," *Proceedings of the 21st Annual ACM Hawaii International Conference on System Sciences*, pp. 369-376.

FIT79 Fitter, M. and Green, T. R. G. 1979. "When do diagrams make good computer languages?", *Journal of Man-Machine Studies* 11, pp. 235-261.

FUK85 Fukunaga, K. 1985. "PROMPTER: A Knowledge Based Support Tool for Code Understanding," *Proceedings of the 8th International Conference on Software Engineering*, pp. 358-363.

GAN79 Gane, Chris and Sarson, Trish. 1979. *Structured Systems Analysis: tools and techniques*, Prentice-Hall Inc., Englewood Cliffs, New Jersey.

GAN82 Gane, Chris and Sarson, Trish. 1982. *Structured System Analysis*, McDonnell Douglas.

GEH84 Gehani, Narain. 1984. *Ada: Concurrent Programming*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

GIL84 Gilmore, D. J. and Smith, H. T. 1984. "An investigation of the utility of flowcharts during computer program debugging," *International Journal of Man-Machine Studies*, No. 20, pp. 357-372.

GRA87 Grau, J. K. and Gilroy, K. A. 1987. "Compliant Mappings of Ada Programs to the DOD-STD-2167 Static Structure," *Ada Letters*, vii. 2-73 to 2-84.

GUG86 Gugerty, L. and Olson, G. M. 1986. "Comprehension differences in debugging by skilled and novice programmers," *Empirical Studies of Programmers: First Workshop*, pp. 13-27.

GUT85 Guttag, John V., Horning, James J., and Wing, Jeannette M. 1985. "The Larch Family of Specification Languages," *IEEE Software*, September, pp. 24-36.

HAB82 Habermann, A. N. and Notkin, D. S. 1982. "The Gandalf Software Development Environment," Carnegie-Mellon University, Pittsburgh, PA, pp. 1-18.

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

HAN83 Hansen, K. 1983. *Data Structured Program Design*, Ken Orr & Associates, Inc., Topeka, Kansas.

HAR83 Harada, J. and Sakashita, S. 1984. "A Documentation Tool to Visualize Program Maintainability," *Proceedings of the 1983 Software Maintenance Workshop*, pp. 275-280.

HRL79 Harel, D., Norvig, P., Rood, J., and To, T. 1979. "A Universal Flowcharter," *Proceedings of AIAA Computers in Aerospace Conference*, pp. 218-224.

HEL85 Helmbold, D. and Luckham, D. 1985. "TSL: Task Sequencing Language," *Ada Letters*, Vol. V, No. 2, pp. 255-274.

HES81 Hester, S. D., Parnas, D. L., and Utter, D. F. 1981. "Using Documentation as a Software Design Medium," *The Bell System Technical Journal*, Vol. 60, No. 8, October, pp. 1941-1977.

HEV88 Hevis, E. 1988. "Executable Specification Languages: A Visual Programming Paradigm for CASE Languages," *2nd International Conference on Computer-Aided Software Engineering*, pp. 8-11 to 8-14.

HIG86 Higgins, David A. 1986. *Data Structured Software Maintenance: The Warnier-Orr Approach*, Dorset House Publishing Co., New York, New York.

JAC83 Jackson, M. A. 1983. *System Development.* Prentice-Hall International, Englewood Cliffs, New Jersey.

JEN79 Jensen, R. W., and Tonies, C. C. 1979. *Software Engineering*, Prentice-Hall, Englewood Cliffs, N. J., pp. 267-273.

JON80 Jones, Clifford B. 1980. *Software Development: A Rigorous Approach*, Prentice-Hall International, Inc., London, England.

KAM75 Kammann, R. 1975. "The comprehensibility of printed instructions and the flowchart alternative," *Human Factors*, No. 17(2), pp. 183-191.

KEM85 Kemmerer, Richard A. 1985. "Testing Formal Specifications to Detect Design Errors," *IEEE Transactions on Software Engineering*, Volume SE-11, No. 1, January, pp. 32-43.

KRO83 Krohn, G. S. 1983. "Flowcharts used for procedural instructions," *Human Factors*, No. 25(5), pp. 573-581.

KUO83 Kuo, J., Ramanathan, J., Soni, D., and Suni, M. 1983. "An Adaptable Software Environment to Support Methodologies," *Proceedings of the 1983 SOFTFAIR - Software Development: Tools, Techniques, and Alternatives*, pp. 363-374.

LET86 Letovsky, S. 1986. "Cognitive processes in program comprehension," *Empirical Studies of Programmers: First Workshop*, pp. 58-79.

LER88 Lerner, M. 1988. "Bringing Order into Software Mess with the Help of a Graph," *2nd International Conference on Computer-Aided Software Engineering*, pp. 15-16 to 15-20.

LIN77 Lindsey, G. H. 1977. "Structure Charts: A Structured Alternative to Flowcharts," *SIGPLAN Notices*, Vol. 12, No. 11.

LUC85 Luckham, David and von Henke, Friedrich W. 1985 "An Overview of Anna, a Specification Language for Ada," *IEEE Software*, Volume 2, Number 2, March, pp. 9-23.

MAP86 Maples, W. and Swigger, K. M. 1986. "The effect of indentation and white space on program retention," *Proceedings of the Human Factors Society - 30th Annual Meeting*, pp. 24-28.

MAR85a Martin, James. 1985. *Fourth-Generation Languages, Volume 1*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

MAR86 Martin, James. 1986. *Fourth-Generation Languages, Volume 2*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

MAR85b Martin, James, and McClure, C. 1985. *Diagramming Techniques for Analysts and Programmers*, Prentice-Hall, Englewood Cliffs, New Jersey.

MAY75 Mayer, R. E. 1975. "Different problem-solving competencies established in learning computer programming with and without meaningful models," *Journal of Educational Psychology*, No. 67, pp. 726-734.

MEY85 Meyer, Bertrand. 1985. "On Formalism in Specifications," *IEEE Software*, January, pp. 6-26.

MOR87a Morrison, K. 1987. "An Executable Specification Language for Ada Tasking Problems," *Proceedings of the ACM 26th Annual Southeast Regional Conference*, pp. 491-495.

MOR87b Morrison, K. 1987. *GRASP: An Executable Specification Language and Support Environment for Ada*, Master's Thesis, Auburn University, Auburn, Alabama.

MOR88 Morrison, K. 1988. "GRASP: An Executable Specification Language for Ada Tasking," *Proceedings of the 1988 ACM Sixteenth Anuual Computer Science Conference*, p. 683.

MYE78 Myers, G. 1978. *Composite Structured Design*, Von Nostrand.

NAS88 NASA/Goddard Space Flight Center. "A Comparison of Software Verification Techniques," *Technical Report SEL-85-001*, April 1985.

NSS73 Nassi, I. and Shneiderman, B. 1973. "Flowchart techniques for structured programming," *SIGPLAN Notices*, No. 8(8), pp. 12-26.

NOS86 Nosek, J. T., and Ahrens, J. D. 1986. "An experiment to test user validation of requirements: data-flow diagrams vs. task-oriented menus," *International Journal of Man-Machine Studies*, No. 25, pp. 675-684.

ORR77 Orr, Kenneth T. 1977. *Structured Systems Development*, Yourdon Press, New York, New York.

ORR81 Orr, Kenneth T. 1981. *Structured Requirements Definition.* Ken Orr and Associates, Inc. Topeka, Kansas.

PET81 Peters, Lawrence J. 1981. *Software Design: Methods & Techniques*, Yourdon Press, New York, New York.

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

RAE85 Raeder, G. 1985. "A survey of current graphical programming techniques," *IEEE Computer*, No. 18, pp. 11-25.

RAJ85 Rajlich, V. 1985. "Paradigms for Design and Implementation in Ada," *Communications of the ACM*, Vol. 28, No. 7, pp. 718-727.

RAJ88 Rajlich, V., Damaskinos, N., Khorshid, W., Linos, P., and Silva, J. 1988. "An Environment for Maintaining C Programs," *2nd International Conference on Computer-Aided Software Engineering*, pp. 15-21 to 15-23.

RIC88 Rich, C. and Waters, R. C. 1988. "Automatic Programming: Myths and Prospects," *IEEE Computer*, August, Vol. 21, No. 8, pp. 40-51.

ROB86 Robillard, P. N. 1986. "Schematic Pseudocode for Program Constructs and its Computer Automation by Schemacode," *Communications of the ACM*, November, Volume 29, Number 11, pp. 1072-1089.

ROS77 Ross, Douglas T. 1977. "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, Volume SE-3, No. 1, January, pp. 16-34.

RUG88 Rugaber, S. 1988. "Hypertext and Software Maintenance," *2nd International Conference on Computer-Aided Software Engineering*, pp. 15-24 to 15-27.

SEI87 Seidewitz, Ed. and Stark, M. 1987. "Towards a General Object-Oriented Software Development Methodology," *Ada Letters*, Vol. 8, No. 4, pp. 4-54 to 4-67.

SEI88 Seidewitz, Ed. 1988. "General Object-Oriented Software Development with Ada: A Life-Cycle Approach," *Collected Software Engineering Papers: Volume VI.*

SCA87 Scanlan, D. A. 1987. "Data-structure students may prefer to learn algorithms using graphical methods," *Proceedings of the Eighteenth SIGCSE Technical Symposium on Computer Science Education*, pp. 302-307.

SCA88 Scanlan, D. A. 1988. "Should short, relatively complex algorithms be taught using both graphical and verbal methods?: six replications," *Proceedings of the Nineteenth SIGCSE Technical Symposium on Computer Science Education*, pp. 185-189.

SHA88 Sharp, H. 1988. "KDA - A Tool for Automatic Design Evaluation and Refinement using the Blackboard Model of Control," *Proceedings of the 10th International Conference on Software Engineering*, pp. 407-416.

SHE79 Sheppard, S. B., Curtis, B., Milliman, P., and Love, T. 1979. "Modern coding practices and programmer performance," *IEEE Computer*, No. 12, pp. 41-49.

SHE81 Sheppard, S. B., Kruesi, E., and Curtis, B. 1981. "The effects of symbology and spatial arrangement on the comprehension of software specifications," *Proceedings: The Fifth International Conference on Software Engineering*, IEEE, pp. 207-214.

SHN76 Shneiderman, B. 1976. "Exploratory experiments in programmer behavior," *International Journal of Computer and Information Sciences*, No. 5, pp. 123-143.

SHN77a Shneiderman, B. 1977. "Measuring computer program quality and comprehension," *International Journal of Man-Machine Studies*, No. 9, pp. 465-478.

SHN77b Shneiderman, B., Mayer, R., McKay, D., and Heller, P. 1977. "Experimental investigations of the utility of detailed flowcharts in programming," *Communications of the ACM*, No. 20, pp. 373-381.

SHN82a Shneiderman, B. 1982. "Control flow and data structure documentation: two experiments," *Communications of the ACM*, No. 25, pp. 55-63.

SHN82b Shneiderman, B. 1982. "How to design with the user in mind," *Datamation*, No. 4, pp. 125-126.

SHO83 Shooman, Martin L. 1983. *Software Engineering: Design, Reliability and Management*, McGraw-Hill Book Company, New York.

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

SIM73 Sime, M. E., Green, T. R. G., and Guest, D. J. 1973. "Psychological evaluation of two conditional constructions used in computer language," *International Journal of Man-Machine Studies*, No. 5, pp. 105-113.

SOL84 Soloway, E., and Ehrlich, K. 1984. "Empirical studies of programming knowledge," *IEEE Transactions on Software Engineering*, SE-10, pp. 595-609.

STA86 Stark, M. 1986. *Abstraction Analysis: From Structured Specification to Object-Oriented Design*, unpublished GSFC report.

STE74 Stevens, W., Myers, G., and Constantine, L. 1974. "Structured Design," *IBM System Journal*, Vol. 13, No. 2, pp. 115-139.

TEI77 Teichroew, Daniel and Hershey, E. A., III. 1977. "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January, pp. 41-48.

TRI88 Tripp, Leonard L. 1988. "A Survey of Graphical Notations for Program Design – An Update," *ACM SIGSOFT Software Engineering Notes*, Vol. 13, No. 4, pp. 39-44.

URB85 Urban, S. D., Urban, J. E., and Dominick, W. D. 1985. "Utilizing an Executable Specification Language for an Information System," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 7, July, pp. 598-605.

VES86 Vessey, I. and Weber, R. 1986. "Structured Tools and Conditional Logic: An Empirical Investigation," *Communications of the ACM*, January, Vol. 29, No. 1, pp. 48-57.

WAD88a Waddel, K. C. and Cross, J. H. 1987. "Empirical Evaluations of Graphical Representations For Algorithms," *Proceedings of the 26th Annual ACM Southeast Regional Conference*, pp. 496-502.

WAD88b Waddel, K. C. and Cross, J. H. 1988. "Survey of Empirical Studies of Graphical Representations For Algorithms," *Proceedings of the 1988 ACM Sixteenth Annual Computer Science Conference*, p. 696.

WAG88 Wagner, J. 1988. "Graphic Computer-Aided Reverse Engineering (CARE)," *2nd International Conference on Computer-Aided Software Engineering*, pp. 15-28 to 15-32.

WAR86 Ward, P. T. 1986. "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February, pp. 198-210.

WRN74 Warnier, Jean Dominique. 1974. *Logical Construction of Programs*, Van Nostrand Reinhold Company, New York.

WRN81 Warnier, Jean Dominique. 1981. *Logical Construction of Systems*, Van Nostrand Reinhold Company, New York.

WAT85 Waters, R. C. 1985. "The Programmer's Apprentice: A Session with KBEmacs," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11, November, pp. 1296-1320.

WEL85 Welch, P. H. 1985. "Structured Tasking in Ada?", *Ada Letters*, Vol. 5, No. 1, 1-17 to v.1-31.

WHE86 Wheeler, T. J. 1986. "An Example of the Developer's Documentation for an Embedded Computer System written in Ada (Part II)," *Ada Letters*, Vol. VI, No. 6, 1-40 to 1-48.

WIE86 Wiedenbeck, S. 1986. "Processes in computer program comprehension," *Empirical Studies of Programmers: First Workshop*, pp. 48-57.

WIN86 Winters, E. 1986. "Requirements Checklist for a System Development Workstation," *ACM SIGSOFT Software Engineering Notes*, Vol. 11, No. 5, October, pp. 57-62.

WRI73 Wright, P., and Reid, F. 1973. "Written information: some alternatives to prose for expressing the outcomes of complex contingencies," *Journal of Applied Psychology*, No. 57, pp. 160-166.

YAU86 Yau, S. S. and Tsai, J. J.-P. 1986. "A Survey of Software Design Techniques," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 6, June, pp. 713-721.

YOU75 Yourdon, Edward and Constantine, L. 1975.*Structured Design*, Yourdon Inc., New York, New York.

YOU78 Yourdon, Edward and Constantine, L. 1978.*Structured Design*, Yourdon Press, New York, New York.

ZAV86 Zave, Pamela and Schell, W. 1986. "Salient Features of an Executable Specification Language and Its Environment," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February, pp. 312-325.

Appendix A

AN EMPIRICAL EVALUATION OF GRAPHICAL REPRESENTATIONS FOR ALGORITHMS

Kathryn C. Waddel

Auburn University
Auburn, Alabama

I. INTRODUCTION

Graphical representations for algorithms (GRAs) have been available to practitioners as comprehension aids since the introduction of the flowchart in 1947 by Goldstein and Von Neumann. Since then, many others (Figure 1) have followed including the Nassi-Shneiderman chart (Nassi and Shneiderman 1973), the Warnier-Orr diagram (Orr 1977), the action diagram (Martin and McClure 1985), and the control structure diagram (Cross 1988). Tripp (1988) provides a concise survey of 18 additional GRAs introduced since 1977. The use of GRAs has experienced somewhat of a revival due to the availability of high-density, bit-mapped graphics. As a result, GRAs are making their way into computer-aided software engineering (CASE) tools.

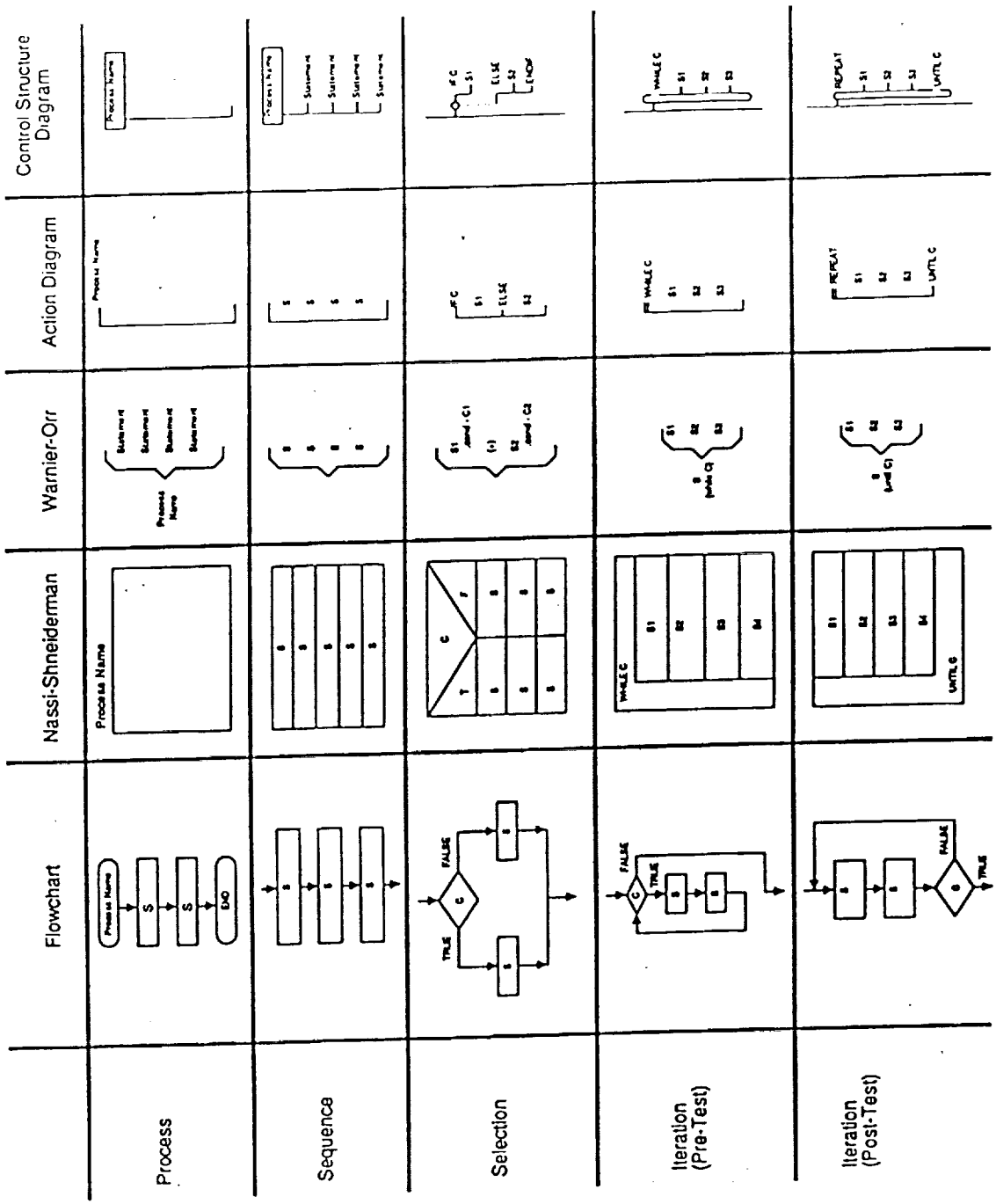


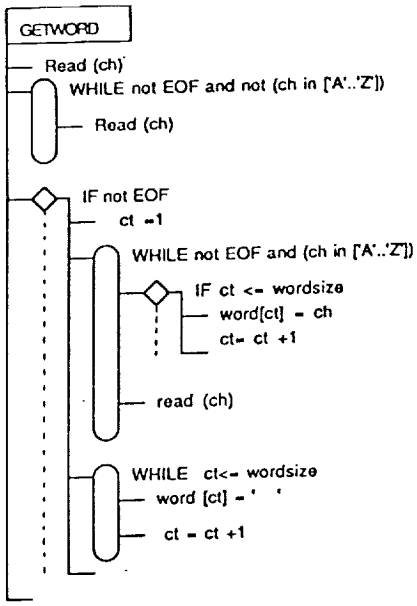
Figure 1. Control Constructs for Some Algorithmic Diagrams.

Various empirical studies have been conducted to determine the effectiveness of GRAs on the process of comprehension. Unfortunately, most of these studies have focused only on the flowchart and have produced mixed results about its effectiveness. Several conclusions can be drawn from these studies: (1) a picture with text may be more useful than text alone, (2) the flowchart may have use in non-programming applications such as the use of a correct flowchart in procedural tasks, but its use in programming applications is questioned, (3) in programming applications the utility of the flowchart may depend upon the task in which it is used and the particular strategy employed and (4) more empirical research is needed to determine the effectiveness of the other graphical notations.

It would be beneficial to both designers and users of software tools to know which, if any, of the graphical representations is the most easily comprehended. In particular, a comprehensive study of this nature would provide developers and users of CASE tools, which rely heavily on graphical representations of software, an empirical basis for the selection of these notations. Professional programmers and students who use control flow diagrams in their work would benefit in using a graphical representation which had been shown by empirical research to enhance understanding. Finally, since maintenance consumes up to 70% to 90% of the total life cycle cost of

software and 50% to 90% of maintenance is understanding the software, the cost of understanding the code could very well be the single most expensive part of the entire software life cycle (Standish 1984). Thus, the use of a tool that has been shown to reduce the time required for comprehension of software could have a significant impact on its overall cost. Any empirical research seeking to find notations which can minimize the cost of understanding software is both necessary and cost-effective.

An empirical study has been completed which compared the comprehensibility, efficiency, and user preference of the conventional flowchart and the control structure diagram (CSD). Pseudocode, a representation which is synonymous with program design language (PDL) or structured English (Pressman 1987), was included in the study to provide a baseline for comparison (Figure 2). Though non-graphical in nature, it will be referred to as a GRA for simplicity in this text, since the three are to be compared in the same manner.



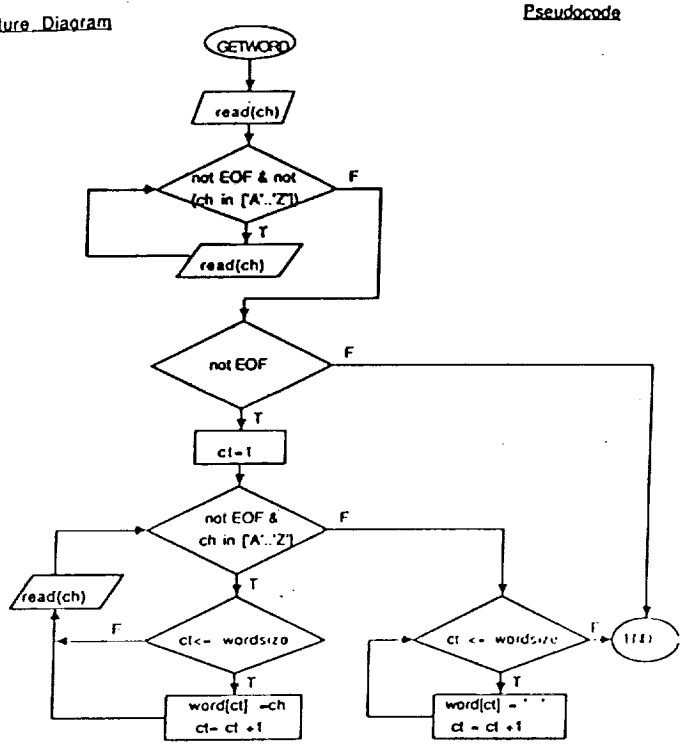
Control Structure Diagram

```
Procedure GETWORD
BEGIN
    Read (ch);
    DO WHILE not EOF and not (ch in ['A'..'Z'])
       Read (ch);
    ENDDO

    IF not EOF
       THEN BEGIN
            ct=1;
            DO WHILE not EOF and (ch in ['A'..'Z'])
               IF ct <= wordsize
                THEN BEGIN
                     word[ct]=ch;
                     ct=ct +1;
                END
               ENDIF
               read(ch);
            ENDDO

            DO WHILE ct <= wordsize
               word[ct]=' ';
               ct=ct + 1;
            ENDDO
       END
    ENDIF
END
```

Pseudocode



Conventional Flowchart

Figure 2. Comparative Diagram of the Control Structure Diagram (CSD), Pseudocode (PDL), and the Flowchart.

II. STATEMENT OF THE PROBLEM

Research Theory and Thesis

The research theory revolved predominantly around the comprehension portion of the project. It was surmised that the comprehension of a program involves an iterative cycle of expectation and hypothesis of its function, revision of the hypothesis, then verification of this hypothesis. This is all accomplished through close scrutiny of the code. Aiding in this cyclical process are beacons, or key lines in the code, which act as clues to the function of the program. Also aiding this process is the inclusion of keywords, lines, meaningful shapes, white space, and indentation, which all contribute to the combining of the program into functional chunks. The main proposal of this study is that the comprehensibility and readability of a GRA are directly linked to how well the GRA aids the chunking process.

Pseudocode (PDL) has unique characteristics: indentations, a concise, linear format much like code, and capitalized keywords, which contribute to its readability. The flowchart is unique with its graphical symbols which clearly show looping and branching, but it is believed that the flowchart could be less comprehensible than PDL because

of the space it requires. While nested loops in PDL can be indented to the right and down in a linear fashion, similar loops and branches in the flowchart can extend in four directions and thus have the potential for causing confusion to the reader when the drawing is continued on a separate page. It was proposed that the CSD would be the most comprehensible of the three notations because it incorporates the linear, textual structure of the PDL which is compact and concise, and easily extends to another page, as well as the graphical nature of the flowchart. The CSD contains special symbols which represent certain constructs in the program; these symbols are similar to the flowchart but are uniform in size so require less room than the flowchart symbols. Also, the CSD extends down and to the right as does PDL, and doesn't require the two-dimensional space needed for the flowchart. Finally, the CSD exploits redundancy since it uses both graphical and textual formats, which would benefit both left and right-brained individuals.

It was also proposed that the CSD and PDL would be the most preferred notations, for the same reasons cited above. It was predicted that users would prefer these notations over the flowchart since they are both linear and top-down in nature, and translate into code more readily.

Thus, the main purpose of the research was to discover which of the three notations: the CSD, conventional flowchart, and pseudocode (PDL) best aids in the chunking

process, a process which translates into understanding of the program. This was measured by a comprehension test, where each subject was shown three algorithms in one of the three notations and answered questions about the function of the algorithms. This measured accuracy of the notations. Also measured was the efficiency of the notations, that is, the extent to which the notation expedited understanding. This was measured by the response time the subjects used to answer the questions and thus all questions were timed. Finally, preference for the notations was measured by means of a short preference survey, where the subject rated the three notations for use in specific tasks.

Goals of the Research

There were four basic goals and associated tasks in this research project. The first was to determine which, if any, of pseudocode (PDL), the conventional flowchart, or the CSD was the most easily comprehended and useful as a debugging aid by novice programmers (less than a year of programming experience), intermediate programmers (one to three years experience), and advanced programmers (three to five years of programming experience). Two measures were observed: efficiency and accuracy of the responses. If the PDL was found to be the most comprehensible, then perhaps the utility of diagrammatical notations in general needs to be reevaluated. If either of the graphically oriented

notations were shown to be more comprehensible, there will be obvious implications for both the computer education community and the professional software community.

The second objective was to determine if there was any difference between the three experience groups in levels of comprehension of the three notations. Novice programmers have not yet attained the skills necessary to efficiently debug or modify a program. It was believed valuable to determine the difference in the error rate between the novice, intermediate, and advanced programmer groups and note the differences across all three notations. Perhaps it would be found that one representation is more easily comprehended by the intermediate and advanced levels but is a hindrance to novices.

The third goal was to observe how each experience level scored in accuracy and efficiency for each of the two types of programming tasks: debugging and general comprehension. The first group of questions in the comprehension section were concentrated on the discovery of bugs in the algorithms; the rest were general flow of control questions. Perhaps one notation would result in better scores on the debugging questions than the others, while another notation would improve general comprehension. These are two programmer tasks which require separate observation.

The next goal was to find the preferred diagrammatical notation between the novice, intermediate, and advanced

groups, via a preference survey. It was valuable to determine which notation was the most preferred, since this notation would be the one most readily used by programmers. Preference was measured in terms of the task for which the notation was to be used, as subjects may have preferred to use one notation for one purpose, but another for a different purpose.

Finally, the preference data and the accuracy/efficiency data were compared. Perhaps the most preferred notation was also the most useful.

III. THE PROJECT

The Automated Instruments

Each subject from a sample of students was given a brief automated summary on the use of one of the notations, followed by an automated test (approximately 70 minutes). This was implemented on IBM-compatible personal computers and required an enhanced graphics adapter (EGA) card, at least 360K RAM memory, and two megabytes of hard disk space. The test contained three algorithms, each representing three difficulty levels: easy, moderate, and difficult. Each test represented the three algorithms in one of the following formats: the conventional flowchart, the control structure diagram (CSD), or pseudocode (PDL). Thus, a given subject observed the three algorithms in one graphical format, a repeated-measures experimental design. The order of all three was randomized among subjects seeing a certain notation. Each algorithm had a number of bugs seeded in it and the subject was to determine the exact nature and location of the bugs. Questions about flow of control followed. All questions were multiple-choice with five candidate responses and each question/response was timed. Following the comprehension test, there was a short

preference survey which included a brief explanation of the other two notations and how they show sequence, selection, and iteration. The subject was asked to select and rate which of the notations he would prefer to use in a number of programming situations. Throughout the session, the subject was given explicit instructions for taking the test. Except for the beginning section which queried the subject for background information, all parts of the instrument were in graphics mode. There were three instruments, one for each graphical representation. Each was written in Turbo Pascal[1] with support for the drawings provided by the Turbo Pascal Graphix Toolbox. Statistical tests were conducted on all the data to determine the significance of the results.

The Subjects

A total of 154 students were tested for this experiment: 22 were tested for the preliminary study, 132 for the main study. One hundred and twenty-nine were Auburn students enrolled in the computer science and engineering department; 15 more were enrolled in Auburn's chemical engineering department, and 10 were from Clemson University. Because the instrument required that all subjects have Pascal knowledge, many of the novice programmers had to be tested at the end of the quarter to ensure that they would

[1] Turbo Pascal is a registered trademark of Borland International.

have the experience needed to participate in the experiment.

Three levels of experience among the students were chosen, the same ones recommended by Shneiderman (1976b): novice programmers (less than one year of programming experience), intermediate programmers (1 to 3 years of experience) and advanced programmers (more than 3 years of experience). The subjects were acquired with the help of faculty in 14 courses offered in the computer science and engineering departments at Auburn and Clemson, and one course in the chemical engineering department.

The novice programmers (mostly sophomores) came from beginning Pascal or PL/1 courses (Pascal is a prerequisite to the PL/1 course at Auburn). The intermediate programmers were students in the data structures, software engineering, and algorithms courses. Advanced programmers (mostly seniors) came from the artificial intelligence, C programming, and compiler courses.

The sessions were somewhat long in that they ranged from 60-90 minutes in length. Any subject was stopped after he had been tested for 90 minutes, whether he was finished or not. Data which was not gathered beyond this time was later changed to a '.' value so that SAS would read it as missing data.

The Experiment

After some refinements to the instruments, the preliminary study was conducted. Twenty-two novice programmers from the PL/1 course were tested in March, 1989 in the Haley Center microcomputer laboratory at Auburn University. Afterwards, each subject was asked to fill out an evaluation of the instrument by noting its strengths and weaknesses with respect to the clarity of instructions, readability of the drawings and text, readability and logic of the questions, ease of use of the keys to input choices, etc. Minor changes were made to the instrument before commencing with the main study.

A sample size of 137 was determined from a formula for case I research for one-tailed tests (R.L. Shavelson, 1988). The main study began on April, 1989 and ran through July, 1989. This study consisted of testing 132 students: 56 novices, 33 intermediates, and 43 advanced student programmers. Testing was conducted at Auburn's Haley Center and Tichenor PC laboratories, and Clemson University. In this study, testing of two novice classes was delayed until the last day of the quarter so that they would have enough experience and knowledge to participate. It was felt that the longer the wait before testing them, the closer they would be in ability to the novices in the PL/1 classes.

IV. STATISTICS AND RESULTS

The Results of the Comprehension Test

Overall GRA Accuracy and Efficiency

Because the data contained missing values, a PROC GLM (as opposed to a MANOVA) was conducted to see if there was a significant difference in GRAs with respect to accuracy, a variable used to measure comprehensibility and readability, and efficiency of response times. A tail probability of 0.05 was used as the cutoff point, so values less than 0.05 would indicate a difference in GRA populations. Table 1 shows that there was no significant difference ($p<0.1671$) between GRAs in accuracy for the entire data set.

Table 1. Overall Mean Accuracy (Number of Answers Correct for All Three Algorithms) for the Entire Data Set.

	n=48 CSD	n=42 FC	n=42 PDL	Tail Prob.	GRA Favored
mean accuracy	8.83	9.50	10.26	0.1671	none

Likewise, there was no significant difference ($p<0.2824$) in efficiency between the GRAs, shown in Table 2.

Table 2. Overall Mean Efficiency (Response Time in Minutes for All Three Algorithms) for the Entire Data Set.

	n=48 CSD	n=42 FC	n=42 PDL	Tail Prob.	GRA Favored
mean efficiency	51.24	56.52	55.93	0.2824	none

A PROC GLM was run again, this time observing the accuracy and efficiency of the GRAs for each subject experience level. In this case, some differences emerged, as shown in Table 3. For the advanced subjects, accuracy scores were significantly better ($p<0.0207$) for the flowchart and PDL, indicated by Duncan's Multiple-Range Test. The Duncan test also detected a difference in the

novice accuracy scores, which favored PDL and the CSD. Although the tail probability was $p<0.0831$, this was an indication that a difference could emerge if a larger sample size of novices was tested. There was no difference among intermediate subjects.

Table 3. Overall Mean Accuracy (Number of Answers Correct for all Three Algorithms) by Experience Level.

	CSD	FC	PDL	Tail Prob.	GRA Favored
Novices (56)	7.50	6.76	8.95	0.0831	none
Intermed.(33)	9.75	10.78	9.67	0.7563	none
Advanced (43)	9.81	11.69	13.18	0.0207	PDL,FC

There was a strong difference in efficiency (Table 4) for the novices ($p<0.0149$). A Duncan test favored the CSD in efficiency, as the reponse times in minutes were much lower for the CSD than for the other two GRAs (keep in mind that lower response times are favorable). There was no difference in efficiency for the intermediate and advanced groups.

Table 4. Overall Mean Efficiency (Response Time in Minutes for All Three Algorithms) by Experience Level.

	CSD	FC	PDL	Tail Prob.	GRA Favored
Novices (56)	41.40	53.32	56.47	0.0149	CSD
Intermed.(33)	56.76	52.42	50.33	0.6900	none
Advanced (43)	59.40	62.23	61.13	0.8705	none

Accuracy and Efficiency of the Algorithms

Accuracy

Since one of the big problems of composing a comprehension test of this nature is in the selection of the algorithms to use, there was concern about how they fared in accuracy and efficiency. Figure 3 shows the overall mean accuracy for each algorithm. Fortunately, the means show the desired trend: the easy algorithm had the best scores, while the difficult had the worst. The mean for the moderate algorithm indicated that it was perhaps too difficult, as it rivalled the mean for the difficult one.

Figure 4 looks at overall mean accuracy of the GRAs with respect to the algorithms. There was one significant difference in the GRAs; for the easy algorithm, accuracy scores were better for the flowchart and PDL. This is substantiated by the tail probabilities in Table 5. It is interesting to note that while the flowchart had the highest

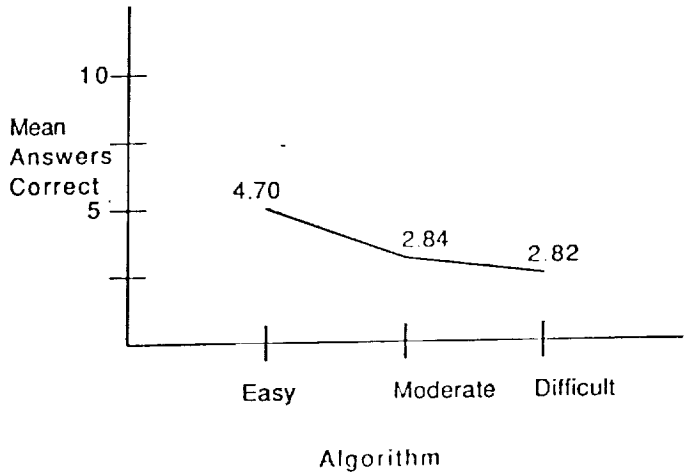


Figure 3. Overall Mean Accuracy, by Algorithm, for the Entire Data Set.

accuracy scores on the easy algorithm, it had the lowest GRA scores (though not significantly lower) on the moderate and difficult algorithms.

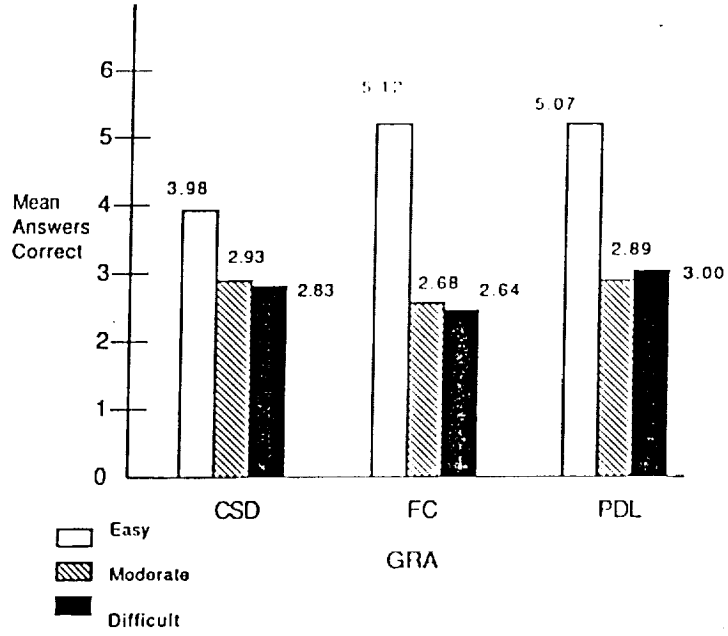


Figure 4. Overall Mean Accuracy (Number of Answers Correct), by GRA and Algorithm, for the Entire Data Set.

Table 5. Probability of Accepting H_o, That the GRAs Do Not Differ in Accuracy, by Algorithm, for the Entire Data Set.

	Tail Prob.	GRA Favored
Easy	0.0414	FC and PDL
Moderate	0.6561	none
Difficult	0.4409	none

The data for the algorithms were analyzed again but this time were observed by subject experience level. Figure 5 shows the mean accuracy scores, by algorithm, for each of the three experience levels. As was

expected, the advanced students did consistently better than the intermediates and novices for all three algorithms. The intermediates did consistently better than the novices, except on the difficult algorithm. Here, the novices did slightly better.

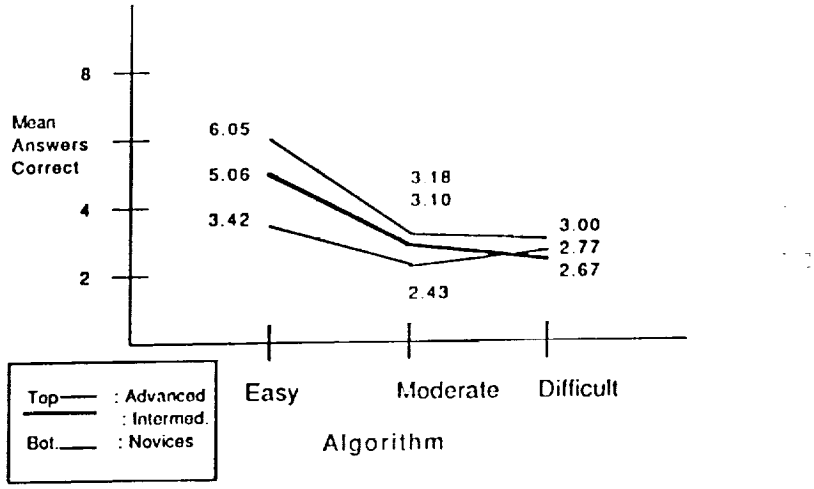


Figure 5. Mean Accuracy (Number Answers Correct), by Algorithm and Experience Level.

With the exception of the novice group, for all levels the desired trend was apparent: the easy algorithm resulted in better accuracy scores than the moderate, which had better scores than the difficult algorithm.

The only algorithm/experience level combination which showed a significant difference (Table 6) in GRAs was that of the easy algorithm for the novice group ($p<0.0520$).

A Duncan test showed that for this group, the flowchart and PDL scores were significantly higher than the CSD scores.

Table 6. The Probability of Accepting H_o, That the GRAs Do Not Differ in Accuracy, by Algorithm and Experience Level.

	N=56 Novices	N=33 Intermed.	N=43 Advanced
Easy	0.0520	0.8454	0.1763
Moderate	0.3490	0.2542	0.6235
Difficult	0.4286	0.6770	0.1991

Efficiency

As was done for accuracy, a PROC GLM was conducted on efficiency of the algorithms. Mean efficiency was actually the mean response time in minutes for all ten answers for an algorithm. Overall mean efficiency is shown in Figure 6. As was expected, the easy algorithm had the lowest response times, but the moderate algorithm earned slightly higher response times than the difficult algorithm. Like the accuracy scores, this suggests that the moderate algorithm was too difficult.

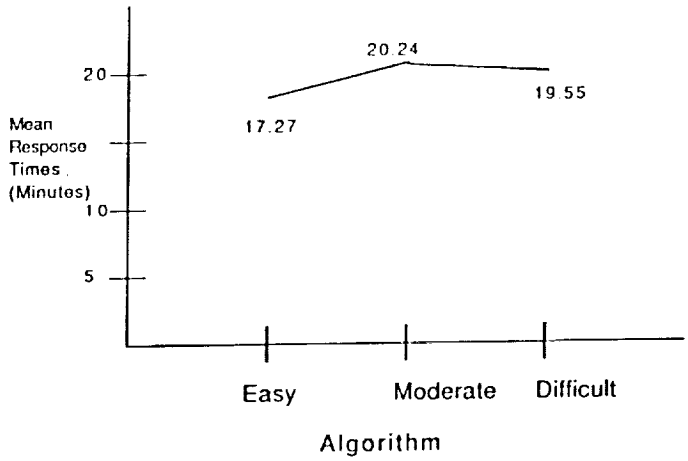


Figure 6. Overall Mean Efficiency (Response Time in Minutes), by Algorithm, for the Entire Data Set.

None of the algorithms showed any difference between GRAs in efficiency, as presented in Table 7. Figure 7 shows response times, by GRA. Although there are visual differences between the GRAs, there was no statistical difference between them.

Table 7. The Probability of Accepting H_o, That the GRAs Do Not Differ in Efficiency, by Algorithm, for the Entire Data Set.

	All Three Levels	GRA Favored
Easy	0.2108	none
Moderate	0.7634	none
Difficult	0.8994	none

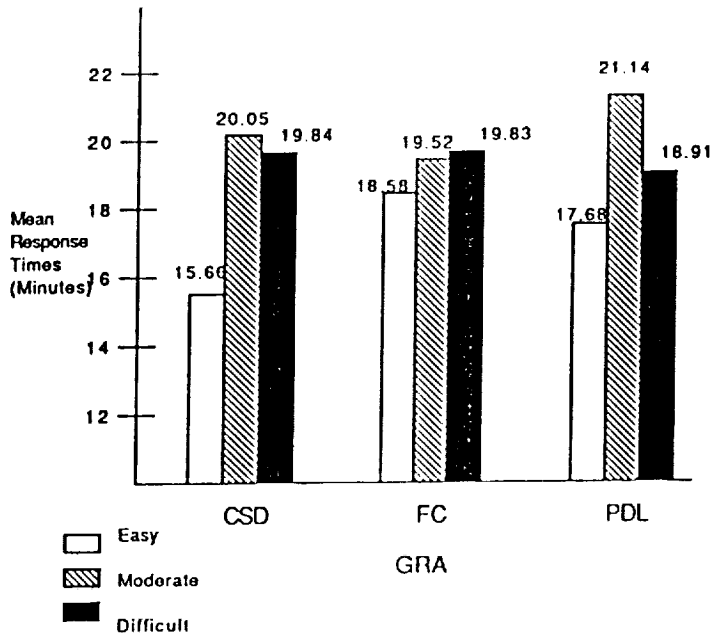


Figure 7. Overall Mean Efficiency (Response Time in Minutes), by GRA and Algorithm, for the Entire Data Set.

The response times by experience level were revealing, as shown in Figure 8. The advanced subjects consistently

took a longer time to respond for all three algorithms than did the novice or intermediate groups. This paid off, as their accuracy scores were higher (see Figure 5). This trend changed for the moderate algorithm where the novices spent more time than the intermediates in responding to the answers but their scores were lower. So, for the novice and intermediate groups, greater response time did not necessarily equal better accuracy; experience and familiarity with the algorithm was also needed for the accuracy to improve.

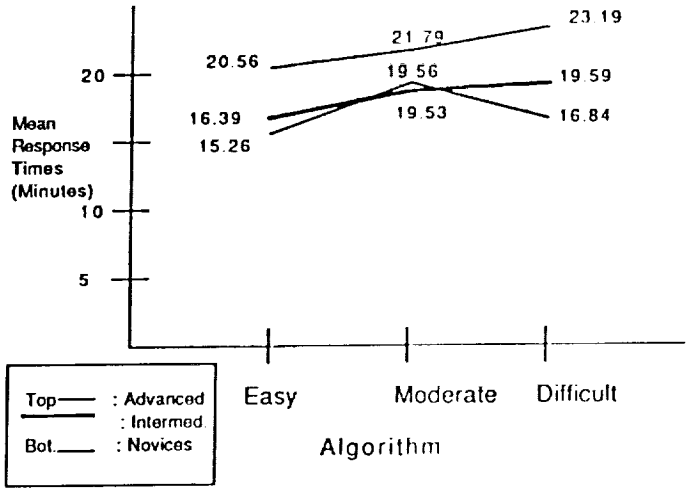


Figure 8. Mean Efficiency (Response Times in Minutes), by Algorithm and Experience Level.

Table 8. The Probability of Accepting H_o, That the GRAs Do Not Differ in Efficiency, by Algorithm and Experience Level.

	N=56 Novices	N=33 Intermediates	N=43 Advanced
Easy	0.0364	0.8478	0.8341
Moderate	0.1336	0.3124	0.7879
Difficult	0.5078	0.2717	0.8848

The Relationship of Accuracy or Efficiency to Experience

Table 9. Means of Subject GPA and Experience (in Months) by Experience Level.

	Novice	Intermed.	Advanced
GPA	2.97	3.00	3.17
CSD Exper.	0.04	1.48	7.29
FC Exper.	7.84	17.48	27.69
PDL Exper.	6.09	17.00	34.14
Pascal Exper.	4.13	21.82	31.00

Table 9 summarizes subject background, based upon experience level; this information was obtained from the subject prior to taking the test. GRA and Pascal experience are measured by months of continuous use. As shown, there is a consistent upward trend of experience and GPA (grade point average) from the novices to the advanced students. Intermediates have about 10 to 17 more months of experience

than the novices, while the advanced have 10 to 17 more than the intermediates. Also noteworthy is the knowledge of the CSD relative to knowledge of the other GRAs and Pascal; the CSD is relatively unknown. This information, though basic, will help determine the extent by which experience and GPA affects accuracy and efficiency.

A correlation was done on the data to determine if there existed a relationship between accuracy or efficiency scores and subject experience in using the GRAs and Pascal and subject grade point average (GPA). There was concern that scores would reflect a bias for the flowchart or PDL, since those two representations are so heavily used in Auburn's computer science curricula.

According to Shavelson (1988), a correlation above 0.1946 is significant for a sample size of 132. As is shown in Table 10, there were many strong correlations in the data. Increased response times correlated with high accuracy scores, which substantiates previous findings, that in most cases, the longer a student spent on a question, the higher the score. High accuracy scores related strongly to a high GPA, PDL experience, and Pascal experience. High accuracy scores did not relate to flowchart or CSD experience.

Increased response times (efficiency) correlated with greater flowchart experience, so the more a student knew the flowchart, the slower his response times were likely to be.

Efficiency was not affected by student GPA or PDL, CSD, and Pascal experience.

There was a strong correlation between flowchart, PDL, CSD, and Pascal experience. PDL and the CSD are very similar in nature. Both were used with Pascal code in this test, which suggests the close relationship between the three. Also, many Auburn students learn the flowchart when they learn either PDL, Pascal, or the CSD. GPA was not affected by GRA experience.

An interesting phenomenon was the role the prologue played in the evaluation. The prologue was the first notation summary the subject saw, the aim being to familiarize him with the notation so he was more successful during the comprehension test. As shown, 'prolog' was negatively correlated with accuracy and close to being positively correlated with high response times. This means that the longer a subject spent reading the summary, the worse the accuracy scores and the longer the response times.

The main question was answered by this correlation: the fact that bias was a factor in the study, at least with PDL and the flowchart. The more a student knew PDL or the flowchart, the more this effected his response time or responses. More PDL knowledge lead to better accuracy scores, while more flowchart knowledge lead to longer response times and therefore less efficiency.

Table 10. Pearson Correlation Coefficients Showing the Relationship Between Accuracy or Efficiency and Subject Experience.

	Effic.	GPA	FC Exp.	PDL Exp.	CSD Exp.	Pascal Exp.	Prolog Exp.
Acc.	.3114*	.3287*	.1821	.3198*	.1023	.4143*	-.2232*
Eff.		.0303	.2036*	.0774	.1700	.0829	.1924
GPA			-.0412	.1082	-.0196	.0443	-.0028
FC Exp.				.6224*	.4307*	.4944*	-.0825
PDL Exp.					.3571*	.6283*	-.2252*
CSD Exp.						.2892*	-.0497
Pas Exp.							-.1686

(* indicates a significant correlation)

The Results of the Preference Survey

There were two questions which needed to be answered by the preference survey: (1) which GRA did the students prefer as a whole and (2) which GRA did each experience level prefer?

It was found that there was strong GRA preference; very few questions yielded no preference at all. Table 11 shows the mean preference scores for each GRA, the tail probability (from a PROC GLM) indicating if a preference was shown or not, and what the Duncan test revealed. Most of the 10 questions resulted in strong differences with the exception of overall readability (question 4) and how well the GRA shows sequence (question 1). Question 3 resulted in a borderline value ($p<0.0904$) which picked up a difference in the Duncan test, favoring PDL and the CSD. Looking at the score, overall preference seemed to be for the PDL; this

was verified by the result of question 10. It is noteworthy that the CSD ranked so highly, although it was the least known of the three notations.

Table 11. Overall Preference, by Task, for the Entire Data Set. (1= Least Useful, 5= Most Useful).

Task	CSD	FC	PDL	Tail Prob.	GRA Favored
1	4.49	4.28	4.44	0.3386	none
2	3.40	4.34	3.98	0.0001	FC
3	3.84	3.68	4.07	0.0904	none
4	4.07	4.17	4.13	0.8258	none
5	3.40	4.07	4.00	0.0001	FC,PDL
6	3.33	3.11	4.24	0.0001	PDL
7	3.73	3.05	3.79	0.0001	CSD,PDL
8	3.43	3.35	4.05	0.0012	PDL
9	3.59	3.93	3.43	0.0165	FC,CSD
10	3.32	3.50	4.00	0.0022	PDL

Where:
1= How well GRA shows sequence
2= How well GRA shows selection
3= How well GRA shows iteration
4= Overall readability
5= Ease of coding from GRA specifications
6= Ease of manual use of GRA
7= Presentation of source code of GRA by prettyprinter
8= Use of GRA in CASE editor
9= How GRA aids comprehension in an automated instrument
10= Overall preference

Table 12 shows a similar analysis by experience level. Although the CSD resulted in better efficiency among novices, they had a strong preference for the flowchart and PDL. The intermediates, who knew the CSD best of all three

levels, were more undecided about GRA preference. Most of their tail probabilities did not reveal a difference; when they did, all three GRAs were preferred for one of the tasks each. The advanced subjects liked the CSD and PDL most of all, which is interesting since they had the greatest flowchart experience.

It was expected that preference for a GRA would relate to experience with that GRA, but this bias did not pervade the study. Correlations between GRA preference and experience were weak. The tail probabilities were: $p<-0.0444$ for PDL preference and PDL experience, $p<0.0742$ for CSD preference and CSD experience, and $p<0.1000$ for flowchart preference and flowchart experience. Preference then had little to do with past experience, so familiarity with PDL, the CSD, or the flowchart did not bias the preference results. It should be added that these correlations looked at the overall sample of subjects and not at each experience level.

Table 12. Preference, by Task, for Each Experience Level. (1= Least Useful, 5= Most Useful).

	Task	CSD	FC	PDL	Tail Prob.	GRA Favored
	1	4.63	4.57	4.63	0.9389	none
	2	3.40	4.20	4.27	0.0077	PDL,FC
N	3	3.40	3.63	4.30	0.0166	PDL
O	4	3.73	4.23	4.30	0.0744	none
V	5	3.17	4.00	4.03	0.0049	PDL,FC
I	6	3.30	3.53	4.27	0.0042	PDL
C	7	3.33	3.30	3.83	0.1661	none
E	8	3.10	3.70	4.00	0.0348	PDL,FC
S	9	3.23	4.00	3.70	0.0390	FC,PDL
	10	2.87	3.60	4.27	0.0002	PDL
I	1	4.40	3.80	4.20	0.2922	none
N	2	3.40	4.60	3.53	0.0116	FC
T	3	4.20	3.67	3.87	0.4924	none
E	4	4.20	3.87	3.80	0.5922	none
R	5	3.40	4.13	3.80	0.1812	none
M	6	3.73	2.60	4.27	0.0010	PDL,CSD
E	7	3.87	3.07	3.67	0.2491	none
D	8	3.73	2.93	3.93	0.1253	none
.	9	3.53	3.80	3.00	0.1266	none
	10	3.60	3.67	3.53	0.9632	none
	1	4.41	4.24	4.38	0.7367	none
A	2	3.41	4.35	3.92	0.0005	FC,PDL
D	3	4.05	3.73	3.97	0.3526	none
V	4	4.29	4.24	4.14	0.7509	none
A	5	3.59	4.11	4.05	0.0854	none
N	6	3.19	2.97	4.21	0.0001	PDL
C	7	4.00	2.84	3.81	0.0001	CSD,PDL
E	8	3.57	3.24	4.14	0.0102	PDL,CSD
D	9	3.89	3.92	3.38	0.0687	none
	10	3.57	3.35	3.97	0.1071	none

Where:
1= How well GRA shows sequence
2= How well GRA shows selection
3= How well GRa shows iteration
4= Overall readability
5= Ease of coding from GRA specifications
6= Ease of manual use of GRA
7= Presentation of GRA by prettyprinter
8= Use of GRA in CASE editor
9= How GRA aids comprehension automated instrument
10= Overall preference

V. CONCLUSIONS AND FUTURE DIRECTIONS

Summary of Results

Results from the entire data set of 132 observations showed no evidence to believe that the three notations (flowchart, CSD, and PDL) differed from each other with respect to accuracy or efficiency. This changed, however, when subject experience level and algorithm difficulty was observed; as an example, for the whole data set, accuracy scores were higher with the flowchart and PDL, but only for the easy algorithm. There were also strong preferences within each subject experience level.

The advanced students made higher accuracy scores with the flowchart and PDL, although none of the three algorithms alone contributed to this difference; efficiency was equally good for all three GRAs. The advanced students preferred PDL for (1) selection, (2) manual use, (3) use in a prettyprinter, and (4) use in a CASE editor. They preferred the CSD for (1) use in a prettyprinter and (2) use in a CASE editor. The flowchart was preferred for showing selection. Thus, for the advanced group, three conclusions can be drawn: (1) the PDL and flowchart were the most

comprehensible notations overall, (2) none of the notations was the most efficient, and (3) PDL was the preferred notation.

The intermediate subject data showed a certain blandness. Accuracy and efficiency were not affected by any of the notations, and the preference data showed little preference: they preferred the CSD and PDL for manual use and the flowchart to show selection. So, for the intermediates, there was no strong pull for or against either notation for accuracy, efficiency, or preference.

Accuracy scores for the novices showed no differences in the notations, except for the easy algorithm; the flowchart and PDL improved scores. The most dramatic evidence was that the CSD improved novice efficiency, particularly for simpler algorithms. This contrasts with novice preference; they preferred the PDL overall. In particular, they rated it best for (1) showing selection, (2) showing iteration, (3) when coding from PDL specifications, (4) manual use, (5) use in a CASE editor, (6) use in an automated tool, and (7) overall. The novices liked the flowchart for (1) showing selection, (2) used in a CASE editor, (3) when coding from flowchart specifications, and (4) use in an automated tool. Amazingly, the novices did not prefer the CSD for any tasks. Thus, for the novices, three conclusions can be drawn: (1) the CSD was the most efficient notation, (2) the flowchart and PDL were the

most accurate notations, but only for easy algorithms, and (3) PDL was the most preferred notation.

The intermediates did consistently better than the novices for all algorithms, except the difficult one. This is curious since most of the novices had very little experience with the use of pointers, a prominent feature in the difficult algorithm. Also, the intermediate subjects spent more time than the novices responding to the difficult algorithm. One can draw two conclusions: (1) the intermediates had forgotten how to use pointers or (2) the novices were better at quessing the answers than the intermediates for this algorithm.

Means of experience and GPA showed a consistent trend: the advanced subjects had 10 to 17 months more of experience than the intermediates, while the intermediates had 10 to 17 more months than the novices. The advanced students also had higher GPAs than did the intermediates, while the intermediates had higher GPAs than the novices.

Correlations showed that accuracy scores were affected by PDL experience, so bias towards PDL was a factor in this study. Thus, the better scores shown by the advanced PDL group were probably enhanced by their PDL experience. This is plausible, since this group had the most experience with PDL. It is interesting that flowchart experience did not bias accuracy scores. Perhaps the higher accuracy scores

for the flowchart were attributed more to the subject's knowledge of PDL or Pascal.

High accuracy scores related strongly to a high GPA and Pascal experience. This conclusion is straightforward, since Pascal knowledge was a prerequisite to participating in this study, and the more one knew Pascal, the easier the algorithms were to understand. Students with high GPAs would score well in accuracy, because the skill required to understand this test are the same skills required in many of their courses.

There was a strong correlation between GRA experience and Pascal experience, which parallels the fact that students at Auburn learn to use PDL or flowchart along with Pascal. As to preference, notation experience did not seem to affect a subject's preference, so there was not a problem of bias in the preference survey. As an example, the intermediates showed little preference for the CSD but, of the three groups, had the most recent experience with its use.

The advanced students had the highest response times (which equals low efficiency) of the three levels, which improved their accuracy scores. In general, for all the levels, longer response times equalled greater accuracy, except for the medium algorithm with the novices and the difficult algorithm for the intermediates. Also, low efficiency correlated with greater flowchart experience. For

some reason, though flowchart experience did not affect accuracy scores, it did cause higher response times. Perhaps this is true because the flowchart is further from being like code than PDL or the CSD. One can only speculate the cause.

As to the algorithms used in the instruments, the moderate one was too difficult; accuracy scores were as low and response times as high as that of the difficult algorithm. This algorithm implemented a stack to find the way out of a two-dimensional maze. One can only guess the cause of the low scores. Perhaps the notion of a stack to keep track of possible moves was too foreign an idea to subjects.

Implications of the Results

With regards to accuracy, the PDL and flowchart were favored, particularly for advanced subjects (although no one algorithm contributed to the difference) and for novices when the algorithm was easy; other than this, there was no difference in the notations. With regards to efficiency, the CSD was favorable for novices when the algorithm was easy.

In retrospect, it is not surprising that PDL was the most accurate of the notations. First of all, there was bias towards PDL in the data; the subjects had as much PDL experience as flowchart and Pascal experience (in the

advanced group, they had more PDL experience) and, since PDL knowledge correlated with high accuracy, better scores resulted. Secondly, PDL is very much like code. These students are accustomed to looking at code, modifying from it and debugging it. There was a high correlation between Pascal experience and PDL experience. Thus, it follows that the notation which looks most like code would also be the most accurate.

It was a surprise that the flowchart resulted in high accuracy scores for advanced subjects. These scores were not affected by flowchart experience among subjects; however, flowchart experience did correlate with PDL and Pascal experience. Pascal knowledge enhanced accuracy scores; perhaps the flowchart did, too, because Pascal can be translated easily into the flowchart.

Noteworthy is the fact that, despite the experience of some subjects with the CSD in their software engineering courses, the CSD was relatively unknown. It is believed that this factor, unfamiliarity, may have accounted for the favorability towards the flowchart and PDL. The subjects simply had more experience with these two notations, and a five-minute summary of the CSD during the test could not compensate for months of use and familiarity with them.

Despite this handicap, the CSD still did well in expediting novice understanding and shows evidence of being a notation which should be used in education of novices.

Even though PDL and the flowchart may increase understanding, the graphical and textual nature of the CSD made understanding faster and more efficient for novices. These factors should be considered when choosing a GRA to use in novice programming classes. This may also have implications for users of fourth generation languages, especially when communicating with non-computer professionals.

Although the advanced group favored the flowchart and PDL in accuracy, no single algorithm contributed to this difference. There was evidence that, overall, the effectiveness of the notations separated for the easy algorithm and failed to show a difference for the more difficult algorithms. It could well be that in the more complex algorithms, the subjects, particularly the novices, ignored the graphical part of the notation and concentrated more on the text, the Pascal-like code. This supports a conjecture that the notation which most resembles code is the one which is used.

Subjects in this study were accustomed to looking at code, not at GRAs. They create, modify, and debug by looking at the code itself, not a graphical representation of the program. Perhaps the results would have been more definitive had the subjects been accustomed to programs output on a flowchart, CSD, or PDL prettyprinter, or had used the notations extensively as design tools (where the

GRA is developed first, then converted to code, not the other way around, which is so often the case). It was not the case that these subjects had this kind of experience, however, and there is evidence to believe that the graphics were important when the task was easy, but when the task became more complex, the graphics were abandoned for that which the subject knew best: the code itself. This could have been an underlying behavior in the novices and should be looked into further in future research.

Preferences were definitive; PDL was preferred over the graphical representations. Again, the subjects seemed to favor that which is most like code itself. The other two should not be ignored, however, as the graphical notations were preferred for manual use and in automated tools.

Future Directions

The question of usefulness of GRAs is an important one since so many CASE tools are being developed which utilize GRAs and since GRAs are being implemented in program documentation as enhancements to understanding during the software maintenance phase.

Usefulness, as was demonstrated, relies very heavily upon user experience. The actual determination of experience level is probably quite complex. A subject claiming to have 5 months of flowchart experience could actually have had one month of design experience with the

flowchart and four months implementation experience: coding, debugging, and testing. This is especially true with college students who work on relatively small programs and find that they can create a program without a design. The type of experience a programmer has is as important as the length of his experience and should be a consideration in studies such as this.

A programmer/analyst will use that with which he is most familiar. Future GRA studies should bear this in mind. Experimental groups need to be set up months in advance of testing. Subjects need to be selected by experience level, then well-trained in a single notation and provided a rich environment in which to use it. This should be done months prior to the test. It is important that the subjects can 'think' in the GRA before the test. Subjects which have experience in a GRA would be divided into three groups, so only one-third of the GRA group actually is tested in that notation. An evaluation such as this might reveal GRA differences. Also, the effects of using GRAs in large programs rather than "textbook programs", needs to be evaluated. In particular, future empirical studies should focus on the robust interactive environment which characterizes present and future CASE tools.

It is hoped that human-factors research (and GRA research) is continued. The possibilities are limitless and, as the relationship between man and machine gets closer, the

implications become more important. However, evaluations of this kind are tricky and require care in selecting the proper human subjects and care in putting together the testing instruments. It is sincerely hoped that, if nothing else, this thesis has spawned within the reader new ideas for evaluating programmer behavior and an insight into how to accomplish this fascinating and ever-changing task.

BIBLIOGRAPHY

Cross, J.H. & Sheppard, S.V. (1988). The control structure diagram: an automated graphical representation for software. Proceedings of the Twenty-First Hawaii International Conference on System Science (January 5-8, 1988), IEEE Computer Society.

Martin, J. & McClure, C. (1985). Diagramming Techniques for Analysts and Programmers. Prentice-Hall, Englewood Cliffs, NJ.

Nassi, I. & Shneiderman, B. (1973). Flowchart techniques for structured programming. SIGPLAN Notices, 8(8), 12-26.

Orr, K.T. (1977). Structured Systems Development. Yourdon Press, New York.

Pressman, R.S. (1987). Software Engineering, A Practitioner's Approach. McGraw-Hill, New York.

Shavelson, R.J. (1988). Statistical Reasoning for the Behaviorial Sciences. Allyn and Bacon, Needham Heights, Massachusetts.

Shneiderman, B. (1976). Human factors experiments for developing quality software. Infotech State of the Art Report on Software Reliability, Infotech International Limited, Berkshire, England, 1977.

Standish, T.A. (1984). An essay on software reuse. IEEE Transactions on Software Engineering, SE-10, 9, 494-497.

Tripp, L.L. (1988). A survey of graphical notations for program design- an update. Software Engineering Notes, 13,4 (October), 39-44.

CASE '89 REPORT
REVERSE ENGINEERING AND MAINTENANCE

James H. Cross II
Auburn University

The goals, approaches, issues, and research areas identified and discussed during the reverse engineering and maintenance sessions held at the Third International Workshop on Computer-Aided Software Engineering, July 17-21, 1989, London, UK (CASE '89) are summarized below. Although software maintenance is an important topic by its own merit, it was considered a subtopic to the reverse engineering of software at this workshop. Thus the ideas presented here are focused on reverse engineering.

An attempt was made to clarify several closely related terms: reverse engineering, re-engineering, restructuring, and reuse.

> *Reverse engineering* is the process of extracting design artifacts and building or synthesizing abstractions which are less implementation dependent. In general, reverse engineering may be considered the front-end to one or more of the following activities.
>
> *Re-engineering* is the process of recasting software into another form which eventually results in an executable product. This normally includes modifications with respect to new requirements not met by the original product.
>
> *Re-structuring* is the process of altering code to attain improved structure in the traditional sense. This is usually done as a form of preventative maintenance and does not normally include major modifications with respect to new requirements.
>
> *Re-use* is the process of identifying, cataloging, and retrieving software components for reuse in another, usually larger, software component.
>
> *Re-documentation* may be considered a weaker form of reverse engineering and normally connotes a predominantly manual approach to recovering previously existing documentation. Whereas, reverse engineering may include abstractions or views of the software not previously available.

GOALS OF REVERSE ENGINEERING

The primary purpose of reverse engineering a software system is to increase the overall comprehensibility of the system. This is reflected in the numerous goals identified by workshop participants as follows:

(1) to develop methods to deal with the shear volume and complexity of the software systems

(2) to generate alternative representations (e.g., complementing graphical representations such as data flow diagrams, structure charts, and control flow diagrams)

(3) to develop automated and/or semi-automated tools and techniques for the recovery of *lost information*

(4) to synthesize extracted information into higher levels of abstraction

(5) to provide information as needed in the form of appropriate levels of abstraction for different categories of users of reverse engineering technologies

(6) to provide a basis for re-engineering and/or re-structuring of software products

(7) to facilitate the identification of software components for re-use

APPROACHES USED FOR REVERSE ENGINEERING

Many approaches which address one or more of the above goals were identified by workshop participants. These are listed below.

(1) Code scanning/parsing - This tends to be the front-end activity to many of other approaches that follow.

(2) System or command level scanning/parsing - This includes analyzing components such as make files.

(3) Analysis of program structure - Included here is the collection of metrics such as levels of hierarchy, levels of complexity with respect to control constructs (especially for the purpose of re-structuring), and cohesion and coupling among modules.

(4) Analysis of data structure - Here the emphasis may be on the construction or reconstruction of a data dictionary (including scoping information and aliases), the identification of data dependency relationships, or front-end for re-structuring data to increase data abstraction.

(5) Abstraction of data - This approach is concerned with the synthesis of more abstract or higher level data structures from existing data structures. These may or may not have been identified in the original design.

(6) Abstraction of design - Here the focus is on synthesis of existing design artifacts (perhaps the source code itself) into a more independent and usually higher level design representation.

(7) Design recovery - Design recovery is closely related to abstraction of design but may emphasize the reconstruction of pre-existing design artifacts.

(8) Domain analysis - The domain of input values, which in turn defines the range of output values, is a candidate for analysis with respect to reverse

engineering of requirements specifications. For example, a software system can be totally defined/specified by enumerating all input/output possibilities.

(9) Language translation - A weak form of reverse engineering is the translation software from one language to another language at the same or different level.

(10) Presentation of other forms of representation - This is related to the language translation approach described above but includes graphical representations in the form of alternative or complementing views as well abstracted views of the software.

(11) Program optimization - The process of program optimization may include automated, semi-automated, and manual elements of reverse engineering.

(12) Identification of reusable components - This provides a basis for leveraging existing software into new products.

(13) Identification of misused items - This approach to reverse engineering emphasizes the a corrective or preventative approach to re-engineering.

ISSUES

Numerous issues were identified that must be addressed in order to achieve the goals outlined above. Many are obvious while others are much more subtle. They are as follows:

(1) Reverse engineering models - Models that capture the nature of the reverse engineering process and methodologies that utilize the models are lacking. For example, elements to include in these model (i.e., what to capture and analyze) must be addressed.

(2) Abstraction/summarization without domain knowledge - This is perhaps the most important and perplexing issue facing reverse engineering researchers. The fact that the code has lost much of the original real-world requirements information probably means reverse engineering cannot be a fully-automated process.

(3) Inclusion of concepts from run-time measurement - While reverse engineering is not normally considered an activity related run-time, some of the concepts from fields such as simulation may be applicable.

(4) Conceptual understanding of software is a psychological process - This is an extremely important issue which is overlooked by many software researchers. However, software psychology is an emerging area that will benefit efforts in reverse engineering. For example, empirical studies are needed to address concepts such as cross-level abstraction versus utility at the same level.

Understandability of re-structured code - Although re-structured code may be better according to some pre-defined metric, experience indicates that the re-structured code may, in fact, be less understandable to those who had been most familiar with it. .

(5) Vendor participation - Most CASE vendors appear not to be actively pursuing reverse engineering. However, the research efforts described in the literature may be a precursor to vendor research and development.

(6) Legal definition - An important issue with potentially far-reaching ramifications is what can and cannot be reverse engineered from a legal perspective (e.g., reverse engineering a product to recover/steal the design from a competitor). Much may depend on the ultimate capability of reverse engineering technology.

RESEARCH IN PROGRESS

Several reverse engineering projects currently in progress were cited at the workshop. These are representative of state-of-the-art efforts in the field.

(1) Auburn University - GRASP/Ada project is focused on the generation of graphical representations at various levels of abstraction (e.g., procedural, architectural and system) from Ada source code.

(2) ESPRIT - REDO is focused on the restructuring of data.

(3) ESPRIT - PRACTITIONER is focused on pragmatic support for re-use of concepts from existing software.

(4) MCC - A large reverse engineering effort is underway which is focused on design recovery.

RESEARCH NEEDED

While many of the issues described above are considered areas for research, workshop participants selected the following as being of special interest.

(1) Deriving abstraction - This is a general term which encompasses much of the essence of reverse engineering.

(2) Design to requirements backtracking including the capture of decisions - The future success of reverse engineering may rely on improved forward engineering CASE tools which explicitly capture requirements/design relationships when they are initially created.

(3) Configuration management - Reverse engineering may make it feasible to pass back changes to design documents during the maintenance phase of the life cycle. This would be of special significance in large, long-life systems where frequent turnover of maintenance personnel is experienced.

(4) Controlling complexity of I/O among reverse engineering tools - Currently, many experimental reverse engineering tools are under development. Since most of these are special purpose, little has been done with respect to standardization of tool interfaces or an underlying data base.

(5) Automatic layout of graphical representations - A more immediate research problem concerns the presentation of extracted graphical representations. For example, arc routing for data flow diagrams can seriously detract from the readability of data flow diagrams if not done well.

(6) Nature of re-usable artifacts - The support of software re-use is an important impetus for reverse engineering. However, a key to re-use of artifacts lies in determining and controlling their stability.

(7) Re-usable code versus understanding code - The level of re-use determines whether or not code must be understood as a prerequisite to re-use. Modules that reach the status of "built-in" functions/procedures would rarely, if ever, be read. However, a module that is seldom used or of which only a subcomponent is required would presumably have to be read, understood, and tested in order to establish a suitable level of confidence.

CONCLUDING REMARKS

The CASE '89 workshop was extremely successful in its mission of bringing together practitioners, vendors, and researchers in various areas of computer-aided software engineering. It provided an opportunity for the participants to share their ideas and experiences and, as a result, assess the current state-of-the-art of CASE tools.