A Graphically Oriented Specification Language for Automatic Code Generation

Technical Report CSE-89-05

James H. Cross II
Kelly I. Morrison
Charles H. May, Jr.
Kathryn C. Waddel
email: jcross@AUDUCVAX.bitnet

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

December, 1988

A Graphically Oriented Specification Language for Automatic Code Generation

Technical Report CSE-89-05

James H. Cross II
Kelly I. Morrison
Charles H. May, Jr.
Kathryn C. Waddel
email: jcross@AUDUCVAX.bitnet

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

December, 1988

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)**

Semi-Annual 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

December, 1988

Table of Contents

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 are based on 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 automatic generation of Ada source code. Figure 1 shows the three phases with respect to three basic classes of GRs. Phase I concentrates on the derivation of control structure diagrams (CSD) from Ada source code or Ada PDL. Phase II includes the generation of 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.

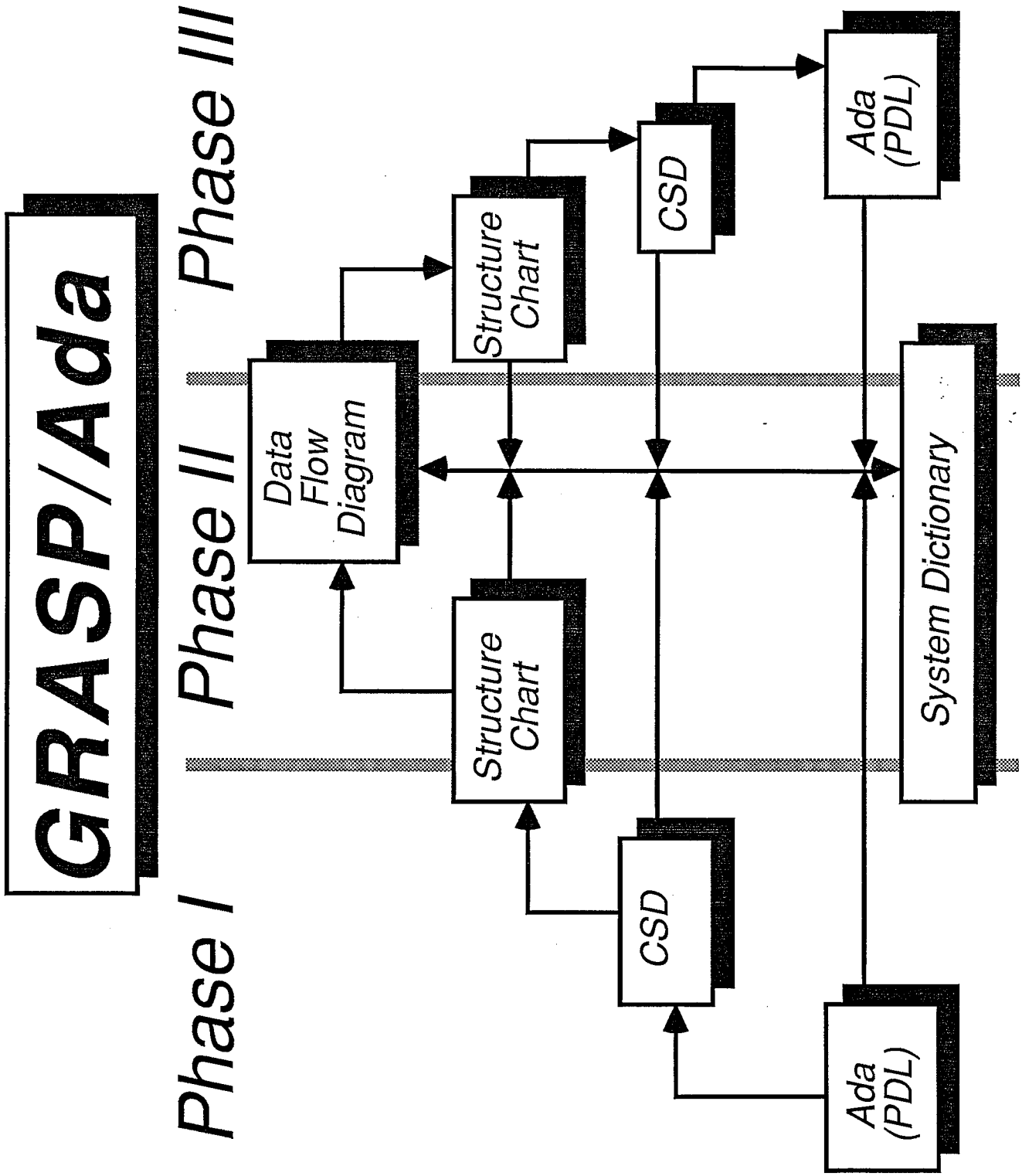

Figure 1. The Planned Three Year GRASP/Ada Research and Development Schedule.

While 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* (*G*raphical *R*epresentations for *A*lgorithms, *S*tructure, and *P*rocesses) is intended to provide a theoretical, as well as practical, foundation for the project. It includes 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 is focused on the general problem of graphical representation of several integrated views of algorithms, structure, and processes. The entire GRASP/Ada project is expected to take on an object-oriented flavor in that the concept of "object" may prove to be the most strategic and tactical way in which to organize the algorithms, structure, and processes of a software system.

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 will provide 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.

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

Figure 2(a). Sample Ada Source Code.

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

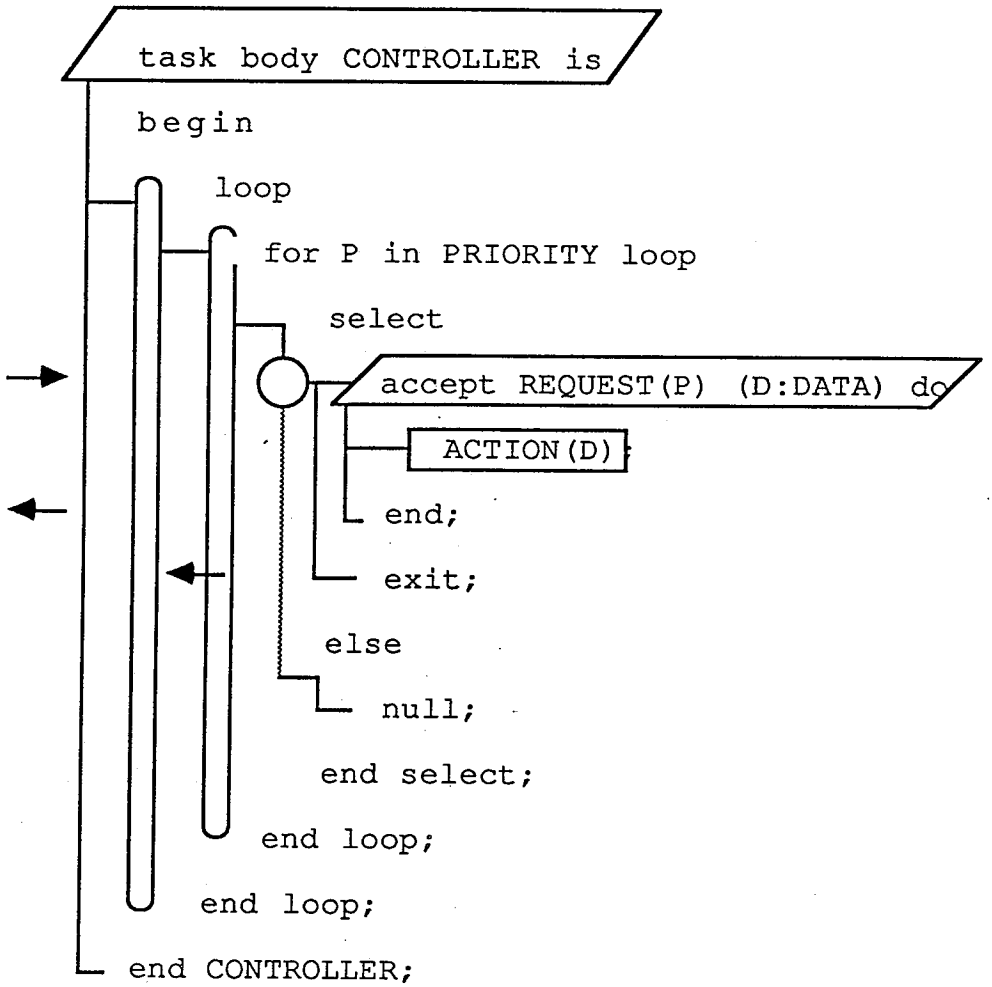

Figure 2(b). Sample Ada Source Code Overlaid with Control Structure Diagram.

Second, the GRASP/Ada automated tool 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 for algorithms, and reverse engineering. Section 3 describes the baseline requirements for Phase I of GRASP/Ada. In addition, an empirical experiment on the comprehensibility of graphical representations for algorithms is defined. Finally, section 4 describes the activities planned for the second half of Phase I.

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) <u>procedural methods</u>, which typically use high level and very high level languages; (2) <u>deductive methods</u>, which create programs after first finding "a constructive proof of the (program) specification's satisfiability"; (3) <u>transformational methods</u>, 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

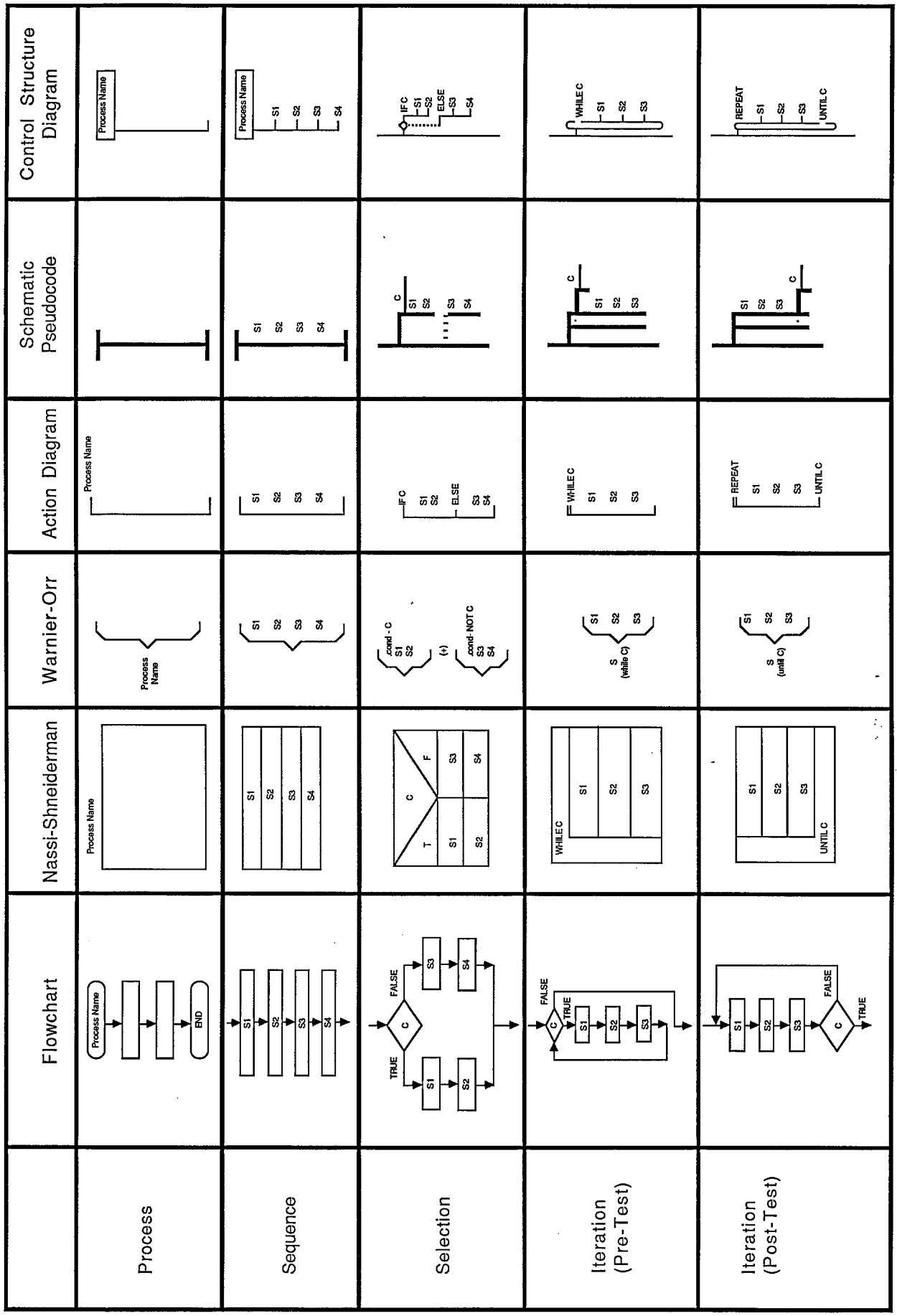

Figure 5. An Overview of Common Graphical Representations.

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 baseline requirements for the 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 tentative CSD constructs for Ada are introduced.

3.1 Introduction

The overall structure of Phase I 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 and records program information (e.g. procedure names, parameters, variable names, and scoping information) in the system dictionary. The primary output of the Phase I prototype is the procedural GR (CSD). Layered structure charts and system dictionary reports are outputs planned for late Phase I or early Phase II.

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. It accomplishes this using an attributed Ada

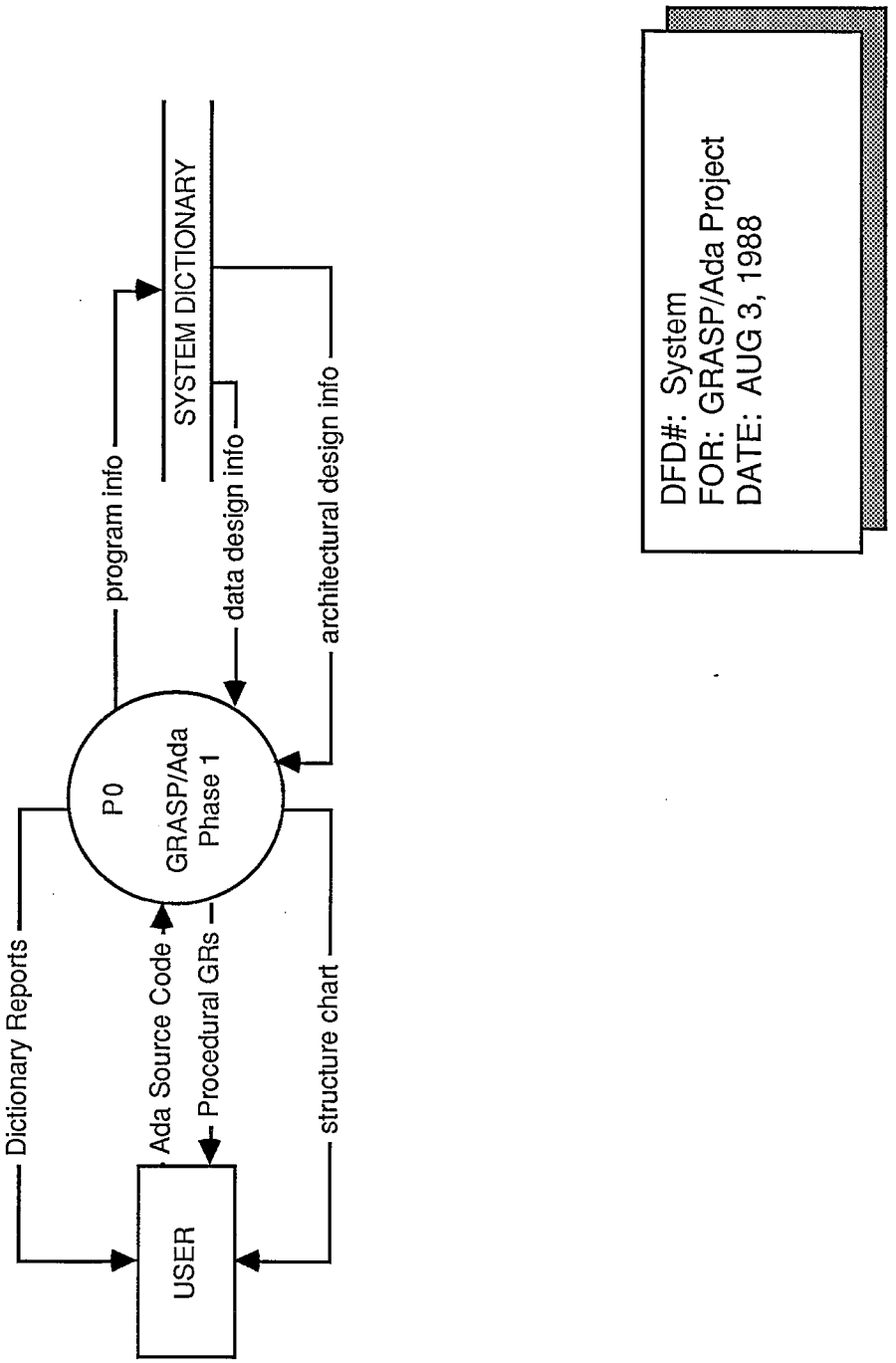

Figure 6. The System Data Flow Diagram (DFD) for GRASP/Ada (Phase I).

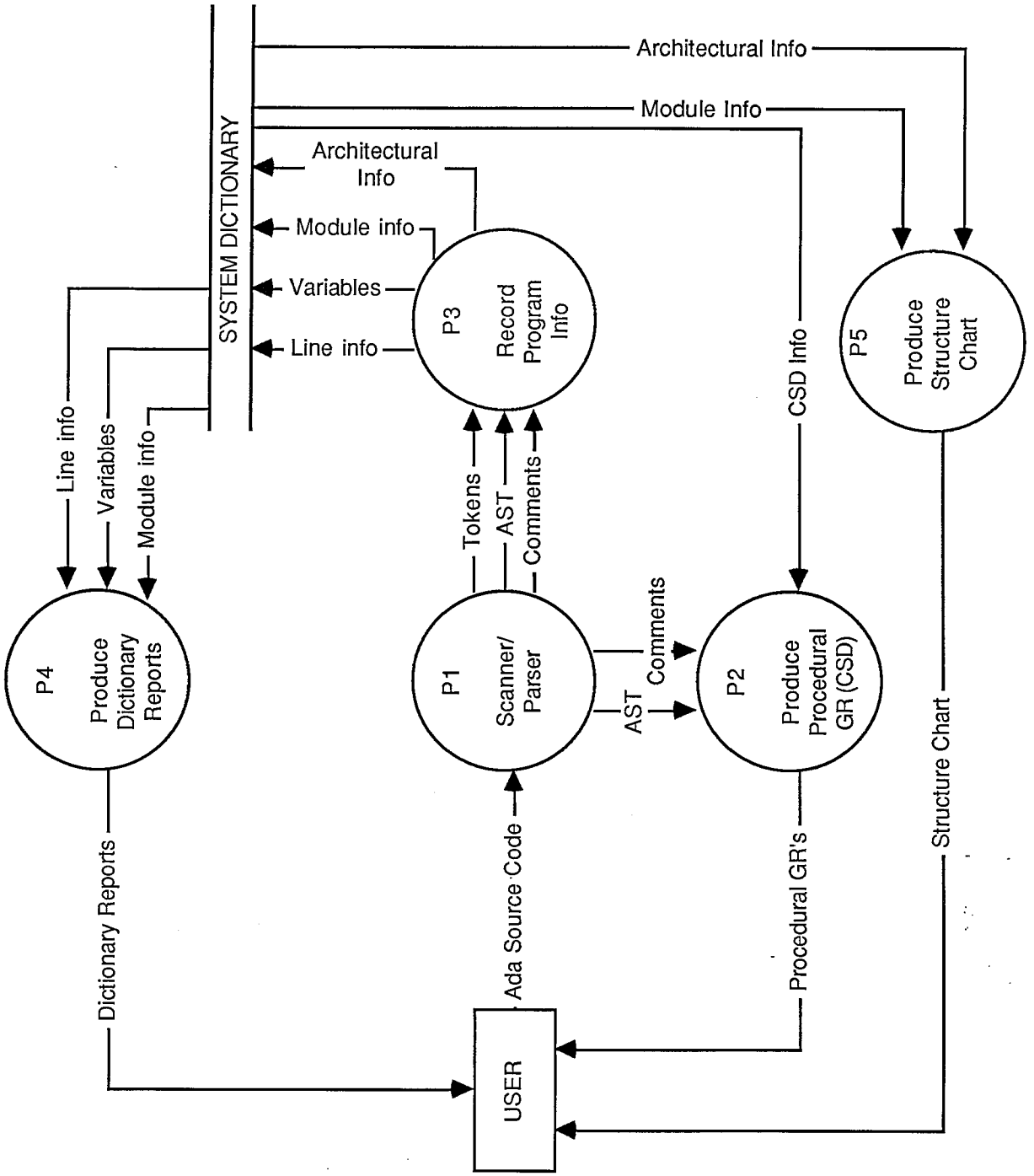

Figure 7. First Breakdown of the GRASP/Ada System (Phase I).

grammar. Certain actions for printing CSD constructs are embedded in the Ada grammar used by the parser, so that when enough of a construct has been recognized by the parser, the appropriate routine is called to produce its CSD representation. Similarly, there are embedded actions for recording assorted architectural and data information in the system dictionary.

P4 produces formatted and enhanced data dictionary reports from the system dictionary. P5 provides a standard layered structure chart that shows the modules in the system being documented. The user will have the option to either depict control and data items directly on the chart, or to list them in an accompanying table.

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 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 planned contributions that are foreseen at this particular phase of the project (Phase I) are twofold. First will be the evolution of an improved CSD with constructs for concurrency and other features specifically adapted for Ada. Second, a tool will be

provided for automatically documenting existing Ada programs with the CSD. This tool will 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

Most of the requirements for the CSD generator prototype have been purposefully generalized in order to fit a variety of implementation environments. At this time, the target environment for the CSD generator has not been finalized, although the VAX 11/780 is considered the default. 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 is 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. Mainframe 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.

Given these rather large differences in computing environments, it becomes difficult to specify exactly what the prototype should produce. If the Macintosh were selected as the target environment, the prototype would be able to produce any CSD construct that could be drawn by making use of the Macintosh's QuickDraw graphics routines. On the other hand, if the IBM PC were selected as the target environment, there would be several

possibilities. If the PC were used in text mode, then the CSD constructs would be limited to those that could be produced from the ASCII character set. If the PC were used in graphics mode, then almost any CSD construct could be represented, but at a cost of much programming overhead and program efficiency since the PC is not tailored for graphics.

The VAX 11/780 is considered the default candidate for several reasons. It is readily available (Auburn University does not have suitable development environments for the IBM PC or Apple Macintosh), and all project members have ready access to a terminal. It has C, Pascal, and Ada compilers installed, so the prototype can 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 has been 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 will be briefly discussed in the following paragraphs.

Manual coding. The first possibility was to manually code the scanner and parser totally from scratch, 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. There are several currently available both in commercial markets and in academic settings, and several

of these were considered. There were three locally available tools at Auburn University that were considered for use in the GRASP/Ada project.

CODE$IT. The first, designed by Dr. Mel Phillips of Auburn, is CODE$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 is 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 is 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 require too much time for what is a fairly unimportant part of the research effort.

LEX/YACC. The third locally available tool is the combination of LEX, a lexical analyzer generator, and YACC, an LR parser generator. Both of these tools 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, the Ada grammar is too large for our YACC version to handle.

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, Dr. Homer Carlisle, who is working with a related research contract (QUEST/Ada), examined the Ada grammar provided with the LALR tool and found that it was not an accurate Ada grammar. He 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.

FLEX/BISON. Dr. Carlisle located an accurate LALR(1) grammar and lexical description for Ada from a public domain exchange for Ada software at SIMTEL-20.ARMY. He found an updated lexical analyzer generator (FLEX) from a software library at Purdue University (J.CC.PURDUE). Finally, he 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.

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). Although no errors were found, the

scanner/parser will be tested more extensively with substantially larger and more complex Ada programs. It is expected that Ada source code from NASA projects will provide a primary testbed.

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 mean translating the scanner and parser each time they are altered, which would add 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 is 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. The automatic removal of comments was briefly considered: in certain situations, the presence of comments could complicate or confuse the CSD graphical representation, so it was decided to have a software switch for removing selected comments. Double and triple spacing certain portions of Ada code was determined to be necessary in order to properly represent certain CSD constructs such as the procedure header and package header. Options to provide forced page breaks, headers and footers will be available.

The prettyprinter will be implemented by embedding certain actions in the Ada grammar. These actions will perform certain operations such as "start a new line" or "force a page break." Still to be decided is how the software switches will be implemented. The switches will be either menu selections that may be toggled when the CSD generator is run, or formal comments that are embedded in the Ada code itself. The advantage of the first is

that the Ada code itself remains unchanged, and the program may be run in a batch setting, with no interaction from the programmer. The advantage of the second is that settings, once chosen, remain until the user changes the formal comments: there is no need to reset the options each time the program is prettyprinted.

The CSD constructs will be implemented in three steps. In each step, one of the three groups of CSD representations described above will be implemented and tested. A package of objects and operations is currently being designed which will be used to generate the CSD graphical representations. By making the primitive operations for generating the CSD object-oriented, the program will be more portable. If the CSD tool were to be moved from the VAX mainframe to, say, the Apple Macintosh II, only the underlying representation and operations would have to be modified: the upper level coding of the program would remain much the same.

Figure 8 shows a tentative object visualization of the package PREFIX which will be used to generate the CSD representations. There are two objects in the PREFIX package: (1) CSD_SYM, which is a piece of a CSD construct (for example, a LOOP_TOP), and (2) PREFIX, which is the set of CSD symbols found before a specified line of Ada code. The operations defined on these objects allow the programmer to build, print, and modify a prefix without understanding the underlying representation of these objects. Therefore, the program is relatively independent and may be modified to work on differing environments with a minimum of code rewriting.

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].

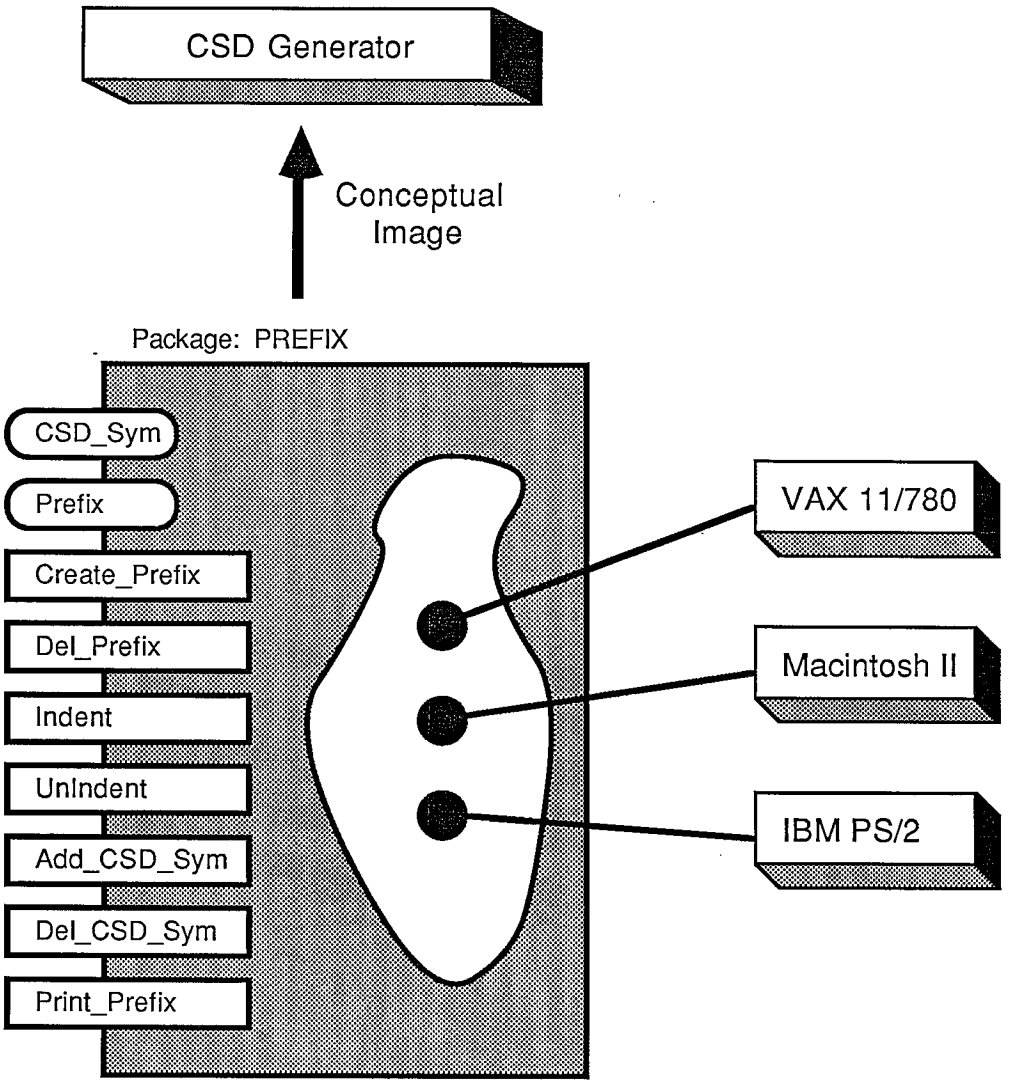

Figure 8. The PREFIX package.

One of the problems involved with creating grammatical representations for CSD constructs is that of determining how far to break up the graphical constructs. It is possible that, properly done, 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 would be of immense aid in designing 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. In order to accomplish this, it may be necessary to embed codes in the CSD, so that the same graphical symbol might have any of several different values, each having a different meaning in the reconstruction of the parse stack; i.e., graphics characters would be overloaded, so that the CSD would look the same to the user but would also carry hidden information needed by the CSD generator for parse stack reconstruction.

A tentative description of the data dictionary format has now been drafted. When it is formalized, actions will be embedded in the Ada grammar that will write the appropriate information to the data dictionary file. The requirements for the data flow diagram and structure chart aspects of the GRASP/Ada documentation will be specified during the second half of Phase I. Structure charts are fairly well defined in the literature, but data flow diagrams are rather loosely described. They need to be formalized before they are suitable for use in automatic code generation. Once a formal list of requirements for the data flow diagrams and structure charts is composed, grammatical representations akin to those for the CSD constructs will be drafted for each.

3.3 Tentative CSD Constructs for Ada

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

During the summer and fall quarters, the Ada programming language was examined, and all of the control and tasking constructs were identified. Next, tentative CSD

representations for each of these constructs were created and iteratively refined. At present, the Ada constructs have been divided into three groups. Group I (see Figure 9) 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 10) 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 11) has all of the Ada constructs related to tasking and parallel processing: task specifications, rendezvouses (both calls and receives), select statements, guarded selects, aborts, and terminations.

These constructs are expected to go through several iterations before they are finalized. Some of the criteria which may effect changes in these constructs include: readability, ease of implementation (can these be represented using ASCII characters, for example?), consistency with other CSD representations, and consistency with other graphical representations.

While developing the CSD constructs, 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 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 mainframes and IBM PC's. For this reason, the CSD constructs have been simplified as much as possible so that they can be constructed with the "graphical characters" specifically designed for the CSD. At this time, the CSD is targeted for the VAX mainframe environment, since it is 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, the CSD was targeted for microcomputer environments such as the Macintosh II or IBM PS/2, but none were readily available. Without local hardware for development, the CSD would be extremely difficult to implement for these environments.

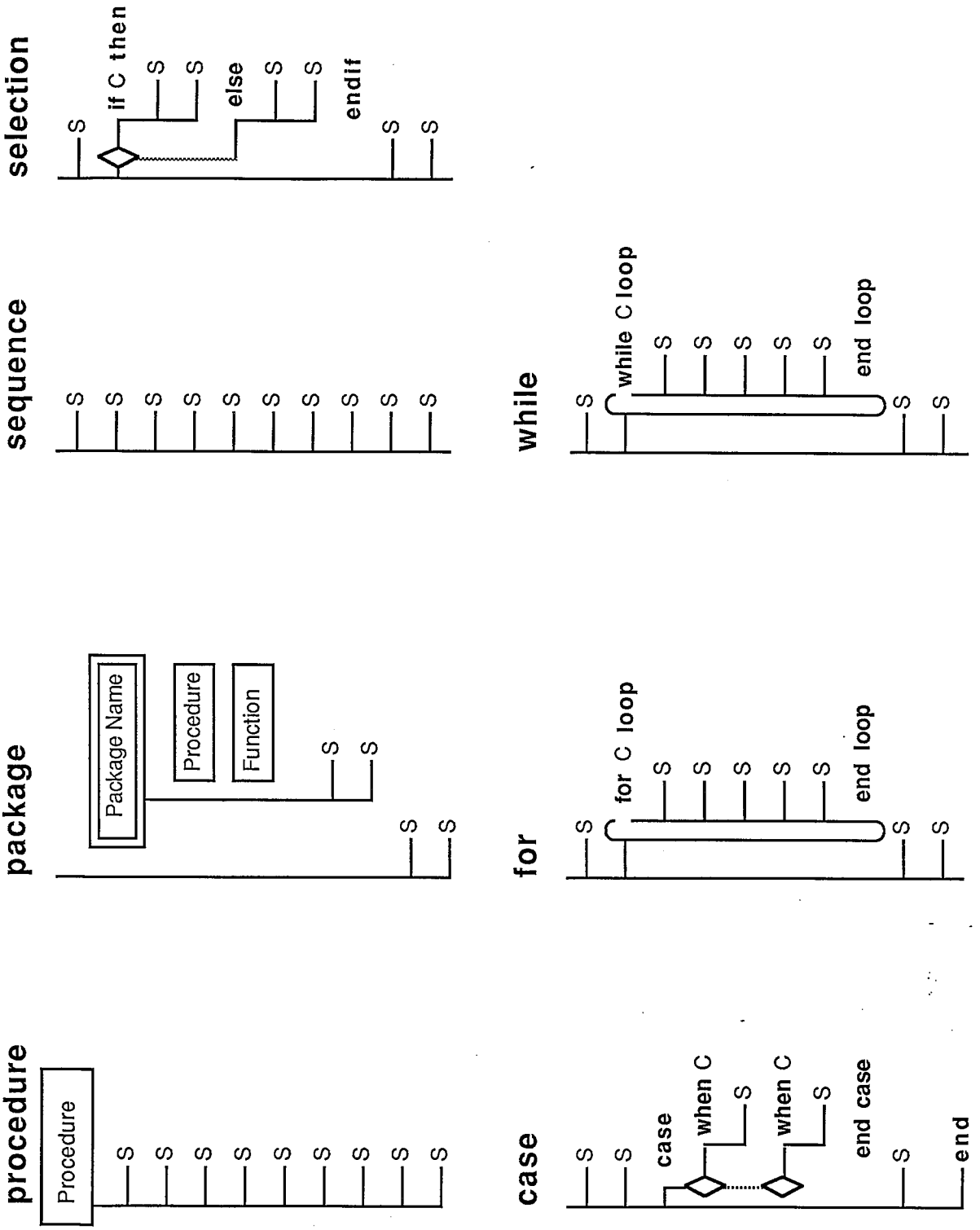

Figure 9. Group I CSD Constructs for Ada.

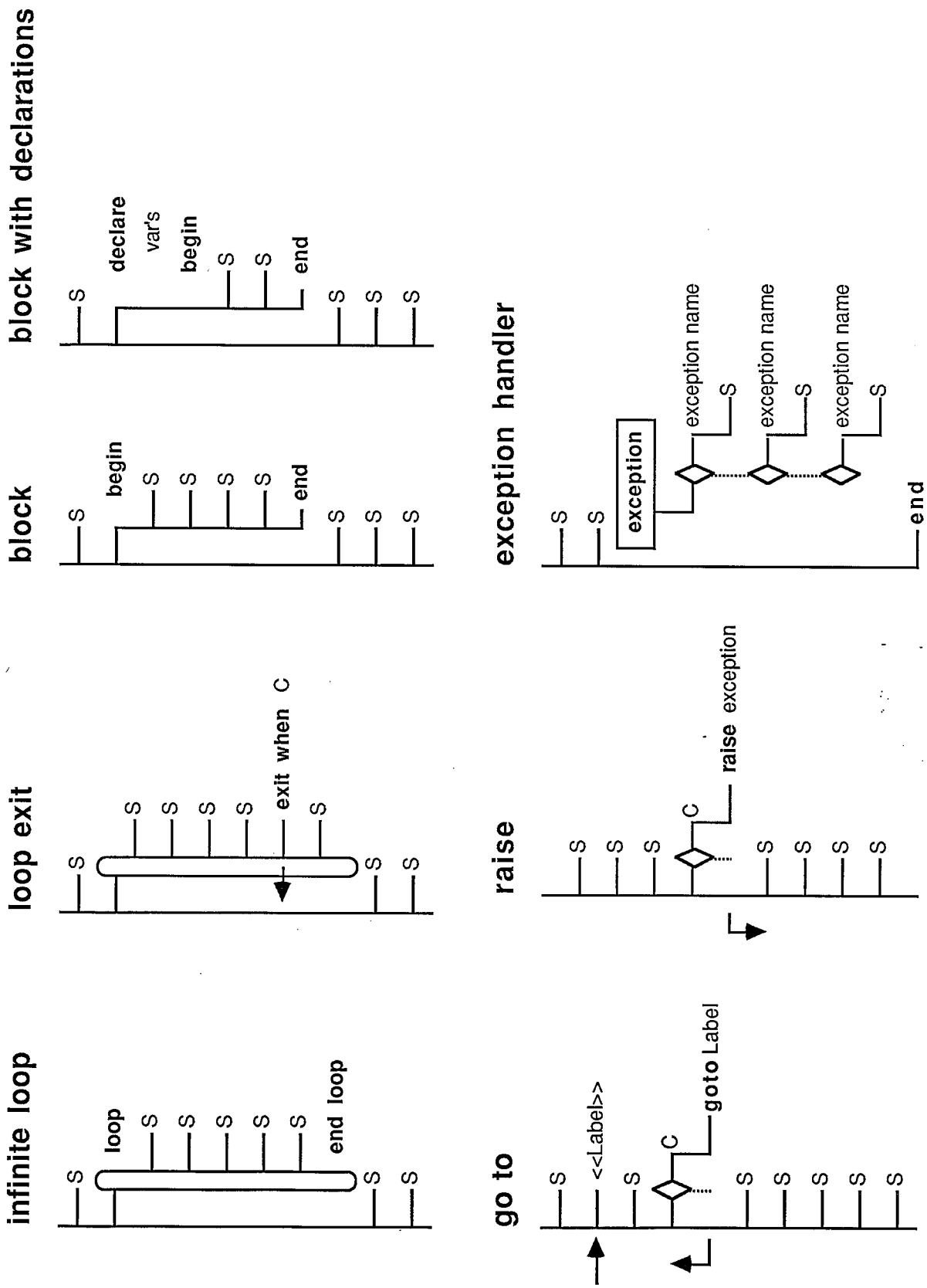

Figure 10. Group II CSD Constructs for Ada.

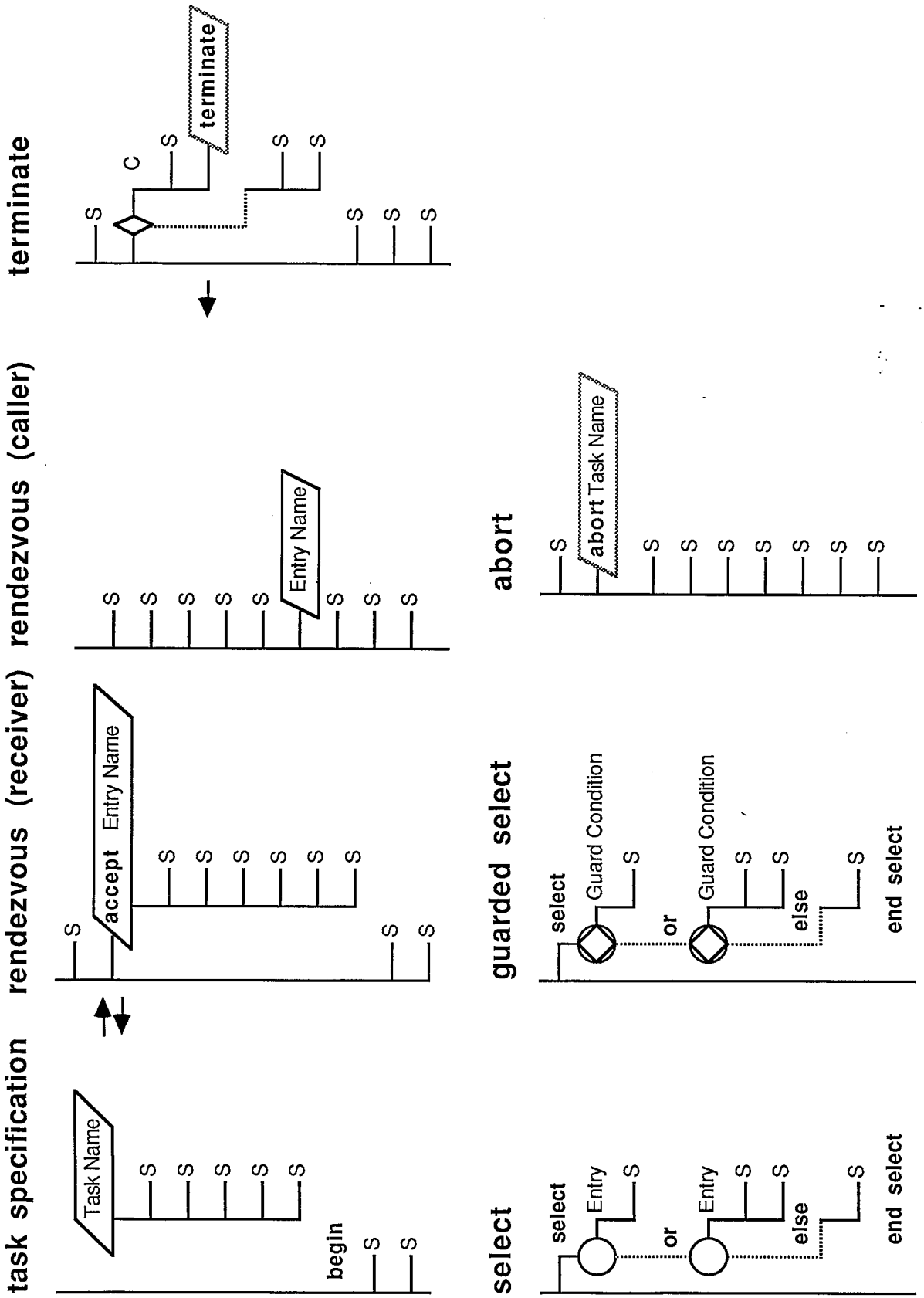

Figure 11. Group III CSD Constructs for Ada.

There are a few requirements that are still in negotiation at the moment: all of which relate to hardware. Since the target environment in which the CSD will be implemented is still uncertain, it is difficult to specify certain requirements. For example, will the Ada code be printed in at most 80 columns, or will an expanded 132 column format be considered? Will the page length be set for computer printout sheets, or for 8 1/2" by 11" pages? Will the ASCII graphics character set be available on the environment's hardcopy device?

3.4 Empirical Study Instrument

An important element of this research is determining the utility of the CSD. In this section, a formal empirical study is defined.

3.4.1 Objectives of Empirical Evaluation

There are four basic objectives and associated tasks in this research project. The first is 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 will be observed: efficiency and accuracy. These notations will be tested using subjects from two experience categories: novice programmers (less than a year of programming experience) and experienced programmers (seniors or juniors in a computer science or engineering curriculum). A third category, professional programmers, may also be included depending on early results and logistics.

The second objective is 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 efficiently debug or modify 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 is 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 will be compared. Perhaps the most preferred notation is also the most useful.

3.4.2 Overview of Procedure

Each subject from a sample of students and professional programmers will be given a brief automated summary on the use of one of the proposed notations, followed by an automated test (approximately 50 minutes). This test will contain three algorithms, each representing three difficulty levels: easy, moderate, and difficult. Each test will represent the three algorithms in one of the following formats: the conventional flowchart, the control structure diagram, or pseudocode. Thus, a given subject will observe the three algorithms in one graphical format. Each algorithm will have a number of bugs seeded in it; the subject must determine the exact nature and location of the bugs. The subject will also be asked questions about flow of control. The questions will be multiple-choice with five candidate responses each. Each question/response will be timed. Following the comprehension test, there will be a short preference survey. After a brief explanation of all the notations and how they show sequence, selection, and iteration, the subject will be asked to select and rate which of the notations he or she would use in a number of programming situations. Subsequently, statistical tests will be conducted on all the data to determine the significance of the results.

3.4.3 Implementation Plan

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

1. Develop the instrument on paper.

For each experience group (novice, experienced or expert), the tests given will be the same. Each subject will observe three algorithms (easy, moderate, and difficult) represented by one graphical format. Within each format group, the order in which the algorithms are presented will be randomized, for six possible combinations. All algorithms will contain constructs for sequence, selection, and iteration. The easy algorithm will contain simple variables, the moderate algorithm will deal with arrays and simple records, and the difficult algorithm will consist of pointers and more complex records. Possible input data but no output data will be provided. It is anticipated that the easy algorithm will be between 50 and 100 lines long, while the moderate and difficult algorithms will range between 150 and 200 lines in length.

A preference test will also be created. This survey will ask 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 will be included at the end of each comprehension test.

2. Develop the automated instruments.

Once all algorithms have been determined and the questions about them developed, they will be transferred onto an IBM PC-compatible computer. Then the user interface will be produced. This interface will first take the user through a discussion of the notation and the way it represents selection, iteration, and sequence. When the subject is ready to begin the test, the computer will give instructions about the content of the test and how the questions are to be answered. The user will go through each algorithm, answering questions about where each bug is located and about general control flow. The interface

will allow the subject to answer only one question at a time; once answered, the subject cannot go back to a previous question. The user will be allowed to view the algorithm as long as required. Each question is timed. As an answer is given, the computer will store the question, the response, and the elapsed time on a record that was made for each subject. When the user is finished with the test, the interface will display a "sign-off" message and prepare for the next subject, who will be 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 (around 30) the instruments must be tested to make sure there are no problems and that they test what was intended. For example, one of the algorithms may prove to be too simple for the experience level and must be replaced. The presentation of the test must also be checked. If text is so condensed that it is hard to read, or the instructions unclear, then these problems must be corrected.

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

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

After the instruments have been developed and tested, they will be implemented at Auburn University. In addition, students at other universities may be utilized in order to attain a sample size sufficient for our results to be statistically accurate.

5. Statistics and evaluations.

Once all the data is collected and stored, the IBM mainframe version of SAS will be used to analyze and determine the results. A more detailed explanation of the statistics is presented in the next section.

3.4.4 Analysis of the Data

Two SAS analyses will be conducted: one on the comprehension test and one on the preference survey. The model, in SAS, for the comprehension test will be:

Response_time Score = GRA Skill GRA x Skill ID (GRA Skill)

Algorithm Algorithm x Skill Algorithm x GRA Algorithm x Skill x GRA.

Response_time and Score are the dependent variables. The other measures are independent variables.

A multiple analysis of variance (MANOVA) will be used on this data with the aid of SAS, a powerful automated statistics package, on the IBM mainframe. This analysis will answer the following questions:

- Which notation is the most accurate? most efficient? most preferred?
- Is the most efficient notation also the most accurate?
- Is the most efficient and the most accurate notation necessarily the most preferred?
- For what task is the most preferred notation most likely to be used?

These tests will be compared across experience levels so that comparisons can be made between novices, experienced and expert programmers. Also, preferences will be compared between these three groups.

There may be a problem of bias in favor of the flowchart for students at universities (or professional programmers on the job) where flowcharts are emphasized. They tend to be more comfortable with this notation simply because they have had more experience with it. There is no way to eliminate bias such as this from the data, but two attempts can be made to compensate for this problem: (1) make sure that all the subjects have experience in using each of the formats and (2) include a question in the test asking the subject how many quarters or months of experience he or she has had in using the flowchart, the control structure diagram, pseudocode, and other graphical representations. This data can be a part of the analysis, to observe the correlation, if any, of subject experience with preference and

comprehension. After the comprehension test, the easy algorithm will be presented in each notation. This will familiarize the subject with each so that he or she can better respond to a series of preference type questions.

4.0 Future Work (December 1988 - May 1989)

The work planned for December 1988 - May 1989 is summarized in the Gantt charts in Figures 12 and 13. The most critical task for the immediate future is to finalize the overall architectural design for GRASP/Ada. It has been stated previously that the project will assume an object-oriented perspective, and thus it is necessary to identify the relevant objects and their associations.

The bulk of the work for the winter quarter and much of the spring quarter will be the implementation of the operational prototype of the CSD generator. The first step in this task will be to implement a prettyprinter for Ada code. The implementation of the CSD constructs has been divided into three steps, corresponding to the partition of the CSD constructs described above. Each step of the implementation will be appropriately tested with Ada code containing the corresponding control constructs. A package of objects and operations is currently being designed to generate the CSD graphical representations. It is hoped that this object-oriented approach will facilitate the transport of the CSD tool to other environments.

Another task to be confronted is the design of appropriate actions for extracting information from the Ada programs for insertion into the system dictionary. The system dictionary will contain most of the information needed in Phase II of the GRASP/Ada project for generation of the structure charts and data flow diagrams. A tentative description of the system dictionary has been drafted; once it is formalized, actions will be embedded in the Ada grammar that will write the appropriate information to the dictionary.

In addition, the requirements for a structure chart generator and a data flow diagram generator will be specified. The analysis phase for these tools is expected to be somewhat symbiotic with the development of the system dictionary, in that the analysis phases for these tools should reveal needed system dictionary contents which may have been previously overlooked. Structure charts are fairly well defined in the literature; the corresponding analysis should not present too many surprises. Data flow diagrams, on the

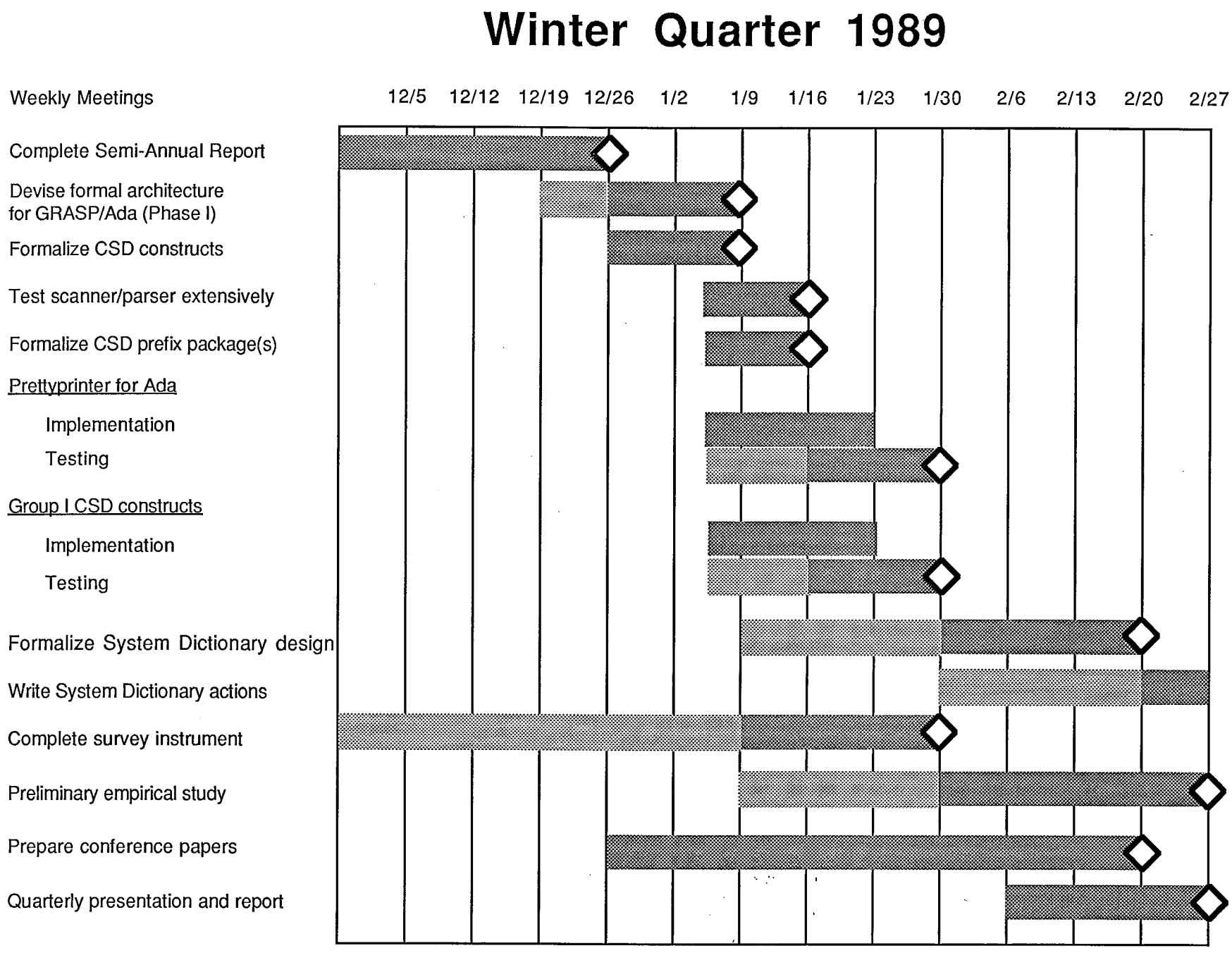

Figure 12. GRASP/Ada Project Schedule

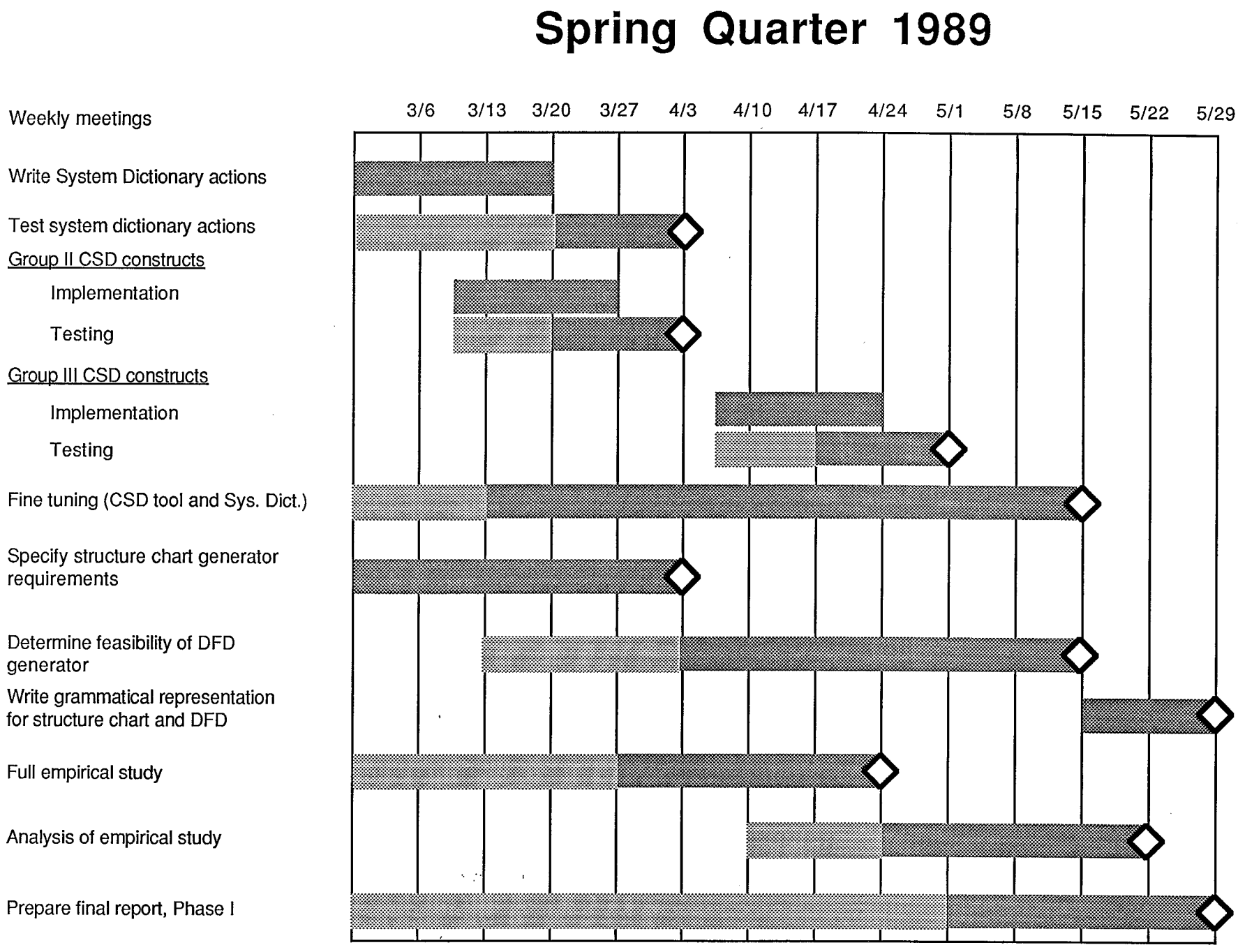

Figure 13. GRASP/Ada Project Schedule

other hand, are not as well defined; a strict formalization is in order before data flow diagrams will be suitable for use in automatic code generation. The analysis for a data flow diagram generator is expected to take the form of a feasibility study.

The empirical study of algorithmic representations will also continue during this period. The most immediate task is the completion of the instrument with which this study will be conducted. Once the instrument is completed, a preliminary study will by conducted on a small sample of roughly thirty experienced programmers (i.e. junior- and senior-level computer science students). This will affirm the effectiveness of the instrument, particularly regarding the effectiveness of the questions posed to the subjects; appropriate adjustments will afterwards be made. In addition, the preliminary study will determine the sample size of the main empirical study which will take place during the spring quarter.

Finally, preparation of papers reporting the work in progress will be conducted primarily during the late winter quarter. These papers are planned for the Twenty-Seventh Annual Conference of the Southeast Region of the ACM to be held in Atlanta in early April. Possible topics include the tentative CSD constructs for Ada and the plans for the empirical study of algorithmic representations.

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. *Collected Software Engineering Papers: Volume VI*. November 1988.

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.